МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

ЧЕРКАСЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ ІМЕНІ БОГДАНА ХМЕЛЬНИЦЬКОГО

Авраменко В.С., Авраменко А.С.

ПРОЕКТУВАННЯ ІНФОРМАЦІЙНИХ СИСТЕМ

Навчальний посібник

Черкаси 2017

УДК 004.4 (075.8)
ББК 32.973.2-018я73-1
А21

Рецензенти: *В.М. Рудницький,* доктор технічних наук, професор, завідувач кафедри інформаційної безпеки та комп'ютерної інженерії Черкаського державного технологічного університету;
Г.В. Косенюк, кандидат технічних наук, доцент кафедри інформаційних технологій Черкаського національного університету імені Богдана Хмельницького.

Рекомендовано до друку рішенням Вченої ради
Черкаського національного університету імені Богдана Хмельницького
(Протокол № 5 від 05.04.2016 р.)

Авраменко В.С., Авраменко А.С.
А21 Проектування інформаційних систем: навчальний посібник / В.С. Авраменко, А.С. Авраменко. – Черкаси: Черкаський національний університет ім. Б. Хмельницького, 2017. – 434 с.: іл.

ISBN 978-966-920-208-6

Навчальний посібник призначений для студентів, які навчаються за спеціальностями «121 Інженерія програмного забезпечення», «122 Комп' ютерні науки та інформаційні технології», «124 Системний аналіз» з дисципліни «Проектування інформаційних систем». Матеріал посібника може використовуватися і в практичній діяльності студентів інших спеціальностй і фахівців, що займаються розробкою і впровадженням інформаційних систем.

Мета цього посібника – формування навичок самостійного практичного застосування сучасних методів і засобів проектування інформаційних систем. У посібнику детально описані методи аналізу та проектування інформаційних систем з використанням мови UML.

УДК 004.4 (075.8)
ББК 32.973.2-018я73-1
ISBN 978-966-920-208-6

ЗМІСТ

ВСТУП **12**
ЧАСТИНА 1. ОРГАНІЗАЦІЯ ПРОЦЕСУ РОЗРОБКИ ІС **15**
РОЗДІЛ 1. ОСНОВНІ ПОНЯТТЯ ТЕХНОЛОГІЇ ПРОЕКТУВАННЯ ІС ... **15**
1.1. Тенденції розвитку інформаційних технологій 15
1.2. Поняття інформаційної системи 17
1.3. Етапи розвитку інформаційних систем 17
1.4. Порівняння ІС з традиційними програмними продуктами 19
1.5. Класифікація інформаційних систем 20
1.6. Сфери застосування і приклади реалізації ІС 23
Контрольні питання до розділу 1 24
РОЗДІЛ 2. МЕТОДОЛОГІЇ І ТЕХНОЛОГІЇ РОЗРОБКИ ІС **25**
2.1. Організація процесу розробки 25
2.1.1. Основні поняття програмної інженерії 25
2.1.2. Основні процеси життєвого циклу ІС 26
2.1.3. Базис процесів розробки ІС 26
2.2. Життєвий цикл програмного забезпечення ІС 27
2.3. Моделі життєвого циклу ІС 29
2.3.1. Стратегії конструювання ІС 29
2.3.2. Класична або каскадна модель 29
2.3.3. Компонентні моделі 34
2.3.4. Макетування (прототипування) 34
2.3.5. Ітеративні (інкрементні) моделі 35
2.3.6. Спіральна модель 36
Контрольні питання до розділу 2 37
РОЗДІЛ 3. ТЕХНОЛОГІЇ СТВОРЕННЯ ІС **38**
3.1. Канонічне проектування ІС 38
3.1.1. Організація канонічного проектування ІС 38
3.1.2. Обстеження об’єкту і обгрунтування створення ІС 39
3.1.3. Технічне завдання 40
3.1.4. Ескізний проект 41
3.1.5. Технічний проект 42

3.1.6. Робоча документація і випробування ІС 42
3.2. Уніфікований процес Rational 43
3.2.1. Основні принципи RUP 43
3.2.2. Фази проекту 43
3.2.3. Ключові ідеї RUP 44
3.3. Екстремальне програмування (XP-процес) 47
3.4. Scrum методологія 49
3.4.1. Історія 49
3.4.2. Загальний опис 50
3.4.3. Ролі 50
3.4.4. Практики 50
Контрольні питання до розділу 3 51
РОЗДІЛ 4. УПРАВЛІННЯ ПРОЕКТОМ ПРИ РОЗРОБЦІ ІС 52
4.1. Основні поняття управління проектом 52
4.1.1. Поняття проекту 52
4.1.2. Підбір команди і організаційні питання 52
4.1.3. Керівник проекту 55
4.1.4. Командні ролі 56
4.1.5. Комунікації 57
4.2. Планування програмного проекту 58
4.2.1. Структура плану управління програмним проектом 59
4.2.2. Структура графіку робіт програмного проекту 60
4.3. Управління персоналом 61
4.3.1. Підбір членів команди 61
4.3.2. Взаємодії в команді 62
4.3.3. Склад групи 64
4.4. Процес розробки 64
4.5. Прогнозуюче і адаптивне планування 66
4.6. Вибір процесу розробки 67
4.7. Налаштування UML під процес 69
4.8. Управління конфігурацією 70
Контрольні питання до розділу 4 71

ЧАСТИНА 2. ІНСТРУМЕНТАЛЬНІ ЗАСОБИ ПРОЕКТУВАННЯ ІС 72
РОЗДІЛ 5. ОСНОВНІ ВІДОМОСТІ ПРО МОВУ UML 72
5.1. Поняття UML 73
5.1.1. UML – це мова 73
5.1.2. UML – це мова моделювання 74
5.1.3. UML – це уніфікована мова моделювання 74
5.1. 4. Історія UML 75
5.2. Призначення UML 76
5.2.1. Специфікація і візуалізація 77
5.2.2. Проектування і документування 78
5.2.3. Інструментальна підтримка 79
5.2.4. Способи використання UML 80
5.2.5. Метод визначення UML 80
5.2.6. Термінологія і нотація 81
5.3. Модель і її елементи 83
5.3.1. Сутності 83
5.3.2. Відношення 85
5.3.3. Діаграми 87
5.3.4. Класифікація діаграм 89
5.4. Загальні діаграми 92
5.4.1. Діаграма використання 92
5.4.2. Діаграма класів 93
5.4.3. Діаграма автомата 94
5.4.4. Діаграма діяльності 95
5.4.5. Діаграма послідовності 95
5.4.6. Діаграма комунікації 97
5.4.7. Діаграма компонентів 97
5.4.8. Діаграма розміщення 98
5.5. Спеціальні діаграми 99
5.5.1. Діаграма об’єктів 99
5.5.2. Діаграма внутрішньої структури 100
5.5.3. Оглядова діаграма взаємодії 101

5.5.4. Діаграма синхронізації 102
5.5.5. Діаграма пакетів 102
5.6. Моделі і їх представлення 103
5.6.1. Призначення і рівні моделей 103
5.6.2. Класичні представлення з UML 1 і 2 105
5.7. Механізми розширення 108
Контрольні питання до розділу 5 110
РОЗДІЛ 6. ПРАКТИКА ЗАСТОСУВАННЯ UML 111
6.1. Рівні моделювання 111
6.2. Практика застосування елементів UML 112
6.3. Вплив UML на процес розробки 115
6.3.1. Технологія програмування 116
6.3.2. Поради з впровадження UML в організації 118
Контрольні питання до розділу 6 119
РОЗДІЛ 7. ОГЛЯД CASE-ЗАСОБІВ ДЛЯ ПОБУДОВИ ДІАГРАМ UML . 120
7.1. IBM Rational Rose 120
7.2. Borland Together 121
7.3. Microsoft Visio 122
7.4. StarUML, Dia, Draw.io 122
7.5. Основні відомості про CASE-засоби Rational Rose 123
7.5.1. Введення в Rational Rose 123
7.5.2. Робота в середовищі Rational Rose 124
Контрольні питання до розділу 7 131
ЧАСТИНА 3. СТРУКТУРНИЙ АНАЛІЗ І ПРОЕКТУВАННЯ ІС 132
РОЗДІЛ 8. ЗАСТОСУВАННЯ СТРУКТУРНОГО ПІДХОДУ В АНАЛІЗІ ВИМОГ І ВИЗНАЧЕННІ СПЕЦИФІКАЦІЙ ПЗ 133
8.1. Основні відомості 133
8.2. Діаграми переходів станів 135
8.3. Функціональні діаграми 138
8.4. Діаграми потоків даних 141
8.5. Діаграма «сутність-зв’язок» 145
Контрольні питання до розділу 8 150

РОЗДІЛ 9. ПРОЕКТУВАННЯ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ ПРИ СТРУКТУРНОМУ ПІДХОДІ 151
9.1. Структурна схема 151
9.2. Функціональна схема 151
Контрольні питання до розділу 9 152
ЧАСТИНА 4. ОБ'ЄКТНО-ОРІЄНТОВАНИЙ АНАЛІЗ І ПРОЕКТУВАННЯ ІС 153
РОЗДІЛ 10. ОБ'ЄКТНО-ОРІЄНТОВАНА РОЗРОБКА ВИМОГ 154
10.1. Моделювання предметної області 154
10.1.1. Поняття предметної області 154
10.1.2. Основні елементи моделювання предметної області 159
10.2. Формування та аналіз вимог 163
10.2.1. Види вимог до програмного забезпечення 164
10.2.2. Формування вимог 167
10.2.3. Аналіз вимог 169
10.2.4. Специфікація вимог 176
10.2.5. Управління вимогами 177
10.3. Формування вимог за допомогою діаграми Use Case 177
10.3.1. Актори і елементи Use Case 180
10.3.2. Відношення в діаграмах Use Case (використання) 183
10.3.3. Робота з елементами Use Case 186
10.3.4. Специфікація елементів Use Case 187
10.3.5. Банкомат – приклад діаграми Use Case 188
10.3.6. Побудова моделі вимог 192
10.3.7. Реалізація варіантів використання 197
10.3.8. Рекомендації до розробки діаграм варіантів використання 198
10.4. Аналіз вимог за допомогою діаграм діяльності 199
10.4.1. Характеристика діаграм діяльності 202
10.4.2. Декомпозиція операції в діаграмі діяльності 205
10.4.3. Розділи діаграми діяльності 206
10.4.4. Рекомендації до побудови діаграм діяльності 207
10.5. Аналіз вимог за допомогою діаграм взаємодії 208
10.5.1. Об'єкти і ролі 209

10.5.2. Діаграми взаємодії 210
10.5.3. Діаграми послідовностей 210
10.5.4. Діаграми комунікації 218
10.5.5. Рекомендації з побудови діаграм послідовності і комунікації 222
Контрольні питання до розділу 10 223
РОЗДІЛ 11. ОБ'ЄКТНО-ОРІЄНТОВАНЕ ПРОЕКТУВАННЯ ІС 224
11.1. Принципи моделювання структури 224
11.1.1. Дескриптори 225
11.1.2. Призначення структурного моделювання 225
11.1.3. Класифікатори 228
11.1.4. Ідентифікація класів 232
11.2. Архітектурне проектування 233
11.2.1. Діаграми пакетів 234
11.2.2. Діаграми компонентів 238
11.3. Детальне проектування 243
11.3.1. Діаграми класів 244
11.3.2. Приклади діаграм класів 253
11.3.3. Створення початкової діаграми класів 255
11.4. Розгортання програмної системи на апаратних засобах 256
11.4.1. Артефакти і вузли 257
11.4.2. Побудова діаграм розгортання 260
Контрольні питання до розділу 11 263
РОЗДІЛ 12. МОДЕЛЮВАННЯ ПОВЕДІНКИ СИСТЕМИ 264
12.1. Модель поведінки 264
12.2. Діаграма кінцевого автомата 266
12.3. Дії в станах 268
12.4. Умовні переходи 269
12.5. Композитні (складені) стани 270
12.6. Псевдостани управління 274
12.7. Застосування діаграм кінцевих автоматів 278
12.8. Діаграми діяльності 279
12.8.1. Дія і діяльність 279

12.8.2. Граф діяльності ... 280
12.9. Діаграми взаємодії ... 282
12.9.1. Повідомлення ... 283
12.9.2. Діаграми послідовності ... 284
12.9.3. Діаграми комунікації ... 286
Контрольні питання до розділу 12 ... 287
РОЗДІЛ 13. ОСОБЛИВОСТІ РОЗРОБКИ БАЗ ДАНИХ З ЕЛЕМЕНТАМИ UML ... 288
13.1. Основні поняття баз даних: моделі даних ... 288
13.2. Розширення UML для моделювання баз даних ... 288
13.2.1. Типи моделей даних ... 289
13.2.2. Таблиці, сутності, представлення і відношення ... 290
13.2.3. Ключі, обмеження, тригери і процедури, що зберігаються ... 292
13.3. Відображення атрибутів об'єктів і класів в реляційну БД ... 294
13.4. Відображення дерев спадкоємства в реляційну БД ... 296
13.4.1. Відображення дерева спадкоємства в єдину таблицю ... 296
13.4.2. Відображення кожного конкретного класу в окрему таблицю ... 297
13.4.3. Відображення кожного класу в окрему таблицю ... 298
Контрольні питання до розділу 13 ... 299
РОЗДІЛ 14. ПРОЕКТУВАННЯ ІНТЕРФЕЙСУ КОРИСТУВАЧА ... 300
14.1. Основні правила створення інтерфейсу ... 300
14.2. Принципи розробки інтерфейсу користувача ... 301
14.3. Взаємодія між користувачем і комп'ютером ... 301
14.4. Розміщення інформації на екрані ... 302
14.5. Запобігання, виявлення і виправлення помилок ... 307
Контрольні питання до розділу 14 ... 308
РОЗДІЛ 15. ВИБІР СТРАТЕГІЇ ТЕСТУВАННЯ І РОЗРОБКА ТЕСТІВ ... 309
15.1. Рівні тестування ... 309
15.2. Технології тестування ... 310
15.3. Програмні помилки ... 310
Контрольні питання до розділу 15 ... 311
ЧАСТИНА 5. ПРИКЛАДИ МОДЕЛЮВАННЯ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ ІС ... 312

РОЗДІЛ 16. МОДЕЛЮВАННЯ ВИКОРИСТАННЯ ІС ВІДДІЛУ КАДРІВ 312
16.1. Значення моделювання використання 313
16.1.1. Підходи до моделювання 314
16.1.2. Переваги моделювання використання 315
16.2. Діаграми використання 315
16.2.1. Дійові особи 316
16.2.2. Варіанти використання 316
16.2.3. Коментарі 317
16.2.4. Відношення на діаграмах використання 318
16.2.5. Способи застосування моделей використання 323
16.3. Реалізація варіантів використання 325
16.3.1. Реалізація діаграмами діяльності 325
16.3.2. Реалізація діаграмами взаємодії 328
РОЗДІЛ 17. МОДЕЛЮВАННЯ СТРУКТУРИ ІС ВІДДІЛУ КАДРІВ 331
17.1. Сутності на діаграмі класів 331
17.2. Відношення на діаграмі класів 334
17.3. Діаграми реалізації 348
РОЗДІЛ 18. МОДЕЛЮВАННЯ ПОВЕДІНКИ ІС ВІДДІЛУ КАДРІВ 354
18.1. Моделювання поведінки за допомогою кінцевих автоматів 354
18.1.1. Кінцеві автомати 354
18.1.2. Діаграми станів (автомата) 355
18.2. Діаграми діяльності 361
18.2.1. Дія і діяльність 361
18.2.2. Граф діяльності 362
18.2.3. Доріжки і розбиття 364
18.2.4. Траєкторія об’єкту і потік даних 367
18.3. Діаграми взаємодії 370
18.3.1. Повідомлення 370
18.3.2. Діаграми послідовності 371
18.3.3. Діаграми комунікації 374
РОЗДІЛ 19. ВИКОНАННЯ НАВЧАЛЬНОГО ПРОЕКТУ В СЕРЕДОВИЩІ RATIONAL ROSE 376

19.1. Постановка завдання 376
19.2. Складання глосарію проекту 377
19.3. Опис додаткових специфікацій 377
19.4. Створення моделі варіантів використання 378
19.5. Аналіз системи 386
19.5.1. Архітектурний аналіз 386
19.5.2. Аналіз варіантів використання 389
19.6. Проектування системи 400
19.6.1. Проектування архітектури 400
19.6.2. Моделювання розподіленої конфігурації системи 405
19.6.3. Проектування класів 407
19.6.4. Проектування баз даних 413
19.7. Реалізація системи 415
19.7.1. Створення компонентів 415
19.7.2. Генерація коду 416
РОЗДІЛ 20. ВАРІАНТИ ЗАВДАНЬ ДЛЯ САМОСТІЙНОЇ РОБОТИ 419
20.1. Постановка завдання на проектування ІС 419
20.2. Варіанти завдань на проектування ІС 420
СПИСОК ЛІТЕРАТУРИ 430

ВСТУП

Розробка інформаційної системи (ІС) – від початкової фази до розгортання – складається з трьох послідовних і поступальних етапів: аналізу, проектування і реалізації. У посібнику описуються методи і прийоми, використовувані на етапах аналізу і проектування. Питання, пов'язані з реалізацією, включаючи приклади програм, розглядаються тільки в тій мірі, в якій вони потрібні на етапі проектування.

Проектування ІС – логічно складна, трудомістка і тривала робота, що вимагає високої кваліфікації фахівців, що беруть участь в ній. Проте до теперішнього часу проектування ІС нерідко виконується на інтуїтивному рівні неформалізованими методами, що включають елементи мистецтва, практичний досвід, експертні оцінки і дорогі експериментальні перевірки якості функціонування ІС. Крім того, в процесі створення і функціонування ІС інформаційні потреби користувачів постійно змінюються або уточнюються, що ще більше ускладнює розробку і супровід таких систем.

Основна доля трудовитрат при створенні ІС доводиться на прикладне програмне забезпечення (ПЗ) і бази даних (БД). Виробництво ПЗ сьогодні – найбільша галузь світової економіки, в якій зайняті близько трьох мільйонів фахівців (програмістів, розробників ПЗ і т. п.).

На початку 70-х рр. в США була відмічена криза програмування (software crisis). Це виражалося в тому, що великі проекти стали виконуватися з відставанням від графіку або з перевищенням кошторису витрат, розроблений продукт не мав необхідних функціональних можливостей, продуктивність його була низька, якість отримуваного програмного забезпечення не влаштовувала споживачів.

У числі причин можливих невдач фігурують: нечітке і неповне формулювання вимог до ПО, недостатнє залучення користувачів в роботу над проектом, відсутність необхідних ресурсів, незадовільне планування, часту зміну вимог і специфікацій, новизна використовуваної технології для організації, відсутність грамотного управління проектом, недостатня підтримка з боку вищого керівництва.

Потреба контролювати процес розробки ПО, прогнозувати і гарантувати вартість розробки, терміни і якість результатів привела у кінці 70-х рр. до необхідності переходу від кустарних до індустріальних способів створення ПЗ і появи сукупності інженерних методів і засобів створення ПО, об'єднаних загальною назвою «Програмна інженерія» (software engineering).

В процесі становлення і розвитку програмної інженерії можна виділити два етапи: 70-і і 80-і рр. – систематизація і стандартизація процесів створення ПЗ (на основі структурного підходу) і 90-і рр. – початок переходу до складального, індустріального способу створення ПЗ (на основі об'єктно-орієнтованого підходу).

У основі програмної інженерії лежить одна фундаментальна ідея: *проектування ПЗ є формальним процесом, який можна вивчати і удосконалювати* [13]. Освоєння і правильне застосування методів і засобів

створення ПЗ дозволять підвищити якість ІС, забезпечити керованість процесу проектування ІС і збільшити термін її життя.

Як відмічає Фредерік Брукс, керівник проекту розробки операційної системи OS/360, найістотнішою і невід'ємнішою властивістю програмних систем є їх складність. Завдяки унікальності і несхожості своїх складових частин програмні системи принципово відрізняються від технічних систем (наприклад, комп'ютерів), в яких переважають елементи, що повторюються.

Самі комп'ютери складніші, ніж більшість продуктів людської діяльності. Кількість їх можливих станів дуже велика, тому їх так важко розуміти, описувати і тестувати. У програмних систем кількість можливих станів на порядок величин перевищує кількість станів комп'ютерів.

Складність ПЗ є істотною, а не другорядною властивістю. Багато проблем розробки ПЗ виходять з цієї складності і її нелінійного росту при збільшенні розміру. Складність є причиною утруднень, що виникають в процесі спілкування між розробниками, що веде до помилок в продукті, перевищення вартості розробки, затягування виконання графіків робіт. Складність викликає труднощі розуміння усіх можливих станів програм, що призводить до зниження їх надійності. Складність структури стримує розвиток ПЗ і можливості додавання нових функцій.

Для успішної реалізації проекту об'єкт проектування (ПЗ) має бути передусім адекватно описаний, тобто мають бути побудовані повні і несуперечливі моделі архітектури ПО, що обумовлює сукупність структурних елементів системи і зв'язків між ними, поведінку елементів системи в процесі їх взаємодії, а також ієрархію підсистем, що об'єднують структурні елементи.

Під моделлю розуміється повний опис системи ПЗ з певної точки зору. Моделі є засобами для візуалізації, опису, проектування і документування архітектури системи. На думку одного з авторитетних фахівців в області об'єктно-орієнтованого підходу Гради Буча, моделювання є центральною ланкою усієї діяльності по створенню якісного ПЗ. Моделі будуються для того, щоб зрозуміти і осмислити структуру і поведінку майбутньої системи, полегшити управління процесом її створення і зменшити можливий ризик.

Мова моделювання повинна включати: елементи моделі – фундаментальні концепції моделювання і їх семантику; нотацію – візуальне представлення елементів моделювання; керівництво по використанню – правила застосування елементів у рамках побудови тих або інших типів моделей ПЗ.Такі можливості має уніфікована мова моделювання UML, який останнім часом став загальноприйнятою мовою розробки програмних систем. UML придатний для моделювання будь-яких систем: від информа-ционных систем масштабу підприємства до розподілених Web-приложений і навіть вбудованих систем реального часу. У посібнику основна увага приділяється застосуванню елементів мови UML, а не особливостям самої мови.

Очевидно, що кінцева мета розробки ПЗ – це не моделювання, а отримання працюючих застосувань (кода). Діаграми кінець кінцем – це усього лише наочні зображення, тому, використовуючи графічні мови моделювання, дуже важливо розуміти, чим вони допоможуть при написанні коду програм.

Наочність і суворість засобів структурного і об'єктно-орієнтованого аналізу дозволяють розробникам і майбутнім користувачам системи із самого початку неформально брати участь в її створенні, обговорювати і закріплювати розуміння основних технічних рішень. Проте широке застосування цих методів і наслідування їх рекомендацій при розробці конкретних ІС стримувалося відсутністю адекватних інструментальних засобів, оскільки при неавтоматизованій (ручний) розробці усі їх переваги практично зведені до нуля.

Дійсно, вручну дуже важко розробити і графічно представити строгі формальні специфікації системи, перевірити їх на повноту і несуперечність і тим більше змінити. Якщо все ж вдається створити строгу систему проектних документів, то її переробка при появі серйозних змін практично неосуществима. Ручна розробка зазвичай породжувала наступні проблеми: неадекватна специфікація вимог, нездатність виявляти помилки в проектних рішеннях, низьку якість документації, що знижує експлуатаційні характеристики, затяжний цикл і незадовільні результати тестування.

Перераховані проблеми породили потребу в програмно-технологічних засобах спеціального класу – CASE-средствах, реалізовуючих CASE-технологию створення і супроводу ПЗ ІС.

CASE-технология є сукупністю методів проектування ІС, а також набір інструментальних засобів, що дозволяють в наочній формі моделювати предметну область, аналізувати цю модель на всіх стадіях розробки і супроводу ІС, і розробляти додатки відповідно до інформаційних потреб користувачів.

Мета цього посібника – формування навичок самостійного практичного застосування сучасних методів і засобів об'єктно-орієнтованого аналізу і проектування (ООАП) ПЗ інформаційних систем, заснованих на використанні візуального моделювання і CASE-средств.

ЧАСТИНА 1. ОРГАНІЗАЦІЯ ПРОЦЕСУ РОЗРОБКИ ІС

РОЗДІЛ 1. ОСНОВНІ ПОНЯТТЯ ТЕХНОЛОГІЇ ПРОЕКТУВАННЯ ІС

1.1. Тенденції розвитку інформаційних технологій

Пройшли часи, коли програміст економив комп'ютерний час і пам'ять на кожній програмній команді, функції, модулі. Нестримно росте потужність ЕОМ і складність вирішуваних завдань. Акцент в реалізації і розробці програм з ефективного кодування і нестандартних рішень переміщається на просту і швидку розробку, взаєморозуміння усередині колективів розробників.

У свою чергу, інформація у сучасному світі перетворилася на один з найбільш важливих ресурсів, а інформаційні системи (ІС) стали необхідним інструментом практично в усіх сферах діяльності. Різноманітність задач, що вирішуються з допомогою ІС, привела до появи безлічі різнотипних систем, що відрізняються принципами побудови і закладеними в них правилами обробки інформації.

Аналіз сучасного стану ринку ІС показує стійку тенденцію росту попиту на інформаційні системи організаційного управління. Причому попит продовжує рости саме на інтегровані системи управління [1].

Перш ніж перейти до поняття інформаційних технологій дамо визначення поняттю технологія.

Слово ***технологія***, з'явившись у древніх греків як термін для позначення майстерності виготовлення речей, в сучасному трактуванні означає комплекс наукових і інженерних знань про способи і чинники виробництва для створення якого-небудь продукту або послуги. Засновником технології, як окремої дисципліни, являється німецький вчений Йоганн Бекман (1739-1811), що написав ряд творів за технологією і запропонував сам цей термін.

Інформаційна технологія – це системно-організована послідовність операцій (збір, реєстрація, передача, накопичення і обробка інформації), що виконуються над інформацією з використанням засобів і методів автоматизації. Процедури обробки інформації є головними в інформаційних технологіях. Інші процедури носять допоміжний характер.

Тенденції розвитку сучасних інформаційних технологій призводять до постійного зростання складності ІС, створюваних в різних областях діяльності. Сучасні великі проекти ІС характеризуються, як правило, наступними особливостями:

- складність опису (досить велика кількість функцій, процесів, елементів даних і складні взаємозв'язки між ними), що вимагає ретельного моделювання і аналізу даних і процесів;
- наявність сукупності тісно взаємодіючих компонентів (підсистем), що мають свої локальні завдання і цілі функціонування, використовують нерегламентовані запити до даних великого об'єму);

- відсутність прямих аналогів, що обмежує можливість використання яких-небудь типових проектних рішень і прикладних систем;
- необхідність інтеграції (композиції) існуючих і знову таких, що розробляються додатків;
- функціонування в неоднорідному середовищі на декількох апаратних платформах;
- роз'єднаність і різнорідність окремих груп розробників по рівню кваліфікації і традиціям використання тих або інших інструментальних засобів, що склалися.

Для успішної реалізації ІС об'єкт проектування має бути адекватно описаний, мають бути побудовані повні і несуперечливі функціональні і інформаційні моделі *архітектури* ІС, що обумовлює сукупність структурних елементів системи і зв'язків між ними, поведінку елементів системи в процесі їх взаємодії, а також ієрархію підсистем, що об'єднують структурні елементи. Під моделлю розуміється повний опис системи ПЗ з певної точки зору.

Моделі є засобами для візуалізації, опису, проектування і документування архітектури системи. На думку одного з авторитетних фахівців в області об'єктно-орієнтованого підходу Гради Буча, моделювання є центральною ланкою усієї діяльності по створенню якісного ПЗ [16]. Моделі будуються для того, щоб зрозуміти і осмислити структуру і поведінку майбутньої системи, полегшити управління процесом її створення і зменшити можливий ризик, а також документувати проектні рішення, що приймаються.

Хороші моделі є основою взаємодії учасників проекту і гарантують коректність архітектури. Оскільки складність систем підвищується, важливо мати в розпорядженні ефективні методи моделювання. Хоча є багато інших чинників, від яких залежить успіх проекту, наявність строгого стандарту мови моделювання є дуже істотною.

Накопичений до теперішнього часу досвід проектування ІС показує, що це логічно складна, трудомістка і тривала за часом робота, що вимагає високої кваліфікації фахівців, що беруть участь в ній [2]. Усе це сприяє появі різних технологій програмування і програмно-технологічних засобів спеціального класу – CASE-средств (Computer Aided Software Engineering), реалізовуючих CASE-технологию створення і супроводу ІС.

Термін CASE має дуже широке тлумачення. Первинне значення терміну CASE обмежувалося питаннями автоматизації розробки тільки програмного забезпечення, а нині воно набуло нового сенсу і охоплює процес розробки складних ІС в цілому. Тепер під терміном CASE-средства розуміються програмні засоби, що підтримують процеси створення і супроводу ІС відповідно до інформаційних потреб користувачів, включаючи: аналіз і формулювання вимог, в наочній формі моделювання предметної області, аналіз цієї моделі на усіх етапах розробки і супроводу ІС.

У 70-80-х рр. при розробці ПЗ досить широко застосовувалися структурні методи, що базуються на строгих формалізованих методах опису ПЗ і технічних рішень, що приймаються. Нині такого ж поширення набувають об'єктно-орієнтовані методи. Ці методи засновані на використанні наочних графічних

моделей: для опису архітектури ПЗ в різних аспектах (як статичної структури, так і динаміки поведінки системи) використовуються схеми і діаграми. Наочність і суворість засобів структурного і об'єктно-орієнтованого аналізу дозволяють розробникам і майбутнім користувачам системи із самого початку неформально брати участь в її створенні, обговорювати і закріплювати розуміння основних технічних рішень.

1.2. Поняття інформаційної системи

Під *системою* розуміють будь-який об'єкт, який одночасно розглядається і як єдине ціле, і як об'єднана в інтересах досягнення поставлених цілей сукупність різнорідних елементів. З безлічі визначень поняття системи (Вікіпедія) виберемо загальне. Тобто, ***система*** (греч. – «складене з частин», «з'єднання») – це сукупність взаємозв'язаних між собою елементів, підлеглих єдиної мети, щоб могла реалізовуватися функція системи.

Інформаційна система – взаємозв'язана сукупність засобів, методів і персоналу, використовуваних для зберігання, обробки і видачі інформації в інтересах досягнення поставленої мети.

Сьогоднішнє, сучасне розуміння інформаційної системи припускає використання як основний технічний засіб переробки інформації персонального комп'ютера. У великих організаціях разом з персональним комп'ютером до складу технічної бази інформаційної системи може входити мейнфрейм або суперЕОМ. Крім того, технічне втілення інформаційної системи саме по собі нічого не означатиме, якщо не врахована роль людини, для якої призначена вироблена інформація і без якого неможливе її отримання і представлення. Очевидно, що існує відмінність між комп'ютерами і інформаційними системами. Комп'ютери, оснащені спеціалізованими програмними засобами, є технічною базою і інструментом для інформаційних систем. Обов'язковою компонентою будь-якої інформаційної системи є також персонал, що взаємодіє з комп'ютерами і телекомунікаціями.

1.3. Етапи розвитку інформаційних систем

У міру розвитку і вдосконалення засобів обчислювальної техніки, мов програмування і математичного забезпечення, автоматизовані системи обробки даних зазнали декілька етапів розвитку.

Перші інформаційні системи з'явилися в 50-х рр. Після війни комп'ютери стали застосовуватися в першу чергу саме для оборонних завдань. Потужність комп'ютерів (першого покоління) була невелика, а програмування для них велося, в основному, в машинному коді. Вирішувалися, головним чином, науково-технічні завдання (рахунок по формулах), завдання на програмування містило, як правило, досить точну постановку завдання, і першими програмістами у більшості своїй були люди, які складали і виводили рівняння. Фізики і математики ретельно розробляли алгоритми. Вони готували детальну і точну документацію, аналізували рішення своїх колег і шукали математичні докази.

У ці ж роки уперше була усвідомлена роль інформації як найважливішого ресурсу підприємства, організації, регіону, суспільства в цілому. У ці роки вони були призначені для обробки рахунків і розрахунку зарплати, а реалізовувалися на електромеханічних бухгалтерських рахункових машинах. Це призводило до деякого скорочення витрат і часу на підготовку паперових документів.

60-е рр. знаменуються зміною відношення до інформаційних систем. Проектування інформаційних систем виконувалося в основному на інтуїтивній технології програмування із застосуванням неформалізованих методів, заснованих на мистецтві, практичному досвіді, експертних оцінках і дорогих експериментальних перевірках якості функціонування програм. Майже відразу приступали до складання програми за завданням, при цьому часто завдання кілька разів змінювалося, враховуючи інформаційні потреби користувачів (що сильно збільшувало час і без того ітераційного процесу складання програми), мінімальна документація оформлялася вже після того, як програма починала працювати.

Інформація, отримана з інформаційних систем, стала застосовуватися для періодичної звітності за багатьма параметрами. Для цього організаціям було потрібне комп'ютерне устаткування широкого призначення, здатне обслуговувати безліч функцій, а не тільки обробляти рахунки і рахувати зарплату, як було раніше. Мета використання – підвищення швидкості обробки документів, спрощення процедури обробки рахунків і розрахунку зарплати.

70-х – початок 80-х рр. отримали широке поширення інформаційні системи і бази даних. До середини 70-х років вартість зберігання одного біта інформації на комп'ютерних носіях стала менше, ніж на традиційних носіях. Це різко підвищило інтерес до комп'ютерних систем зберігання даних.

Інформаційні системи починають широко використовуватися як засіб управлінського контролю, що підтримує і прискорюючого процес ухвалення рішень, системи для вищої ланки управління.

У 80-х – 2010 рр. концепція використання інформаційних систем змінюється. Вони стають стратегічним джерелом інформації і використовуються на усіх рівнях організації будь-якого профілю. Інформаційні системи цього періоду, надаючи вчасно потрібну інформацію, допомагають організації досягти успіху у своїй діяльності, створювати нові товари і послуги, знаходити нові ринки збуту, забезпечувати собі гідних партнерів, організовувати випуск продукції за низькою ціною і багато що інше.

80-е роки характеризуються широким впровадженням персональних комп'ютерів в усі сфери людської діяльності і тим самим створенням великого і різноманітного контингенту користувачів ПЗ.Це привело до бурхливого розвитку призначених для користувача інтерфейсів і створення інформаційних систем нового покоління.

З'являються мови програмування (наприклад, Ада), що враховують вимоги технології програмування. Починається бурхливий процес стандартизації технологічних процесів і, передусім, документації, що створюється в цих процесах. Виходить на передові позиції об'єктний підхід до розробки ІС. Створюються різні інструментальні середовища розробки і

супроводу ІС. Розвивається концепція комп'ютерних мереж. На перший план виходять засоби автоматизації розробки програм, що дістали назву CASE.

Підхід CASE дав розробникам ІС можливість отримати досконаліші інструменти, наприклад, мови четвертого покоління (для них є навіть своя абревіатура, 4GL). Приміром, такі мови четвертого покоління як Visual Basic і Clarion – це корисний, потужний і популярний 4GL-инструментарий, який дозволяє збільшити продуктивність програмістів і скоротити число потенційних помилок.

90-е роки знаменні широким охопленням усього людського суспільства міжнародною комп'ютерною мережею, персональні комп'ютери стали підключатися до неї як термінали. Це поставило ряд проблем (як технологічного, так і юридичного і етичного характеру) регулювання доступу до інформації комп'ютерних мереж. Гостро встала проблема захисту комп'ютерної інформації і передаваних по мережі повідомлень. Стали бурхливо розвиватися **комп'ютерна технологія** (CASE-технология) розробки ІС 2-го покоління і пов'язані з нею формальні методи специфікації програм.

Широке використання комп'ютерних мереж привело до інтенсивного розвитку розподілених обчислень, дистанційного доступу до інформації і електронного способу обміну повідомленнями між людьми. Комп'ютерна техніка із засобу рішення окремих завдань усе більш перетворюється на засіб інформаційного моделювання реального і мислимого світу, здатний просто відповідати людям на питання, що цікавлять їх. Починається етап глибокої і повної інформатизації (комп'ютеризації) людського суспільства. Усе це ставить перед технологією створення ІС нові і досить важкі проблеми.

Незважаючи на високі потенційні можливості CASE-технологии (збільшення продуктивності праці, поліпшення якості програмних продуктів, підтримка уніфікованого і погодженого стилю роботи) далеко не усі розробники ІС, використовуючі CASE-средства, досягають очікуваних результатів.

Існують різні причини можливих невдач, але, мабуть, головною причиною є неадекватне розуміння суті програмування великих систем і застосування CASE-средств.

1.4. Порівняння ІС з традиційними програмними продуктами

Хоча інформаційні системи є звичайним програмним продуктом, вони мають ряд істотних відмінностей від стандартних застосовних програм і систем.

Залежно від предметної області інформаційні системи можуть дуже сильно розрізнятися по своїх функціях, архітектурі, реалізації:

1. ІС призначені для збору, зберігання і обробки інформації. Тому в основі будь-якої з них лежить середовище зберігання і доступу до даних;
2. ІС орієнтуються на кінцевого користувача, що не має високої кваліфікації в сфері застосування обчислювальної техніки. Тому клієнтські застосування ІС повинні мати простий, зручний, легко освоюваний

інтерфейс, який надає кінцевому користувачеві усі необхідні для роботи функції, але в той же час не дає йому можливість виконувати які-небудь зайві дії.

Таким чином, при розробці інформаційної системи доводиться вирішувати дві основні задачі:

- завдання розробки БД, призначеної для зберігання інформації;
- завдання розробки графічного інтерфейсу користувача клієнтських застосувань.

Розглянемо приклади деяких програмних засобів, які є, або не є ІС.

1. 1С-Бухгалтерія 8.0. Використовується з метою формування бухгалтерської звітності підприємства перед податковими органами. Є інформаційною системою.
2. MS Excel. Програмний засіб універсального характеру, призначений для маніпуляцій з даними, представленими в табличній формі для автоматизації розрахунків, формування різноманітних діаграм і для аналізу даних. Не є інформаційною системою.
3. Книга MS Excel, що містить відомості про штатний розклад, працівників підприємства та оснащена макросами, що дозволяють розраховувати заробітну плату і формувати платіжні відомості. Є інформаційною системою.
4. Система комплексної автоматизації діяльності мережі роздрібних магазинів. Є інформаційною системою.
5. Реляційна СУБД DB-2 фірми IBM. Не є інформаційною системою.

1.5. Класифікація інформаційних систем

Інформаційні системи можна класифікувати по цілому ряду різних ознак. У основу даної класифікації покладені найбільш суттєві ознаки, що визначають функціональні можливості і особливості побудови сучасних систем. Залежно від об’єму вирішуваних завдань, використовуваних технічних засобів, організації функціонування, ІС діляться на ряд груп (класів). Розглянемо найбільш важливі з них.

1. За **типом даних**, що зберігаються, ІС діляться на фактографічні і документальні. *Фактографічні системи призначені для зберігання і обробки структурованих даних у вигляді чисел і текстів. У документальних системах* інформація представлена у вигляді документів, що складаються з найменувань, описів, рефератів і текстів.

2. Грунтуючись на **мірі автоматизації інформаційних процесів** в системі управління фірмою, інформаційні системи діляться на ручні, автоматичні і автоматизовані.

Ручні ІС характеризуються відсутністю сучасних технічних засобів переробки інформації і виконанням усіх операцій людиною.

У *автоматичних ІС* усі операції по переробці інформації виконуються без участі людини.

Автоматизовані ІС припускають участь в процесі обробки інформації і людини, і технічних засобів, причому головна роль у виконанні рутинних операцій обробки даних відводиться комп'ютеру.

3. Залежно від **характеру обробки даних** ІС діляться на *інформаційно-пошукові (довідкові) і інформаційно-вирішальні*.

Нині створено і успішно функціонує велике число *інформаційно-довідкових систем різного призначення, які призначені* для задоволення інформаційних запитів користувачів. Характерна особливість таких систем – інформація, знайдена відповідно до запиту, не використовується безпосередньо у рамках цієї ж системи, а видається користувачеві, який використовує отриману інформацію для будь-яких необхідних йому цілей. *Пошук* – одна з основних операцій в таких системах, тому вони є також інформаційно-пошуковими системами (ІПС).

Інформаційно-пошукові системи роблять введення, систематизацію, зберігання, видачу інформації за запитом користувача без складних перетворень даних.

Інформаційно-вирішальні системи здійснюють, крім того, операції переробки інформації по певному алгоритму. За характером використання вихідної інформації такі системи прийнято ділити на ті, що управляють і радять.

Результуюча інформація *керівників ІС* безпосередньо трансформується в рішення, що приймаються людиною. Для цих систем характерні завдання розрахункового характеру і обробка великих об'ємів даних.

Що радять ІС виробляють інформацію, яка приймається людиною до відома і враховується при формуванні управлінських рішень, а не ініціює конкретні дії. Залежно від сфери застосування розрізняють наступні класи ІС.

Інформаційні системи організаційного управління – призначені для автоматизації функцій управлінського персоналу як промислових підприємств, так і непромислових об'єктів (готелів, банків, магазинів). Основними функціями подібних систем є: оперативний контроль і регулювання, оперативний облік і аналіз, перспективне і оперативне планування, бухгалтерський облік.

ІС управління технологічними процесами (ТП) – служать для автоматизації функцій виробничого персоналу по контролю і управлінню виробничими операціями. У таких системах зазвичай передбачається наявність розвинених засобів виміру параметрів технологічних процесів (температури, тиски, хімічного складу і тому подібне).

ІС автоматизованого проектування (САПР) – призначені для автоматизації функцій інженерів-проектувальників, конструкторів, архітекторів, дизайнерів при створенні нової техніки або технології. Основними функціями подібних систем є: інженерні розрахунки, створення графічної документації (креслень, схем, планів), створення проектної документації, моделювання проектованих об'єктів.

4. За **масштабом** ІС поділяються на однокористувацькі, групові та корпоративні.

Інтегровані (корпоративні) ІС – використовуються для автоматизації усіх функцій фірми і охоплюють увесь цикл робіт від планування діяльності до збуту продукції. Вони включають ряд модулів (підсистем), працюючих в єдиному інформаційному просторі і виконуючих функції підтримки відповідних напрямів діяльності.

5. Існує класифікація ІС залежно від **рівня управління**, на якому система використовується.

Інформаційна система оперативного рівня – підтримує виконавців, обробляючи дані про угоди і події (рахунки, накладні, зарплата, кредити, потік сировини і матеріалів). Інформаційна система оперативного рівня є сполучною ланкою між фірмою і зовнішнім середовищем.

Інформаційні системи фахівців – підтримують роботу з даними і знаннями, підвищують продуктивність і продуктивність роботи інженерів і проектувальників. Завдання подібних систем – інтеграція нових відомостей в організацію і допомогу в обробці паперових документів.

Інформаційні системи рівня менеджменту – використовуються працівниками середньої управлінської ланки для моніторингу, контролю, ухвалення рішень і адміністрування. Основні функції цих інформаційних систем:

- порівняння поточних показників з минулими;
- складання періодичних звітів за певний час, а не видача звітів по поточних подіях, як на оперативному рівні;
- забезпечення доступу до архівної інформації і так далі

Стратегічна інформаційна система – комп'ютерна інформаційна система, що забезпечує підтримку ухвалення рішень по реалізації стратегічних перспективних цілей розвитку організації.

Інформаційні системи стратегічного рівня допомагають вищій ланці управлінців вирішувати неструктуровані завдання, здійснювати довгострокове планування.

6. Класифікація по **архітектурі**.

Архітектура "Файл-сервер" – історично перша архітектура інформаційних систем. Як виконавчі модулі, так і дані розміщуються в окремих файлах операційної системи. Доступ до даних здійснюється через шлях (path) і використання файлових операцій (відкрити, считать, записати). Для зберігання даних використовується виділений сервер (окремий комп'ютер), який і є файловим сервером. Виконувані модулі зберігаються або на робочих станціях, або на файловому сервері. В останньому випадку спрощується процедура їх адміністрування, але при цьому зростають вимоги до надійності мережі.

Архітектура "Клієнт-сервер" – це не тільки архітектура, це – нова парадигма, що прийшла на зміну застарілим концепціям. Суть її полягає в тому, що клієнт (виконуваний модуль) запитує ті чи інші сервіси відповідно до визначеного протоколом обміну даними. При цьому, на відміну від ситуації з файловим сервером, немає необхідності у використанні прямих шляхів операційної системи: клієнт їх "не знає", йому "відомі" лише ім'я джерела даних

та інші спеціальні відомості, що використовуються для авторизації клієнта на сервері. Сервер, який фізично може перебувати на тому ж комп'ютері, а може – на іншому кінці земної кулі, обробляє запит клієнта і, зробивши відповідні маніпуляції з даними, передає клієнту запитувану порцію даних.

В рамках напряму "клієнт-сервер" існують два основних "діалекти": "тонкий" і "товстий" клієнт.

У системах на основі тонкого клієнта використовується потужний сервер баз даних, це – високопродуктивний комп'ютер і бібліотека так званих процедур, які дозволяють робити обчислення, що реалізують основну логіку обробки даних, безпосередньо на сервері. Клієнтська програма, відповідно, пред'являє невисокі вимоги до апаратного забезпечення робочої станції. Основна перевага таких систем – відносна дешевизна клієнтських станцій.

Системи з товстим клієнтом, навпаки, реалізують основну логіку обробки на клієнті, а сервер являє собою в чистому вигляді сервер баз даних, що забезпечує виконання тільки стандартизованих запитів на маніпуляцію з даними (як правило, – читання, запис, модифікацію даних в таблицях реляційної бази даних). У системах такого класу вимоги до робочої станції вище, а до сервера – нижче. Гідність архітектури – переносимість серверної компоненти на сервери різних виробників: всі промислові сервери баз даних реляційного типу підтримують роботу зі стандартизованою мовою маніпулювання даними SQL, але внутрішня вбудована мова обробки даних, необхідна для реалізації логіки обробки, на сервері у кожного з серверів своя.

1.6. Сфери застосування і приклади реалізації ІС

Якщо раніше мало не єдиною областю, в якій застосовувалися інформаційні системи, була автоматизація бухгалтерського обліку, то зараз спостерігається впровадження інформаційних технологій у безліч інших областей.

Розглянемо найбільш важливі завдання, що вирішуються за допомогою спеціальних програмних засобів.

Бухгалтерський облік. Це класична сфера застосування інформаційних технологій і завдання, що найчастіше реалізовується на сьогодні. Таке положення цілком з’ясовне. По-перше, помилка бухгалтера може коштувати дуже дорого, тому очевидна вигода використання можливостей автоматизації бухгалтерії. По-друге, завдання бухгалтерського обліку досить легко формалізується, так що розробка систем автоматизації бухгалтерського обліку не представляє технічно складної проблеми.

Управління фінансовими потоками. Впровадження інформаційних технологій в управління фінансовими потоками також обумовлено критичністю цієї області управління підприємства до помилок. Неправильно побудувавши систему розрахунків з постачальниками і споживачами, можна спровокувати кризу готівки навіть при налагодженій мережі закупівлі, збуту і хорошому маркетингу.

Управління складом, асортиментом, закупівлями. Далі, можна автоматизувати процес аналізу руху товару, тим самим відстеживши і зафіксувавши ті двадцять відсотків асортименту, які приносять вісімдесят відсотків прибутку.

Управління виробничим процесом. Основними механізмами тут є планування і оптимальне управління виробничим процесом. Автоматизоване рішення подібної задачі дає можливість грамотно планувати, враховувати витрати, проводити технічну підготовку виробництва, оперативно управляти процесом випуску продукції.

Управління маркетингом. Управління маркетингом має на увазі збір і аналіз даних про фірми-конкурентів, їх продукцію і цінову політику, а також моделювання параметрів зовнішнього оточення для визначення оптимального рівня цін, прогнозування прибули і планування рекламних кампаній.

Документообіг. Добре відлагоджена система облікового документообігу відбиває поточну виробничу діяльність, що реально відбувається на підприємстві, і дає управлінцям можливість впливати на неї.

Оперативне управління підприємством. Інформаційна система, вирішальна завдання оперативного управління підприємством, будується на основі бази даних, в якій фіксується уся можлива інформація про підприємство. Така інформаційна система є інструментом для управління бізнесом і зазвичай називається корпоративною ІС.

Надання інформації про фірму. Практично кожне підприємство, що поважає себе, зараз має свій web-сервер підприємства для вирішення завдань, з яких можна виділити дві основні:

- створення іміджу підприємства;
- розвантаження довідкової служби компанії для отримання необхідної інформації про фірму, пропоновані товари, послуги і ціни.

Контрольні питання до розділу 1

1. Дайте визначення поняттю *система*.
2. Що таке *інформаційна система*.
3. Дайте визначення *фактографічній системі*.
4. Що таке *інформаційно-пошукова (довідкова) система*.
5. Що таке *інтегрована (корпоративна) система*.
6. Що таке *моніторингова інформационная система*.
7. Які ви знаєте сфери застосування ІС.

РОЗДІЛ 2. МЕТОДОЛОГІЇ І ТЕХНОЛОГІЇ РОЗРОБКИ ІС

З ростом складності архітектури ІС росте трудомісткість їх розробки, ускладнюються сценарії інтеграції, ростуть витрати на супровід. Традиційне інтегроване середовище розробки (integration development environment, IDE) надає базові інструменти – редактори коду, компілятори, засоби розробки призначеного для користувача інтерфейсу і відладчики. Проте за їх межами залишаються ключові етапи життєвого циклу додатків, такі як визначення вимог, моделювання, тестування, розгортання і управління змінами.

Розробка інформаційних систем – це циклічний процес створення унікального продукту, який удосконалюється від версії до версії; різні, не завжди послідовні етапи можуть впливати один на одного. По суті, створенню програмного продукту більше відповідає не модель управління проектом з чітким початком і кінцем, а модель управління усім життєвим циклом продукту.

Циклічному ходу розробки більше підходять ітеративні методики, що мають на увазі постійну співпрацю між різними функціональними групами в команді програмного проекту і кінцевими користувачами, і тісний взаємозв'язок між різними етапами життєвого циклу додатка.

2.1. Організація процесу розробки

У цьому розділі визначаються базові поняття програмної інженерії (інженерії програмного забезпечення). Як і у будь-якій інженерній дисципліні, основними складовими програмної інженерії є продукти (програмні системи) і процеси, що забезпечують створення продуктів.

2.1.1. Основні поняття програмної інженерії

Програмна інженерія – система інженерних принципів для створення економічного ПО, яке надійне і ефективно працює в реальних комп'ютерах [3]. Програмна інженерія – порівняно молода дисципліна, її вік трохи більше сорока років. Перші двадцять років (70-80-і роки) в ній домінував класичний, процедурний підхід, а ось починаючи з 1990 по 2015-і роки – об'єктно-орієнтований підхід до розробки ПО.

Розрізняють методи, засоби і процеси програмної інженерії.

Методи забезпечують рішення широкого спектру технічних завдань.

Засоби (утиліти) програмної інженерії забезпечують автоматизовану або автоматичну підтримку методів. В цілях спільного застосування утиліти можуть об'єднуватися в системи автоматизованої розробки ПЗ.Такі системи прийнято називати CASE-системами.

Процеси є «клеєм», який сполучає методи і утиліти так, що вони забезпечують безперервний технологічний ланцюжок розробки.

2.1.2. Основні процеси життєвого циклу ІС

Основні процеси життєвого циклу ІС складаються з п’яти процесів, які реалізуються під управлінням основних сторін, залучених в життєвий цикл ПЗ.Основними сторонами є замовник, постачальник, розробник. Основними процесами вважають:

1. **Процес замовлення**. Визначає роботи замовника.
2. **Процес постачання**. Визначає роботи постачальника.
3. **Процес розробки**. Визначає роботи розробника.
4. **Процес експлуатації**. Визначає роботи оператора.
5. **Процес супроводу**. Визначає роботи супроводжуючої організації, яка надає послуги з супроводу програмного продукту.

2.1.3. Базис процесів розробки ІС

Вважатимемо, що моделі процесів складаються з таких будівельних елементів, як види діяльності, дії і завдання.

Діяльність – найбільший елемент, який орієнтований на досягнення вагомої мети і застосовується незалежно від прикладної області, розміру проекту, складності витрат. Діяльність складається з дій.

Дія – середній елемент, охоплює набір завдань, які роблять етапний робочий продукт.

Завдання – найдрібніший елемент. Завдання фокусується на маленькій, але добре визначеній меті (наприклад, на проведенні тестування модуля), яка призводить до відчутного реального результату. Мета – створювати ПЗ за прийнятний час і з достатньою якістю.

Підготовка. До будь-якої технічної роботи треба правильно підготуватися. Підготовка полягає в тісній співпраці із замовником і іншими зацікавленими особами.

Планування. План визначає порядок інженерної роботи. Він описує технічні завдання, які потрібно виконувати, найбільш вірогідні чинники ризику, що підстерігають команду, необхідні ресурси, робочі продукти (моделі, документи, дані, звіти, форми тощо), які будуть створені, і розклад роботи.

Моделювання. У будь-якій людській діяльності щодня працюють з моделями. Моделі дозволяють краще зрозуміти загальну картину – ескіз майбутнього рішення. При необхідності ескіз доповнюється деталями. Зазвичай моделювання включає дві дії: *аналіз і проектування*.

Конструювання. Ця діяльність об’єднує дві дії: генерацію програмного коду ПЗ (ручну або автоматичну) і тестування, яке потрібно для виявлення помилок в коді.

Розгортання. ПЗ (остаточний варіант або частково завершена версія) поставляється замовникові, який оцінює отриманий продукт і забезпечує зворотний зв’язок, що дає можливість поліпшення продукту.

2.2. Життєвий цикл програмного забезпечення ІС

Розробка програмного забезпечення – різновид людської діяльності. Виділити її компоненти можна, визначивши набір завдань, які треба вирішити для досягнення кінцевої мети, – побудови досить якісної системи у рамках заданих термінів і ресурсів. Для вирішення кожного такого завдання організовується допоміжна діяльність, до якої можна також застосувати декомпозицію на окремі дрібніші діяльності, і так далі. У результаті повинно стати зрозуміло, як вирішувати кожну окрему підзадачу і усе завдання цілком на основі наявних рішень для підзадач.

В якості прикладів діяльностей, які треба проводити для побудови програмної системи, можна привести ***проектування*** – виділення окремих модулів і визначення зв'язків між ними з метою мінімізації залежностей між частинами проекту і досягнення кращої його осяжності в цілому, кодування – розробку коду окремих модулів, розробку призначеної для користувача документації, яка потрібна для досить складної системи.

Одним з базових понять методології проектування ІС являється поняття життєвого циклу її програмного забезпечення (ЖЦПЗ). ЖЦПЗ – це безперервний процес, який починається з моменту ухвалення рішення про необхідність його створення і закінчується у момент його повного вилучення з експлуатації.

В ході життєвого циклу створення ІС проходить через аналіз предметної області, збір вимог, проектування, кодування, тестування, супровід і інші види діяльності.

При розробці ІС створюються і переробляються різного роду ***артефакти*** – створювані людиною інформаційні сутності, документи в досить загальному сенсі, що беруть участь в якості вхідних даних і виходять в якості результату різних деятельностей. Прикладами артефактів є: модель предметної області, опис вимог, технічне завдання, архітектура системи, проектна документація на систему в цілому і на її компоненти, прототипи системи і компонентів, власне, початковий код, призначена для користувача документація, документація адміністратора системи та ін.

Структура ЖЦПЗ за стандартом ISO/IEC 12207 базується на трьох групах процесів:

- основні процеси ЖЦПЗ (придбання, постачання, розробка, експлуатація, супровід);
- допоміжні процеси, що забезпечують виконання основних процесів (документування, управління конфігурацією, забезпечення якості, верифікація, атестація, оцінка, аудит, рішення проблем);
- організаційні процеси (управління проектами, визначення, оцінка і поліпшення самого ЖЦ, навчання).

Розробка включає усі роботи із створення ІС і його компонент відповідно до заданих вимог, включаючи оформлення проектної і експлуатаційної документації. Розробка ІС включає, як правило:

1. **Планування і оцінка проекту.** Під час цієї фази виявляються властивості, які повинна мати готова система, тобто, створюється задум системи в остаточному варіанті. Під час цієї фази приймаються рішення відносно розмірів, часу відгуку і інших параметрів системи. Визначення здійснюється за допомогою двох основних категорій – вимог і *специфікацій.*
2. **Аналіз системних і програмних вимог.** *Системний аналіз* задає роль кожного елементу в комп'ютерній системі, взаємодію елементів один з одним. *Аналіз вимог* відноситься до програмного елементу – програмного забезпечення. Уточнюються і деталізуються його функції, характеристики і інтерфейс.
3. **Проектування алгоритмів, структур цих і програмних структур**. Проектування полягає в створенні представлень:
   - архітектури ПЗ і модульної структури ПЗ;
   - алгоритмічної структури ПЗ і структури даних;
   - вхідного і вихідного інтерфейсу (вхідних/вихідних форм даних).
4. **Реалізація (кодування)** полягає в перекладі результатів проектування в текст на мові програмування.
5. **Тестування відповідає за** виконання програми для виявлення дефектів у функціях, логіці і формі реалізації програмного продукту.
6. **Введення в дію (супровід) –** це внесення змін до експлуатованої ІС. Цілі змін:
   - виправлення помилок;
   - адаптація до змін зовнішньої для ІС середовища;
   - удосконалення ІС за вимогами замовника.
7. **Експлуатація** включає роботи по впровадженню компонентів ІС в експлуатацію, у тому числі:
   - бази цих і робочих місць користувачів;
   - забезпечення конфігурація експлуатаційною документацією;
   - проведення навчання персоналу і так далі;
   - модифікацію (модернізацію) ІС.

Вимоги – ця властивість, необхідна для вирішення проблеми або досягнення мети. При описі вимог використовуються такі поняття, як якість, можливості або інші характеристики системи, передбачуваного її використання або середовища. Найкращою мовою опису вимог, що забезпечує строге, точне і усеосяжне експертне їх вираження, є професійно-природна мова експертів у сфері діяльності організації користувача.

Специфікація – цей опис на мові розробника зовні відомих характерних особливостей поведінки системи.

Верифікація – це процес визначення того, чи відповідає поточний стан розробки, досягнутий на цьому етапі, вимогам цього етапу. Перевірка дозволяє оцінити відповідність параметрів розробки з початковими вимогами. На кожному етапі слід переконуватися, що зробили саме те, що планували, і це відповідає загальній логіці розробці – *«Ми створюємо систему правильно».*

Валідація (перевірка правильності) – процес перевірки того, що реализованая система задовольняє пред'явленим вимогам і працює так, як передбачалося – *«Ми створюємо (створили) правильну систему»*.

2.3. Моделі життєвого циклу ІС

2.3.1. Стратегії конструювання ІС

Модель ЖЦ інформаційних систем (ЖЦІС) залежить від специфіки програмного продукту і специфіки умов, в яких він створюється і функціонує. Існують три основні стратегії конструювання ІС:

1. ***Одноразовий прохід*** (водоспадна стратегія) – лінійна послідовність етапів конструювання.
2. ***Інкрементна (чи ітеративна) стратегія***. На початку процесу визначаються усі призначені для користувача і системні вимоги, частина конструювання, що залишилася, виконується у вигляді послідовності версій (заплановане поліпшення продукту).
3. ***Еволюційна стратегія***. Система також будується у вигляді послідовності версій, але на початку процесу визначені не усі вимоги. Вимоги уточнюються в результаті розробки версій.

Розглянемо коротко декілька моделей життєвого циклу.

2.3.2. Класична або каскадна модель

Найбільш широко відомою і вживаною довгий час залишалася класична або так звана ***каскадна (70-85 р.р.)*** або ***водоспадна (waterfall)*** модель життєвого циклу. Вона була уперше чітко сформульована в стандартах міністерства оборони США (автор Уїнстон Ройс, 1970). Ця модель припускає послідовне виконання різних видів діяльності, починаючи з вироблення вимог і закінчуючи супроводом, з чітким визначенням меж між етапами, на яких набір документів, виробленою на попередній стадії, передається в якості вхідних даних для наступної. Дуже часто класичний життєвий цикл називають каскадною або водоспадною моделлю, підкреслюючи, що розробка розглядається як послідовність етапів, причому перехід на наступний, ієрархічно нижній етап відбувається тільки після повного завершення робіт на поточному етапі (рис. 2.1).

Охарактеризуємо зміст основних етапів.

Стратегія. Визначення стратегії припускає обстеження системи. Основне завдання обстеження – оцінка реального об'єму проекту, його цілей і завдань, а також отримання визначень суті і функцій на високому рівні. На цьому етапі притягуються висококваліфіковані бізнес-аналітики, які мають постійний доступ до керівництва фірми; етап припускає тісну взаємодію з основними користувачами системи і бізнес-експертами. Основне завдання взаємодії – отримати якомога повнішу інформацію про систему (повне і однозначне розуміння вимог замовника) і передати дану інформацію у формалізованому вигляді системним аналітикам для подальшого проведення етапу аналізу.

Рисунок 2.1 – Каскадна модель ЖЦІС

Результатом етапу визначення стратегії є документ, де чітко сформульовано: що отримає замовник, якщо погодиться фінансувати проект; коли він отримає готовий продукт (графік виконання робіт); скільки це коштуватиме (для крупних проектів повинен бути складений графік фінансування на різних етапах робіт).

Виконана на даному етапі робота дозволяє відповісти на питання, чи варто продовжувати даний проект і які вимоги замовника можуть бути задоволені за тих або інших умов. Може так статися, що проект продовжувати не має сенсу, наприклад через те, що ті або інші вимоги не можуть бути задоволені за якимись об'єктивними причинами. Якщо ухвалюється рішення про продовження проекту, то для проведення наступного етапу аналізу вже є уявлення про об'єм проекту і кошторис витрат.

Аналіз. Етап аналізу припускає докладне дослідження бізнес-процесів (вимог і функцій, визначених на етапі вибору стратегії) і інформації, необхідної для їх виконання (сутності, їх атрибутів і зв'язків (відносин)).

Вся інформація про систему, зібрана на етапі визначення стратегії, формалізується і уточнюється на етапі аналізу. Особливу увагу слід приділити повноті переданої інформації, аналізу інформації на предмет відсутності суперечностей, а також пошуку невживаною взагалі або інформації, що дублюється. Як правило, замовник не відразу формує вимоги до системи в цілому, а формулює вимоги до окремих її компонентів. Тому необхідно приділити увагу узгодженості цих компонентів. Фактично ці вимоги визначають повне завдання на розробку.

Проектування (моделювання). Моделювання присвячене виконанню двох дій – аналізу вимог і проектуванню. Результати цих дій моделі – зазвичай записуються на графічній мові моделювання, мові картинок.

Розгляд результатів аналізу – це процес передачі інформації від аналітиків проектувальникам. На практиці це інтерактивний процес. У проектувальників неминуче виникатимуть питання до аналітиків, і навпаки.

Інформація про систему постійно уточнюватиметься. При розробці схеми бази даних може змінитися інформаційна модель, отримана на етапі аналізу, наприклад, тому, що наявне проектне рішення нестабільне або поволі працює при реалізації його за допомогою вибраної СУБД або через інші причини.

Робота проектувальників бази даних в значній мірі залежить від якості інформаційної моделі. Інформаційна модель не повинна містити ніяких незрозумілих конструкцій, які не можна реалізувати в рамках вибраної СУБД. Слід зазначити, що інформаційна модель створюється для того, щоб на її основі можна було побудувати модель даних, тобто повинна враховувати особливості реалізації вибраної СУБД.

Побудова логічної і фізичної моделей даних є основною частиною проектування бази даних. Отримана в процесі аналізу інформаційна модель спочатку перетвориться в логічну, а потім у фізичну модель даних. Після цього для розробників інформаційної системи створюється пробна база даних. З нею починають працювати розробники коду. У ідеалі до моменту початку розробки модель даних повинна бути стійка. Проектування бази даних не може бути відірване від проектування модулів і додатків, оскільки бізнес-правила можуть створювати об'єкти в базі даних, наприклад серверні обмеження.

Головна мета проектування полягає у відображенні функцій, отриманих на етапі аналізу, в модулі інформаційної системи. Визначення модулів розкриваються в технічній специфікації програм. При проектуванні модулів визначають розмітку меню, вид вікон, гарячі клавіші і пов'язані з ними виклики.

Тестування. Проектування процесу тестування, як правило, слідує за процесом функціонального проектування і проектування схеми бази даних. На цьому етапі можна використовувати складні схеми тестування, а можна обмежитися і простими. Коли генерація модуля завершена, виконують автономний тест, який переслідує дві основні цілі:

- виявлення відмов модуля (жорстких збоїв);
- відповідність модуля специфікації (наявність всіх необхідних функцій, відсутність зайвих функцій).

Після того, як автономний тест пройшов успішно, група модулів, що згенерували, проходить тести зв'язків, які повинні відстежити взаємний вплив модулів.

Далі група модулів тестується на надійність роботи, тобто проходять, по-перше, тести імітації відмов системи, а по-друге, тести напрацювання на відмову. Перша група тестів показує, наскільки добре система відновлюється після збоїв програмного забезпечення, відмов апаратного забезпечення. Друга група тестів визначає ступінь стійкості системи при штатній роботі і дозволяє оцінити час безвідмовної роботи системи. У комплект тестів стійкості повинні входити тести, що імітують пікове навантаження на систему.

Потім весь комплект модулів проходить системний тест – тест внутрішнього приймання продукту, що показує рівень його якості. Сюди входять тести функціональності і тести надійності системи.

Останній тест інформаційної системи – приймально-здавальні випробування. Такий тест передбачає показ інформаційної системи замовникові і повинен містити групу тестів, що моделюють реальні бізнес-процеси, щоб показати відповідність реалізації вимогам замовника.

Реалізація (розробка, кодування). На етапі розробки здійснюється тісна взаємодія проектувальників, розробників і груп тестерів. У разі інтенсивної розробки тестер фактично є членом групи розробки. Проектувальник на даному етапі виконує функції «ходячого довідника», оскільки постійно відповідає на питання розробників, що стосуються технічної специфікації. Найчастіше на етапі розробки міняються інтерфейси користувача. Це обумовлено у тому числі і тим, що модулі періодично демонструються замовникові. Істотно можуть мінятися і запити до даних. Взаємодія тестера і розробника без централізованої передачі частин проекту допустимо, але тільки у випадку, якщо необхідно терміново перевірити якусь правку. Дуже часто етап розробки і етап тестування взаємозв'язані і йдуть паралельно.

При розробці повинні бути організовані постійно оновлювані сховища готових модулів проекту і бібліотек, які використовуються при збірці модулів. Бажано, щоб процес оновлення сховищ контролювала одна людина. Одне з сховищ повинне бути призначене для модулів, що пройшли функціональне тестування, а інше – для модулів, що пройшли тестування зв'язків. Перше з них — це чернетки. Друге – те, з чого вже можна збирати дистрибутив системи і демонструвати його замовникові для проведення контрольних випробувань або здачі яких-небудь етапів робіт.

Введення в дію (упровадження, розгортання) – етап класичного життєвого циклу націлений на дві дії: постачання розробленого продукту замовникові і супровід процесу експлуатації цього продукту.

Згідно із статистичними даними, 65% супроводи пов’язано з удосконаленням ПЗ, 18% відводиться на адаптацію і 17% пов’язано з виправленням помилок.

Експлуатація перекриває процес тестування, система вводиться в експлуатацію не повністю, а поступово. Введення в експлуатацію проходить три фази:

- первинне завантаження інформації;
- накопичення інформації;
- вихід на проектну потужність.

Первинне завантаження інформації ініціює досить вузький круг помилок – в основному це проблеми розузгодження даних при завантаженні і власні помилки завантажувачів, тобто те, що не було відстежене на тестових даних. Подібні помилки повинні бути виправлені щонайшвидше.

В період накопичення інформації виявиться найбільша кількість помилок, допущених при створенні інформаційної системи. Як правило, це помилки,

пов'язані з багатокористувальницьким доступом. Часто на етапі тестування таким помилкам не приділяється належної уваги — мабуть, із-за складності моделювання і дорожнечі засобів автоматизації процесу тестування інформаційної системи в умовах багатокористувальницького доступу. Деякі помилки виправити буде складно, оскільки вони є помилками проектування. Жоден хороший проект від них не застрахований. Це означає, що про всяк випадок треба резервувати час на локалізацію і виправлення таких помилок.

Друга категорія виправлень пов'язана з тим, що користувача не влаштовує інтерфейс. Тут не завжди потрібно виконувати абсолютно всі побажання користувача, інакше процес введення в експлуатацію не кінчиться ніколи.

Позитивні сторони застосування каскадного підходу полягають в наступному:

- дає план і часовий графік по усіх етапах проекту, упорядковує хід конструювання;
- на кожному етапі формується закінчений набір проектної документації, що відповідає критеріям повноти і узгодженості;
- виконувані в логічній послідовності етапи робіт дозволяють планувати терміни завершення усіх робіт і відповідні витрати.

Каскадний підхід добре зарекомендував себе при побудові ПЗ, для якого на самому початку розробки можна досить точно і повно сформулювати усі вимоги з тим, щоб надати розробникам свободу реалізувати їх якнайкраще з технічної точки зору.

Основними недоліками каскадного підходу є:

- істотне запізнювання з отриманням результатів, оскільки реальні проекти часто вимагають відхилення від стандартної послідовності кроків;
- вимога повного закінчення фази-діяльності, закріплення результатів у вигляді детального початкового документу (технічного завдання, проектної специфікації);
- узгодження результатів з користувачами робиться тільки в точках, що плануються після завершення кожного етапу робіт;
- користувачі і замовник не можуть ознайомитися з варіантами системи під час розробки, і бачать результат тільки в самому кінці;
- вимоги до ПЗ «заморожені» у вигляді технічного завдання на увесь час її створення, тому, користувачі можуть внести свої зауваження тільки після того, як робота над системою буде повністю завершена.

Незважаючи на наполегливі рекомендації компаній – експертів в області проектування і розробки ПЗ, багато компаній продовжують використати каскадну модель на практиці замість якого-небудь варіанту ітераційної моделі.

Головні причини, по яких *каскадна модель* зберігає свою популярність, наступні: звичка, розробка невеликих проектів, ілюзія зниження ризиків учасників проекту (замовника і виконавця), проблеми впровадження при використанні ітераційної моделі.

2.3.3. Компонентні моделі

У більшості програмних проектів застосовується повторне використання деяких програмних модулів (компонентів). Це зазвичай трапляється там, де розробники проекту знають про раніше створені програмні продукти, у складі яких є компоненти, що приблизно задовольняють вимогам компонентів, що розробляються.

Цей підхід заснований на наявності великої бази існуючих програмних компонентів, які можна інтегрувати в створювану нову систему. Часто такими компонентами є програмні продукти, що вільно продаються на ринку, які можна використати для виконання певних спеціальних функцій.

У цьому підході початковий етап специфікації вимог і етап атестації (тестування і введення в експлуатацію) такі ж, як і в інших моделях процесу створення ПЗ. А етапи, розташовані між ними, мають наступний сенс.

1. ***Аналіз компонентів***. Маючи специфікацію вимог, на цьому етапі здійснюється пошук компонентів, які могли б задовольняти сформульованим вимогам.
2. ***Модифікація вимог***. На цій стадії аналізуються вимоги з урахуванням інформації про компоненти, отриманої на попередньому етапі. Вимоги модифікуються так, щоб максимально використати можливості відібраних компонентів.
3. ***Проектування системи***. На цьому етапі проектується структура системи або модифікується існуюча структура повторно використовуваної системи.
4. ***Розробка і зборка системи***. Це етап безпосереднього створення системи. У рамках даного підходу зборка системи є швидше частиною розробки системи, чим окремим етапом.

Основні достоїнства описуваної моделі полягають в тому, що скорочується кількість компонентів, що безпосередньо розробляються, і зменшується загальна вартість створюваної системи.

2.3.4. Макетування (прототипування)

Досить часто замовник не може сформулювати детальні вимоги по введенню, обробці або виведенню даних для майбутнього програмного продукту. У цих випадках доцільно використати макетування.

Основна мета макетування: зняти невизначеності у вимогах замовника. Макетування (прототипування) – це процес створення моделі необхідного програмного продукту.

Модель може приймати одну з трьох форм:

1) паперовий макет або макет на основі ПК (зображує або малює человекомашинный діалог);
2) працюючий макет (прототип, версія програми), що виконує деяку частину необхідних функцій;
3) існуюча програма (версія), характеристики якої потім мають бути поліпшені.

Як показано на рис. 2.2, макетування грунтується на багатократному повторенні ітерацій, в яких беруть участь замовник і розробник.

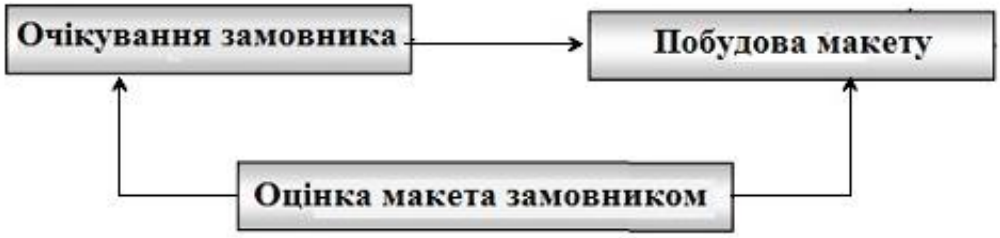


Рисунок 2.2 – Макетування (прототипування)

2.3.5. Ітеративні (інкрементні) моделі

Ітеративна модель є класичним прикладом інкрементної стратегії розробки. Вона об'єднує елементи послідовної водоспадної моделі з ітераційним макетуванням. Кожна лінійна послідовність тут виробляє інкремент (версію) ПЗ, що поставляється.

Основний недолік каскадного підходу – відсутність гнучкості. Саме цей недолік долається каскадно-поворотним підходом, в якому дозволені повернення до попередніх стадій і перегляд або уточнення раніше прийнятих рішень. Каскадно-поворотний підхід відбиває ітеративний або ітераційний характер розробки ПЗ.

Ітеративні моделі припускають розбиття створюваної системи на набір кроків, які розробляються за допомогою декількох послідовних проходів усіх робіт або їх частини. На першій ітерації розробляється частина системи, не залежний від інших. При цьому велика частина або навіть повний цикл робіт проходиться на нього, потім оцінюються результати і на наступній ітерації або перша переробляється, або розробляється наступний, який може залежати від першого, або якось поєднується доопрацювання першу ітерацію з додаванням нових функцій.

В результаті на кожній ітерації можна аналізувати проміжні результати робіт і реакцію на них усіх зацікавлених осіб, включаючи користувачів, і вносити зміни, що коригують, на наступних ітераціях. Кожна ітерація може містити повний набір видів діяльності від аналізу вимог, до введення в експлуатацію чергової частини ПЗ. Каскадна модель з можливістю повернення на попередній крок стає ітеративною (рис. 2.3).

Процес ітераційної розробки має цілий ряд достоїнств.

Замовникові немає необхідності чекати повного завершення розробки системи, щоб отримати про неї представлення. Компоненти, отримані на перших кроках розробки, задовольняють найбільш критичним вимогам (оскільки зазвичай мають найбільший пріоритет) і їх можна оцінити на самій ранній стадії створення системи.

Замовник може використати компоненти, отримані на перших кроках розробки, як прототипи і провести з ними експерименти для уточнення вимог до тих компонент, які розроблятимуться пізніше.

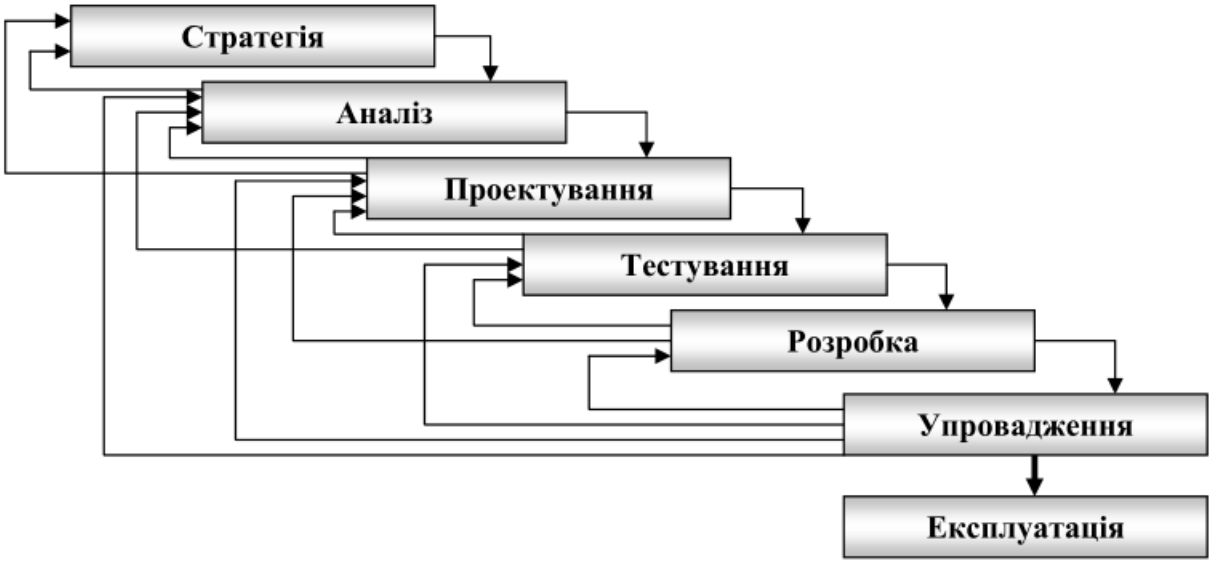


Рисунок 2.3 – Можливий хід робіт по ітеративній каскадній моделі

2.3.6. Спіральна модель

Розвитком ідеї ітерацій є ***спіральна модель життєвого циклу ПЗ, запропонована Барри Боэмом*** в 1988 році з метою скоротити можливий ризик розробки. Спіральна модель використовує поняття ***прототипу*** – програми, що реалізовує часткову функціональність створюваного програмного продукту. Створення прототипів здійснюється за декілька витків спіралі, кожен з яких складається з «аналізу ризику», «деякого процесу» і «верифікації».

Спіральна модель (рис. 2.4) – класичний приклад застосування еволюційної стратегії конструювання, була запропонована для подолання перерахованих проблем в попередніх моделях.

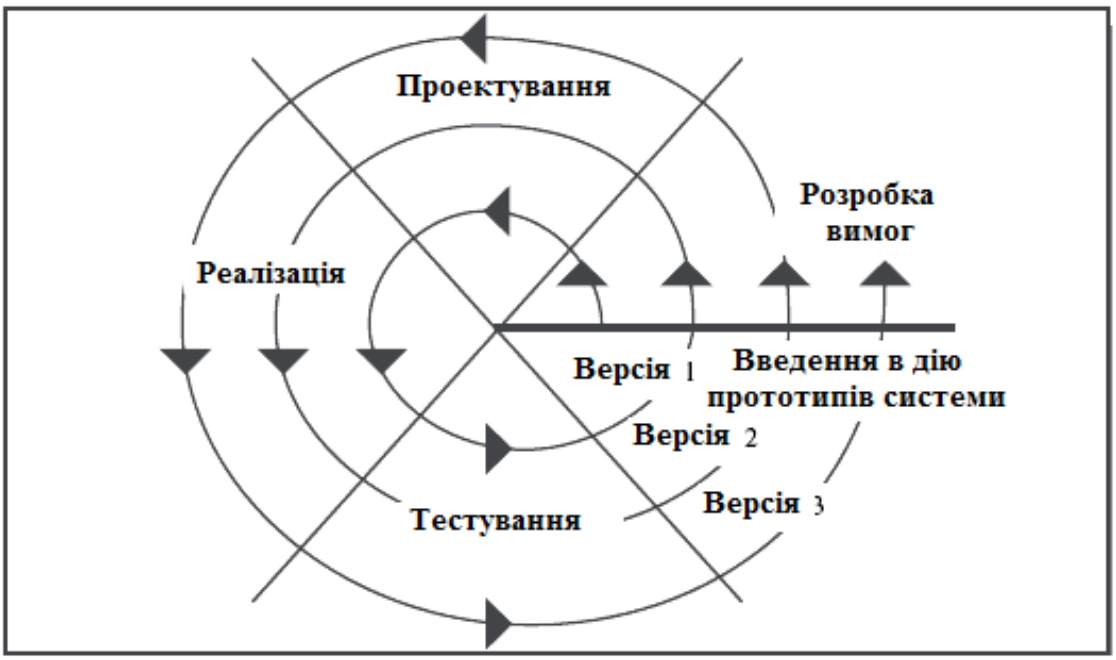


Рисунок 2.4 – Спіральна модель ЖЦІС

Кожен виток спіралі відповідає створенню фрагмента або версії ПЗ, на нім уточнюються цілі і характеристики проекту, визначається його якість і плануються роботи наступного витка спіралі. Таким чином поглиблюються і послідовно конкретизуються деталі проекту і в результаті вибирається обгрунтований варіант, який доводиться до реалізації.

Достоїнствами спіральної моделі є:

1. Неповне завершення робіт на кожному етапі дозволяє переходити на наступний етап, не чекаючи повного завершення роботи на поточному.
2. Кожен виток спіралі відповідає створенню фрагмента або версії ПЗ, на нім уточнюються цілі і характеристики проекту.
3. Дозволяє явно враховувати ризик на кожному витку еволюції розробки.
4. Використовує моделювання для зменшення ризику і вдосконалення програмного виробу.

Серед проблем спірального циклу можна виділити наступні:

1. Визначення моменту переходу на наступний етап. Для її вирішення необхідно ввести тимчасові обмеження на кожного з етапів життєвого циклу.
2. Підвищені вимоги до замовника.
3. План складається на основі статистичних даних, отриманих в попередніх проектах, і особистого досвіду розробників.

У спіральній моделі немає фіксованих етапів, таких як розробка специфікації або проектування. Ця модель може включати будь-які інші моделі розробки систем на певному витку спіралі.

Контрольні питання до розділу 2

1. Дайте визначення *життєвого циклу ПЗ*.
2. Які ви знаєте основні процеси життєвого циклу ПЗ?
3. Що таке верифікація ПЗ?
4. Що таке валідація ПЗ?
5. Які ви знаєте основні стратегії конструювання ІС?
6. Перерахуйте основні переваги і недоліки каскадної моделі ЖЦІС.
7. Перерахуйте основні переваги і недоліки спіральної моделі ЖЦІС.

РОЗДІЛ 3. ТЕХНОЛОГІЇ СТВОРЕННЯ ІС

У наступних розділах ми коротко розглянемо **канонічне проектування ІС** і три сучасні процеси розробки – **уніфікований процес Rational** (Rational Unified Process, RUP), **екстремальне програмування** (Extreme Programming, XP), **Scrum**-технологию. Усі три сучасні технології розробки є прикладами ітеративних процесів, але побудовані вони на основі різних припущень про природу розробки ПЗ.

RUP є прикладом так званого «важкого» процесу, детально описаного і припускаючого підтримку власне розробки початкового коду ПЗ великою кількістю допоміжних дій.

Екстремальне програмування, навпаки, представляє так звані «живі» (agile) методи розробки, які роблять упор на використанні хороших розробників, а не на добре відлагоджені процеси розробки. Живі методи уникають фіксації чітких схем дій, щоб забезпечити велику гнучкість в кожному конкретному проекті.

Scrum-технология (метод Scrum). Достоїнствами XP є велика гнучкість і простота розуміння, проте в повному об'ємі XP не була використана навіть її авторами. Крім того, відомі і успішно впроваджуються окремі практики XP, як, наприклад, парне програмування, колективне володіння кодом, і, звичайно ж, рефакторинг кода. Тому ми розглянемо поліпшену технологію – Scrum-технологию.

3.1. Канонічне проектування ІС

3.1.1. Організація канонічного проектування ІС

Організація канонічного проектування ІС орієнтована на використання головним чином каскадної моделі життєвого циклу ІС. Стадії і етапи роботи описані в стандарті ГОСТ 34.601-90.

Стадії і етапи створення ІС, виконувані організаціями-учасниками, прописуються в договорах і *технічних завданнях* на виконання робіт:

Стадія 1. Формування вимог до ІС.

Стадія 2. Розробка концепції ІС: вивчення об'єкту автоматизації, розробка варіантів концепції ІС, що задовольняють вимогам користувачів, оформлення звіту і затвердження концепції.

Стадія 3. Технічне завдання: Розробка і затвердження *технічного завдання* на створення ІС.

Стадія 4. Ескізний проект: розробка попередніх проектних рішень по системі і її частинам, розробка ескізної документації на ІС.

Стадія 5. Технічний проект: розробка проектних рішень по системі і її частинам, розробка документації на ІС і її частини.

Стадія 6. Розробка робочої документації на ІС і її частини.

Стадія 7. Введення в дію: комплектація ІС виробами (програмними і технічними засобами), що поставляються, проведення попередніх випробувань, проведення дослідної експлуатації, проведення приймальних випробувань.

Стадія 8. Супровід ІС: виконання робіт відповідно до гарантійних зобов'язань, післягарантійне обслуговування.

3.1.2. Обстеження об'єкту і обгрунтування створення ІС

Обстеження **–** це вивчення і діагностичний аналіз організаційної структури підприємства, його діяльності і існуючої системи обробки інформації. Матеріали, отримані в результаті обстеження, використовуються для:

- обгрунтування розробки і поетапного впровадження систем;
- складання *технічного завдання* на розробку систем;
- розробки технічного і робочого проектів систем.

На етапі *обстеження* доцільно виділити дві складові: визначення стратегії впровадження ІС і детальний аналіз діяльності організації.

Основне завдання першого етапу *обстеження* – оцінка реального об'єму проекту, його цілей і завдань на основі виявлених функцій і інформаційних елементів об'єкту високого рівня, що автоматизується. Ці завдання можуть бути реалізовані або замовником ІС самостійно, або із залученням консалтингових організацій. Етап припускає тісну взаємодію з основними потенційними користувачами системи і бізнес-експертами.

Результатом етапу визначення стратегії є документ (*техніко-економічне обгрунтування проекту*), де чітко сформульовано, що отримає замовник, якщо погодиться фінансувати проект, коли він отримає готовий продукт (графік виконання робіт) і скільки це коштуватиме (для великих проектів має бути складений графік фінансування на різних етапах робіт).

На етапі детального аналізу діяльності організації вивчаються завдання, що забезпечують реалізацію функцій управління, організаційна структура, штати і зміст робіт по управлінню підприємством, а також характер підлеглості вищестоящим органам управління.

Одній з найбільш трудомістких, хоча і завдань цього етапу, що добре формалізуються, являється опис документообігу організації. При *обстеженні* документообігу складається схема маршруту руху документів, яка повинна відбити:

- кількість документів;
- місце формування показників документу;
- взаємозв'язок документів при їх формуванні;
- маршрут і тривалість руху документу;
- місце використання і зберігання цього документу;
- внутрішні і зовнішні інформаційні зв'язки.

За результатами *обстеження* встановлюється перелік завдань управління, рішення яких доцільно автоматизувати, і черговість їх розробки.

Результати *обстеження представляють об'єктивну основу для формування технічного завдання* на інформаційну систему.

3.1.3. Технічне завдання

Технічне завдання – це документ, що визначає цілі, вимоги і основні початкові дані, необхідні для розробки автоматизованої системи управління.

При розробці технічного завдання необхідно вирішити наступні завдання:

- встановити спільну мету створення ІС, визначити склад підсистем і функціональних завдань;
- розробити і обгрунтувати вимоги, що пред'являються до підсистем;
- розробити і обгрунтувати вимоги, що пред'являються до інформаційної бази, математичного і програмного забезпечення, комплексу технічних засобів (включаючи засоби зв'язку і передачі даних);
- встановити загальні вимоги до проектованої системи;
- визначити перелік завдань створення системи і виконавців;
- визначити етапи створення системи і терміни їх виконання;
- провести попередній розрахунок витрат на створення системи і визначити рівень економічної ефективності її впровадження.

Приведемо типові вимоги до складу (розділи) і змісту технічного завдання (ГОСТ 34.602- 89).

1. Загальні відомості:

- повне найменування системи і її умовне позначення;
- шифр теми або шифр (номер) договору;
- найменування підприємств розробника і замовника;
- перелік документів, на підставі яких створюється ІС;
- планові терміни початку і закінчення робіт;
- відомості про джерела і порядок фінансування робіт;
- порядок оформлення і пред'явлення замовникові результатів робіт із створення системи, її частин і окремих засобів.

2. Призначення і цілі створення (розвитку) системи:

- вид діяльності, що автоматизується, перелік об'єктів, на яких передбачається використання системи;
- найменування і необхідні значення технічних, виробничо-економічних та ін. показників об'єкту, які мають бути досягнуті при впровадженні ІС.

3. Характеристика об'єктів автоматизації:

- короткі відомості про об'єкт автоматизації;
- зведення про умови експлуатації і характеристики ІС довкілля.

4. Вимоги до системи.

Вимоги до системи в цілому:

- вимоги до структури і функціонування системи (перелік підсистем, рівні ієрархії, способи інформаційного обміну, взаємодія з суміжними системами, перспективи розвитку системи);

- вимоги до персоналу (чисельність користувачів, кваліфікація, режим роботи, порядок підготовки);
- показники призначення (міра пристосовності системи до змін процесів управління і значень параметрів);
- вимоги до надійності, безпеки, ергономіки, транспортабельності, експлуатації, захисту і збереження інформації, захисту від зовнішніх дій.

Вимоги до функцій (по підсистемах):

- перелік тих, що підлягають автоматизації завдань;
- часовий регламент реалізації кожної функції;
- вимоги до якості реалізації кожної функції.

Вимоги до видів забезпечення:

- математичному (склад і сфера застосування мат. моделей і методів, типових алгоритмів, що розробляються);
- інформаційному (склад, структура і організація даних, обмін даними між компонентами системи, сумісність з суміжними системами, СУБД);
- лінгвістичному (мови програмування, мови взаємодії користувачів з системою, системи кодування, мови введення-виводу);
- програмному (незалежність програмних засобів від платформи, якість програмних засобів);
- організаційному (структура і функції експлуатуючих підрозділів, захист від помилкових дій персоналу).

5. Склад і зміст робіт із створення системи:

- перелік стадій і етапів робіт;
- терміни виконання;
- склад організацій – виконавців робіт.

6. Порядок контролю і приймання системи:

- види, склад, об’єм і методи випробувань системи;
- загальні вимоги до приймання робіт по стадіях.

7. Вимоги до складу і змісту робіт з підготовки об’єкту до введення системи в дію:

- перетворення вхідної інформації до машиночитаемому виду;
- зміни в об’єкті автоматизації;
- терміни і порядок комплектування і навчання персоналу.

8. Вимоги до документування:

- перелік тих, що підлягають розробці документів;
- перелік документів на машинних носіях.

9. Джерела розробки:

- документи і інформаційні матеріали, на підставі яких розробляється ТЗ і система.

3.1.4. Ескізний проект

Ескізний проект передбачає розробку попередніх проектних рішень по системі і її частинам.

Виконання стадії ескізного проектування не є строго обов'язковим. Якщо основні проектні рішення визначені раніше або досить очевидні для конкретної ІС і об'єкту автоматизації, то ця стадія може бути виключена із загальної послідовності робіт.

Як правило, на етапі ескізного проектування визначаються:

- функції ІС і її підсистем, їх цілі і очікуваний ефект від впровадження;
- склад комплексів завдань і окремих завдань;
- концепція інформаційної бази і її укрупнена структура;
- функції системи управління базою даних;
- склад обчислювальної системи і інших технічних засобів;
- функції і параметри основних програмних засобів.

За результатами виконаної роботи оформляється, узгоджується і затверджується документація в об'ємі, необхідному для опису повної сукупності прийнятих проектних рішень.

На основі *технічного завдання (і ескізного проекту) розробляється технічний проект* ІС.

3.1.5. Технічний проект

Технічний проект системи – це технічна документація, що містить загальносистемні проектні рішення, алгоритми рішення завдань, а також оцінку економічної ефективності автоматизованої системи управління і перелік заходів по підготовці об'єкту до впровадження.

На цьому етапі здійснюється комплекс науково-дослідних і експериментальних робіт для вибору основних проектних рішень і розрахунок економічної ефективності системи.

3.1.6. Робоча документація і випробування ІС

На стадії «**робоча документація»** здійснюється створення програмного продукту і розробка усієї супроводжуючої документації. Документація повинна містити усі необхідні і достатні відомості для забезпечення виконання робіт по введенню ІС в дію і її експлуатації, а також для підтримки рівня експлуатаційних характеристик (якості) системи. Розроблена документація має бути відповідним чином оформлена, погоджена і затверджена.

Для ІС, які є різновидом автоматизованих систем, встановлюють наступні основні види випробувань: *попередні випробування, дослідна експлуатація і приймальні випробування*.

Попередні випробування проводять для визначення працездатності системи і вирішення питання про можливість її приймання в *дослідну експлуатацію*. Попередні випробування слід виконувати після проведення розробником відладки і тестування програмних і технічних засобів системи і представлення, що поставляються, ним відповідних документів про їх готовність до випробувань.

Дослідну експлуатацію системи проводять з метою визначення фактичних значень кількісних і якісних характеристик системи і готовності персоналу до роботи в умовах її функціонування, а також визначення фактичної ефективності і коригування, при необхідності, документація.

Приймальні випробування проводять для визначення відповідності системи *технічному завданню, оцінки якості дослідної експлуатації* і вирішення питання про можливість приймання системи в постійну експлуатацію.

3.2. Уніфікований процес Rational

Уніфікований процес Rational (Rational Unified Process, RUP) [4] є досить складною, детально пропраццьованною ітеративною моделлю життєвого циклу ПЗ.RUP є програмним продуктом в якості доповнення до мови моделювання UML, розроблений компанією Rational Software, яка нині входить до складу IBM.

3.2.1. Основні принципи RUP

Історично RUP є розвитком моделі процесу розробки, прийнятої в компанії Ericsson в 70-х-80-х роках XX століття. Ця модель була створена Джекобсоном (Ivar Jacobson), який у 1987 заснував власну компанію Objectory AB саме для розвитку технологічного процесу розробки ПЗ як окремого продукту, який можна було б переносити в інші організації. Після вливання Objectory в Rational в 1995 розробки Джекобсона були інтегровані з роботами Ройса (Walker Royce, сина автора «класичної» каскадної моделі), Крухтена (Philippe Kruchten) і Буча (Grady Booch), а також з тим, що розвивалося паралельно універсальною мовою моделювання (Unified Modeling Language, UML) [17].

Основними принципами RUP є:

1. Ітераційний і інкрементний (нарощуваний) підхід до створення ПЗ.
2. Планування і управління проектом на основі функціональних вимог до системи – варіантів використання.
3. Побудова системи на базі архітектури ПЗ.

Перший принцип є визначальний. Відповідно до нього розробка системи виконується у вигляді декількох короткострокових міні-проектів фіксованої тривалості (від 2 до 6 тижнів), що називаються ітераціями. Кожна ітерація включає свої власні етапи аналізу вимог, проектування, реалізації, тестування, інтеграції і завершується створенням працюючої системи.

3.2.2. Фази проекту

З точки зору етапів (фаз) проектування, то вони, загалом, аналогічні «водоспаду». Але RUP виділяє в життєвому циклі 4 основних фази, у рамках кожної з яких можливе проведення декількох ітерацій.

1. **Фаза початку проекту (Inception).** Основна мета цієї фази – досягти компромісу між усіма зацікавленими особами відносно завдань проекту і ресурсів, що виділяються на нього. На цій фазі визначаються основні цілі проекту, керівник проекту і бюджет проекту, основні засоби його виконання – технології, інструменти, ключові виконавці, а також відбувається, можливо, апробація вибраних технологій з метою підтвердження можливості досягти цілей з їх допомогою. На цю фазу може йти близько 10% часу і 5% трудомісткості одного циклу.

Результатами початкової стадії є:

- загальний опис системи: основні вимоги до проекту, його характеристики і обмеження;
- початковий бізнес-план;
- план проекту, що відбиває стадії і ітерації;
- один або декілька прототипів.

2. Фаза проектування (Elaboration). Основна мета цієї фази – на базі основних, найбільш суттєвих вимог розробити стабільну базову архітектуру продукту, яка дозволяє вирішувати поставлені перед системою завдання і надалі використовується як основа розробки системи. На цю фазу може йти близько 30% часу і 20% трудомісткості одного циклу.

3. Фаза побудови (Construction). Основна мета цієї фази – детальне прояснення вимог і розробка системи, що задовольняє їм, на основі спроектованої раніше архітектури. В результаті повинна вийти система, що реалізовує усі виділені варіанти використання. На цю фазу йде близько 50% часу і 65% трудомісткості одного циклу.

Результатами стадії розробки є:

- модель варіантів використання (завершена принаймні на 80%), що визначає функціональні вимоги до системи;
- перелік додаткових вимог, включаючи вимоги нефункціонального характеру;
- опис базової архітектури майбутньої системи, працюючий прототип;
- план розробки усього проекту, що відбиває ітерації і критерії оцінки для кожної ітерації.

4. Фаза впровадження (Transition). Мета цієї фази – зробити систему повністю доступною кінцевим користувачам. На цій стадії відбувається розгортання системи в її робочому середовищі, бета-тестування, підгонка дрібних деталей під потреби користувачів. На цю фазу може йти близько 10% часу і 10% трудомісткості одного циклу.

3.2.3. Ключові ідеї RUP

Робочі продукти (артефакти), що виробляються в ході проекту, можуть бути представлені у вигляді баз даних і таблиць з інформацією різного типу, різних видів документів, початкового коду і об’єктних модулів, а також моделей, що складаються з окремих елементів.

Найбільш важливі з точки зору RUP артефакти проекту – це моделі, що описують різні аспекти майбутньої системи. Більшість моделей є наборами діаграм UML.

Перерахуємо основну техніку, використовувану в RUP:

1. Вироблення концепції проекту на його початку для чіткої постановки завдань.
2. Управління за планом.
3. Зниження ризиків і відстежування їх наслідків, як можна більше ранній початок робіт по подоланню ризиків.
4. Ретельне економічне обгрунтування усіх дій – робиться тільки те, що треба замовникові.
5. Як можна більше раннє формування базової архітектури.
6. Використання компонентної архітектури.
7. Прототипирование, інкрементна розробка і тестування.
8. Регулярні оцінки поточного стану.
9. Управління змінами, постійний відробіток змін ззовні проекту.
10. Націленість на створення продукту, працездатного в реальному оточенні.
11. Націленість на якість.
12. Адаптація процесу під потреби проекту.

Як видно, фази в цілому відповідають моделі водоспаду, з тим виключенням, що фаза супроводу не розглядається як окрема. В цілому, при розробці проекту RUP базується на чотирьох ключових ідеях.

1. Ключовою ідеєю процесу є його итеративность – кожна фаза, починаючи з розвитку, включає декілька ітерацій, на кожній з яких виконується свій шматочок аналізу, проектування, реалізації і тестування готового продукту (вірніше за його частину).
2. Увесь хід робіт спрямовується підсумковими цілями проекту (ітераціями), вираженими у вигляді ***прецедентів використання*** або ***варіантів використання (use cases)*** – сценаріїв взаємодії результуючої програмної системи з користувачами або іншими системами. Вимоги аналізуються також як і проект на кожній ітерації і в цілому дуже ретельно, але не до кінця – діє правило 70-80% – розробник повинен уявляти собі вимоги саме настільки, перш ніж починати кодування.
3. На етапі розвитку приймаються ключові рішення, що стосуються ***архітектури*** системи в цілому, її властивостей, функцій, використовуваних технологій і так далі. У кінці цієї фази реалізується 10-30% системи, але усі основні рішення вже прийняті (до 70-80%) і на етапі конструювання у рамках цих рішень система доводиться до кінця. Архітектура встановлює набір компонентів, з яких буде побудовано ПО, відповідальність кожного з компонентів (тобто вирішувані ним підзадачі у рамках загальних завдань системи), чітко визначає інтерфейси, через які вони можуть взаємодіяти, а також способи взаємодії компонентів один з одним.

4. Основою процесу розробки є ***плановані і керовані ітерації,*** об'єм яких визначається на основі архітектури (функціональність, що реалізовується у рамках ітерації, і набір компонентів).

RUP як продукт входить до складу інтегрованого комплексу інструментальних засобів **Rational Suite**, який існує в наступних варіантах:

- Rational Suite AnalystStudio – призначений для визначення і управління повним набором вимог до системи, що розробляється;
- Rational Suite DevelopmentStudio – призначений для проектування і реалізації ПЗ;
- Rational Suite TestStudio – є набір продуктів, призначених для автоматичного тестування додатків;
- Rational Suite Enterprise – забезпечує підтримку повного життєвого циклу ПЗ і призначений як для менеджерів проекту, так і окремих розробників, що виконують декілька функціональних ролей в команді розробників.

До складу Rational Suite, окрім самої технології RUP як продукту, входять наступні компоненти:

- Rational Rose – засіб візуального моделювання (аналізу і проектування), що використовує мову UML;
- Rational XDE – засіб аналізу і проектування, інтегрований з платформами MS Visual Studio .NET і IBM WebSphere Studio Application Developer;
- Rational Requisite Pro – засіб управління вимогами, призначене для організації спільної роботи групи розробників;
- Rational Rapid Developer – засіб швидкої розробки додатків на платформі Java 2 Enterprise Edition;
- Rational SoDA – засіб автоматичної генерації проектної документації;
- Rational Quantify – засіб кількісного визначення вузьких місць, що впливають на загальну ефективність роботи програми;
- Rational TestManager – засіб планування функціонального і навантаження тестування;
- Rational TestFactory – засіб тестування надійності;
- Rational Quality Architect – засіб генерації коду для тестування.

Засоби автоматичної генерації коду, використовуючи інформацію, що міститься в діаграмах класів і компонентів, формують файли описів класів. Створюваний таким чином скелет програми може бути уточнений шляхом прямого програмування на відповідній мові (основні мови, підтримувані Rational Rose, – C++ і Java).

Уніфікований процес RUP хоча і є досить складною ітеративною моделлю розробки життєвого циклу ПЗ, все ж може з успіхом застосуються, якщо на деяких етапах (фазах) допускати деякі «вільності».

Серед проблем власне кодування, слід виділити небажання деяких розробників (а також деяких замовників) використати компоненти сторонніх виробників. Щоб не винаходити велосипед, при написанні коду також можуть використовуватися готові програмні конструкції (готові рішення), типові

рішення, шаблони, так звані «**патерни» (**design patterns – зразки проектування або просто patterns).

Проте які б поліпшення ми не вносили, техніку RUP властиві серйозні обмеження і недоліки:

1. Він уніфікований, тобто підходить для різнорідних процесів, проектів, розробок, що робить його трохи заплутаним і неконкретним (мало конкретних чітких рекомендацій).
2. Він важкуватий для невеликих проектів і колективів розробників, особливо коли бюджет і терміни проектів невеликі.
3. Він вимагає глибокого осмислення вимог, як і традиційний водоспад (хоча і залишає на потім 30%). У реальних ситуаціях і 70% відразу отримати важко!

3.3. Екстремальне програмування (XP-процес)

Останнім часом все більшу популярність стали набирати так звані «**гнучкі» (*«живі»)*** методи розробки ПЗ (Agile Software Development). Серед них найпоширенішим являться ***Екстремальне програмування (eXtreme Programming, XP)*** – полегшений (рухливий) процес (чи методологія), головний автор якого – Кент Бек (1999) [5; 35].

XP-процес орієнтований на групи малого і середнього розміру, що будують програмне забезпечення в умовах невизначених або швидко таких, що змінюються вимог.

Основні принципи «живої» розробки ПЗ зафіксовані в маніфесті «живої» розробки, що з’явився в 2000 році:

1. Люди, що беруть участь в проекті, і їх спілкування важливіші, ніж процеси і інструменти.
2. Працююча програма важливіша, ніж вичерпна документація.
3. Співпраця із замовником важливіша, ніж обговорення деталей контракту.
4. Відробіток змін важливіший, ніж наслідування планів.

«Живі» методи з’явилися як протест проти надмірної бюрократизації розробки ПЗ, великої кількості побічних що не є необхідними для отримання кінцевого результату документів, які доводиться оформляти при проведенні проекту відповідно до більшості «важких» процесів. Велика частина таких робіт і документів не має прямого відношення до розробки ПЗ і забезпечення його якості, а призначена для дотримання формальних пунктів контрактів на розробку, отримання і підтвердження сертифікатів на відповідність різним стандартам.

Основна ідея XP-процесу – усунути високу вартість зміни, характерну для додатків з використанням об’єктів, **патернів** (рішення типових проблем в певному контексті або готові програмні конструкції) і реляційних баз даних. Тому XP-процес має бути високо динамічним процесом. XP-група має справу зі змінами вимог на всьому протязі ітераційного циклу розробки, причому цикл складається з дуже коротких ітерацій.

За твердженням авторів XP, ця методика є не стільки наслідуванням якихось загальних схем дій, скільки застосування комбінації відповідної техніки (практик). Кожна техніка важлива, і без її використання розробка вважається такою, що йде не по XP.

1. Гра в планування (Planning game) – швидке визначення зони дії наступної реалізації шляхом об'єднання ділових пріоритетів і технічних оцінок. Замовник формує зону дії, пріоритетність і терміни з точки зору бізнесу, а розробники оцінюють і простежують просування (прогрес).

2. Часта зміна версій (Small releases) – швидкий запуск у виробництво простої системи, тобто найперша працююча версія повинна з'явитися якнайшвидше, і тут же повинна почати використовуватися. Для реалізації нових версій вводяться ще жорсткіші обмеження на тривалість однієї ітерації.

3. Метафора (Metaphor) – уся розробка проводиться на основі простої, загальнодоступної історії про те, як працює уся система. Метафора в досить простому і зрозумілому команді виді повинна описувати основний механізм роботи системи. Це поняття нагадує архітектуру, але повинне набагато простіше, усього у вигляді однієї-двох фраз описувати основну суть прийнятих технічних рішень.

4. Просте проектування (Simple design) – проектування виконується настільки просто, наскільки це можливо в даний момент. Не потрібно додавати функції заздалегідь – тільки після явного прохання про це. Уся зайва складність віддаляється, як тільки виявляється.

5. Тестування (Testing) – безперервне написання тестів для модулів, які повинні виконуватися бездоганно. Розробники спочатку пишуть тести, потім намагаються реалізувати свої модулі так, щоб тести спрацьовували.

6. Реорганізація (рефакторинг, Refactoring) – наведення ладу в коді, переробка окремих файлів, видалення максимальної кількості незрозумілих фрагментів коду, об'єднання класів на основі їх схожої функціональності, корекція коментарів, осмислене перейменування об'єктів. Система реструктурується у зв'язку з додаванням нової функціональності, але її поведінка не змінюється; мета – усунути дублювання, спростити систему.

7. Парне програмування (Pair programming) – увесь код пишеться двома програмістами, працюючими на одному комп'ютері. Об'єднання в пари довільно і міняється від завдання до завдання. Той, в чиїх руках клавіатура, намагається найкращим способом вирішити поточне завдання. Другий програміст аналізує роботу першого і дає поради, обмірковує наслідки тих або інших рішень, нові тести, менш прямі, але гнучкіші рішення.

8. Колективне володіння кодом (Collective ownership) – будь-який розробник може покращувати будь-який код системи у будь-який час. Ніхто не повинен виділяти свою власну область відповідальності, уся команда в цілому відповідає за увесь код.

9. Безперервна інтеграція (Continuous integration) – система інтегрується і будується багато разів в день, у міру завершення кожного завдання. Безперервне регресійне тестування, тобто повторення попередніх тестів, гарантує, що зміни вимог не приведуть до регресу функціональності.

10. 40-годинний тиждень (40 – hour week) – як правило, працюють не більше 40 годин в тиждень. Не можна подвоювати робочий тиждень за рахунок наднормових робіт.

11. Локальний замовник (On – site customer) – в групі увесь час повинен знаходитися представник замовника, дійсно готовий відповідати на питання розробників. Його обов'язком є досить оперативні відповіді на питання будь-якого типу, що стосуються функцій системи, її інтерфейсу, необхідної продуктивності, правильної роботи системи в складних ситуаціях і ін.

12. Стандарти (стилі) кодування (Coding standards) – повинні витримуватися правила, що забезпечують однакове представлення програмного коду в усіх частинах програмної системи. Код розглядається як найважливіший засіб спілкування усередині команди. Ясність коду – один з основних пріоритетів.

13. Відкритий робочий простір (Open workspace). Команда розміщується в одному, досить просторому приміщенні, для спрощення комунікації і можливості проведення колективних обговорень при плануванні і ухваленні важливих технічних рішень.

14. Зміна правил з потреби (just rules). Кожен член команди повинен прийняти перераховані правила, але при виникненні необхідності команда може поміняти їх, якщо усі її члени прийшли до згоди з приводу цієї зміни.

Недоліками цього підходу деякі фахівці вважають нездійсненність в такому стилі досить великих і складних проектів, неможливість планувати терміни і трудомісткість проекту на досить довгу перспективу і чітко передбачити результати тривалого проекту в термінах співвідношення якості результату і витрат часу і ресурсів.

3.4. Scrum методологія

Достоїнствами XP є велика гнучкість і простота розуміння, проте в повному об'ємі XP не була використана навіть її авторами. Крім того, відомі і успішно впроваджуються окремі практики XP, як, наприклад, парне програмування, колективне володіння кодом, і, звичайно ж, рефакторинг кода. Ідея простого, ненадмірного дизайну проекту також зробила значний вплив на світ розробників ПО.

Більше практичним «гнучким» методом розробки є Scrum [35].

3.4.1. Історія

У 1986 японські фахівці Hirotaka Takeuchi і Ikujiro Nonaka опублікували повідомлення про новий підхід до розробки нових сервісів і продуктів (не обов'язково програмних). Основу підходу складала згуртована робота невеликої універсальної команди, яка розробляє проект на усіх фазах. Наводилася аналогія з регбі, де уся команда рухається до воріт супротивника як єдине ціле, передаючи (пасуючи) м'яч своїм гравцям як вперед, так і назад. На початку 90-х років цей підхід став застосовуватися в програмній індустрії і набув назви **Scum** (термін з регбі, що означає, – сутичка), в 1995 році Jeff

Sutherland і Ken Schwaber представили опис цього підходу на OOPSLA ‘95 – одній з найавторитетніших конференцій в області програмування. Відтоді метод активно використовується в індустрії і багаторазово описаний в літературі.

3.4.2. Загальний опис

Метод Scrum дозволяє гнучко розробляти проекти невеликими командами (7 чоловік плюс/мінус 2) в ситуації вимог, що змінюються. При цьому процес розробки итеративен і надає велику свободу команді.

Спочатку створюються вимоги до усього продукту. Потім з них вибираються найактуальніші і створюється план на наступну ітерацію. Впродовж ітерації плани не міняються (цим досягається відносна стабільність розробки), а сама ітерація триває 2-4 тижні. Вона закінчується створенням працездатної версії продукту (робочий продукт), яку можна пред’явити замовникові, запустити і продемонструвати, хай і з мінімальними функціональними можливостями. Після цього результати обговорюються і вимоги до продукту коригуються.

3.4.3. Ролі

У Scrum є всього три види ролей.

Власник продукту (Product Owner) – це менеджер проекту, який представляє в проекті інтереси замовника. У його обов’язки входить розробка початкових вимог до продукту (Product Backlog), своєчасна їх зміна вимог, призначення пріоритетів, дат постачання і ін. Важливо, що він абсолютно не бере участь у виконанні самої ітерації.

Scrum-мастера (Scrum Master) забезпечує максимальну працездатність і продуктивну роботу команди – як виконання Scrum-процесса, так і рішення господарських і адміністративних завдань. Зокрема, його завданням є обгороджування команди від усіх дій ззовні під час ітерації.

Scrum-команда (Scrum Team) – група, що складається з п’яти-дев’яти самостійних, ініціативних програмістів. Першим завданням команди є постановка для ітерації реально досяжних і пріоритетних для проекту в цілому завдань (на основі Project Backlog і при активній участі власника продукту і Scum-мастера). Другим завданням є виконання цього завдання щоб то не було, у відведені терміни і із заявленою якістю.

3.4.4. Практики

У Scrum визначені наступні практики.

Sprint Planning Meeting. Проводиться на початку кожного Sprint. Спочатку Product Owner, Scurm-мастер, команда, а також представники замовника і інші зацікавлені особи визначають, які вимоги з Project Backlog найбільш пріоритетні і їх слід реалізовувати у рамках цього Sprint. Формується Sprint Backlog. Далі Scurm-мастер і Scrum-команда визначають те, як саме буде

досягнута певна вище мета з Sprint Backlog. Для кожного елементу Sprint Backlog визначається список завдань і оцінюється їх трудомісткість.

Daily Scrum Meeting – п'ятнадцятихвилинна щоденна нарада, метою якої є досягти розуміння того, що сталося з часу попередньої наради, скоректувати робочий план згідно реаліям сьогоднішнього дня і позначити шляхи рішення існуючих проблем. Кожен учасник Scrum-команди відповідає на три питання: що я зробив з часу попередньої зустрічі, мої проблеми, що я робитиму до наступної зустрічі?

У цій нараді (15-20 хвилин) може брати участь будь-яка зацікавлена особа, але тільки учасники Scrum-команди мають право приймати рішення. На них лежить відповідальність за їх власні слова, і, якщо хтось з боку втручається і приймає рішення за них, тим самим він знімає відповідальність за результат з учасників команди.

Sprint Review Meeting. Проводиться у кінці кожного Sprint. Спочатку Scrum-команда демонструє Product Owner зроблену впродовж Sprint роботу, а той у свою чергу веде цю частину мітингу і може запросити до участі усіх зацікавлених представників замовника. Product Owner визначає, які вимоги з Sprint Backlog були виконані, і обговорює з командою і замовниками, як краще розставити пріоритети в Sprint Backlog для наступної ітерації. У другій частині мітингу робиться аналіз минулого спринту, який веде Scrum-мастер. Scrum-команда аналізує в останньому Sprint позитивні і негативні моменти спільної роботи, робить висновки і приймає важливі для подальшої роботи рішення. Scrum-команда також шукає шляхи для збільшення ефективності подальшої роботи. Потім цикл повторюється (рис. 3.6).

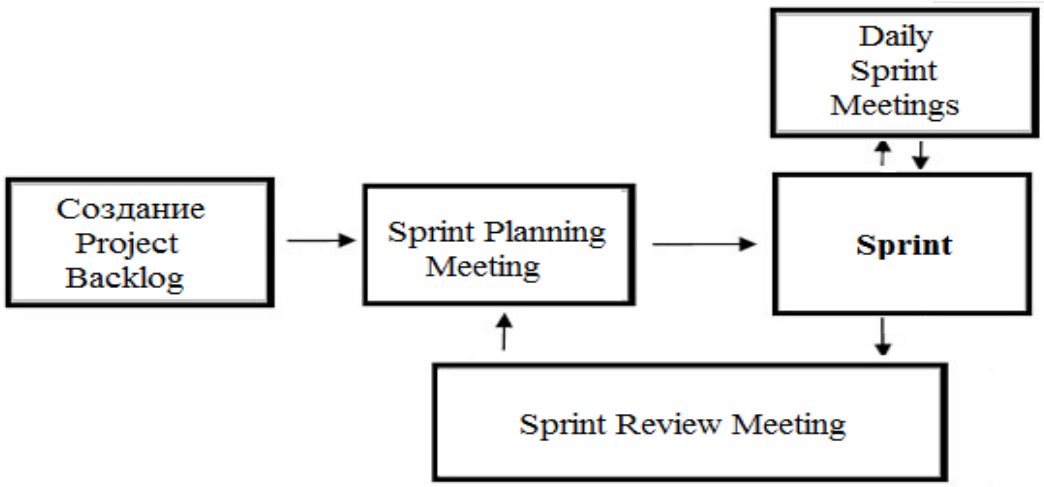


Рисунок 3.1 – Ітераційна схема створення ПЗ

Контрольні питання до розділу 3

1. Перерахуйте переваги і недоліки канонічної технології проектування ІС.
2. Перерахуйте переваги і недоліки RUP-технологии проектування ІС.
3. Перерахуйте переваги і недоліки XP-технологии.
4. Перерахуйте переваги і недоліки SCRUM-технологии.
5. Які ви знаєте стадії і етапи створення ІС із застосуванням канонічної технології?

РОЗДІЛ 4. УПРАВЛІННЯ ПРОЕКТОМ ПРИ РОЗРОБЦІ ІС

Цей розділ присвячений управлінню проектами по розробці або модифікації програмного забезпечення ІС, спробуємо обговорити стандартні компоненти життєвого циклу (ЖЦ) розробки ІС і причини необхідності дотримання ім. Також пояснимо, чому методологія ЖЦРП така важлива, і чому використання окремих компонент цієї методології зможе зробити рішення задачі простішим. Нарешті, ми розглянемо, як обійти деякі типові помилки, що виникають в процесі реалізації проекту [27; 28].

4.1. Основні поняття управління проектом

4.1.1. Поняття проекту

Інформаційна система розробляється як деякий проект. Багато особливостей управління проектами і фази розробки проекту (фази життєвого циклу) є загальними, не залежними не лише від предметної області, але і від характеру проекту (неважливо, інженерний це або економічний).

Проект – ця обмежена за часом цілеспрямована зміна окремої системи із спочатку чітко визначеним цілями, досягнення яких визначає завершення проекту, а також зі встановленими вимогами до термінів, результатів, ризику, рамок витрачання засобів і ресурсів.

Мета будь-якого програмного проекту полягає у виробництві певного програмного продукту. Поняття «продукт» задає не лише текст на мові програмування і двійковий код, що відкомпілювався, але і туди включаються документація, звіти по проміжних підсумках, результати перевірок, оцінки якості і т. д. Ці елементи зазвичай називають *артефактами*.

Розглядаючи планування проектів і управління ними, необхідно чітко усвідомлювати, що йдеться про управління деяким динамічним об’єктом. Тому система управління проектом має бути досить гнучкою, щоб допускати можливість модифікації без глобальних змін в робочій програмі.

4.1.2. Підбір команди і організаційні питання

При розробці великих проектів керівникам цих проектів доводилося шукати відповіді на безліч питань, що розпочинаються із слова «чому». Чому надвисокого IQ недостатньо для того, щоб ефективно керувати програмістами? Чому тільки менше 20% проектів розробки ПЗ завершуються в строк і укладаються до бюджету, а майже третина проектів анулюється до їх завершення?

Виявилось, що використання кращих мов і технологій програмування, найдосконаліших інструментів розробки і систем якості не гарантують успішність програмного проекту. «Саме людські якості забезпечують успіх тому або іншому проекту, саме вони є чинником першорядної ваги, грунтуючись на якому потрібно будувати прогнози про проект» [6].

Таким чином, головний, насправді, елемент в розробці будь-якого проекту – це зовсім не новітні комп'ютери, або новітні технології, або новітня версія C++. **Головне – це люди, талановиті люди**. На них усе тримається.

Тобто, усе вирішують таланти. Проста мобілізація засобів і зусиль вже не може забезпечити прогрес. Згадаємо Ф. Брукса [7], «Якщо проект не укладається в терміни, то додавання робочої сили затримає його ще більше». Ідею багатства тепер зв'язують не з грошима, а з людьми, не з фінансовим капіталом, а з «людським». Ринок праці перетворюється на ринок незалежних фахівців і його учасникам все більше відомо про можливі варіанти вибору. Працівники інтелектуальної праці починають самостійно визначати собі ціну.

Як нам вже відомо, суть усіх гнучких технологій: Scrum, xProgramming і інших, – полягає саме в тому, що вони декларують своєю вищою цінністю командну роботу, орієнтованість на людей і їх взаємодію, а не на процеси і засоби. Кожна розробка збирає навколо себе команду проекту. Ця команда проекту складається із зацікавлених в проекті осіб, які можуть грати в нім наступні ролі.

Кінцеві користувачі. Здійснюють введення в систему, яку ми розробляємо, забезпечують зворотний зв'язок в нашому проекті інтерфейсу, проводять бета-тестування і допомагають управляти визначенням досягнення кінцевої мети.

Розробники. Один з розробників є керівником команди проекту, її лідером, що регулює потік інформації між членами команди. Це найбільш авторитетна людина в команді проекту, до чиєї думки прислухаються, хто приймає більшість технічних рішень по ходу проекту. У команді може бути і менеджер проекту, в класичному розумінні що виконує тільки адміністративні обов'язки. Часто лідер і менеджер проекту – одна особа.

Начальник відділу. Ця особа, що очолює відділ, для якого ми пишемо додаток. На начальника відділу покладається відповідальність за достовірність даних, що видаються цим відділом інформаційній системі.

Відповідальний за гарантію якості. Ця роль в кожному проекті полягає в тому, щоб переконатися, що:

1. Проект додатка досягне намічених цілей.
2. Проект додатка відповідає опису системи.
3. План тестування відповідає вимогам.
4. План перетворення даних відповідає вимогам.

За якість проекту зазвичай відповідає одна людина, але стежать за якістю усі члени команди проекту.

Відповідальні за бета-тестування (дослідна експлуатація). Зазвичай групу команди бета-тестирования входять можливі кінцеві користувачі. Існує два типи тестування. Один використовує плани спеціального тестування, розроблені відповідальним за гарантію якості. Інший використовує тести, які розробили самі відповідальні за бета-тестування, застосовуючи критерії відповідальної за гарантію якості.

Перш, ніж стурбуватися створенням основ технології, керівникові (менеджерові, начальникові) проекту необхідно по максимуму вирішити усі організаційні проблеми.

1. Постарайтеся, наскільки можливо, розділити територіально розробників ПЗ і інших фахівців. При сучасному підході до розробки ПЗ дуже високо цінується легкість комунікації. Поза сумнівом, прекрасно, коли розробник може, зробивши два кроки, уточнити у безпосереднього замовника спосіб вирішення проблеми, вхідні дані або що-небудь ще, але, посадивши менеджера по продажах і розробника за сусідні столи, ви найменше в чверть понизите продуктивність праці останньої. Основний принцип: усі розробники мають бути доступні один для одного, як організаційно, так і фізично. Структура в групі – однорангова, тобто існує тільки професійний авторитет, але не адміністративне ділення.

2. Забороніть, кому б то не було, окрім менеджера проекту, безпосередньо давати вказівки розробникам. Проект з великою часткою вірогідності вийде з-під контролю, якщо розробники виконуватимуть такі, нехай навіть термінові, розпорядження. Вимоги не фіксуватимуться, тести не робитимуться, модифікації здійснюватимуться на ходу.

3. Виділіть або людину, або час на обговорення завдань. Практика показує, що завдання по супроводу-розробці ПЗ виникають постійно впродовж дня, обставини майже завжди вимагають негайного з'ясування. Якщо ви маєте двох розробників, нехай вони по черзі спілкуються з представниками замовника, якщо у вас є спеціально виділений для цього менеджер – нехай спілкується він.

4. Виділіть для розробників окремий телефонний номер. Не залучайте розробників до секретарської роботи. Розробники часто відповідальні і пунктуальні люди – у менеджерів, при усьому до них повазі, часто немає часу.

5. Ніколи не залучайте розробників до некваліфікованої праці. Якщо ви, звичайно, не боїтеся втратити розробника. Пам'ятайте, розробник надзвичайно цінує свою кваліфікацію. Хороший розробник завжди зможе знайти собі нову роботу, а ви навряд чи легко знайдете нового розробника.

6. Не завантажуйте розробників одночасно декількома завданнями. Постійне відвернення уваги розробника від завдання до завдання, особливо якщо вони належать до різних предметних областей або розробляються на різних мовах програмування, окрім зниження ефективності роботи, має ще один негативний результат – настання апатії (чи як говорять деякі програмісти – «алергії») до завдання або до роботи взагалі.

7. Забезпечте для розробників навчання і обмін досвідом. Як керівник, ви не можете бути в курсі усіх останніх змін у світі розробки ПЗ. Ви можете вважати, що людині, яка уміє програмувати вчитися більше не потрібно, цілком достатньо досвіду, що отримується в процесі роботи. Проте якщо розробники не будуть в курсі останніх технологій, ваше ПЗ застаріє дуже швидко і стане стримуючим чинником на шляху до нових завоювань ринку.

4.1.3. Керівник проекту

Однією з поширених помилок є вибір керівника проекту, що не має відповідних технічних знань для реалізації цього проекту. Ця проблема зазвичай зустрічається на великих проектах, де потрібна велика команда програмістів. Часто існує технічний лідер, який може управляти проектом також добре, як і вирішувати технічні питання.

В процесі аналізу вимог замовника важливо, щоб в переговорах брав участь один з членів команди розробників, а у кращому разі, провідний технічний фахівець або технічний керівник проекту.

Якщо в процесі обговорення бере участь тільки адміністративна особа, або керівник проекту, далекий від проблем безпосереднього кодування, то виникає безліч проблем і питань, пов'язаних з можливою оптимізацією окремих операцій, створенням словника баз даних, системними вимогами до створюваного програмного забезпечення і так далі. З іншого боку, участь в обговоренні проекту технічних фахівців може привести до помітного спрощення проекту за рахунок приведення окремих вимог користувача до вже існуючих і раніше розроблених технологій задоволення цих вимог.

Якщо вказані рекомендації будуть дотримані, то технічна сторона розробки буде розглянута більш повно, що дасть можливість згодом уникнути багатьох помилок, пов'язаних з нерозумінням тією або іншою стороною технічних особливостей проекту.

Є багато обов'язків хорошого керівника проектом, але ми перерахуємо тут тільки деякі.

1. Сучасний керівник – це цілеспрямований організатор, у якого є натхнення, який створює силові поля, що притягують таланти, а не просто службовців, прагнучих зайняти робочі місця.

2. Ніхто більше не вірить в керівників, які завжди праві і прикидаються, що знають більше, ніж підлеглі. «Людьми не потрібно «управляти». Завдання – направляти людей. Мета – зробити максимально продуктивними специфічні навички і знання кожного окремого працівника» [8].

3. Керівник відповідає за щоденне керівництво цим проектом. Керівник може делегувати (передавати) іншій людині виконання яких-небудь деятельностей керівника, але не відповідальність.

4. Керівник не повинен дозволяти втручатися комусь іншому в керівництво проектом. Тому що за проект відповідає саме керівник, це його особиста відповідальність.

5. Керівник проекту, окрім планування завдання, здійснює ще і розподіл ресурсів, ґрунтуючись на своєму відчутті і дуже хорошому знанні завдання.

6. Керівник відповідає за переговори із замовником. Тільки він їх веде. Він представляє інтереси групи і берет на себе зобов'язання з її боку.

7. Людський чинник – один з найважливіших в управлінні проектами, але частенько його ігнорують. Головне – це люди, талановиті люди. Тому одне з головних завдань керівника – це уміння знаходити і виховувати талановитих розробників.

Інше завдання керівника – відводити різні проблеми від групи, щоб група розробників, працюючих над проектом, просто не знала про ті бурі, які бушують за стінами цього колективу, і не відволікалася на них. Неправильно поступають ті керівники, які, отримавши прочухан від замовника або власного керівництва, передають його тут же своїм виконавцям. Це абсолютно неприпустимо.

4.1.4. Командні ролі

Фахівці в області оцінки і розвитку команд стверджують, що існує лише дев'ять командних ролей, баланс яких є вирішальним чинником успіху або невдачі в командній роботі [9]. Приведемо деякі з них.

Генератор ідей. Оригінальний мислитель, який дає життя новим ідеям. Незалежний співробітник з розвиненою уявою, але подібно до інших людей має негативні риси вдачі – може бути надто чутливий до критики. Для успіху генератору ідей потрібні конструктивні стосунки з керівником або координатором групи.

Дослідник ресурсів. Так само, як і генератор ідей, в змозі привнести нові ідеї до групи, але ці ідеї будуть запозичені ззовні, завдяки широким контактам. Дещо безцеремонний, гнучкий і шукає сприятливі можливості. До негативних якостей характеру відносяться лінь, самовдоволення.

Координатор. Зазвичай формальний лідер групи. Керує і направляє групу у бік досягнення цілей. Може заздалегідь визначити, хто з працівників хороший для виконання необхідних завдань. Зазвичай спокійний, упевнений і розпорядливий. Проте іноді схильний до зайвого домінування, і група стає продовженням його сильного «Я».

Мотиватор. Енергійний і в змозі впроваджувати ідеї. Бачить світ як проект, який вимагає впровадження. Зазвичай упевнений, динамічний, емоційний і імпульсивний. Мотор групи, але може бути дратівливим, невгамовним, нелюб'язним.

Аналітик. Оцінює пропозиції і займає позицію спостерігача за просуванням. Не дає групі рухатися неправильним шляхом. Обачний, безпристрасний, має аналітичний склад розуму. Може здаватися байдужим, незацікавленим, іноді стає надмірно критичним.

Натхненник команди. Прагне об'єднувати і вносити гармонію в стосунки між членами групи. Займає позицію того, що розуміє чужі проблеми, прагне допомогти і згладжує конфлікти. За вдачею людина добра, прагне налагоджувати неформальні стосунки. Проте буває нерішучим в складних або кризових ситуаціях.

Фахівець. Професіонал, самостійний прагне стати експертом у своїй області. Має високу професійну/технічну експертизу і знання, гордиться своєю роботою. Приносить вклад тільки у вузькій сфері своєї професійної експертизи.

4.1.5. Комунікації

Найважливішим неформальним макропоказником стану проекту є комунікації, їх якість і кількість. Тільки завдяки ефективним комунікаціям можна досягати синергетичного ефекту, який відрізняє команду від просто групи. Вже з XP-технології відомо, що освоєння нової технології парою програмістів, які здійснюють інтенсивний обмін знаннями, відбувається мінімум в 3 рази швидше, ніж у разі, коли ту ж роботу виконує один програміст.

Недостатня кількість комунікацій свідчить, як правило, про відсутність команди, кожен поглиблений у своє завдання і не цікавиться, що роблять його колеги. В результаті буде зроблено не те, що треба, а то, що буде зроблено, навряд чи вдасться інтегрувати в єдину систему.

Для побудови ефективної комунікації необхідно обов'язково враховувати індивідуальні особливості людей. Для кожного типу особи існує свій найбільш ефективний спосіб спілкування. Якщо той, що говорить не бере до уваги індивідуальні особливості того, що слухає, комунікації, як правило, заходять у безвихідь. При цьому буде відсутня (повністю або частково) передача інформації.

Не може бути ефективної команди, якщо учасники не знають і не уміють робити свою справу. Р. М. Белбин у своїй книзі [9] приводить список чинників незрілості і неефективності співробітника:

1. Не дотримується інструкції.
2. Погано контролює час.
3. Не любить, коли контролюють його роботу.
4. Не звертає увагу на якість роботи.
5. Не може сконцентруватися на роботі.
6. Має особисті проблеми.
7. Перебільшує свої здібності.
8. Не виконує свою долю роботи.
9. Не любить змін в роботі.
10. Не повідомляє про помилки.
11. Не лояльний по відношенню до своєї компанії.

У більшості своїй ці негативні якості носять тимчасовий характер і походять від відсутності досвіду і недостатньої самостійності фахівця. Дружня підтримка і допомога, як правило, дозволяють впоратися з більшістю перерахованих проблем.

У свою чергу, ефективний програміст окрім технічних знань і умінь повинен володіти ще і особистими компетенціями, необхідними для командної роботи:

1. Займає активну позицію, прагне розширити свою відповідальність і збільшити особистий вклад в загальну справу.
2. Постійно придбає нові професійні знання і досвід, висуває нові ідеї, спрямовані на підвищення ефективності досягнення спільних цілей, домагається поширення своїх знань, досвіду і ідей серед колег.

3. Отримує задоволення від своєї роботи, гордиться її результатами і прагне, щоб ці ж почуття переживали усі колеги.
4. Чітко усвідомлює свої особисті і загальні цілі, розуміє їх взаємообумовленість, наполегливо прагне до їх досягнення.
5. Упевнений в собі і у своїх колегах, об'єктивно оцінює їх досягнення і успіхи, уважно відноситься до їх інтересів і думок, активно шукає взаємовигідне рішення в конфліктах.
6. Сприймає кожну нову проблему, як додаткову можливість підтвердити власний професіоналізм.

Буває так, що, начебто, сильний фахівець приносить команді більше шкоди, ніж користі. До патологій поведінка, яка неприйнятна в команді, слід віднести наступні:

1. Непорядність, брехливість, відсутність совісті і почуття справедливості.
2. Неповага і неувага до партнерів. Схильність до негативних оцінок інших. Грубість. «Кожен сам за себе! – ніхто тобі не допоможе»!.
3. Завищена самооцінка. Відчуття власної переваги.
4. Мудрування. Людина сильно переоцінює свій особистий вклад в загальну справу і тому вважає, що він повинен працювати менше, ніж його «менш здібні» колеги.

Оптимальний варіант, коли користувач має уявлення про технічну сторону обговорюваного завдання, а команда програмістів має досвід у сфері діяльності користувача. Коли поєднуються такі якості користувача і розробника, приблизно половина питань відразу знімається з обговорення за непотрібністю.

4.2. Планування програмного проекту

Ефективність керівництва програмним проектом цілком визначається правильністю планування робіт, які потрібні для його виконання. План допомагає передбачати можливі проблеми розробки ПЗ і ввести захисні заходи для їх попередження і рішення. План створюється на початковому етапі проекту і розглядається менеджерами і інженерами-розробниками як керівний документ, виконання якого повинне привести до успішного завершення проекту. Цей початковий план повинен максимально детально описувати усі етапи роботи проекту.

Планування є багатокроковим ітераційним процесом. Дуже важливо, щоб план регулярно переглядався – адже у міру роботи в проект безперервно поступають нові відомості. Важливими чинниками, які повинні враховуватися при розробці плану, є контрактні зобов'язання фірми, вимоги замовника, бюджетні і тимчасові обмеження.

Планування розпочинається з оцінки предметної області програмної системи, її розміру, трудовитрат і часу на розробку. Формується команда виконавців, розподіляються їх функції. При цьому враховуються обмеження за часом, бюджету, наявності і можливостям співробітників, матеріально-

технічному забезпеченню. Потім визначаються етапи розробки, контрольні віхи проекту і перелік артефактів – результатів кожного етапу.

Якщо зафіксовані внутрішні проблеми або замовник змінив/розширив список вимог, можливий перегляд первинних оцінок проекту. Такий перегляд може привести до модифікації початкового (чи вже проміжного) графіку робіт. Якщо зміни впливають на терміни завершення або вартість проекту, із замовником узгоджуються нові обмеження проекту. Після цього продовжують проект – переходять до наступного етапу роботи.

4.2.1. Структура плану управління програмним проектом

План управління проектом має бути складений так, щоб кожен знав, що і коли йому потрібно робити. Існує безліч стандартів для таких планів. Але у будь-якому випадку більшість планів містять наступні розділи [10].

1. Вступ.

1.1. Огляд проекту. Повинен визначати проект, але не намагатися охопити усі вимоги до нього. Самі вимоги наводяться в Специфікації вимог до програмного забезпечення.

1.2. Результуючі артефакти проекту. Список усіх документів, початкових файлів і кінцевих програмних продуктів, які мають бути створені.

1.3. Розвиток плану. Напрями очікуваного розширення і зміни.

1.4. Посилальні матеріали.

1.5. Визначення і абревіатури.

2. Організація проекту.

2.1. Модель процесу. Посилаються на тип процесу розробки, який буде використаний (наприклад, водоспадний, спіральний, инкрементальный).

2.2. Організаційна структура. Описується внутрішня організація команди.

2.3. Організаційні рамки і взаємозв’язки. Шляхи можливої взаємодії між організаціями. Усе це залежить від зацікавлених в проекті сторін. Наприклад, тут визначається, яким чином здійснюватиметься взаємодія між відділом розробки і маркетинговим, чи будуть це регулярні зустрічі або листування по електронній пошті і т. д.

2.4. Відповідальність за проект. Визначає межі відповідальності, тобто хто за що відповідає. Наприклад, за що несе відповідальність координатор підвищення ефективності команди при горизонтальній організації? Чи відповідає він за загальний успіх проекту, чи надає персональні рекомендації або займається тільки керівництвом?

3. Аналіз ризиків.

3.1. Цілі і пріоритети. Проголошується робоча філософія проекту.

3.2. Допущення, залежності і обмеження.

3.3. Управління рисками.

3.4. Механізми моніторингу і контролю. Визначають, хто управлятиме, контролюватиме і (чи) здійснюватиме перевірку проекту, а також пропонує, як і коли це повинно бути зроблено.

3.5. План розставляння кадрів.

4. Технічний процес.

4.1. Методи, інструменти і технології. Накладаються обмеження на мови і використовувані інструменти. Може містити інформацію про повторно використовувані вимоги і використання такої техніки, як зразки проектування.

4.2. Документація програмного забезпечення.

4.3. Функції супроводу проекту. Описані дії для підтримки процесу розробки, такі як управління конфігурацією і забезпечення якості. Якщо ж функція підтримки представлена в різних документах (наприклад, в плані управління конфігураціями або в плані якості), то в цьому пункті будуть посилання на ці документи. Інакше тут повністю специфікуються функції підтримки.

5. Розподіл робіт, графік і бюджет.

5.1. Розподіл робіт. Описує те, як робота повинна розподілятися і надаватися після виконання. Оскільки програмна архітектура ще не затверджена, перша версія цього пункту буде поверхневою. Деталі з'являються в подальших версіях плану.

5.2. Потреби в ресурсах. Оцінюються трудовитрати, апаратне і програмне забезпечення, необхідні для зборки і технічної підтримки продукту. Можуть бути приведені результати оцінки вартості. Цей пункт уточнюється і деталізується з кожною ітерацією.

5.3. Виділення бюджету і ресурсів. Розподіляються ресурси між різними частинами проекту впродовж усього його життєвого циклу. Наводяться оцінки вартості людино-дня, можуть вказуватися оцінки вартості апаратури і програмного забезпечення.

5.4. План-графік. Містить розклад, визначальний, як і коли мають бути виконані різні етапи процесу.

У міру виконання проекту план повинен регулярно переглядатися.

4.2.2. Структура графіку робіт програмного проекту

Складання графіку – одна з найвідповідальніших робіт менеджера проекту. Тут менеджер оцінює тривалість проекту, визначає ресурси, необхідні для реалізації робочих завдань, і розгортає послідовність завдань в часі. Як правило, ця дія виконується за допомогою спеціалізованої утиліти планування проекту.

Плануючі утиліти дозволяють:

- визначити критичний шлях (ланцюжок завдань, задаючих тривалість усього проекту);
- визначити тривалість критичного шляху;
- встановити для кожного завдання найбільш вірогідну тимчасову оцінку;
- вичислити межі, що визначають тимчасове вікно для окремого завдання.

Першими виконуваними завданнями є збір вимог і аналіз вимог. Вони закладають фундамент для подальших паралельних завдань. Збір вимог проводиться з метою:

- з'ясування потреб замовника;

- оцінки здійснимості програмної системи;
- виконання економічного і технічного аналізу;
- визначення вартості і обмежень планування;
- створення системної специфікації.

Аналіз вимог дає можливість:

- уточнити функції і характеристики програмного продукту;
- позначити інтерфейс продукту з іншими системними елементами;
- визначити проектні обмеження програмного продукту;
- побудувати моделі: процесу, даних, режимів функціонування продукту;
- створити такі форми представлення інформації і функцій системи, які можна використати в ході проектування.

Результати аналізу зводяться в специфікацію аналізу, що містить конкретизовані вимоги до програмного продукту.

Завдання по проектуванню і плануванню тестів можуть бути распараллелены. Завдяки модульній природі ПЗ для кожного модуля можна передбачити паралельний шлях для детального (процедурного) проектування, кодування і тестування. Після отримання усіх модулів ПЗ вирішується завдання тестування інтеграції – об'єднання елементів в єдине ціле. Далі проводиться тестування правильності, яке забезпечує перевірку відповідності ЗА вимогами замовника.

4.3. Управління персоналом

Найціннішим ресурсом програмного проекту є люди. В першу чергу важливі технічні навички інженерів-розробників. Проте ці навички необхідно застосовувати для вирішення проблем в потрібний час і в потрібному місці. Отже, передбачається комбінація двох стилів: робота в команді і лідерство. Організація команди, що забезпечує ефективну роботу, є дуже складним завданням для менеджера – керівника проекту. Хороша команда повинна демонструвати сплав найрізноманітніших якостей: професійні навички, досвід, згуртованість, дух товариства. Структура команди повинна стимулювати творчу роботу усіх і кожного [6].

4.3.1. Підбір членів команди

Передусім, керівник повинен організувати правильний підбір членів команди: вони можуть доповнювати один одного по навичках і досвіді і мають бути сумісні один з одним психологічно. При роботі з кандидатом менеджер зазвичай враховує наступні аспекти:

- досвід роботи у багатьох апаратно-програмних середовищах;
- знання мов програмування;
- освіта і досвід роботи за фахом;
- комунікабельність і здатність адаптуватися;
- особові якості.

Пояснимо деякі з перерахованих аспектів.

Освіта є комплексним показником початкових знань і навичок кандидата, а також його здібності до навчання. Досвід же характеризує кінцеві знання і навички фахівця.

Комунікабельність характеризує можливості спілкування з колегами, керівниками і іншими зацікавленими в проекті особами. Здатність до адаптації може пояснювати «послужний список» – наявний робочий стаж.

Особові якості оцінити, мабуть, найважче. Тут і психологічний портрет, і темперамент, і ініціативність, і цілеспрямованість, і багато що інше. Саме ці якості визначають сумісність кандидата з колективом.

Важливо також правильно вибрати лідера команди. Він відповідає за технічне керівництво або за адміністративне управління (можливо і поєднання цих обов'язків). Лідери мають бути в курсі повсякденної діяльності команди, забезпечуючи її ефективну роботу і співпрацю з керівництвом проекту. Вони повинні ладнати з усіма членами колективу, згладжуючи напруженості і усувати неприємності.

При програмуванні без персоналізації усі робочі продукти (моделі, код, документація) **вважаються власністю усієї команди**, а не окремого співробітника, який займався їх створенням.

Переваги розробки без персоналізації:

- спрощення процедур перевірки, критики недоліків, підвищення їх об'єктивності;
- заохочення стилю невимушеного обговорення робочих завдань, достоїнств і недоліків окремих рішень;
- активізація дружніх стосунків, підвищення рівня щирості;
- швидкий ріст майстерності (завдяки роботі пліч-о-пліч);
- поліпшення якості, вдосконалення результатів роботи.

4.3.2. Взаємодії в команді

Для команди програмного проекту просто потрібна розвинена система взаємодії, іншими словами, спілкування і хороші засоби зв'язку між співробітниками. Співробітники повинні інформувати один одного про хід роботи, рішення, що приймаються, а також про зміни, які внесені в попередні рішення. Постійна взаємодія теж сприяє згуртованості і підвищенню якості роботи, оскільки співробітники спільно обговорюють рішення, починають краще розуміти мотивацію своїх колег.

На ефективність взаємодії впливають наступні параметри.

1. Розмір/структура команди. З ростом числа учасників кількість зв'язків по взаємодії росте квадратично. Наприклад, між трьома учасниками є три зв'язки, чотири учасники мають шість зв'язків, п'ять чоловік – десять зв'язків, тобто n чоловік мають $(n – 2) +... + 1 = n(n – 1)/2$ зв'язків (кожен з кожним). Отже, 50 чоловік повинні брати участь в 1225 взаємодіях. Але ж це неможливо!

На початку 1990-х років засновник фірми Borland, Філіп Кан постулював формулу продуктивності розробки програмного забезпечення на конференції

COMDEX (Лас-Вегас, 1992). Він назвав цю формулу продуктивності розробки ПЗ як «Закон Філіпа». Закон свідчить, що продуктивність розробника програмного забезпечення в команді з «*N*» *людей зменшується, розділивши його на корінь кубічний з «N»*.

Графік (рис.4.1) показує: чим більше в команді розробки ПЗ фахівців, тим нижче продуктивність кожного з них. Якщо вірити закону Філіпа, то створення прикладного коду об'ємом в 1 Гбайт за допомогою традиційної техніки зажадало б велетенських трудовитрат.

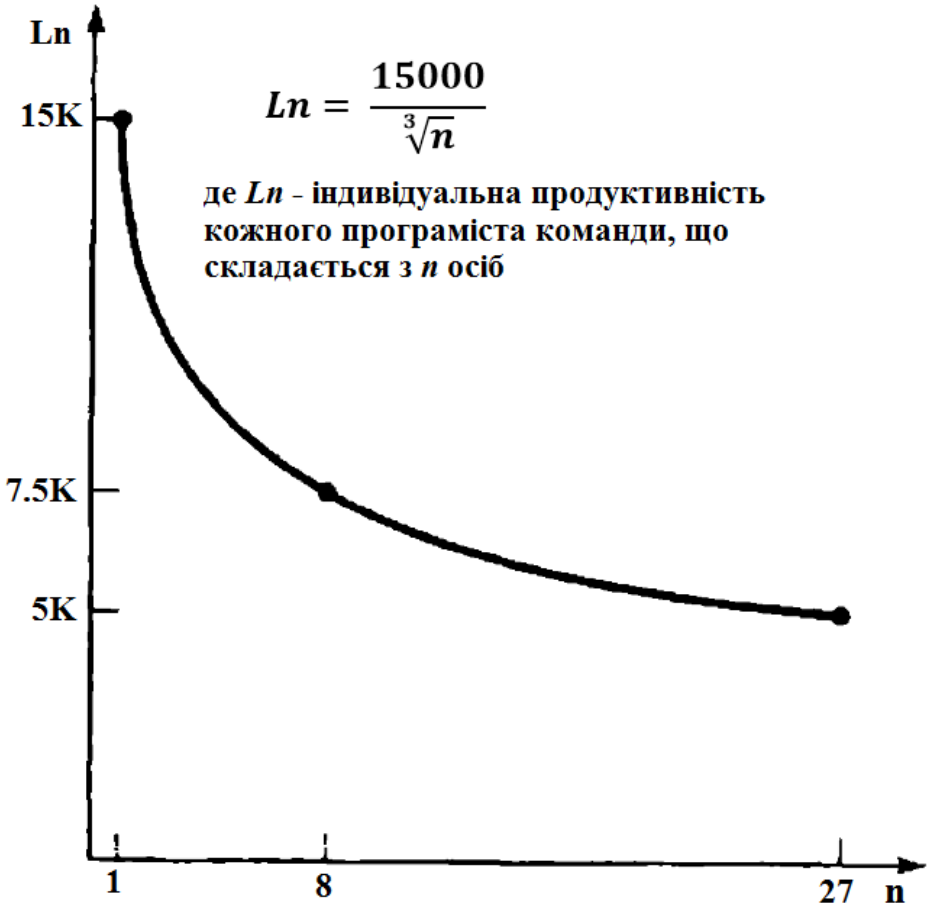


Рисунок 4.1 – Закон Філіпа

Звідси витікає, що традиційна техніка розробки ПО, залежна від того, як люди набирають код, поступово стає неефективною. Для побудови складних, великих додатків майбутнього за допомогою сьогоднішніх засобів просто не вистачить ресурсів. Таким чином, технікові розробки ПЗ необхідно змінити докорінно. Тому багато фахівців в області розробки ПЗ вважають, що об'єктно-орієнтовані методи і є частковим рішенням цієї задачі.

Для великих команд альтернативою також є їх розділення на групи. Кожна група відповідає за певну частину проекту і працює над нею. Зазвичай чисельність групи не перевищує восьми чоловік. У таких групах проблеми взаємодії зникають. Для взаємодії з іншими групами в кожній групі виділяється один співробітник. Така структура зберігає переваги невеликих команд, але дозволяє великій кількості людей створювати великі програмні продукти.

2. Ієрархія команди. Співробітники в команді з горизонтальною організацією (один рівень, усі співробітники рівні) легше спілкуються між собою, чим в командах з багаторівневою організацією і ієрархією стосунків (начальники/підлеглі). У останніх командах взаємодія відбувається між рівнями, в ієрархічній послідовності.

4.3.3. Склад групи

Не існує універсального рецепту для визначення оптимального складу групи розробників. Цей склад залежить від великого числа різноманітних чинників: стилю менеджменту, прийнятого в організації, предметної області і розміру проекту, професійних можливостей співробітників організації і т. д. Перерахуємо типові ролі:

1. Аналітик – відповідає за розвиток і інтерпретацію вимог замовника; має бути експертом в предметній області, але працювати в тісному контакті з іншими співробітниками.

2. Архітектор – впередсмотрящий; відповідає за проектування і розвиток архітектури продукту, є одним з найбільш кваліфікованих фахівців, що мають досвід ухвалення стратегічних рішень; окрім досвіду проектування, архітектор повинен уміти програмувати, оскільки його рішення утілюються в програмному коді.

3. Конструктор компонентів – головний творець компонентів (будівельної цегли, з якої компонується продукт).

4. Фахівець з інтеграції – відповідає за зборку сумісних версій компонентів і перевірку правильності їх спільної роботи, підтримує випуск версій продукту.

5. Фахівець з документації – документує усі реалізовані рішення, готує документацію для користувача.

6. Системний програміст – відповідає за створення і адаптацію програмних утиліт, що полегшують розробку в проекті.

7. Системний адміністратор – управляє фізичними комп’ютерними ресурсами в проекті.

Зрозуміло, не кожен проект вимагає виконання усіх цих ролей. У невеликих проектах співробітники можуть грати відразу декілька ролей.

4.4. Процес розробки

Найбурхливіша дискусія про процеси розробки розгортається навколо вибору між ітеративною і водоспадною моделями.

При організації роботи в стилі **водоспаду** проект ділиться на підставі виду робіт. Щоб створити програмне забезпечення, необхідно зробити певні дії: проаналізувати вимоги, створити проект, виконати кодування і тестування. Такий річний проект може включати двомісячну фазу аналізу, за якою йде чотиримісячна фаза дизайну (моделювання), а потім тримісячна фаза кодування і, нарешті, тримісячна фаза тестування.

Ітеративний стиль ділить проект за принципом функціональності продукту. Можна узяти рік і розділити його на тримісячні ітерації. У першій ітерації береться чверть вимог і виконується повний цикл розробки програмного забезпечення для цієї чверті: аналіз, дизайн, кодування і тестування. До кінця першої ітерації у вас є система, що має чверть необхідної функціональності. Потім ви приступаєте до другої ітерації і через шість місяців отримуєте систему, що робить половину того, що їй покладене.

При розробці способом водоспаду після кожного етапу зазвичай в якому-небудь виді виконується формальна здача, але має і місце повернення назад. В процесі кодування можуть з'ясуватися обставини, що змушують знову повернутися до етапів аналізу і дизайну. Звичайно, на початку кодування не слід думати, що аналіз завершений. І рішення, прийняті на стадії аналізу і дизайну, неминуче переглядатимуться пізніше. Проте ці зворотні потоки є виключеннями і мають бути по можливості зведені до мінімуму.

При ітеративному процесі розробки перед початком реальної ітерації зазвичай спостерігається деяка дослідницька активність. Як мінімум на вимоги буде кинутий поверхневий погляд, достатній, принаймні, для розділення вимог на ітерації для подальшого виконання. В процесі такого дослідження можуть бути прийняті деякі рішення по дизайну самого вищого рівня.

З іншого боку, незважаючи на те що в результаті кожної ітерації повинне з'явитися інтегроване програмне забезпечення, готове до постачання, часто буває, що воно ще не готове і потрібний деякий стабілізаційний період для виправлення останніх помилок.

Звичайно, можна не передавати систему на реалізацію у кінці кожної ітерації, але вона повинна знаходитися в стані виробничої готовності. Проте ітеративний процес найчастіше передбачає передачу підсистеми в реалізацію, оскільки можна на кожній ітерації оцінити працездатність системи і отримати якіснішу зворотну реакцію. У цій ситуації часто говорять про проект, що має декілька версій, кожна з яких ділиться на декілька ітерацій.

Можливий і змішаний підхід (так званий життєвий цикл **поетапної доставки** [33]), відповідно до якого спочатку виконуються аналіз і проектування верхнього рівня в стилі водоспаду, а потім кодування і тестування, розділені на ітерації.

Більшість авторів публікацій по процесу створення програмного забезпечення, що особливо належить до об'єктно-орієнтованого співтовариства, останні п'ять років випробовує неприязнь до підходу в стилі водоспаду. З усієї безлічі причин цього явища найголовніша полягає в тому, що при використанні методу водоспаду дуже важко стверджувати, що розробка якогось проекту дійсно йде у вірному напрямі.

Занадто легко оголосити перемогу на ранньому етапі і приховати помилки планування. Зазвичай єдиний спосіб, яким ви дійсно можете показати, що слідуєте по такому шляху, полягає в тому, щоб отримати протестоване, інтегроване програмне забезпечення. У разі ітеративного процесу це повторюється багаторазово, і в результаті, якщо щось йде не так, як потрібно, ви своєчасно отримуєте відповідний сигнал.

Саме тільки з цієї причини рекомендується уникати методу водоспаду в чистому вигляді. Принаймні, необхідно застосовувати поетапну доставку, якщо неможливо використати ітеративний метод в повному об'ємі.

Особливо важливо, щоб при ітеративному процесі кожна ітерація завершувалася створенням протестованого, інтегрованого програмного продукту, який би мав якість, як можна ближчу до якості серійної продукції.

Загальним прийомом при ітеративній розробці є упаковка за часом. Таким чином, ітерація займатиме фіксований проміжок часу. Якщо виявилося, що ви не в змозі виконати усе, що планували зробити за час ітерації, то необхідно викинути деяку функціональність з цієї ітерації, але не слід змінювати дату виконання. У більшості проектів, заснованих на ітеративному процесі, протяжність ітерацій однакова, і це дозволяє вести розробку в постійному ритмі.

Одним з найбільш загальних аспектів ітеративної розробки є питання переробки (рефакторинга). Ітеративна розробка недвозначно припускає, що ви перероблятимете і видалятимете існуючий код на останній ітерації проекту. У багатьох областях людської діяльності, наприклад в промисловому виробництві, переробка вважається збитком. Але створення програмного забезпечення не схоже на промислове виробництво – часто буває вигідніший переробити існуючий код, чим латати код невдало спроектованої програми.

Усі ці технічні прийоми пропагувалися нещодавно в книзі «Extreme Programming» [5], хоча вони застосовувалися і раніше і могли (і повинні були) застосовуватися, незалежно від того, чи використовувався XP (eXtreme Programming) або який-небудь інший гнучкий процес.

4.5. Прогнозуюче і адаптивне планування

Прогнозуючий підхід спрямований на виконання роботи на початковому етапі проекту, для того щоб краще зрозуміти, що треба робити надалі. Таким чином, настає момент, коли частину проекту, що залишилася, можна оцінити з достатньою мірою точності. В процесі **прогнозуючого планування** проект розділяється на дві стадії. На першій стадії складаються плани, і тут передбачати важко, але друга стадія більше передбачувана, оскільки плани вже готові.

Проте все ще йдуть гострі дискусії про те, чи багато проектів можуть бути передбачуваними. Суть цього питання полягає в аналізі вимог. Одна з найістотніших причин складності програмних проектів полягає в трудності розуміння вимог до програмних систем.

Більшість програмних проектів піддаються істотному **перегляду вимог**: зміні вимог на пізній стадії виконання проекту. Наслідки перегляду можна запобігти, заморозивши вимоги на ранній стадії проекту і не дозволяючи змінам з'являтися, але це призводить до ризику поставити клієнтові систему, яка більше не задовольняє вимогам користувачів.

Прибічники іншої школи стверджують, що перегляд вимог неминучий, що у багатьох проектах важко стабілізувати вимоги в такому ступені, щоб була

можливість використати прогнозуюче планування. Це може бути або слідством того, що виключно важко уявити, що може робити програмний продукт, або слідством того, що умови ринку диктують непередбачувані зміни. Ця школа підтримує адаптивне **планування** відповідно до твердження, що прогнозованість – це ілюзія.

Відмінність між прогнозуючими і адаптивними проектами проявляється різними шляхами, коли люди говорять про стан проекту. Коли стверджується, що виконання проекту йде добре, оскільки робота ведеться відповідно до плану, то мається на увазі метод прогнозування.

При адаптивній розробці не можна сказати «відповідно до плану», оскільки план увесь час міняється. Це не означає, що адаптивні проекти не плануються; зазвичай планування займає значний час, але план трактується як основна лінія проведення послідовних змін, а не як пророцтво майбутнього.

На основі прогнозуючого плану можна розробити контракт з фіксованою функціональністю за фіксованою ціною. У такому контракті точно вказується, що повинно бути створено, скільки це коштує і коли продукт буде поставлений.

У адаптивному плані така фіксація неможлива. Ви можете позначити бюджет і терміни постачання, але ви не можете точно зафіксувати функціональність продукту, що поставляється. Адаптивний контракт припускає, що користувачі співпрацюватимуть з командою розробників, щоб регулярно переглядати необхідну функціональність і переривати проект, якщо прогрес занадто незначний. Як такий процес адаптивного планування може визначати проект зі змінними межами функціональності за фіксованою ціною.

Природно, адаптивний підхід менш бажаний, оскільки усі віддають перевагу великій передбачуваності програмних проектів. Проте передбачуваність залежить від точності, коректності і стабільності безлічі вимог. Звідси витікають дві важливі поради.

1. Не складайте прогнозуючий план до тих пір, поки не отримаєте точні і коректні вимоги і не будете упевнені, що вони не піддадуться істотним змінам.
2. Якщо ви не можете отримати точні, коректні і стабільні вимоги, то використайте метод адаптивного планування.

4.6. Вибір процесу розробки

За останні десять років виріс інтерес до гнучких процесів розробки програмного забезпечення. Гнучкий (agile) – це широкий термін, що охоплює велику кількість процесів, що мають загальну безліч величин і понять, визначених Маніфестом гнучкої розробки програмного забезпечення (Manifesto of Agile Software Development, http://agileManifesto.org). Прикладами таких процесів є XP (Extreme Programming – екстремальне програмування), Scrum (сутичка, зіткнення), FDD (Feature Driven Development – розробка, керована можливостями) і DSDM (Dynamic Systems Development Method – метод розробки динамічних систем).

Гнучкі процеси виключно адаптивні за своєю природою. Вони також мають чітку орієнтацію на людину. Гнучкі підходи припускають, що найбільш важливим чинником успішного завершення проекту є кваліфікація виконавців і їх хороша спільна робота з людської точки зору. Значущість процесів або інструментів, ними використовуваних, безперечно стоїть на другому місці.

Гнучкі методи в основному спрямовані на використання коротких, обмежених за часом ітерацій, що найчастіше закінчуються через місяць або раніше. Оскільки їх вклад в документацію невеликий, то в гнучкому підході не передбачається застосування UML в режимі проектування. Частіше усього UML використовується в режимі эскизирования і рідше в якості мови програмування.

У більшості своїй гнучкі процеси не занадто **формалізовані.** Сильно формалізовані або ваговиті процеси мають багато документації і постійний контроль під час виконання проекту. Гнучкий підхід припускає, що формалізм заважає проведенню змін і суперечить природі талановитих осіб. Тому гнучкі процеси часто називають **полегшеними** (lightweight). Важливо розуміти, що недостатня формализованность є наслідком адаптивності і орієнтації фахівців, а не фундаментальною властивістю.

Один з прибічників ітеративного процесу розробки (Мартін Фаулер) говорить у своїй книзі [36]: «Застосовуйте ітеративний метод розробки тільки в проектах, яким ви бажаєте успіху».

При грамотному застосуванні ітеративної розробки, вона є дуже важливим методом, здатним допомогти в ранньому виявленні можливих ризиків і в поліпшенні керованості процесом розробки. Проте це не означає, що можна зовсім обійтися без керівництва проектом («хоча, якщо бути справедливим, я повинен відмітити, що деякі використовують її саме для цієї мети» [36]). Ітеративна розробка вимагає ретельного планування. Але це дуже надійний підхід, і тому будь-яка книга з об’єктно-орієнтованої розробки рекомендує його застосовувати – і небезпідставно.

Хоча уніфікований процес, розроблений компанією Rational (Rational Unified Process, RUP), не залежить від UML, їх часто згадують разом. Тому буде доречно сказати тут про цей процес ще декілька слів.

У разі застосування RUP в першу чергу необхідно вибрати шаблон розробки (development case) – процес, який ви збираєтеся використати в проекті. Шаблони розробки можуть дуже значно варіюватися. При виборі шаблону розробки відразу потрібно людину, добре знайому з RUP, – той, хто зможе пристосувати RUP до певних вимог проекту. В якості альтернативи існує набір розподілених по пакетах шаблонів, що постійно збільшується, розробки, з яких можна почати.

Незалежно від шаблону розробки RUP по суті також є ітеративним процесом. Метод водоспаду не сумісний з філософією RUP, хоча можна відмітити, що проекти, в яких застосовуються процеси в стилі водоспаду, нерідко обряджають в одяг RUP.

Іноді RUP називають просто уніфікованим процесом (Unified Process, UP). Так зазвичай поступають організації, які хочуть застосувати термінологію

і загальний підхід RUP, але не хочуть користуватися ліцензійними продуктами фірми Rational Software. Можна думати про RUP як про продукт фірми Rational, заснований на UP, а можна рахувати RUP і UP одним і тим же [36].

Одна з великих переваг ітеративної розробки полягає в можливості часто удосконалювати процес. У кінці кожної ітерації слід проводити її ретроспективний аналіз, збираючи команду на наради, щоб розглянути, як йдуть справи і що можна поліпшити. Якщо ітерація коротка, то досить двох годин. У кінці розробки проекту або його основної версії можна провести формальніший ретроспективний аналіз проекту, який може зайняти пару днів; детальнішу інформацію можна знайти на http://www.retrospectives.com/.

4.7. Налаштування UML під процес

При розгляді графічних мов моделювання зазвичай про них думають в контексті водоспадного процесу. Водоспадний процес, як правило, супроводжується документами, що виступають прес-релізами між стадіями аналізу, дизайну і кодування. Часто графічні моделі можуть займати основну частину цих документів. Дійсно, багато хто із структурних методів 70-х і 80-х років часто говорить про моделі аналізу і дизайну, подібних до цієї [36].

Незалежно від того, застосовуєте ви метод водоспаду або ні, так або інакше ви проводите аналіз, дизайн, кодування і тестування. Використання UML не має на увазі обов’язкову розробку документів або завантаження складних CASE-систем.

Аналіз вимог. В процесі аналізу вимог необхідно зрозуміти, що клієнти і користувачі програмного забезпечення чекають від системи. У вашому розпорядженні є безліч прийомів UML:

1. Прецеденти, які описують, як люди взаємодіють з системою.
2. Діаграма класів, яка будується з точки зору концептуальної перспективи і може служити хорошим інструментом для побудови точного словника предметної області.
3. Діаграма діяльності, яка показує робочий потік організації, способи взаємодії програмного забезпечення і користувачів. Діаграма діяльності може показати контекст для використання прецедентів, а також деталі роботи складних прецедентів.
4. Діаграма станів, яка може виявитися корисною, якщо концепція має своєрідний життєвий цикл з різними станами і подіями, які змінюють ці стани.

Аналізуючи стани, пам’ятайте, що найважливіше – ця взаємодія з вашими користувачами і клієнтами. Звичайно це непрограмісти, і вони не знайомі з UML і іншими подібними технологіями.

Будьте готові у будь-який момент відійти від правил UML, якщо це допоможе поліпшити взаєморозуміння. Найбільший ризик у разі застосування UML для аналізу полягає в тому, що ви будуєте діаграми, не зовсім зрозумілі фахівцям в конкретній предметній області. Така діаграма гірша, ніж даремна; вона лише здатна вселити в розробників неправдиве почуття упевненості.

Проектування. При розробці моделі ви можете широко застосовувати діаграми. Можна використати більше нотацій і при цьому бути точнішим. Ось деякі корисні прийоми:

1. Діаграми класів з точки зору програмного забезпечення. Вони показують класи програми і їх взаємозв'язок.
2. Діаграми послідовності для загальних сценаріїв. Правильний підхід полягає у витяганні найбільш важливих і цікавих сценаріїв з прецеденту, а також у використанні і діаграм послідовності з метою зрозуміти, що відбувається в програмі.
3. Діаграми пакетів, що показують високорівневу організацію програмного продукту.
4. Діаграми станів для класів із складним життєвим циклом.
5. Діаграми розгортання, що показують фізичну конфігурацію програмного забезпечення.

Створення моделей зазвичай відбувається на ранній стадії ітерації і може бути зроблене по частинах для різних розділів функціональності, призначеної для цієї ітерації. Крім того, ітерація має на увазі зміну існуючої моделі, а не побудову кожного разу нової моделі.

Застосування UML в режимі эскизирования – рухливіший процес. Один з підходів полягає у виділенні пари днів на початку ітерації на створення ескізного дизайну для цієї ітерації. Можна також проводити короткі сесії проектування у будь-який момент в ході ітерації, влаштовуючи короткі півгодинні наради всякий раз, коли розробники починають сперечатися з приводу нетривіальної функції.

У режимі проектування очікується, що програмна реалізація будуватиметься відповідно до діаграм. Зміну моделі слід вважати відхиленням, яке вимагає, щоб проектувальники, що створили цю модель, її переглянули. Ескіз зазвичай трактується як перший зріз дизайну. Якщо в ході кодування виявляється, що ескіз не зовсім точний, то розробники повинні мати можливість змінити дизайн. Розробники, впроваджувальні дизайн, повинні самі вирішувати, чи варто влаштовувати широку дискусію, щоб зрозуміти усі можливі варіанти.

4.8. Управління конфігурацією

Управління конфігурацією – це координація різних версій і частин документації і програмного коду. Управління конфігурацією ПЗ – захисна діяльність, вживана на усіх етапах життєвого циклу ПЗ. Вона забезпечує управління змінами в ПЗ, яке включає наступні дії:

1. Ідентифікація зміни.
2. Контроль зміни.
3. Гарантія правильної реалізації зміни.
4. Формування повідомлення про зміни.

Управління конфігурацією стартує з початком програмного проекту і закінчується з припиненням використання ПЗ.

За час життя програмний код продукту зазнає зміни двох категорій: додавання нових частин, підключення нових версій існуючих частин. Звичайно, слід враховувати обидві категорії змін.

Інформацію на виході процесу розробки ПЗ можна розділити на три категорії:

1. Комп'ютерні програми (у вигляді виконуваних кодів).
2. Документи, що описують програми (як для технічного персоналу, так і для користувачів).
3. Структури даних (як зовнішні, так і внутрішні).

Сукупність усіх елементів інформації, що виробляються як частину процесу розробки ПО, називають конфігурацією ПО.

З розвитком процесу розробки ПЗ кількість елементів конфігурації нестримно росте.

Мінімальна конфігурація ПЗ включає наступні базові елементи:

1. Системна специфікація.
2. План програмного проекту.
3. Специфікація вимог до ПЗ.Працюючий або паперовий макет.
4. Попередній посібник користувача.
5. Специфікація проектування.
6. Лістинги початкових текстів програм.
7. План і методика тестування. Тестові варіанти і отримані результати.
8. Керівництво по роботі і інсталяції.
9. Виконуваний код програм.
10. Опис бази даних.
11. Посібник користувача по налаштуванню.
12. Документи супроводу. Звіти про проблеми ПЗ.Запити супроводу. Звіти про зміни.
13. Стандарти і методики розробки ПО.

Управління конфігурацією полягає в застосуванні дій для управління змінами впродовж усього життєвого циклу ПО.

Зміна – неодмінний факт життєвого циклу ПЗ.Замовники хочуть змінити вимоги. Розробники хочуть модифікувати технічний підхід.

Контрольні питання до розділу 4

1. Дайте визначення поняттю *об'єкту*.
2. Перерахуйте командні ролі проекту.
3. Перерахуйте обов'язки керівника проекту.
4. Дайте визначення поняттю *адаптивні вимоги і адаптивний замовник*.
5. Що таке *ітеративний стиль* розробки ПО?
6. Що таке *прогнозуюче і адаптивне планування*?
7. У чому суть уніфікованого процесу RUP?
8. Дайте визначення поняттю *управління конфігурацією*.

ЧАСТИНА 2. ІНСТРУМЕНТАЛЬНІ ЗАСОБИ ПРОЕКТУВАННЯ ІС

РОЗДІЛ 5. ОСНОВНІ ВІДОМОСТІ ПРО МОВУ UML

Трудомісткість створення сучасних додатків на початкових етапах проекту, як правило, оцінюється значно нижче зусиль, що реально витрачаються, що служить причиною незапланованих витрат і затягування остаточних термінів готовності програм. У процесі розробки додатків змінюються *функціональні вимоги* замовника, що ще більше віддаляє момент закінчення роботи програмістів. Збільшення розмірів програм змушує залучати понадштатних програмістів, що, у свою чергу, вимагає додаткових ресурсів для організації їх погодженої роботи. У розробці і впровадженні сучасних корпоративних інформаційних систем бере участь багато фахівців різної кваліфікації, для яких однакове розуміння архітектури і функціональності є серйозною проблемою.

Таким чином, усі ці особливості призводять до наполегливої необхідності моделювання структури і процесу функціонування програмних систем до початку написання відповідного коду. При цьому неодмінною умовою успішного завершення проекту стає побудова попередньої *моделі* програмної системи.

Модель **–** абстракція фізичної системи, яка створюється з метою досягнення чого-небудь перед тим, як воно буде створено, і представлена на деякій мові або в графічній формі.

Моделі бувають різні – матеріальні і нематеріальні, штучні і природні, декоративні і математичні.

Наведемо декілька прикладів. Знайомі усім нам пластмасові іграшкові автомобільчики – це не що інше, як *матеріальна штучна декоративна* модель реального автомобіля. Звичайно, в такому «авто» немає двигуна, ми не заповнюємо його бак бензином, але як модель ця іграшка свої функції цілком виконує: вона дає дитині уявлення про автомобіль, оскільки відображає його характерні риси – наявність чотирьох коліс, кузова, дверей, вікон, здатність їхати тощо.

В ході медичних досліджень досліди на тваринах часто передують клінічним випробуванням медичних препаратів на людях. У такому разі тварина виступає в ролі *матеріальної природної* моделі людини.

Рівняння

$$mg - a\frac{dx}{dt} = m\frac{d^2x}{dt^2}$$

– це теж модель, але це модель математична, і описує вона рух матеріальної точки під дією сили тяжіння.

Оскільки модель не містить несуттєвих деталей, працювати з нею виявляється простіше, ніж з модельованою суттю. Абстрагування – це одне з найважливіших людських умінь, яке дає нам можливість працювати із складними речами.

Інженери, художники і ремісники тисячоліттями займалися створенням моделей, на яких вони випробовували свої плани перед тим, як утілювати їх в життя. Проектування апаратних і програмних систем не є виключенням. Щоб створити складну систему, розробник повинен абстрагувати різні її представлення, побудувати моделі, використовуючи чітку систему позначень, перевірити, чи моделі задовольняють вимогам, пред'явленим до системи, і поступово додавати деталі, перетворюючи моделі на реалізацію.

Основна вимога до моделі програмної системи – вона має бути зрозуміла замовникові і усім фахівцям проектної групи, включаючи бізнес-аналітиків і програмістів, оскільки моделі програмних систем можуть відображати різні аспекти системи. Наприклад, структурна модель відображає статичну організацію системи, а модель поведінки підкреслює динамічні процеси, властиві системі.

Саме для розробки такої нотації знадобилися зусилля групи фахівців провідних фірм виробників програмного і апаратного забезпечення, які привели до появи мови UML. Розробка і використання *моделей мови UML здійснюється у рамках загальної концепції об'єктно-орієнтованого аналізу і проектування,* яка, у свою чергу, є узагальненням методології *об'єктно-орієнтованого програмування* [11; 19; 20; 26].

5.1. Поняття UML

Предмет, що обговорюється, позначається ідентифікатором UML, який є абревіатурою повної назви *Unified Modeling Language.* Правильний переклад цієї назви українською мовою – ***уніфікована мова моделювання***. Таким чином, предмет, що обговорюється, характеризується трьома словами, кожне з яких є точним терміном.

5.1.1. UML – це мова

Головним словом в цьому поєднанні є слово «мова».

Мова – це знакова система для зберігання і передачі інформації.

Розрізняються мови *формальні,* правила вживання яких строго і явно визначені, і *неформальні,* вживання яких засноване на практиці, що склалася. Розрізняються також мови *природні,* що з'являються як би самі собою в результаті неперсоніфікованих (тобто загальних) зусиль маси людей, і мови *штучні*, що є плодом видимих зусиль певних осіб.

Мова, на якій написаний цей розділ, є неформальною і природною. З іншого боку, переважна більшість мов програмування є формальними і штучними. Зустрічаються і інші комбінації: наприклад, мова формул алгебри вважається формальною і природною, а есперанто – неформальною і штучною.

Так от, UML можна охарактеризувати як **формальну штучну мову**, хоча і не в такому ступені, як багато поширених мов програмування. Ознакою штучності служить наявність трьох загальновизнаних авторів – Гради Буча, Айвара Якобсона і Джеймса Рамбо [11].

Для опису формальних штучних мов (зокрема, для опису мов програмування) придумано і використовується безліч різних способів. Проте на практиці склалася загальноприйнята структура таких описів. Вважається, що формальна штучна мова описана належним чином, якщо цей опис містить, щонайменше, такі частини:

- ***синтаксис***, тобто визначення правил складання конструкцій мови;
- ***семантика***, тобто визначення правил приписування сенсу конструкціям мови;
- ***прагматика***, тобто визначення правил використання конструкцій мови для досягнення певної мети.

Як формальна штучна мова UML має синтаксис, семантику і прагматику, хоча ці частини названі в деяких випадках інакше і описані інакше, ніж це прийнято в текстових мовах програмування. По-перше, оскільки **UML мова графічна, а не текстова,** а по-друге, оскільки **UML мова моделювання, а не програмування**.

5.1.2. UML – це мова моделювання

Слово «моделювання», що входить в назву UML, має багато змістових відтінків і способів вживання, що склалися. Зокрема, англійські слова modeling і simulation перекладаються одним словом «моделювання», хоча означають різні речі.

У першому випадку йдеться про складання моделі, яка використовується тільки для опису об’єкту або явища, що моделюється. У другому випадку мається на увазі складання моделі, яка може бути використана для отримання істотної інформації про модельований об’єкт або явище. При цьому в другому випадку зазвичай додається уточнювальний прикметник: *чисельне моделювання, математичне моделювання та ін.* UML є мовою моделювання в першому сенсі.

Стосовно розробки програмного забезпечення так склалося, що результати фаз аналізу і проектування, оформлені засобами певної мови, прийнято називати ***моделлю***. *Діяльність зі складання моделей природно назвати* ***моделюванням***.

Таким чином, ***модель UML* – це, передусім, опис об’єкту або явища**, а також і дещо інше, а саме усе, що авторам UML вдалося включити в мову, не порушуючи принципу уніфікації, до викладу якого ми перейдемо в наступному розділі.

5.1.3. UML – це уніфікована мова моделювання

Поштовхом до зміни ситуації послужили такі обставини. По-перше, масове поширення отримав *об’єктно-орієнтований підхід* до розробки програмних систем, внаслідок чого виникла потреба у відповідних засобах. Іншими словами, появи чогось подібного на UML з нетерпінням чекали практики. По-друге, три видатні фахівці в цій області, автори найпопулярніших

методів, наважилися об’єднати зусилля саме з метою уніфікації своїх (і не лише своїх) розробок відповідно до соціального замовлення.

Доклавши заслуговуючі поваги зусилля, автори UML за підтримки і сприяння усієї міжнародної програмістської громадськості змогли звести воєдино (**уніфікувати**) велику частину того, що було відомо і до них. В результаті уніфікації вийшла теоретично витончена і практично корисна річ – UML.

Але уніфікація UML носить не лише історичний характер. UML докладає зусиль (і в основному успішно) в уніфікації декількох різних областей: UML надає візуальний синтаксис для моделювання упродовж усього життєвого циклу розробки програмного забезпечення – від постановки вимог до реалізації; UML використовується для моделювання усіх аспектів – від апаратних вбудованих систем реального часу до систем підтримки ухвалення рішень; UML є незалежною від мов і платформ.

5.1. 4. Історія UML

Існує багато технологій і інструментальних засобів, за допомогою яких можна реалізувати оптимальний проект ІС, починаючи з етапу аналізу і закінчуючи створенням програмного коду системи.

У більшості випадків ці технології пред’являють дуже жорсткі вимоги до процесу розробки і використовуваних ресурсів, а спроби трансформувати їх під конкретні проекти виявляються безуспішними. Ці технології представлені CASE-засобами верхнього рівня або CASE-засобами повного життєвого циклу (upper CASE tools або full life – cycle CASE tools). Вони не дозволяють оптимізувати діяльність на рівні окремих елементів проекту, і, як наслідок, багато розробників перейшли на так звані CASE-засоби нижнього рівня (lower CASE tools). Проте вони зіткнулися з новою проблемою – проблемою організації взаємодії між різними командами, що реалізовують проект.

Уніфікована мова об’єктно-орієнтованого моделювання Unified Modeling Language (UML) стала засобом досягнення компромісу між цими підходами. Існує достатня кількість інструментальних засобів, що підтримують за допомогою UML життєвий цикл інформаційних систем, і, одночасно, UML є досить гнучкою для налаштування і підтримки специфіки діяльності різних команд розробників.

Потужний поштовх до розробки цього напряму інформаційних технологій дало поширення *об’єктно-орієнтованих* мов програмування у кінці 1980-х – на початку 1990-х років. Користувачам хотілося отримати єдину мову моделювання, яка об’єднала б у собі всю потужність об’єктно-орієнтованого підходу і давала б чітку модель системи, що відбиває усі її суттєві сторони.

До середини дев’яностих явними лідерами в цій області стали методи Booch (Гради Буч, Grady Booch), OMT – 2 (Джим Рамбо, Jim Rumbaugh), OOSE – Object – Oriented Software Engineering (Айвар Якобсон, Ivar Jacobson). Проте ці три методи мали свої сильні і слабкі сторони: OOSE був кращим на стадії аналізу проблемної області і аналізу вимог до системи, OMT – 2 був

найприйнятнішим на стадіях аналізу і розробки інформаційних систем, Booch краще всього підходив для стадій дизайну і розробки. Усе йшло до створення єдиної мови, яка об'єднувала б сильні сторони відомих методів і забезпечувала найкращу підтримку моделювання. Такою мовою виявилася *UML*.

Створення UML почалося в жовтні 1994 р., коли Джим Рамбо і Гради Буч з Rational Software Corporation стали працювати над об'єднанням своїх методів OMT і Booch [11]. Восени 1995 р. побачила світ перша чорнова версія об'єднаної методології, яку вони назвали Unified Method 0.8. Після приєднання у кінці 1995 р. до Rational Software Corporation Айвара Якобсона і його фірми Objectory, зусилля трьох творців найпоширеніших *об'єктно-орієнтованих* методологій були об'єднані і спрямовані на створення UML.

Історія розвитку UML зовсім не завершена – мова постійно удосконалюється, збагачується і розширюється. На рис. 5.1 наведена картинка, що ілюструє історію розвитку UML.

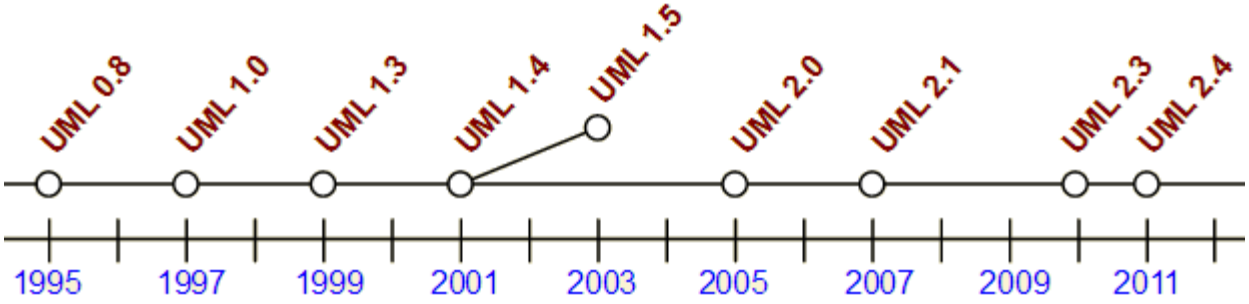


Рисунок 5.1 – Історія розвитку UML

Нині консорціум користувачів *UML* Partners включає представників таких грандів інформаційних технологій, як Rational Software, Microsoft, IBM, Hewlett – Packard, Oracle, DEC, Unisys, IntelliCorp, Platinum Technology.

Мова UML прийнята на озброєння практично усіма найбільшими компаніями – виробниками ПЗ (Microsoft, IBM, Hewlett – Packard, Oracle, Sybase та ін.). Крім того, практично усі світові виробники CASE-засобів, окрім Rational Software (Rational Rose), підтримують UML у своїх продуктах.

5.2. Призначення UML

Основна ідея UML – можливість моделювати програмне забезпечення і інші системи як набори взаємодіючих об'єктів. Це, звичайно ж, чудово підходить для ОО програмних систем і мов програмування, але також дуже добре працює і для бізнес-процесів і інших прикладних завдань.

Таким чином, UML призначена для моделювання. Самі автори UML визначають своє дітище таким чином.

Мова UML – це графічна мова моделювання загального призначення, призначена для специфікації, візуалізації, проектування і документування усіх артефактів, що створюються при розробці програмних систем. Велике значення в цьому визначенні дається вибору ключових слів і порядку, в якому вони перераховані.

5.2.1. Специфікація і візуалізація

У типових випадках у процесі розробки додатків беруть участь як мінімум дві дійові особи: ***замовник*** (конкретна людина або група осіб, або організація) і ***розробник*** (це може бути програміст-одинак, тимчасова команда проекту або ціла організація, що спеціалізується на розробці програмного забезпечення). Через те, що дійових осіб двоє, дуже багато що залежить від міри їх взаєморозуміння.

Одним з ключових етапів розробки додатка є визначення того, яким вимогам повинен задовольняти додаток, що розробляється. У результаті цього етапу з'являється формальний або неформальний документ (артефакт), який називають по-різному, маючи на увазі приблизно одне і те ж: *постановка завдання, вимоги, технічне завдання, зовнішні специфікації*.

Аналогічні за призначенням, але, можливо, відмінні за формою і змістом артефакти з'являються і на інших етапах розробки, особливо якщо в розробку включено багато дійових осіб. Для них також використовуються різні назви: *функціональні специфікації, архітектура додатка та ін.* Усі такі артефакти називатимемо *специфікаціями*.

Специфікація **–** це декларативний опис того, як щось влаштоване або працює. У нашому випадку специфікація – детальний опис системи, який повністю визначає її мету і функціональні можливості.

Необхідно брати до уваги три тлумачення специфікацій:

- те, яке має на увазі дійову особу, що є джерелом специфікації (наприклад, замовник);
- те, яке має на увазі дійову особу, що є споживачем специфікації (наприклад, розробник);
- те, яке об'єктивно обумовлене природою об'єкту, що специфікується.

Ці три трактування специфікацій можуть не співпадати, і, на жаль, як показує практика, часто не співпадають, причому значно.

Замовник може не усвідомлювати своїх об'єктивних потреб, або невірно їх інтерпретувати, або помилятися стосовно природи своїх утруднень. Розробник може не розбиратися в предметній області замовника і інтерпретувати формулювання специфікацій абсолютно іншим чином. Якщо ж у формулюванні специфікацій бере участь розробник, то зловживання технічною термінологією може абсолютно дезорієнтувати замовника.

Не слід також забувати, що замовник і розробник мають, як правило, абсолютно різне розуміння сенсу цього артефакту. Адже окрім цього є ще аналітики, менеджери, бізнес-консультанти. Кожен з них називає специфікації по-своєму: *вимоги користувача, постановка завдання, технічне завдання, функціональна специфікація, архітектура системи*. Причому усі ці люди, будучи фахівцями в абсолютно різних предметних областях, говорять кожний своєю мовою і часто просто не розуміють один одного. Ось через те і виникає проблема, яку може вирішити тільки наявність єдиного, уніфікованого засобу створення специфікацій, досить простого і зрозумілого для усіх зацікавлених осіб.

Розрізняють специфікації трьох видів: словесні специфікації на природній мові, модельні специфікації і формальні специфікації.

Словесні специфікації на природній мові якраз і викликають масу проблем, оскільки створюються різними фахівцями на «їх мові».

Іншим видом специфікацій є *формальні специфікації.* Зрозуміло, що формальна специфікація є, по суті, математичною моделлю завдання і тому для обчислювальних завдань усе виглядає досить просто. Формалізація ж завдань з інших галузей знань може виявитися складнішою і трудомісткою проблемою.

Коли ми говоримо про те, що UML – цей засіб ***візуалізації,*** то ми маємо на увазі *модельні специфікації.* Усі ми знаємо, що вивчення чогось нового йде набагато простіше, якщо документ містить не лише текст, а ще і ілюстрації до нього.

Так от, такі картинки наочні і інтуїтивно зрозумілі, причому майже однозначно розуміються будь-якими зацікавленими особами, так що можуть використовуватися як засіб спілкування між людьми. UML дозволяє створювати такі прості і зрозумілі картинки (моделі), що описують систему з різних сторін, які можна показати замовникові і обговорити з ним, тобто такі моделі служить засобом комунікації в команді.

Таким чином, основне призначення **UML –** надати, з одного боку, досить формальний, з іншого боку, досить зручний, і, з третього боку, **досить універсальний засіб,** що дозволяє деякою мірою понизити ризик розбіжностей в тлумаченні специфікацій.

5.2.2. Проектування і документування

У оригіналі це призначення UML визначене за допомогою слова construct, яке ми передаємо терміном «проектування». Йдеться про те, що UML призначена не лише для опису абстрактних моделей додатків, але і для безпосереднього маніпулювання артефактами, що входять до складу цих додатків, у тому числі такими, як програмний код [12; 22].

Іншими словами, одним з призначень UML є, наприклад, створення таких моделей, для яких можлива **автоматична генерація програмного коду** (чи фрагментів коду) відповідних додатків. Більше того, природа моделей UML така, що можливий і зворотний процес: автоматична побудова моделі за кодом готового додатка. Англійською мовою автоматична побудова моделі за кодом готового додатка називається ***reverse engineering*** і зазвичай перекладається як «***зворотне проектування».*** Є непоганий альтернативний варіант: *«інженерний аналіз програм»*, але він не отримав поширення.

І останнє з цього набору ключових слів – «***документування».*** За великим рахунком, UML-моделі самі по собі вже є документами і дуже зрозумілими, навіть для неспеціаліста. Причому будь-який елемент на будь-якій діаграмі може бути забезпечений текстовим коментарем. Тобто, побудова набору діаграм вже є процесом документування майбутньої системи. Більше того, більшість інструментів UML-проектирования уміють витягати текстову інформацію з моделей і генерувати відносно легкі для читання тексти.

5.2.3. Інструментальна підтримка

Розглянемо, як співвідноситься сьогоднішня практика використання UML з призначенням мови, що декларується вище.

Проводячи в життя принцип навчання на прикладах, для ілюстрації вищесказаного звернемося до однієї з діаграм UML – діаграми використання (детальний опис цього типу діаграм наведений нижче).

На нашу думку, можна виділити три основні варіанти використання UML (рис. 5.2).

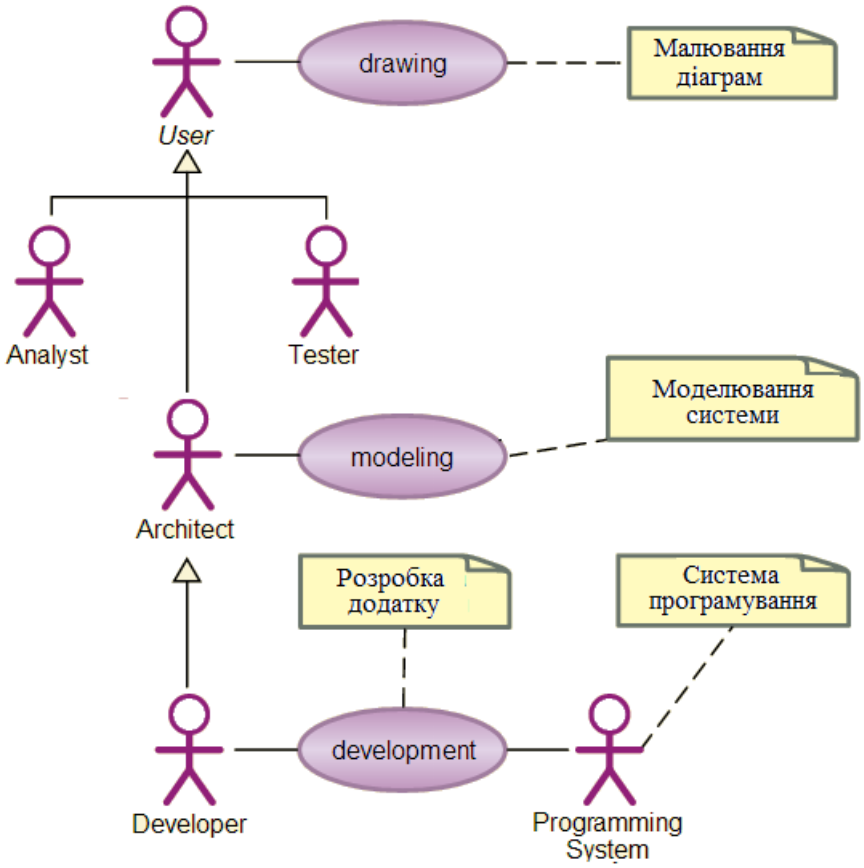


Рисунок 5.2 – Інструментальна підтримка

Варіант використання **drawing** («Малювання діаграм») має на увазі *зображення діаграм UML* з метою обдумування, обміну ідеями між людьми, документування і тому подібного. Значимим для користувача **User** результатом в цьому випадку є саме зображення діаграм. Іноді малювання діаграм від руки фломастером з подальшим фотографуванням цифровим апаратом може виявитися практичним.

Варіант використання **modeling** («Моделювання систем») має на увазі *створення і зміну моделі* системи в термінах тих елементів моделювання, які передбачаються метамоделлю UML. Значимим результатом в цьому випадку є **машинно-читаний артефакт** з описом моделі. Скорочено називатимемо такий артефакт просто ***моделлю***, діяльність зі складання моделі називатимемо моделюванням, а суб'єкта моделювання називатимемо архітектором Architect.

Варіант використання **development** («Розробка додатків») має на увазі *детальне моделювання, реалізацію і тестування* додатка в термінах UML.

Значимим для користувача **Developer** результатом в цьому випадку є працюючий додаток, який може бути скомпільований в мову, підтримувану конкретною системою програмування або відразу інтерпретований середовищем виконання інструменту. Цей варіант використання найскладніший в реалізації.

Сучасні інструменти підтримують вказані варіанти використання далеко не в рівній мірі. Усі інструменти уміють (погано або добре) візуалізувати усі типи діаграм UML, деякі інструменти дозволяють побудувати модель, що допускає якесь подальше використання, але тільки небагато інструментів можуть генерувати виконуваний код і то, ні в якому разі, не для усіх діаграм.

5.2.4. Способи використання UML

Із сказаного вище видно, що UML призначена для вирішення різних завдань, відповідно вона може бути використана і практично використовується по-різному. Далі ми перераховуємо різні способи використання UML.

Малювання картинок. Графічні засоби UML можна і треба використати безвідносно до усього іншого. Навіть малювання діаграм олівцем на папері дозволяє упорядкувати думки і зафіксувати для себе істотну інформацію про модельований додаток або іншу систему.

Обмін інформацією. Співтовариство людей, що застосовують і розуміють UML, нестримно росте. Якщо ви використовуватимете UML, то вас розумітимуть інші, і ви розумітимете інших «з напівпогляду».

Специфікація систем. Це найважливіший спосіб використання UML. І хоча не в усіх випадках UML виявляється абсолютно адекватним засобом специфікації, сподіваємося, що у міру розвитку мови все менше залишатиметься таких виключень, де UML непридатна.

Повторне використання архітектурних рішень. Повторне використання раніше розроблених рішень – ключ до підвищення ефективності. На жаль, моделі UML поки що повторно використовуються в дуже обмежених масштабах.

Генерація коду. Генерувати код треба і можна, але можливості наявних інструментів не варто переоцінювати.

5.2.5. Метод визначення UML

У основу опису UML покладений метод розкручування, тобто використання мови, що визначається, для визначення цієї мови. А саме, основні конструкції UML формально визначені за допомогою UML. Це описано в UML неформально, за допомогою текстів природною (англійською) мовою.

Метод розкручування часто застосовується при визначенні формальних мов. Наприклад, можна дуже витончено визначити операційну семантику мови програмування, якщо написати транслятор або інтерпретатор цієї мови цією ж мовою. Одним з перших цей прийом використав Н. Вірт, створюючи мову Паскаль.

У описі UML використовуються три мовні рівні:

1. *Мета-метамодель*, тобто опис мови, якою описана метамодель.
2. *Метамодель*, тобто опис мови, якою описуються моделі.
3. *Модель*, тобто опис самої модельованої предметної області.

Використання приставки **мета** (від грецького μετα, що означає «між», «після», «через») може бути дещо незвичним, нехай ця мова називається *X*. Якщо опис хороший, то на самому початку вказується мова (іноді її так і називають – метамова), яка використовується для опису мови *X*. Наприклад, наводиться фраза такого типу: «синтаксис мови *X* описаний за допомогою контекстно-вільної граматики, правила якої записані у формі Бекуса-Наура (БНФ), контекстні умови і семантика описані природною мовою».

Якщо опис не дуже хороший, то така фраза може бути і відсутньою, але вона все одно неявно мається на увазі і від читача вимагається зрозуміти використовувану метамову за контекстом. Далі за допомогою метамови більш менш формально описуються конструкції мови *X*. Усе, що не вдається описати формально, описується природною мовою.

Таким чином, основна ідея опису UML цілком традиційна і узгоджується із загальноприйнятою практикою: ***мета-метамодель*** – це опис використовуваного формалізму; ***метамодель*** – це і є власне опис мови (елементів моделювання); там, де формалізм не спрацьовує, на допомогу приходить природна мова.

5.2.6. Термінологія і нотація

Проблема термінології є однією з найхворобливіших при обговоренні мови UML. По-перше, автори UML намагалися зробити мову **незалежною від конкретних мов програмування**, моделей обчислювальності і тому подібного. З цією метою вони частенько вводили нові терміни для визначуваних понять, щоб випадковий збіг із старим терміном, вже зайнятим у якій-небудь суміжній області програмування, не вводив в оману користувача.

По-друге, мова UML порівняно молода (але вже модна), тому сталої термінологічної традиції доки немає.

Щоб підкреслити, що UML мова графічна, автори називають правила запису (малювання) моделей не синтаксисом, а **нотацією**.

При розробці UML були запропоновані і прийняті розумні рекомендації щодо вибору нотації. Автори виходили з того, що UML використовуватиметься по-різному: починаючи від не дуже акуратного малювання від руки на листку паперу, друку чорно-білих зображень в книгах і закінчуючи створенням складних діаграм за допомогою комп'ютера. Тому в якості основних графічних елементів були вибрані такі, які було б легко використати в усіх випадках. Типів елементів нотації п'ять:

1) фігура (shape);
2) лінія (line);
3) значок (icon);
4) текст (text);
5) рамка (frame).

Фігури (shape) в UML використовуються двовимірні (тобто їх можна намалювати на площині) і замкнуті (тобто внутрішня і зовнішня частини). Фігури можуть міняти свої розміри і форму, зберігаючи при цьому свої інтуїтивно відмітні ознаки.

Наприклад, серед фігур UML є прямокутники і еліпси. Вони можуть бути зображені багатьма способами: різного розміру, з різним співвідношенням довжин сторін, по-різному орієнтовані відносно меж сторінки і так далі, але в усіх випадках прямокутник відмінний від еліпса і не може бути з ним сплутаний.

Усередині фігур можуть поміщатися інші елементи нотації: тексти, лінії, значки і навіть інші фігури. Єдина вимога: має бути однозначно зрозуміло, що елемент нотації знаходиться усередині фігури, зокрема, його зображення не повинне перетинати межу фігури.

Лінії (line) в UML завжди приєднуються своїми кінцями до фігур або значків, вони не можуть бути намальовані самі по собі. Форма ліній довільна: це можуть бути прямі, ламані, плавні криві – значення це не має. Товщина ліній також довільна. А ось стиль лінії має значення. На щастя, в UML використовується тільки два стилі ліній, які важко сплутати: суцільні і пунктирні лінії. До ліній можуть примальовувати різні додаткові елементи: стрілки на кінцях, тексти і так далі. Єдина вимога: повинно бути ясно, що додатковий елемент належить саме до цієї лінії. Лінії можуть перетинатися, і це нічого не означає, але рекомендується уникати таких випадків, оскільки це утрудняє сприйняття.

Значки (icon) в UML схожі на фігури тим, що вони двовимірні, а відрізняються тим, що не мають внутрішності, в яку можна щось помістити, і, як правило, не міняють свою форму і розміри.

Тексти (text) в UML – це, як завжди, послідовності помітних символів деякого алфавіту. Алфавіт не фіксований – він тільки має бути зрозумілий читачеві моделі. Гарнітура, розмір і колір шрифту не мають значення, а ось зображення шрифту має: в UML розрізняються прямі, *курсивні* і <u>підкреслені</u> тексти.

Рамки (frame) з'явилися в UML 2. Рамка – це окремий випадок фігури, яка використовується виключно як контейнер для інших фігур, ліній, значків і текстів. Рамка має прямокутну форму і, як правило, ярличок в лівому верхньому кутку, в якому вказується тип і ім'я рамки.

Загалом, нотація UML досить вільна: малювати можна як завгодно, аби не виникало непорозумінь.

Використання кольорів для заливки фігур і розфарбовування ліній, тіні у значків і фігур, різні шрифти в текстах, нарешті, анімація зображень – усе це, звичайно, корисні речі, оскільки підвищують наочність картинок. Важливо при цьому знати міру, а міра дуже проста і навіть має назву – ***канонічна нотація***. Згідно з нею будь-яка модель може бути описана монохромними рисунками з текстовими поясненнями. При цьому рисунки повинні залишатися зрозумілими після друку на чорно-білому принтері.

У паперовій монохромній книзі зазвичай використовують канонічну нотацію, але в електронній книзі в діаграмах краще використати кольори (для наочності). Наприклад, тут будемо додержуватися певної палітри кольорів, що відображає вибране представлення моделі [22]:

 – моделювання використання (бузковий);

 – моделювання структури (жовтий);

 – моделювання поведінки (блакитний);

 – елементи, використовувані і для моделювання структури, і для моделювання поведінки (зелений);

 – стереотипи, обмеження і інші значимі елементи (сірий).

5.3. Модель і її елементи

Згідно з «The Unified Modeling Language User Guide» [11], UML складається всього з трьох «будівельних блоків»:

1. Сутності – це самі елементи моделі.
2. Відношення, що зв'язують сутності. Відношення визначають, як семантично пов'язані дві або більше за сутності.
3. Діаграми – це представлення моделей UML. Вони показують набори сутностей, які «розповідають» про програмну систему і є нашим способом візуалізації того, що робитиме система (аналітичні діаграми) або як вона робитиме це (проектні діаграми).

Модель UML **–** це сукупність кінцевої множини конструкцій мови, головні з яких **–** це ***сутності*** і ***відношення*** між ними.

Розглядаючи модель UML з найзагальніших позицій, можна сказати, що це граф, в якому вершини і ребра навантажені додатковою інформацією і можуть мати складну внутрішню структуру. Вершини цього графа називаються *сутностями,* а ребра – *відношеннями*.

5.3.1. Сутності

Для зручності огляду сутності в UML можна розділити на чотири групи: 1) структурні; 2) поведінкові; 3) такі, що групують; 4) анотації.

1. Структурні сутності призначені для опису структури. Зазвичай до структурних сутностей відносять такі. На рис. 5.3 наведена стандартна нотація в мінімальному варіанті для структурних сутностей.

Об'єкт **(object)** 1 **–** сутність, що має унікальність і інкапсулює в собі стан і поведінку.

Клас **(class)** 2 **–** опис множини об'єктів із загальними атрибутами, що визначають стан, і операціями, що визначають поведінку.

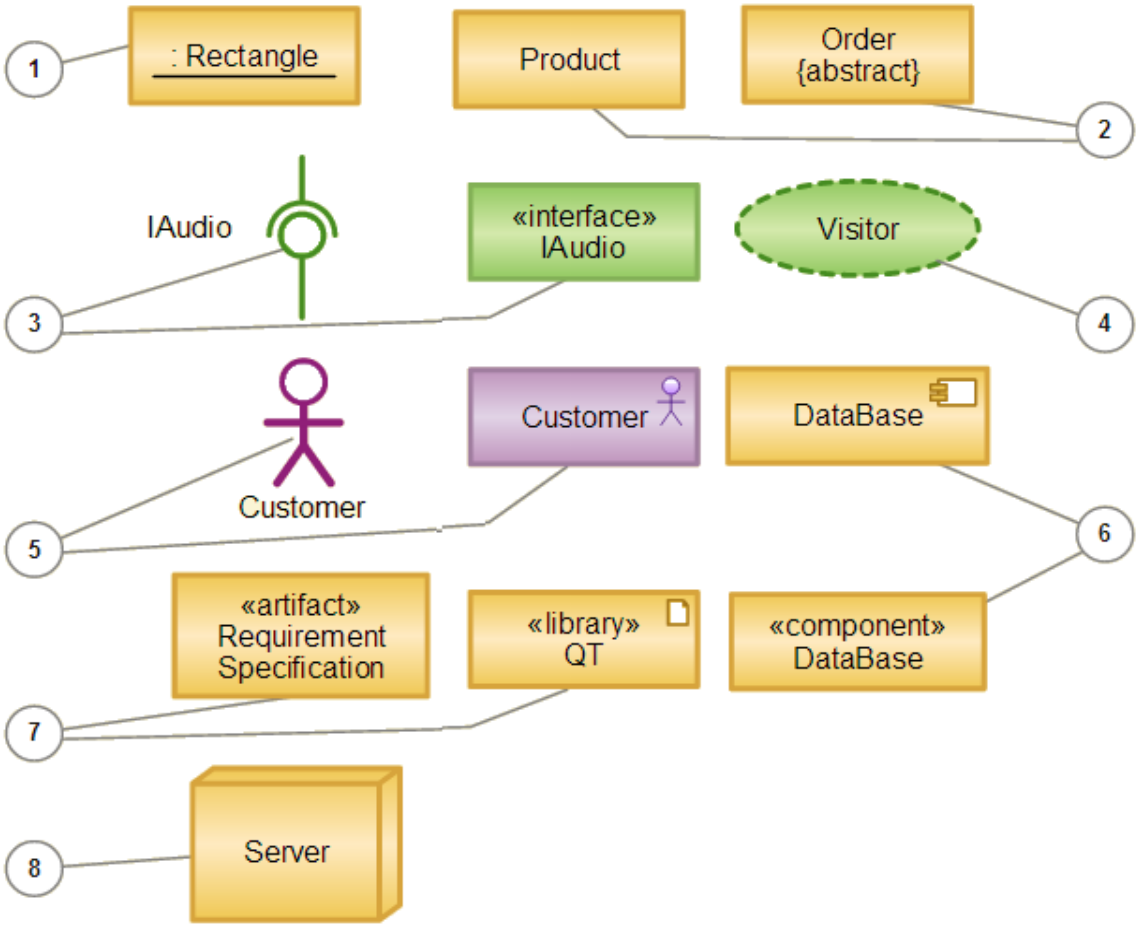


Рисунок 5.3 – Нотація структурних сутностей

***Інтерфейс* (interface)** 3 – іменована множина операцій, що визначає набір послуг, які можуть бути запитані споживачем і надані постачальником послуг.

***Кооперація* (collaboration)** 4 – сукупність об'єктів, які взаємодіють для досягнення певної мети.

***Дійова особа* (actor)** 5 – сутність, що знаходиться поза модельованою системою і безпосередньо взаємодіє з нею.

***Компонент* (component)** 6 – модульна частина системи з чітко певним набором необхідних інтерфейсів, що надаються.

***Артефакт* (artifact)** 7 – елемент інформації, який використовується або породжується в процесі розробки програмного забезпечення. Іншими словами, артефакт – це фізична одиниця реалізації, що отримується з елементу моделі.

***Вузол* (node)** 8 – обчислювальний ресурс, на якому розміщуються і при необхідності виконуються артефакти.

2. Поведінкові сутності (рис. 5.4) призначені для опису поведінки. Основних поведінкових сутностей всього дві**:** *стан* і *дія* (іноді вживається ще і *діяльність*, яка можна розглядається як особливий випадок стану).

***Стан* (state)** 1 – період в життєвому циклі об'єкту, знаходячись в якому об'єкт задовольняє деякій умові і здійснює власну діяльність або чекає настання деякої події.

***Діяльність* (activity)** 2 можна вважати частковим випадком стану, який характеризується тривалими (за часом) не атомарними обчисленнями.

***Дія* (action)** 3 – примітивне атомарне обчислення.

***Варіант використання* (use case)** 4 – множина сценаріїв, об'єднаних за деяким критерієм і виконуваних системою дій, що описують послідовності, які доставляють значущий для деякої дійової особи результат. На рис. 5.4 наведена стандартна нотація в мінімальному варіанті для поведінкових сутностей.

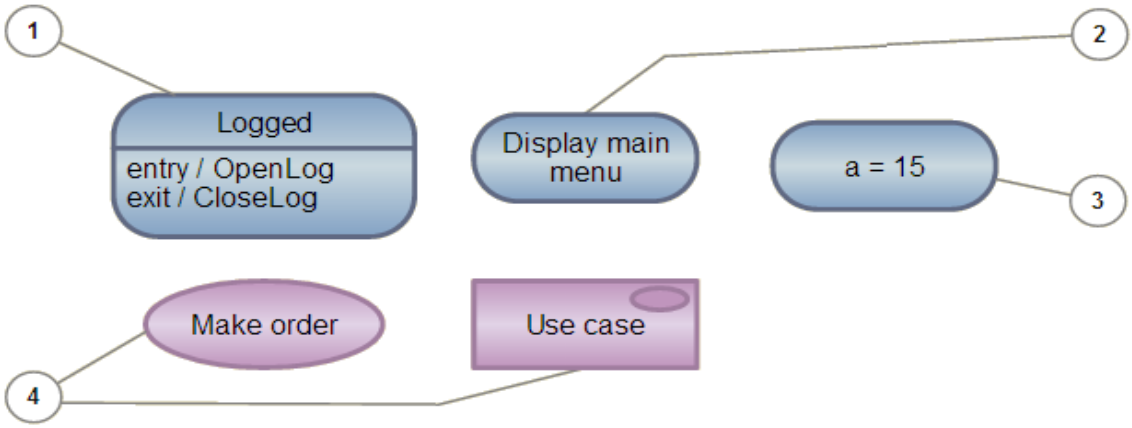


Рисунок 5.4 – Нотація поведінкових сутностей

3. Групуюча сутність в UML одна – пакет, зате універсальна. ***Пакет* (package)** 1 – група елементів моделі (у тому числі пакетів, рис. 5.5).

4. Сутність анотації теж одна – коментар (рис. 5.5). ***Коментар* (comment)** 2 – довільний за форматом і змістом опис одного або декількох елементів моделі.

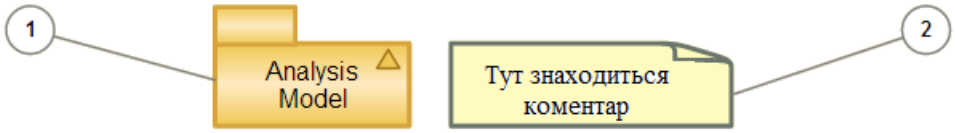


Рисунок 5.5 – Групуюча нотація і коментар

5.3.2. Відношення

У UML використовуються чотири основні типи відношень:

- залежність (dependency);
- асоціація (association);
- узагальнення (generalization);
- реалізація (realization).

Залежність – це найзагальніший тип відношення між двома сутностями.

Відношення залежності вказує на те, що зміна незалежної сутності якимсь чином впливає на залежну сутність.

Графічно відношення залежності зображається у вигляді пунктирної лінії із стрілкою 1, спрямованою від залежної сутності 2 до незалежної 3, як показано на рис. 5.6. Як правило, семантика конкретної залежності уточнюється в моделі за допомогою додаткової інформації. Наприклад, залежність із стереотипом «**use**» означає, що залежна сутність використовує (скажімо, викликає операцію) незалежну сутність.

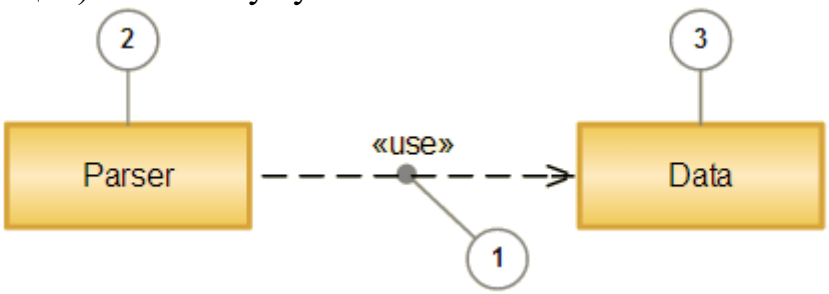


Рисунок 5.6 – Відношення залежності

Відношення асоціації має місце, якщо одна сутність безпосередньо пов'язана з іншою (чи з іншими – асоціація може бути не лише бінарною).

Графічно асоціація зображається у вигляді суцільної лінії 1 з різними доповненнями, що сполучає пов'язані сутності, як показано на рис. 5.7. На програмному рівні безпосередній зв'язок може бути реалізований по-різному, головне, що асоційовані сутності знають одна про іншу. Наприклад, відношення частина-ціле є частковим випадком асоціації і називається *відношенням агрегації*.

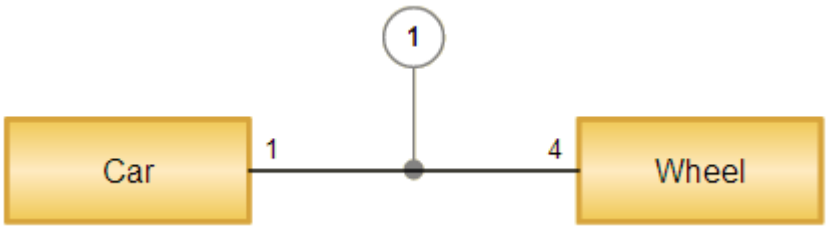


Рисунок 5.7 – Відношення асоціації

Узагальнення – це відношення між двома сутностями, одна з яких є частковим (спеціалізованим) випадком іншої.

Графічно узагальнення зображається у вигляді лінії з трикутною незафарбованою стрілкою на кінці 1, спрямованою від частки 2 (підкласу) до загального 3 (суперкласу), як показано на рис. 5.8.

Відношення реалізації використовується дещо рідше, ніж попередні три типи відношень, оскільки часто маються на увазі за умовчанням.

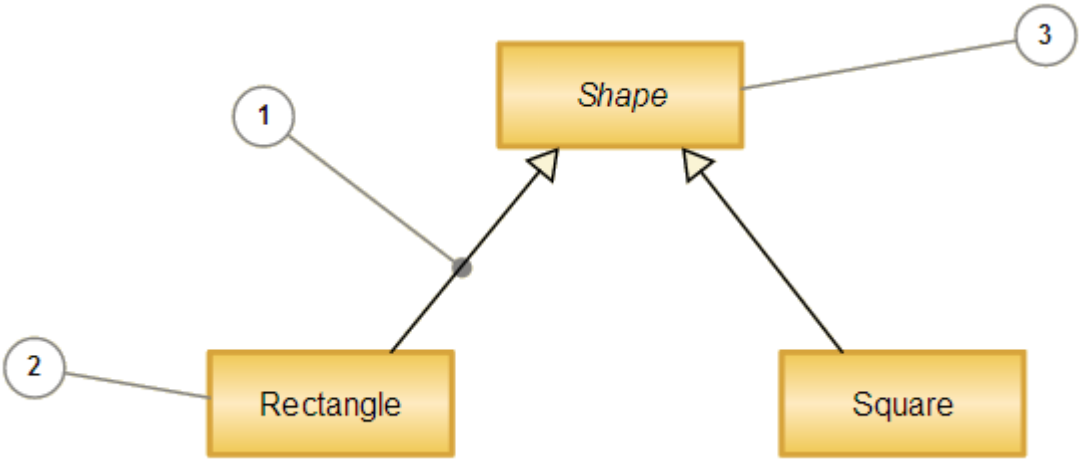


Рисунок 5.8 – Відношення узагальнення

Відношення реалізації вказує, що одна сутність є реалізацією іншої.

Наприклад, клас є реалізацією інтерфейсу. Графічно реалізація зображається у вигляді пунктирної лінії з трикутною незафарбованою стрілкою на кінці 1, спрямованою від реалізовуючої сутності 2 до тієї, що реалізовується 3, як показано на рис. 5.9.

Перераховані типи відношень є основними, різні їх варіації і додаткові відношення (Агрегація (◇——) – цільовий елемент є частиною початкового елементу; Композиція (◆——) – строга, більше обмежена форма агрегації; Включення (⊕——) – початковий елемент містить цільовий елемент) детально розглядаються в подальших розділах.

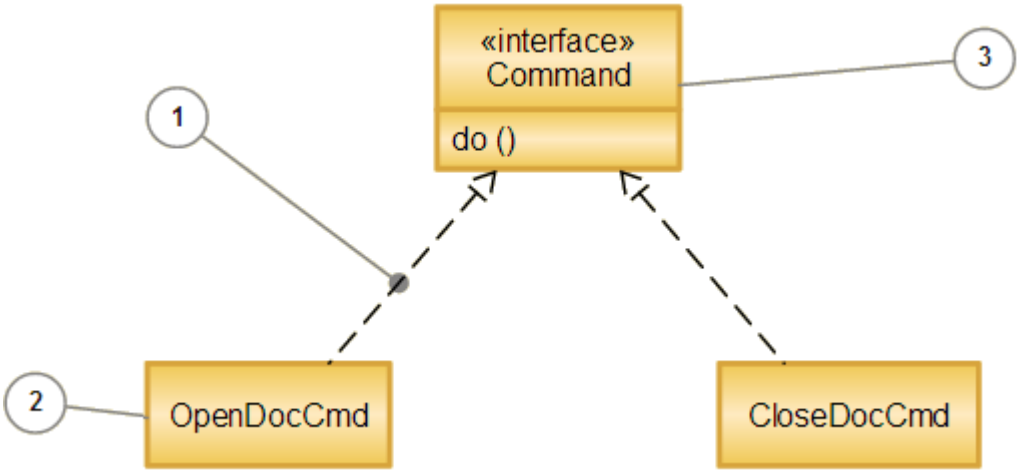


Рисунок 5.9 – Відношення реалізації

5.3.3. Діаграми

У попередньому параграфі ми змалювали множину слів UML (лексем, графічних примітивів, елементів моделювання). Перейдемо до синтаксису, а саме до опису того, як із слів конструюються пропозиції.

На перший погляд, усе дуже просто: беруться сутності і, якщо потрібно, вказуються відношення між ними. В результаті виходить модель, тобто граф (з

різнорідними вершинами і ребрами), навантажений додатковою інформацією. Але при уважнішому розгляді виявляються проблеми.

Розглянемо таку аналогію з природною мовою. Кожна трійка «сутність» – «відношення» – «сутність» в моделі цілком може розглядатися як просте твердження: 2 < 5, ртуть важча за залізо, місяць є супутником Землі (усе це приклади відношень). Поки усе добре, але згадаємо, що в графі (у моделі) ніякої упорядковуючої структури немає: не можна сказати, що ця вершина перша, а ця – друга.

Продовжуючи нашу аналогію, виходить, що модель – це множина незв'язаних між собою пропозицій, ніяк не впорядкована. Якщо узяти який-небудь «нормальний» текст, то можна помітити, що окрім структури пропозицій, є маса додаткових структур: пропозиції об'єднані в абзаци, абзаци зібрані в параграфи і розділи, у яких є заголовки, окрім звичайних абзаців і заголовків є примітки і виноски. І усі ці додаткові структури по суті нічого не додають до змісту книги, але серйозно впливають на її читабельність. Текст, в якому немає цих структур, зрозуміти дуже важко. Звідси висновок: окрім сутностей і відношень, в моделі має бути якась структура, яка б допомагала її побудові і розумінню.

Діаграми UML і є та основна структура, що накладається на модель, яка полегшує створення і використання моделі.

Діаграма **–** це графічне представлення деякої частини графа моделі.

Діаграми – це свого роду картини, або представлення моделі. **Діаграма це не модель!** Насправді, відмінність між діаграмою і моделлю є дуже важливою для розуміння, оскільки сутність або відношення можуть бути видалені з діаграми, або навіть з усіх діаграм, але як і раніше вони продовжують існувати в моделі. Вони залишатимуться в моделі до тих пір, поки не будуть явно видалені з неї.

Взагалі кажучи, в діаграму можна було б включити будь-які (допустимі) комбінації сутностей і відношень, але свавілля в цьому питанні утруднило б розуміння моделей. Тому автори UML визначили набір рекомендованих до використання типів діаграм, які дістали назву ***канонічних.***

Інструменти моделювання, як правило, забезпечують роботу з усіма канонічними діаграмами, але роблять це досить догматично, не дозволяючи відійти від канону ні на крок, навіть якщо це треба по суті завдання. Було б зручно, якби набір канонічних діаграм пропонувався за умовчанням, але користувач міг би налагодити, змінити і перевизначити цей набір у разі потреби, приблизно так, як це робиться з шаблонами Microsoft Word. Деякі інструменти, але далеко не все, підтримують такі можливості.

Відмітимо, що окрім сутностей і відношень на діаграмі є присутніми інші елементи моделі, які ми також називатимемо ***конструкціями*** мови. Це тексти, які можуть бути написані усередині фігур сутностей або поряд з лініями відношень, рамки діаграм і їх фрагментів, значки, що приєднуються до ліній або поміщаються всередину фігур.

5.3.4. Класифікація діаграм

У UML 1 усього визначено 9 канонічних типів діаграм. Нижче перераховані їх назви, прийняті в цьому посібнику (у інших джерелах можуть бути відмінності):

1. Діаграма використання (Use Case diagram).
2. Діаграма класів (Class diagram).
3. Діаграма об'єктів (Object diagram).
4. Діаграма станів (State chart diagram).
5. Діаграма діяльності (Activity diagram).
6. Діаграма послідовності (Sequence diagram).
7. Діаграма кооперації (Collaboration diagram).
8. Діаграма компонентів (Component diagram).
9. Діаграма розміщення (Deployment diagram).

Канонічні діаграми зовсім не утворюють повного ортогонального набору: вони перетинаються як за включеними в них засобами, так і за сферою застосування. Більше того, деякі з них є окремими випадками інших, є просто семантично еквівалентні пари, можна навести приклади допустимих діаграм, для яких складно вказати однозначно, до якого саме з канонічних типів діаграма належить.

Сказане можна проілюструвати умовною класифікацією діаграм, наведеною нижче (рис. 5.10). Тут на рисунку діаграма використання поміщена окремо, вона не належить ні до діаграм опису структури, ні до діаграм опису поведінки. У більшості джерел діаграми використання відносять до опису поведінки. Окрім відношень узагальнення, на діаграмі класів, додатково показані деякі залежності між окремими UML діаграмами. Ці залежності у кожному конкретному випадку носять різний характер, що відображено за допомогою використання різних стереотипів. Детальніше про стандартні стереотипи залежностей буде сказано пізніше.

У UML 2 внесені значні корективи як в список канонічних діаграм, а саме їх число збільшилося до 13, так і в список доступних конструкцій мови, що значно розширило сферу її застосування. Окрім цього, дві діаграми були перейменовані: **діаграма кооперації** була перейменована в **діаграму комунікації,** а **діаграма станів** в **діаграму автомата**.

Список нових діаграм і їх назв наведений нижче.

1. Діаграма внутрішньої структури (Composite Structure diagram).
2. Діаграма пакетів (Package diagram).
3. Діаграма автомата (State machine diagram).
4. Діаграма комунікації (Communication diagram).
5. Оглядова діаграма взаємодії (Interaction Overview diagram).
6. Діаграма синхронізації (Timing diagram).

На рис. 5.10, 5.11, 5.12 наведені діаграми класів, які відбражають взаємозв'язок діаграм в UML 1 и UML 2.

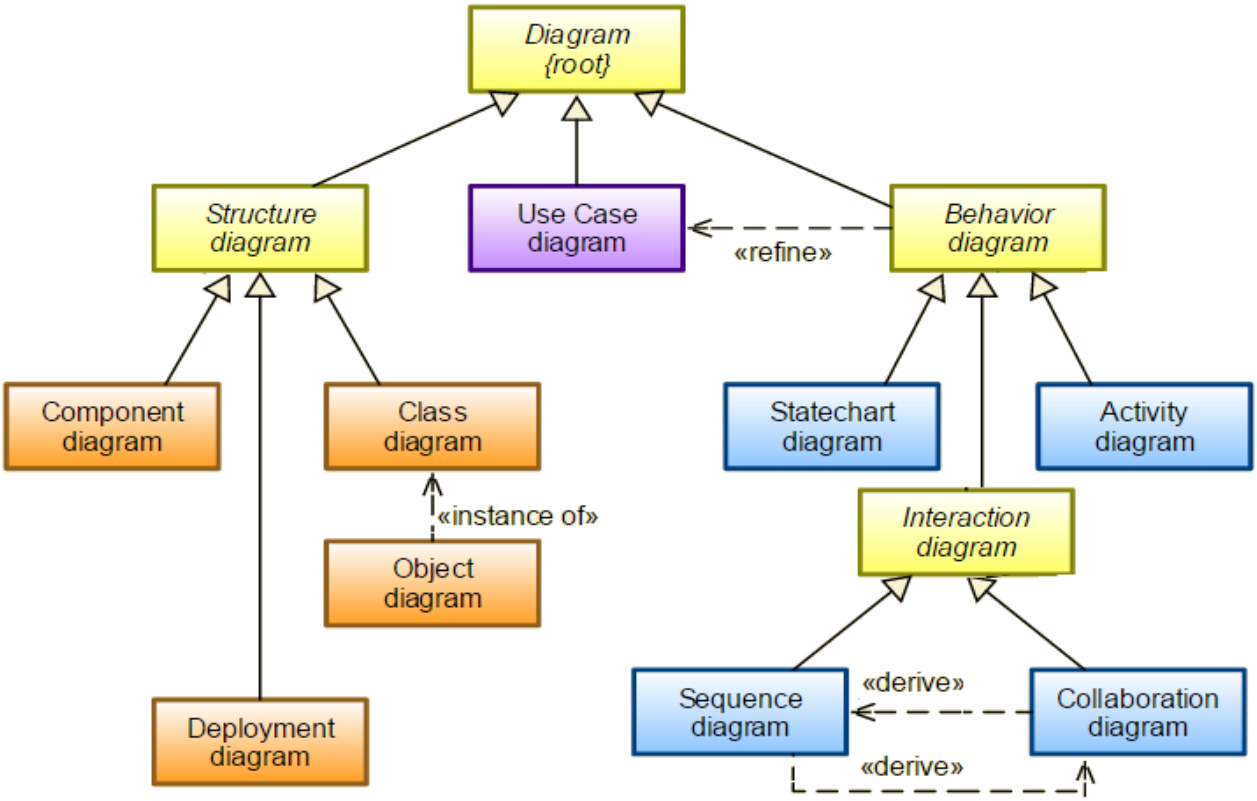


Рисунок 5.10 – Ієрархія типів діаграм для UML 1

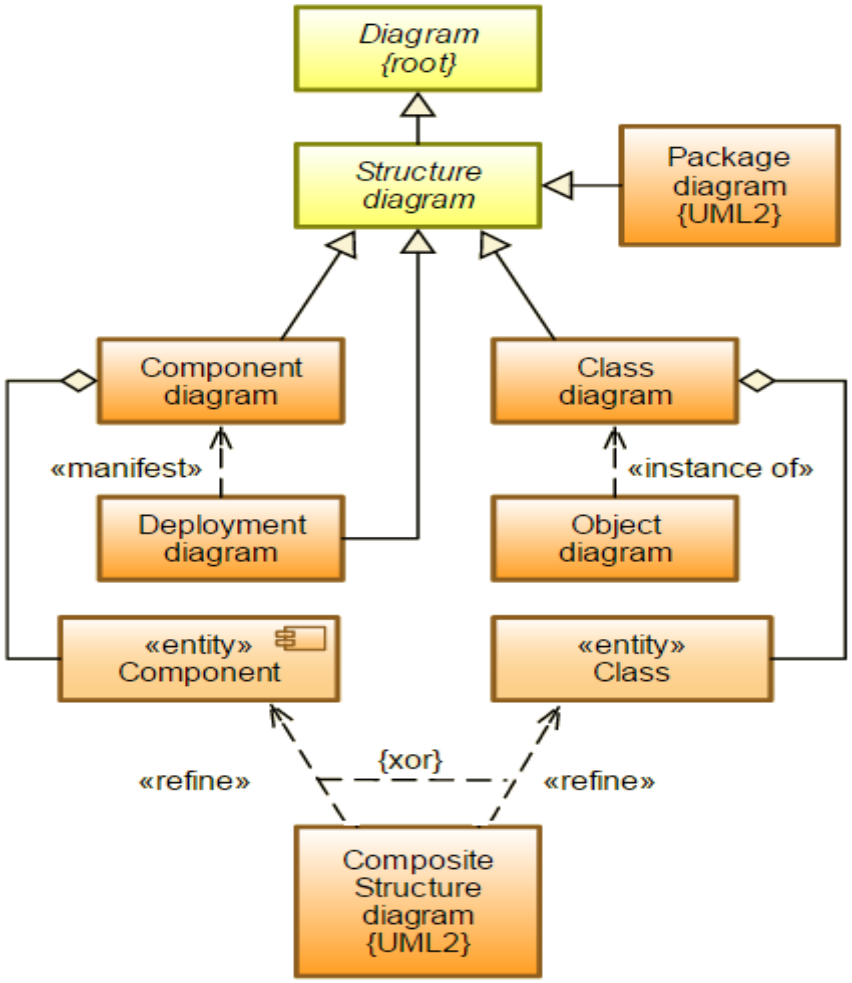


Рисунок 5.11 – Ієрархія типів діаграм для UML 2 (частина 1)

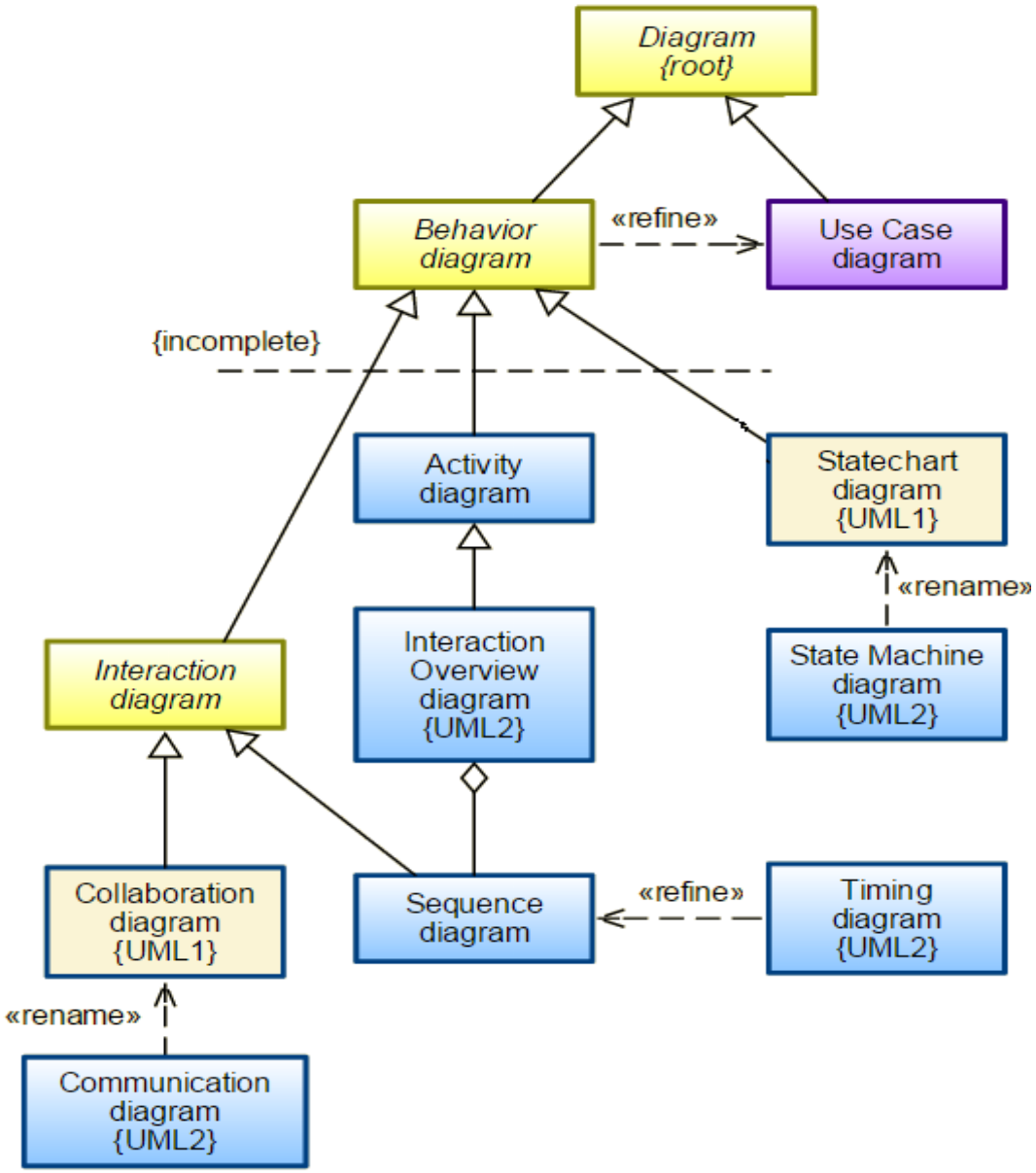


Рисунок 5.12 – Ієрархія типів діаграм для UML 2 (частина 2)

Основних елементів оформлення два: зовнішня рамка і ярличок з назвою діаграми. Якщо з рамкою усе просто – це прямокутник, що обмежує область в якому повинні знаходитися елементи діаграми, то назва діаграми записується в спеціальному форматі, наведеному на рис. 5.13.

Рисунок 5.13 – Нотація для діаграм

Вказана складна форма ярличка підтримується не усіма інструментами. Втім, це не обов'язково, оскільки семантика первинна, а нотація вторинна. Далі скрізь використовуватимемо як ярличок діаграми прямокутник.

Можливі **теги** (типи) для діаграм приведені в таблиці 5.1. Теги, що пропонуються стандартом, записані в другий стовпець. Проте, як показала практика, пропоновані стандартом правила не завжди зручні і логічно обгрунтовані, тому третій стовпець таблиці містить альтернативні теги.

Таблиця 5.1 – Типи і теги діаграм

Назва діаграми	Тег (стандартний)	Тег (пропонований)
Діаграма використання	**use case** або **uc**	use case
Діаграма класів	class	class
Діаграма автомата	**state machine** або **stm**	state machine
Діаграма діяльності	**activity** або **act**	activity
Діаграма послідовності	**interaction** або **sd**	sd
Діаграма комунікації	**interaction** або **sd**	comm
Діаграма компонентів	**component** або **cmp**	component
Діаграма розміщення	не визначений	deployment
Діаграма об'єктів	не визначений	object
Діаграма внутрішньої структури	class	**class** або **component**
Оглядова діаграма взаємодії	**interaction** або **sd**	interaction
Діаграма синхронізації	**interaction** або **sd**	timing
Діаграма пакетів	**package** або **pkg**	package

5.4. Загальні діаграми

Усі діаграми UML можна умовно розбити на дві групи, перша з яких – *загальні діаграми*. Загальні діаграми практично не залежать від предмета моделювання і можуть застосовуватися у будь-якому програмному проекті.

5.4.1. Діаграма використання

Діаграма використання **(діаграма прецедентів, use case diagram) –** це найзагальніше представлення функціонального призначення системи.

Діаграма використання покликана відповісти на головне питання моделювання: що робить система у зовнішньому світі?

На діаграмі використання (рис. 5.14) застосовуються два типи основних сутностей: варіанти використання 1 і дійові особи (актори – користувачі, пристрої) 2, між якими встановлюються такі основні типи відношень:

1. Асоціація між дійовою особою і варіантом використання 3.
2. Узагальнення між дійовими особами 4.
3. Узагальнення між варіантами використання 5.

4. Залежності (різних типів) між варіантами використання 6.

На діаграмі використання, як і на будь-якій іншій, можуть бути присутніми коментарі 7. Більше того, це настійно рекомендується робити для поліпшення читаності діаграм.

Детальний опис усіх варіацій діаграм використання наведений в параграфі 5.5.2, а на рисунку 5.14 показані тільки основні елементи нотації.

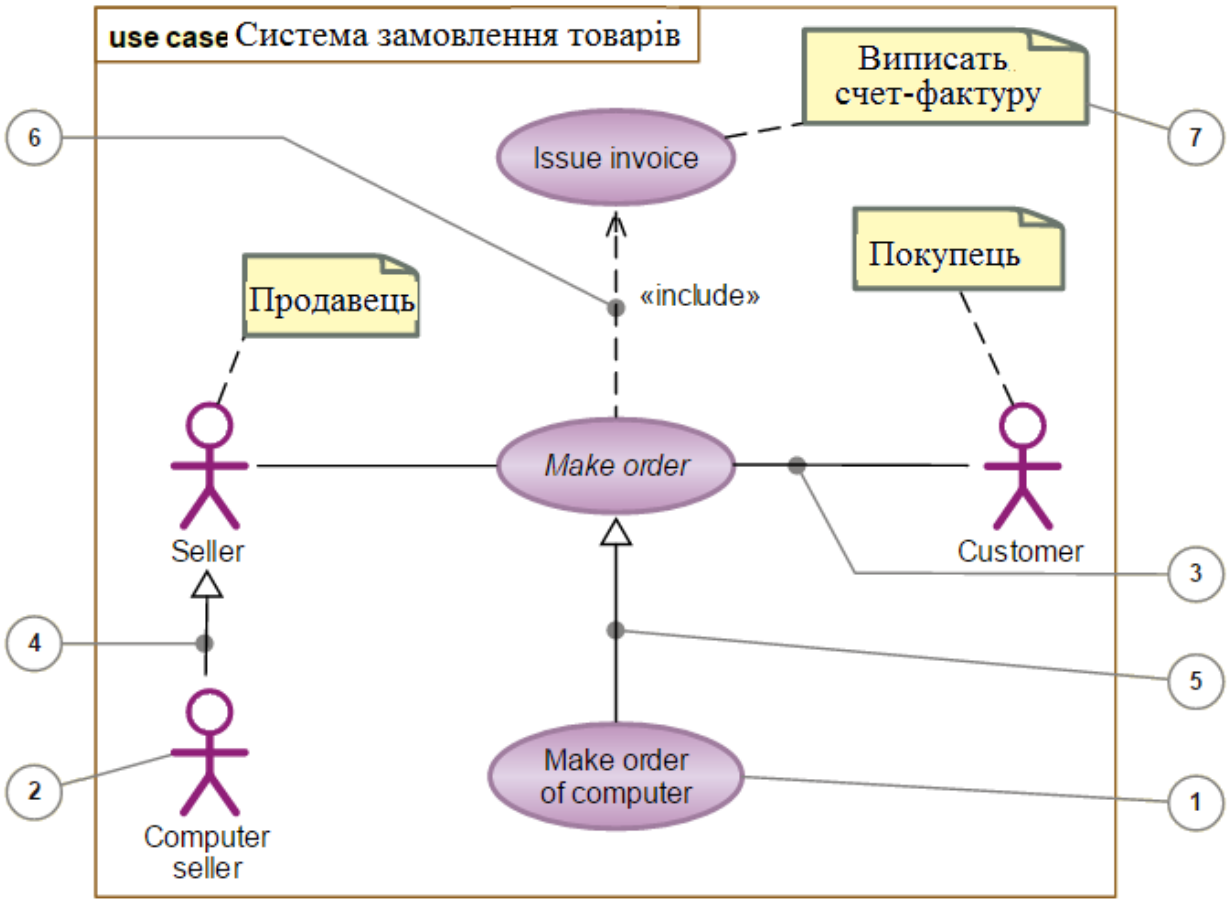


Рисунок 5.14 – Нотація діаграми використання (прецедентів)

5.4.2. Діаграма класів

Діаграма класів **(class diagram)** – основний спосіб опису структури системи.

Це не дивно, оскільки UML в першу чергу об'єктно-орієнтована мова, і класи є основним (якщо не єдиним) «будівельним матеріалом».

На діаграмі класів (рис. 5.15) застосовується один основний тип сутностей: класи 1 (включаючи численні окремі випадки класів: інтерфейси, примітивні типи, класи-асоціації і багато інших), між якими встановлюються наступні основні типи відношень:

- асоціація між класами 2 (з множиною додаткових подробиць);
- узагальнення між класами 3;
- залежності (різних типів) між класами 4 і між класами та інтерфейсами.

Детальний опис усіх варіацій діаграм класів наведений в підрозділі 5.6, а на рисунку 5.15 показані тільки основні елементи нотації.

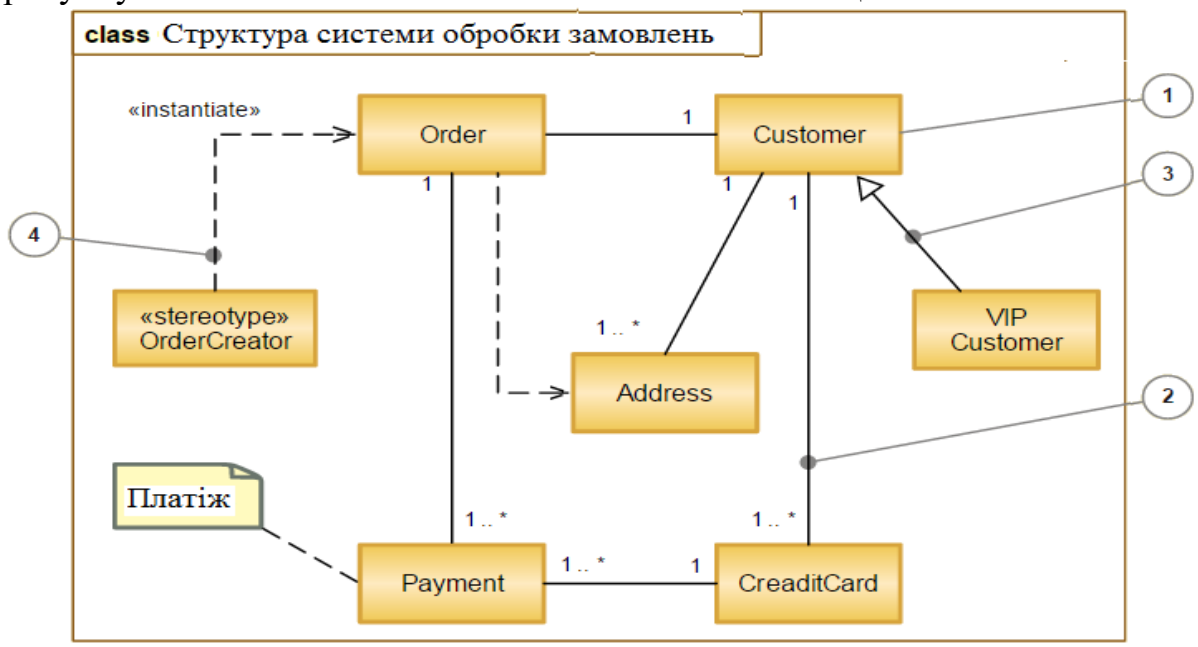


Рисунок 5.15 – Нотація діаграми класів

5.4.3. Діаграма автомата

Діаграма автомата (state machine diagram, *діаграма станів*) – це один із способів детального опису поведінки в UML на основі явного виділення станів і опису переходів між станами. По суті, діаграмою автомата, як це витікає з назви, є граф переходів станів, навантажений множиною додаткових деталей і подробиць. На діаграмі автомата (рис. 5.16) застосовують один основний тип сутностей – стан 1, і один тип відношень – переходи 2, але і для тих і для інших визначено багато різновидів, спеціальних випадків і додаткових позначень. Детальний опис усіх варіацій діаграм автомата наведений в підрозділі 5.7, а на рисунку 5.16 показані тільки основні елементи нотації.

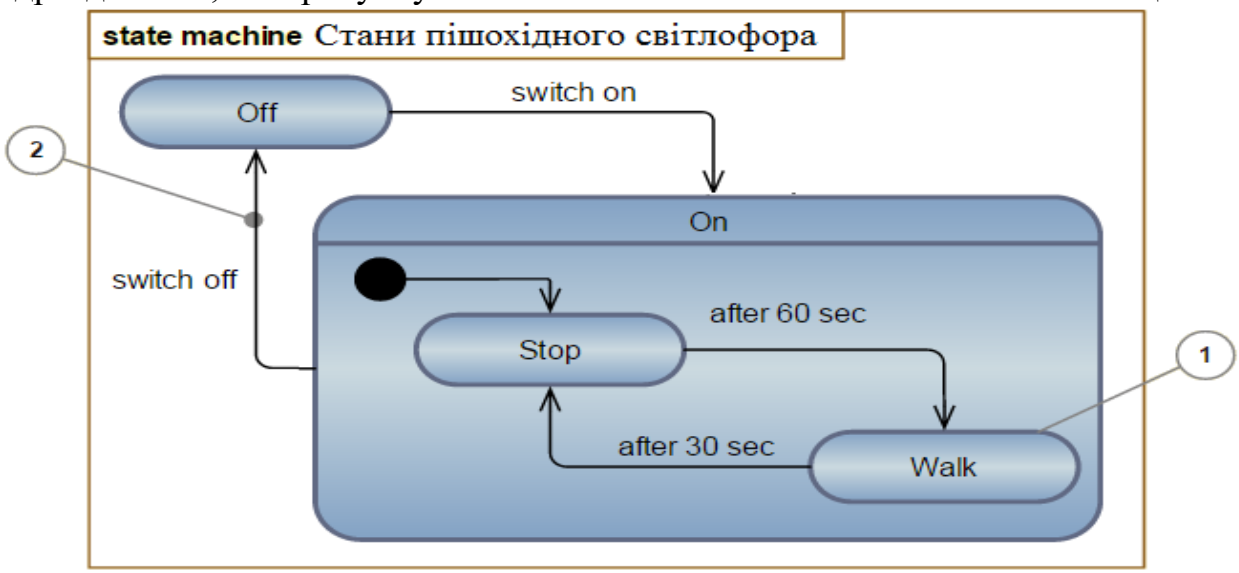


Рисунок 5.16 – Нотація діаграми автомата

5.4.4. Діаграма діяльності

Діаграма діяльності (activity diagram) – спосіб опису поведінки на основі вказівки потоків управління і потоків даних.

Діаграма діяльності – ще один спосіб опису поведінки, який візуально нагадує стару добру блок-схему алгоритму. Проте за рахунок модернізованих позначень, погоджених з об'єктно-орієнтованим підходом, діаграма діяльності UML є потужним засобом для опису поведінки системи.

На діаграмі діяльності (рис. 5.17) застосовують один основний тип сутностей – дія 1, і один тип відношень – переходи 2 (передачі управління і даних). Також використовуються такі конструкції як розвилки, злиття, з'єднання, галуження 3, які схожі на сутності, але такими насправді не є, а є графічним способом зображення деяких окремих випадків багатомісних відношень. Семантика елементів діаграм діяльності детально розібрана в розділі 5.7. Основні елементи нотації діаграмі діяльності, показані на рис. 5.17.

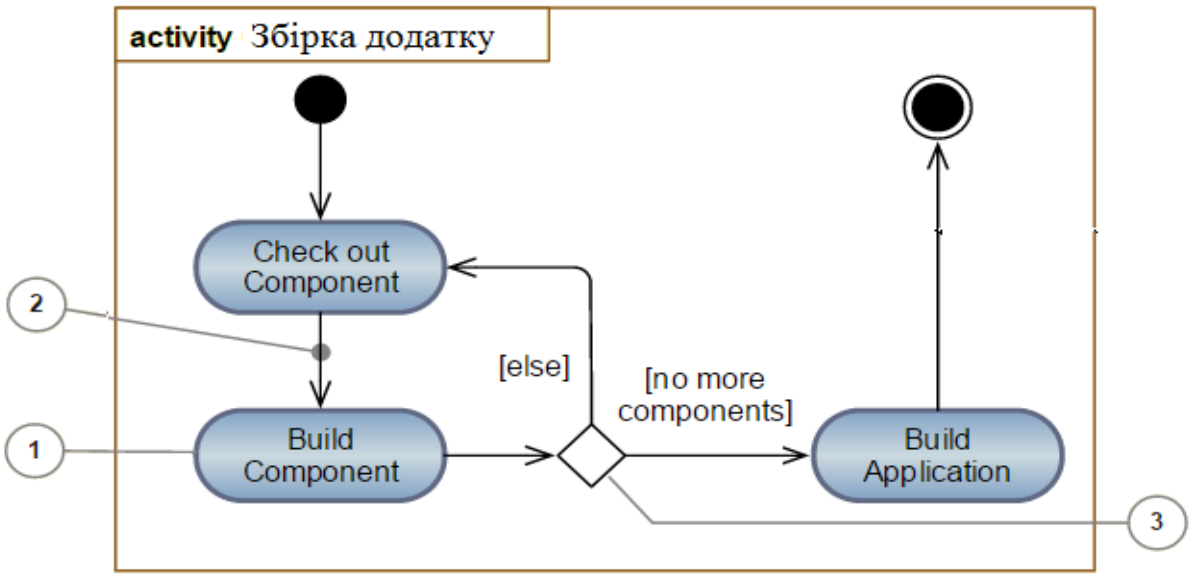


Рисунок 5.17 – Нотація діаграми діяльності

5.4.5. Діаграма послідовності

Діаграма послідовності (sequence diagram) – це спосіб опису поведінки системи на основі вказівки послідовності передаваних повідомлень.

Фактично, діаграма послідовності – це запис протоколу конкретного сеансу роботи системи (чи фрагмента такого протоколу). У об'єктно-орієнтованому програмуванні найістотнішою під час виконання є пересилка повідомлень між взаємодіючими об'єктами. Саме послідовність посилок повідомлень відображається на цій діаграмі, звідси і назва.

На діаграмі послідовності (рис. 5.18) застосовують один основний тип сутностей – екземпляри взаємодіючих класифікаторів 1 (в основному класів, компонентів і дійових осіб), і один тип відношень – зв'язки 2, по яких відбувається обмін повідомленнями 3. Передбачені декілька способів посилки

повідомлень, які в графічній нотації розрізняються видом стрілки, що відповідає відношенню.

Важливим аспектом діаграми послідовності є явне відображення плину часу. На відміну від інших типів діаграм, окрім хіба що діаграм синхронізації, на діаграмі послідовності має значення не лише наявність графічних зв'язків між елементами, але і взаємне розташування елементів на діаграмі. А саме, вважається, що є (невидима) вісь часу, за умовчанням спрямована зверху вниз, і те повідомлення, яке відправлене пізніше, намальоване нижче.

Вісь часу може бути спрямована горизонтально, в цьому випадку вважається, що час тече зліва направо.

На рис. 5.18 показані основні елементи нотації, вживані на діаграмі послідовності. Для позначення самих взаємодіючих об'єктів застосовується стандартна нотація – прямокутник з ім'ям екземпляра класифікатора. Пунктирна лінія, що виходить з нього, називається лінією життя (lifeline) 4. Це не позначення відношення в моделі, а графічний коментар, покликаний направити погляд читача діаграми в правильному напрямі. Фігури у вигляді вузьких смужок, накладених на лінію життя, також не є зображеннями модельованих сутностей. Це графічний коментар, що показує відрізки часу, протягом яких об'єкт володіє потоком управління (execution occurrence) 5 або іншими словами має місце активація (activation) об'єкту. Складені кроки взаємодії (combined fragment) 6 дозволяють на діаграмі послідовності відбивати і алгоритмічні аспекти протоколу взаємодії. Інші деталі нотації діаграми послідовностей див. в підрозділі 5.7.

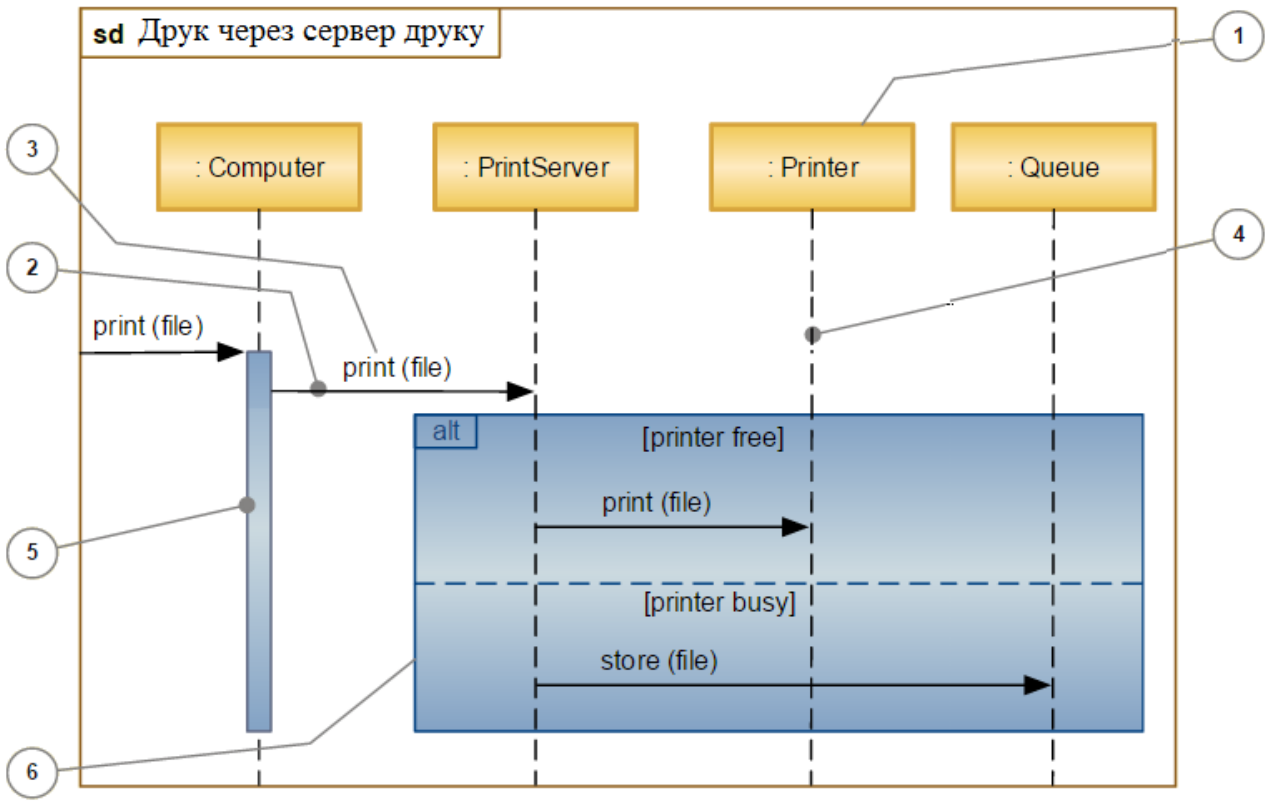


Рисунок 5.18 – Нотація діаграми послідовності

5.4.6. Діаграма комунікації

Діаграма комунікації (communication diagram) – спосіб опису поведінки, семантично еквівалентний діаграмі послідовності.

Фактично, цей такий же опис послідовності обміну повідомленнями взаємодіючих екземплярів класифікаторів, тільки виражений іншими графічними засобами.

Таким чином, на діаграмі комунікації (рис. 5.19) також як і на діаграмі послідовності застосовують один основний тип сутностей – екземпляри взаємодіючих класифікаторів 1 і один тип відношень – зв'язку 2. Проте тут акцент робиться не на часі, а на структурі зв'язків між конкретними екземплярами.

Для позначення самих взаємодіючих об'єктів застосовується стандартна нотація – прямокутник з ім'ям екземпляра класифікатора. Взаємне положення елементів на діаграмі комунікації (кооперації) не має значення – важливі тільки зв'язки (найчастіше екземпляри асоціацій), уздовж яких передаються повідомлення 3. Для відображення впорядкованості повідомлень в часі застосовується ієрархічна десяткова нумерація.

Порівняйте цей рисунок з рис. 5.18. **Нотація діаграми послідовності** (на них зображена одна і та ж поведінка), і вам усе стане зрозуміло. Інші деталі нотації діаграми комунікації приведені в підрозділі 5.7.

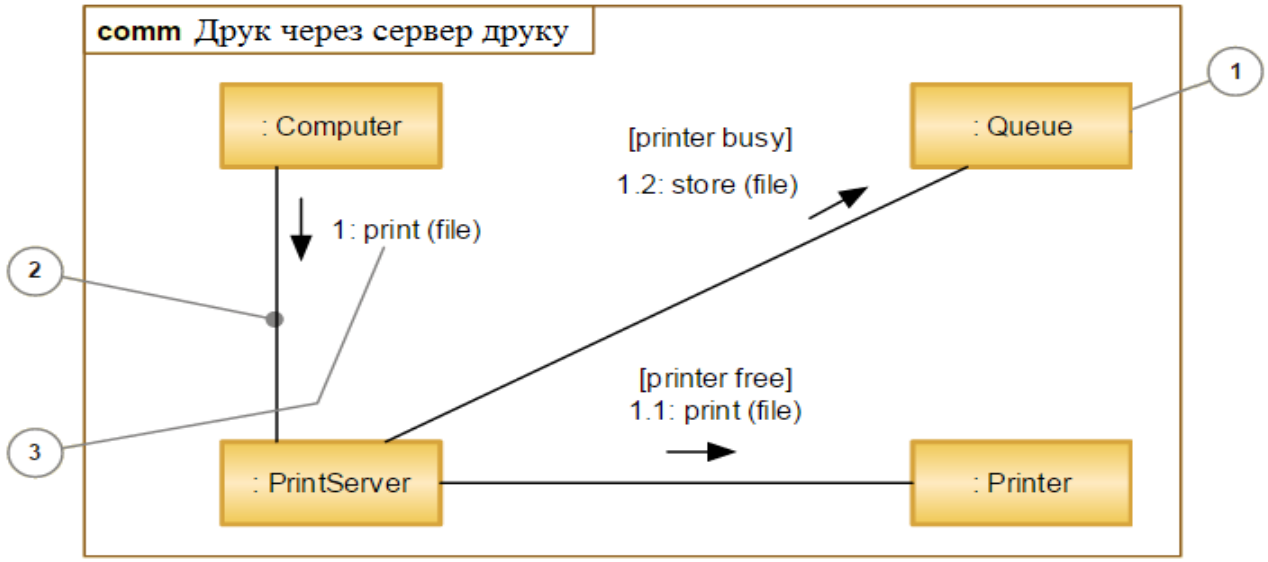


Рисунок 5.19 – Нотація діаграми комунікації

5.4.7. Діаграма компонентів

Діаграма компонентів (component diagram) – показує взаємозв'язки між модулями (логічними або фізичними), з яких складається модельована система.

Основний тип сутностей на діаграмі компонентів (рис. 5.20) – це самі компоненти 1, а також інтерфейси 2, за допомогою яких вказується взаємозв'язок між компонентами. На діаграмі компонентів застосовуються такі відношення:

– реалізації між компонентами і інтерфейсами (компонент реалізує інтерфейс);
– залежності між компонентами і інтерфейсами (компонент використовує інтерфейс) [3].

Детальний опис усіх варіацій діаграм компонентів наведений в підрозділі 5.6, а на рисунку 5.20 показані тільки основні елементи нотації.

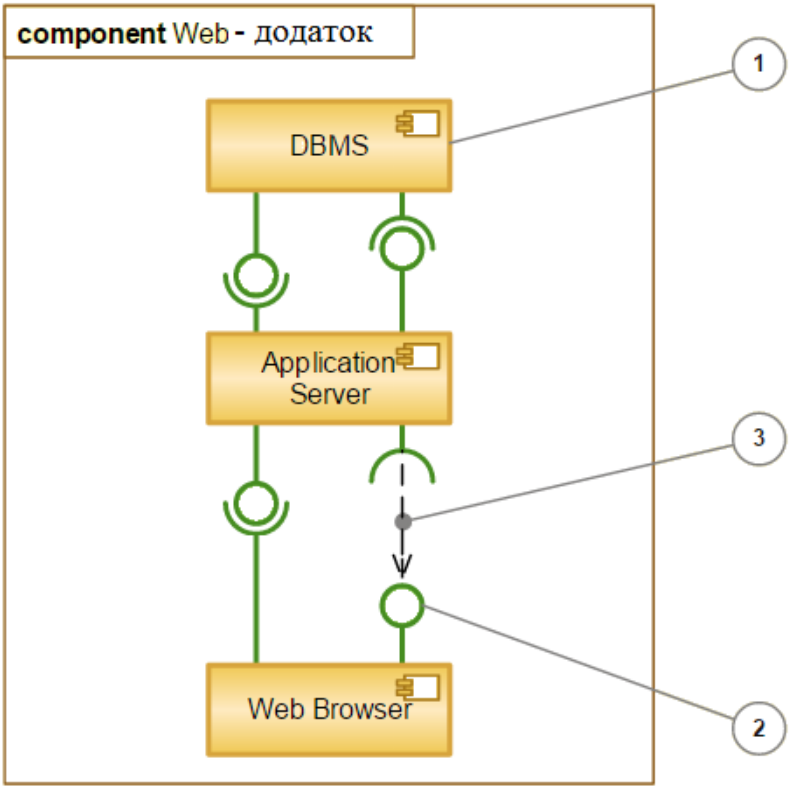


Рисунок 5.20 – Нотація діаграми компонентів

5.4.8. Діаграма розміщення

Діаграма розміщення (deployment diagram) разом з відображенням складу і зв'язків елементів системи показує, як вони фізично розміщені на обчислювальних ресурсах під час виконання.

Таким чином, на діаграмі розміщення (рис. 5.21), в порівнянні з діаграмою компонентів, додається два типи сутностей: артефакт [1], який є реалізацією компонента [2] і вузол [3] (може бути як класифікатор, що описує тип вузла, так і конкретний екземпляр), а також відношення асоціації між вузлами [4], що показує, що вузли фізично пов'язані під час виконання.

На рис. 5.21 показані основні елементи нотації, вживані на діаграмі розміщення. Для того щоб показати, що одна сутність є частиною іншої, застосовується або відношення залежності «deploy» [5], або фігура однієї сутності поміщається всередину фігури іншої сутності [6]. Детальний опис діаграми приведений в підрозділі 5.6.

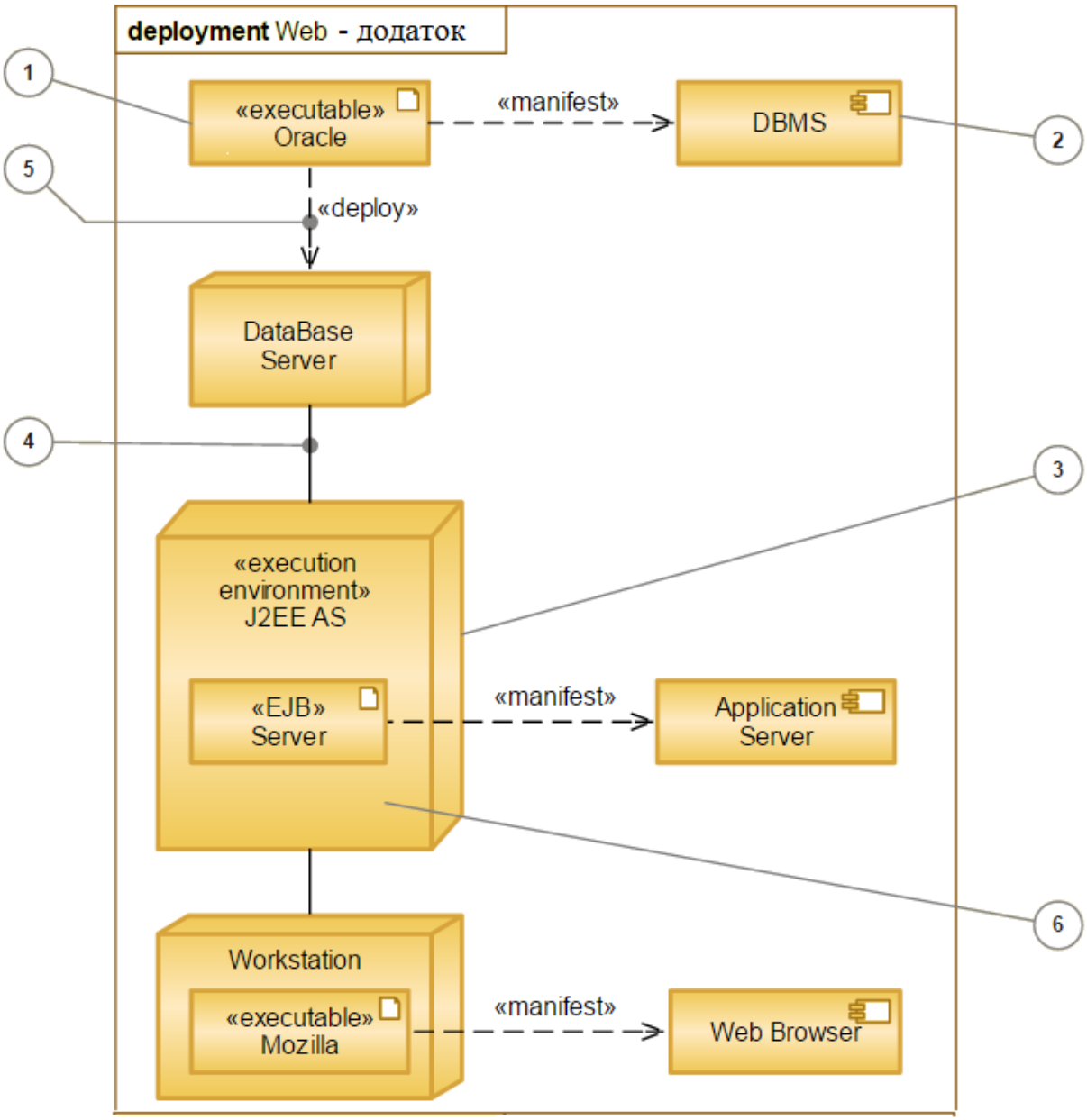


Рисунок 5.21 – Нотація діаграми розміщення

5.5. Спеціальні діаграми

Спеціальні діаграми характеризуються тим, що найчастіше служать для доповнення якої-небудь загальної діаграми, наприклад, є її частковим випадком або ж просто грають допоміжну роль, уточнюючи деякі деталі.

5.5.1. Діаграма об’єктів

Діаграма об’єктів (object diagram) – є екземпляром діаграми класів.

На діаграмі об’єктів (рис. 5.22) застосовують один основний тип сутностей: об’єкти 1 (екземпляри класів), між якими вказуються конкретні зв’язки 2 (найчастіше екземпляри асоціацій).

Діаграми об’єктів мають допоміжний характер – по суті це приклади (можна сказати, дампи пам’яті), що показують, які є об’єкти і зв’язки між ними в деякий конкретний момент функціонування системи.

Основні елементи нотації, вживані на діаграмі об’єктів, показані нижче, а детальний опис приведений в підрозділі 5.6.

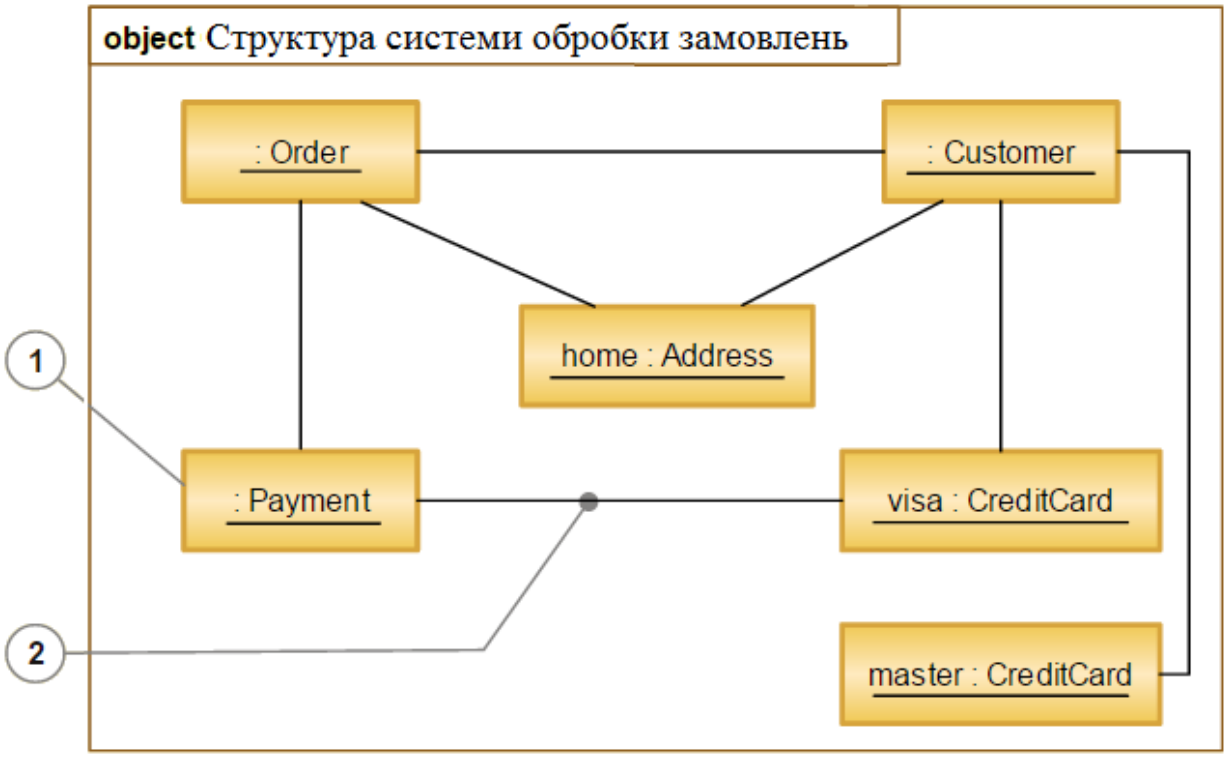


Рисунок 5.22 – Нотація діаграми об’єктів

5.5.2. Діаграма внутрішньої структури

Діаграма внутрішньої структури (composite structure diagram) використовується для детальнішого представлення структурних класифікаторів, передусім класів і компонентів.

Структурний класифікатор (рис. 5.23) зображається у вигляді прямокутника 1, у верхній частині якого знаходиться ім’я класифікатора 2. Усередині знаходяться *частини (parts)* 3. Частин може бути декілька. Кожна частина є екземпляром деякого іншого класифікатора. Частини можуть взаємодіяти одна з одною. Це позначається за допомогою *з’єднувачів (connectors)* 4 різних видів. Місце на зовнішній межі частини, до якої приєднується з’єднувач, називається *портом (port)* 5. Порти розташовуються також на зовнішній межі структурного класифікатора 6, забезпечуючи йому зв’язок із зовнішнім світом.

Детальний опис усіх варіацій діаграм внутрішньої структури наведений в підрозділі 5.6, а на рисунку 5.23 показані тільки основні елементи нотації.

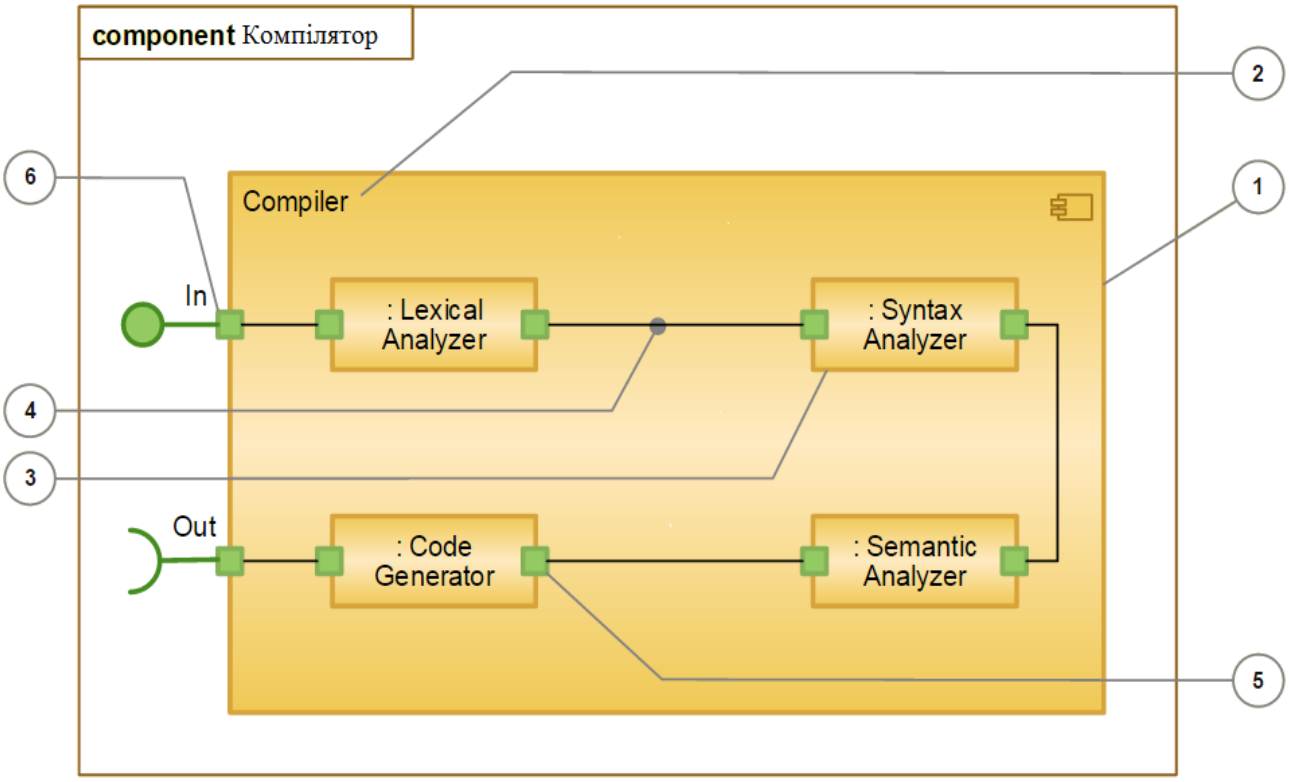


Рисунок 5.23 – Нотація діаграми внутрішньої структури

5.5.3. Оглядова діаграма взаємодії

Оглядова діаграма взаємодії (interaction overview diagram) є різновидом діаграми діяльності з розширеним синтаксисом. Елементами оглядової діаграми взаємодії можуть виступати *посилання на взаємодії* (interaction use) 1, що визначаються діаграмами послідовності (рис. 5.24)**.** Основні елементи нотації показані нижче, а детальний опис наведений в розділі 5.7.

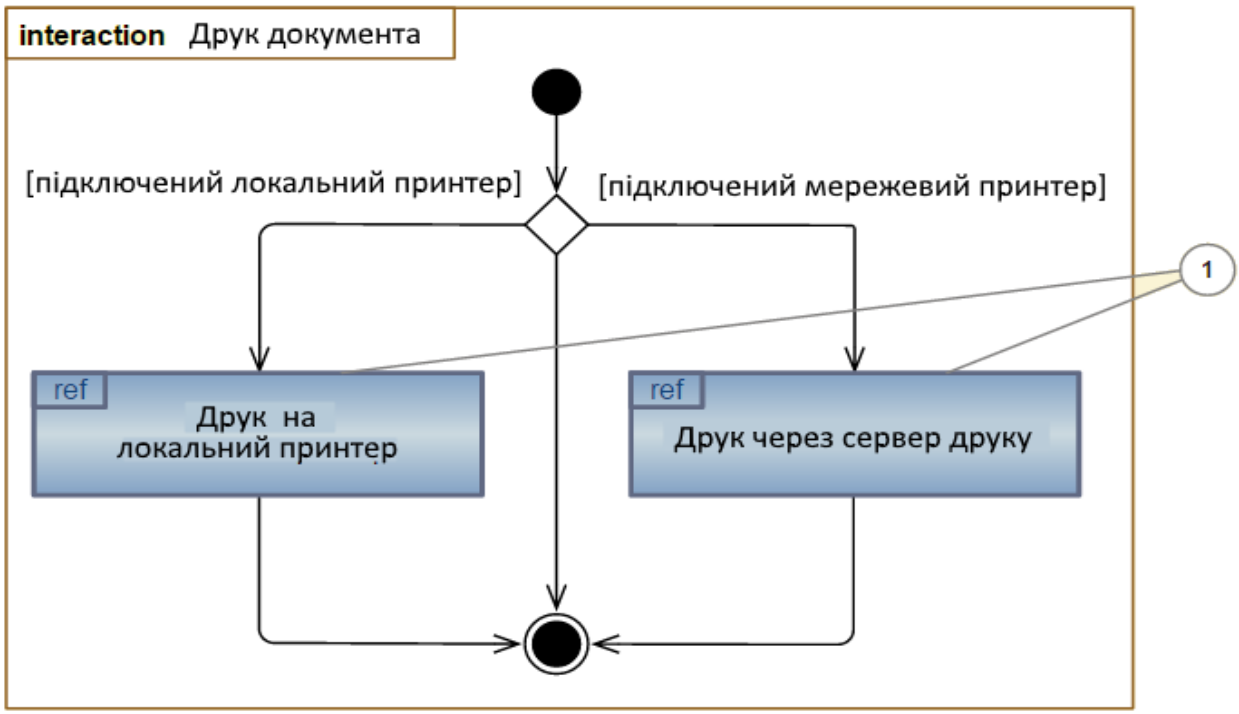


Рисунок 5.24 – Нотація оглядової діаграми взаємодії

5.5.4. Діаграма синхронізації

Діаграма синхронізації (timing diagram) є особливою формою діаграми послідовності, на якій особлива увага приділяється зміні станів 1 різних екземплярів класифікаторів і їх тимчасової синхронізації 2 (рис. 5.25)**.**

Основні елементи нотації показані нижче, а детальний опис наведений в розділі 5.7.

Рисунок 5.25 – Нотація діаграми синхронізації

5.5.5. Діаграма пакетів

Діаграма пакетів (package diagram) – засіб групування елементів моделі.

Діаграма пакетів (рис. 5.26) – єдиний засіб, що дозволяє управляти складністю самої моделі. Основні елементи нотації – пакети 1 і залежності з різними стереотипами 2, що вживаються на діаграмі, показані на рис. 5.26.

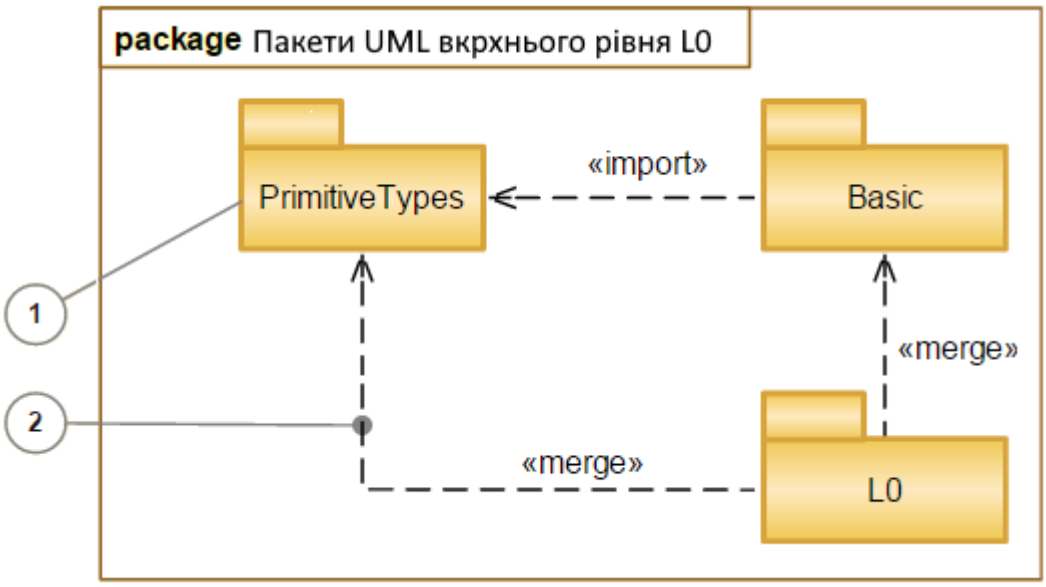


Рисунок 5.26 – Нотація діаграми пакетів

5.6. Моделі і їх представлення

Як показує практичний досвід, починаючи з деякого порогу складності, не вдається описати з єдиної точки зору усі без виключення аспекти модельованої системи. Дійсно, в моделі треба відобразити багато речей: інтерфейси для взаємодії із зовнішнім світом, внутрішню логічну структуру програми, структуру даних, що зберігаються, алгоритми функціонування і багато що інше.

Звідси слідує висновок: **моделювати складну систему слід з декількох різних точок зору**, кожного разу зважаючи на один аспект модельованої системи і абстрагуючись від інших. Ця теза є одним із засадничих принципів UML, може бути найважливішим принципом, що зумовив практичний успіх UML.

Ідея полягає в тому, що абстрактний граф моделі, що складається з безлічі різнотипних сутностей і відношень, не підлягає конструюванню або вивченню в цілому. Кожного разу для візуалізації, зміни або інших маніпуляцій з цього загального графа вичленяють тільки сутності і відношення, релевантні для певного аспекту модельованої системи, а усі інші ігноруються. Такий вид з певної точки зору проекції моделі називають ***представленням*** *(view)* **–** засобом логічної структуризації моделі**.**

5.6.1. Призначення і рівні моделей

Моделі UML можуть створюватися і використовуватися з різними цілями. Іноді буває навіть так, що автор створював модель з однією певною метою, а використовується вона іншими людьми абсолютно несподіваним для автора чином. Таким чином, призначення моделі не є чимось постійним і важко змінюваним. Трансформації призначення моделей цілком можливі, але, проте, практика моделювання підказує, що моделі дають більший ефект, якщо при моделюванні брати до уваги призначення моделі.

Призначення моделей може бути різним, усі випадки описати неможливо. Зазвичай пропонують таку класифікацію призначень моделі, засновану на відповіді на просте питання «а що з цією моделлю трапиться потім?», яке автор моделі повинен задавати собі під час моделювання. Розглядаємо три типові варіанти відповіді.

1. Концептуальна модель (conceptual model). На діаграми такої моделі дивитимуться, їх обмірковуватимуть, але з самою моделлю нічого робити не будуть. Це не означає, що модель не потрібна – це означає, що модель використовується тільки для управління розумовим процесом, для розуміння. Тому такі моделі називають концептуальними (також застосовуються терміни ***модель аналізу*** (analysis model) або ***аналітична модель***).

Такий тип використання моделей один з найважливіших, оскільки так використовуються моделі, які виходять в результаті аналізу предметної області. Концептуальні моделі досить стабільні: якщо не міняється предметна область, то немає потреби міняти і модель. Головна вимога до моделі предметної області: ***концептуальна цілісність***.

2. Модель проектування. Модель проектування є високорівневим (на рівні підсистем) і низькорівневим (на рівні класів, якщо йдеться про використання ООП) описом програмної системи. Модель проектування призначена для того щоб, керуючись нею, розробити програмний код додатка.

Як правило, архітектура (високорівневий опис) і код розробляються ітеративно, і розробникам в процесі розробки необхідно модифікувати модель проектування, щоб вона відповідала рішенням, що приймалися. Головна вимога до моделі проектування – ***зрозумілість*** (usability). Дійсно, розробники повинні повністю розуміти модель, щоб вести розробку.

3. Модель реалізації (implementation model). Модель реалізації призначена для автоматичного перетворення в істотно інший вид, наприклад, в здійснимий код. Таке призначення вимагає вказівки необхідних деталей реалізації в моделі, оскільки «від себе» комп'ютер їх додати не зможе. Головна вимога до моделей реалізації: ***повнота*** (completeness).

Ще раз підкреслимо, що між моделями різного призначення немає непереборного бар'єру, але відмінності все-таки є. На рис. 5.27 наведемо приклад, де розглянуті відношення між авторами «Author» і книгою «Book».

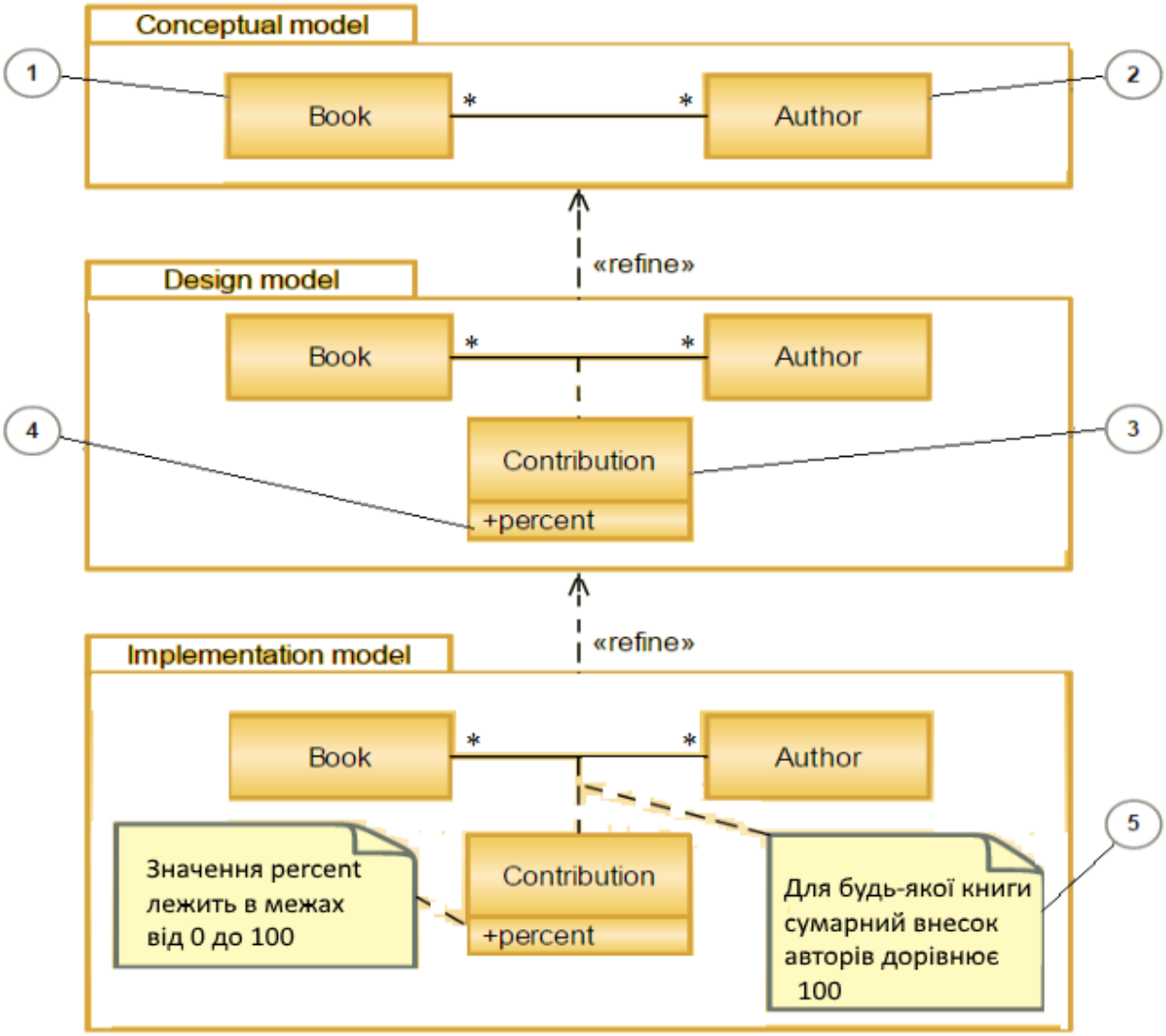


Рисунок 5.27 – Концептуальна модель, модель проектування і модель реалізації

З концептуальної точки зору книга 1 – це тип видання, який має авторів 2, причому співавторів у книги може бути декілька і кожен може бути автором декількох книг. Важливо також врахувати, що в створення книги кожен співавтор вніс певний вклад 3. У моделі проектування необхідно уточнити, що абстрактний творчий вклад вимірюється в конкретних відсотках 4, а в моделі реалізації додати, що необхідно перевіряти інваріантне (що залишається незмінним при тих або інших перетвореннях) співвідношення 5, що полягає в тому, що для кожної книги сума вкладів усіх її авторів має бути рівна 100%.

В якості доповнення до моделі реалізації можна навести пояснюючу діаграму об'єктів (рис. 5.28), де вказати конкретну книгу 1, конкретних авторів 2 і конкретний вклад кожного з авторів 3.

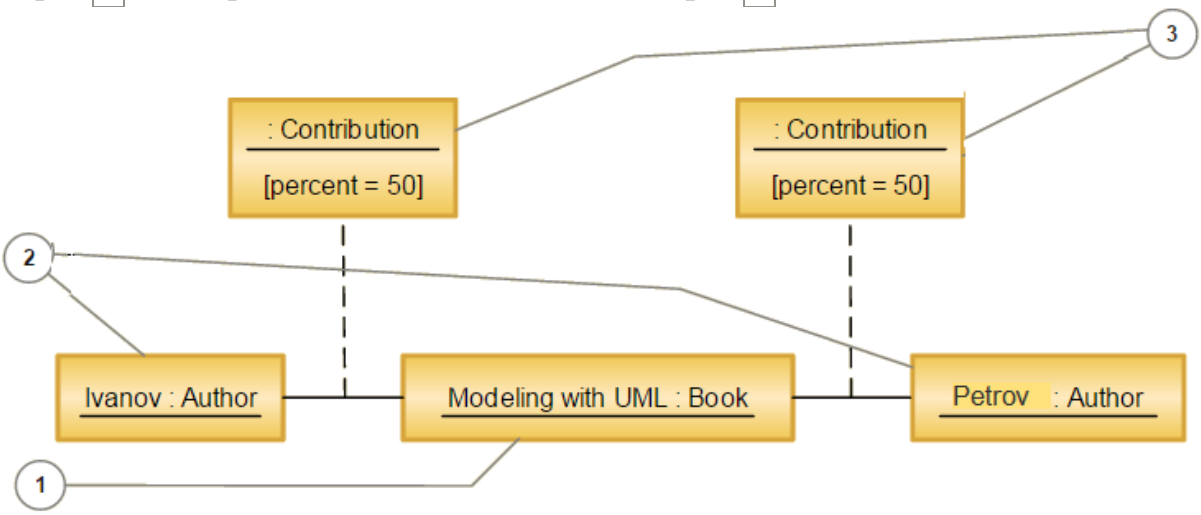


Рисунок 5.28 – Пояснююча діаграма об'єктів

5.6.2. Класичні представлення з UML 1 і 2

Набір використовуваних представлень (видів) моделі є ще менш формальним, чим набір канонічних діаграм і класифікація призначення і рівня моделей.

Одним з найпопулярніших є набір представлень, описаних авторами мови в UML 1 і показаних на рис. 5.29.

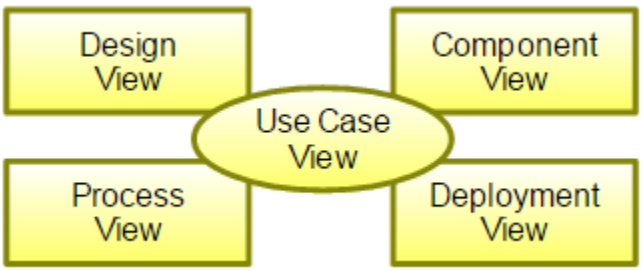


Рисунок 5.29 – Представлення з UML 1

1. Представлення використання (Use Case View) – цей опис поведінки системи в термінах варіантів використання з точки зору зовнішніх стосовно

системи дійових осіб. Це представлення описує не те, як організована система, а ті функціональні вимоги, яким вона повинна задовольняти. При цьому структурні аспекти передаються *діаграмами використання, а поведінкові аспекти – діаграмами взаємодії, станів і діяльності*.

2. Представлення проектування (Design View) призначене для опису словника предметної області, тобто, в парадигмі об'єктно-орієнтованого програмування, класів, а також таких допоміжних сутностей як, наприклад, інтерфейси або кооперації. Структурні аспекти передаються *діаграмами класів і об'єктів,* а поведінкові аспекти – *діаграмами взаємодії, станів і діяльності*.

3. Представлення процесів (Process view) – цей опис взаємодії елементів управління (процесів, потоків) під час роботи системи. Воно відбиває такі нефункціональні вимоги, як, наприклад, забезпечення паралелізму. Структурні аспекти передаються за допомогою концепції *активних класів,* що представляють процеси і потоки, а поведінкові аспекти – *діаграмами взаємодії, станів і діяльності.*

4. Представлення компонентів (Component view) – цей опис системи на рівні артефактів (компонентів, файлів тощо), використовуваних для складання, випуску, конфігурації програмного продукту. Структурні аспекти передаються *діаграмами компонентів,* а поведінкові аспекти – *діаграмами взаємодії, станів і діяльності*.

5. Представлення розміщення (Deployment view) відбиває топологію зв'язків апаратних засобів і розміщення на них компонентів. Структурні аспекти передаються *діаграмами розміщення,* а поведінкові аспекти – *діаграмами взаємодії, станів і діяльності*.

Первинні п'ять представлень, що асоціюються з UML 1, при переході до UML 2 були доповнені і в результаті утворили набір вже з восьми представлень.

1. Статичне представлення (Static view) відображає статичну структуру системи і не описує динаміку у будь-якому її прояві. Найчастіше статичне представлення відбиває логічні концепції додатка, основою якого служать класи і їх відношення.

2. Представлення проектування (Design view) є деталізованішим варіантом статичного представлення, виділяючи класифікатори, що забезпечують необхідну функціональність системи.

3. Представлення використання (Use Case view) описує функціонування системи в термінах варіантів використання і розглядає їх з точки зору зацікавлених дійових осіб.

4. Представлення автоматів (State machine view) специфікує поведінку окремих елементів, для яких можна ввести поняття життєвого циклу, що описується набором станів і переходів між ними.

5. Представлення діяльності (Activity view) описує функціонування системи з точки зору різних елементів діяльності, сполучених потоками управління і потоками даних.

6. Представлення взаємодії (Interaction view) відбиває послідовність обміну повідомленнями між елементами системи під час виконання додатка.

7. Представлення розгортання (Deployment view) описує розміщення артефактів на обчислювальних вузлах під час виконання додатку, а також компоненти і інші елементи, маніфестацією яких є ці артефакти.

8. Представлення управління моделлю (Model Management view) відбиває внутрішню організацію моделі, описуючи її розбиття на пакети і вказуючи відношення між ними.

Враховуючи неформальний характер самого поняття представлення, і спираючись на досвід використання UML, більшість авторів пропонують свій варіант набору представлень. Так, авторами книги [12] пропонується тільки три представлення.

1. Представлення використання. По суті це те ж саме представлення, що було вказано вище. Представлення використання покликане відповідати на питання, **що робить система корисного.** Визначальною ознакою для віднесення елементів моделі до представлення використання являється явне зосередження уваги на факті наявності у системи зовнішніх меж, тобто виділення зовнішніх дійових осіб, що взаємодіють з системою, і внутрішніх варіантів використання, що описують різні сценарії такої взаємодії. Таким чином, єдиним виразним засобом представлення використання виявляються *діаграми використання*.

2. Представлення структури. Представлення структури покликане відповідати (з різною мірою деталізації) на питання: **з чого складається система.** Визначальною ознакою для віднесення елементів моделі до представлення структури являється явне виділення структурних елементів – складових частин системи – і опису взаємозв'язків між ними. Принциповим є чисто статичний характер опису, тобто відсутність поняття часу у будь-якій формі, зокрема, у формі послідовності подій і/або дій. Представлення структури описується, передусім, і головним чином *діаграмами класів,* а також, якщо потрібно, *діаграмами компонентів*, *розміщення, внутрішньої структури* і, в окремих випадках, *діаграмами об'єктів*.

3. Представлення поведінки. Представлення поведінки покликане відповідати на питання: **як працює система.** Визначальною ознакою для віднесення елементів моделі до представлення поведінки являється явне використання поняття часу, зокрема, у формі опису послідовності подій/дій, тобто у формі алгоритму. Представлення поведінки описується *діаграмами автомата* і *діяльності*, а також оглядовою *діаграмою взаємодії, діаграмами комунікації* і *послідовності*.

Такий набір представлень є погодженим з класифікацією діаграм. Більше того, процес моделювання (незалежно від призначення моделі) є не лінійним послідовним, а ітеративним і паралельним, приблизно таким, як показано на рис. 5.30.

Іншими словами, **процес моделювання циклічний**, на кожному кроці може бути присутнім для уточнення представлення використання, за яким йде паралельне моделювання структури і поведінки.

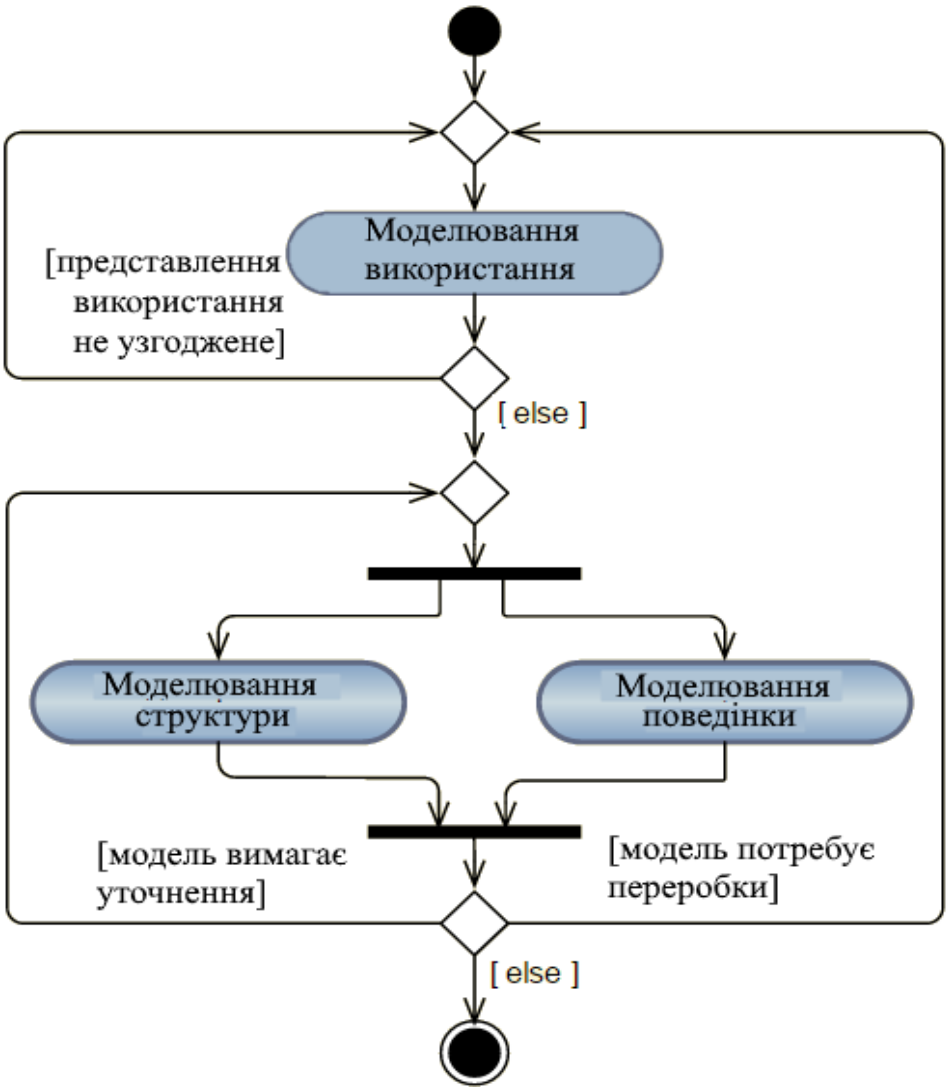


Рисунок 5.30 – Процес моделювання

5.7. Механізми розширення

Механізми розширення **–** це вбудований у мову спосіб її змінити. Автори UML при уніфікації різних методів постаралися включити в мову якомога більше різних засобів (утримуючи об'єм мови у рамках розумного), так щоб мова виявилася застосовною в різних контекстах і предметних областях. Але не можна обійняти неосяжного! Цілком можуть виникати і виникають випадки, коли стандартних елементів моделювання бракує або вони не цілком адекватні.

Механізми розширення дозволяють визначати нові елементи моделі на основі існуючих керованим і уніфікованим способом. Таких механізмів три: *помічені значення*; *обмеження*; *стереотипи*.

Ці механізми не є незалежними – вони тісно пов'язані між собою.

Помічене значення **–** це пара: ім'я властивості і значення властивості, яка може бути додана до будь-якого стандартного елементу моделі.

Іншими словами, помічене значення – це властивість, що додається до представлення моделі. До будь-якого елементу моделі можна додати будь-яке помічене значення, яке зберігатиметься так само, як і усі стандартні властивості

цього елементу. Спосіб обробки поміченого значення, визначеного користувачем, стандартом не описується і віддається на відкуп інструменту.

Помічені значення записуються в моделі у вигляді рядка тексту, що має наступний синтаксис: у фігурних дужках вказується пара: ім'я і значення, розділені знаком рівності. На рис. 5.31 наведений приклад використання помічених значень.

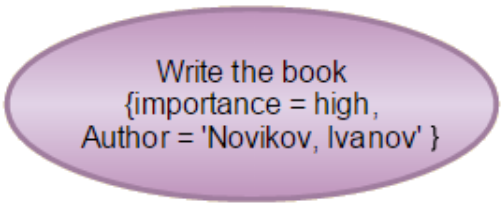


Рисунок 5.31 – Приклад використання помічених значень

Обмеження – це логічне твердження стосовно значень властивостей елементів моделі.

Логічне твердження може мати два значення: істина або хиба, тобто умова, що задається ним, або виконується, або не виконується. Вказуючи обмеження для елементу моделі, ми змінюємо його семантику, вимагаючи, щоб обмеження виконувалося. Обмеження може належати до окремого елементу або до сукупності елементів моделі.

Обмеження записуються у вигляді рядка тексту, поміщеного у фігурні дужки. Це може бути неформальний текст природною мовоюі, логічний вираз мови програмування, що підтримується інструментом, або вираження на деякій іншій формальній мові.

Аналогічно поміченим значенням, обмеження можна написати після імені елементу або в окремому коментарі, приєднаному до елементу. На рис. 5.32 наведений приклад використання обмеження.

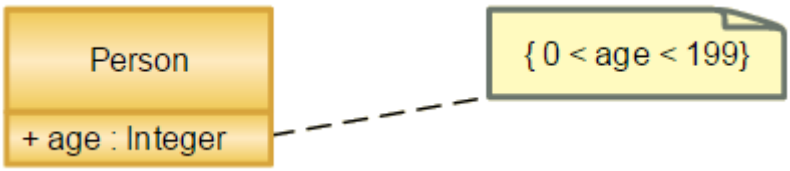


Рисунок 5.32 – Приклад використання обмеження

Для помічених значень і обмежень одна і та ж синтаксична конструкція – текст у фігурних дужках – використовується не випадково. Насправді помічене значення можна вважати обмеженням. А саме, якщо в моделі вказано помічене значення {A = B}, те це можна розглядати як запис умови: «властивість A обов'язково має значення B».

Стереотип – це визначення нового елементу моделювання на основі існуючого.

Визначення стереотипу робиться таким чином. Узявши за основу деякий існуючий елемент моделі, до нього додають нові помічені значення (розширюючи тим самим внутрішнє представлення), нові обмеження (змінюючи семантику) і доповнення, тобто нові графічні елементи.

Після того як стереотип визначений, його можна використати як елемент моделі нового типу. Якщо при створенні стереотипу не використовувалися доповнення і графічна нотація узята від базового елементу моделі, на основі якого визначений стереотип, то стереотип елементу позначається за допомогою імені стереотипу, поміщеного в подвійні кутові дужки (друкарські лапки «»), яке поміщається перед ім'ям елементу моделі. Якщо ж для стереотипу визначена своя нотація, наприклад, новий графічний символ, то вказується цей символ. Втім, можна вказати як ім'я стереотипу, так і графічний символ (значок). На рис. 5.33 наведений приклад визначення і використання стереотипу «PowerUser», який ми визначили на базі стандартного стереотипу дійової особи.

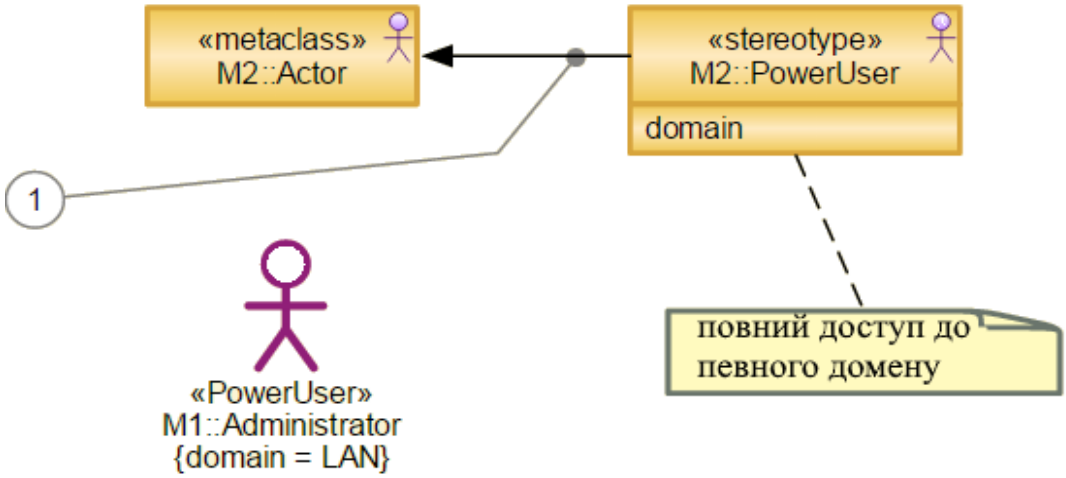


Рисунок 5.33 – Приклад використання стереотипу

Відношення між елементом моделі, який узятий за основу M2::Actor і новим елементом моделі M2::PowerUser називається відношенням ***розширення*** *(extension)* [1]. У UML є велика кількість зумовлених стереотипів. Стереотипи використовуються дуже часто, тому приклади їх застосування розосереджені по всьому посібнику. Стереотипи є потужним механізмом розширення мови, проте ними слід користуватися помірно.

Контрольні питання до розділу 5

1. Дайте визначення поняттю *моделі* програмної системи.
2. Які різновиди моделей ви знаєте?
3. Призначення UML
4. Дайте визначення поняттю *специфікація*.
5. Перерахуйте різні способи використання UML.
6. Які основні графічні елементи UML ви знаєте?
7. Перерахуйте способи використання UML.
8. Що таке *модель UML?*
9. З яких трьох «будівельних блоків» складається UML?
10. Які чотири сутності в UML ви знаєте?
11. Які чотири основні типи відношень в UML ви знаєте?
12. Перерахуйте 9 канонічних типів діаграм в UML 1.

РОЗДІЛ 6. ПРАКТИКА ЗАСТОСУВАННЯ UML

6.1. Рівні моделювання

Перше питання, яке належить поставити перед собою приступаючи до застосування UML: чого ми хочемо досягти? Тут можливі варіанти. Розглянемо три з них, найбільш зрозумілі і часто вживані:

1) концептуальне моделювання;
2) специфікація вимог (закінчений опис поведінки системи);
3) детальне проектування.

Концептуальне моделювання. Для концептуального моделювання повні діаграми використання обов'язкові. Також бажані і корисні діаграми класів (з мінімумом подробиць) в об'ємі словника предметної області, діаграма компонентів або розміщення для перерахування основних артефактів і, можливо, діаграми взаємодії або діяльності для типових сценаріїв основних варіантів використання.

В деяких випадках (тільки у тих, коли це істотно для розуміння логіки роботи додатка) виявляються корисними діаграми станів. Дуже важливо не забути згадати в коментарях ті ключові аспекти, які не відбиті в графічній нотації. Головні вимоги до концептуальної моделі – осяжність і узгодженість. Діаграм не повинно бути багато, і вони не мають бути складними.

Специфікація вимог. У другому варіанті застосування UML основною метою моделювання є отримання артефакту, який міг би послужити технічним завданням на розробку.

Технічне завдання повинне визначати вимоги до системи і забезпечувати ефективну перевірку того, що вимоги дійсно виконані. Так само, як і у разі концептуального проектування, основою специфікації вимог є діаграми використання, але їх детальність і повнота мають бути істотно вищі.

Реалізація варіантів використання у формі діаграм взаємодії обов'язкова – це основа для побудови набору тестів приймально-здавальних випробувань. На діаграми використання покладається велика відповідальність, тому розробляти їх необхідно набагато ретельніше. Наприклад, якщо на діаграмі не зображений деякий варіант використання, то це означає, що система не повинна мати відповідної функціональності. У специфікації вимог діаграма компонентів не просто бажана, а обов'язкова – технічне завдання повинне визначати, хоч би на рівні назв, що саме має бути розроблене.

Між концептуальною моделлю і специфікацією вимог немає непрохідної межі – досить часто концептуальна модель за декілька ітерацій переростає в специфікацію вимог. Взагалі-то, задовільна специфікація вимог виявляється в 3-5 разів об'ємнішою концептуальної моделі.

Детальне проектування. Третій варіант застосування UML вимагає певної рішучості і, одночасно, тверезого розрахунку. Детальне проектування націлене вже не стільки на опис, скільки на конструювання артефактів системи, що розробляється. Модель детального проектування служить основним

документом, яким керуються програмісти при кодуванні, тестувальники при перевірці, а технічні «письменники» – при складанні програмної документації для системи.

Якщо немає упевненості в здатності команди побудувати і використати хороший детальний проект виключно засобами UML, то не варто витрачати на нього ресурси. Краще не мати ніяких детальних діаграм, чим мати погані діаграми. Вважається, що помилка, допущена в детальній діаграмі, обходиться у декілька разів дорожче, ніж звичайна помилка в коді.

Детальне проектування у декілька разів об'ємніше, чим специфікація вимог, тому, як правило, до процесу детального проектування залучаються декілька чоловік, тобто загострюються проблеми цілісності моделі, узгодженості позначень, збереження артефактів.

У моделі детального проектування, як правило, використовується велика частина елементів мови UML. Особливе значення мають діаграми класів, які мають бути визначені з максимальною мірою детальності. Кількість класифікаторів зростає у декілька разів в порівнянні з концептуальною моделлю. Окрім основних сутностей із словника предметної області треба визначити типи даних, інтерфейси, сигнали, виключення тощо. Як правило, сама модель має складну структуру і без використання діаграм пакетів також не обійтися.

Наведена класифікація рівнів моделювання не претендує на універсальність. У літературі описані і інші варіанти, наприклад, застосування UML для опису бізнес-процесів, у тому числі процесу розробки програмного забезпечення. Але для розробки, наприклад, офісних додатків вказані способи застосування UML є типовими.

6.2. Практика застосування елементів UML

UML є результатом зведення воєдино і уніфікації десятків мов моделювання, що існували в 90-і роки XX століття. Побічним ефектом такої уніфікації стало включення в мову величезного числа різноманітних елементів на усі випадки життя. Повний об'єм специфікації UML складає більше 1000 сторінок. Таким величезним об'ємом інформації важко оперувати, не кажучи вже про те, що його просто нелегко засвоїти.

На щастя, є підстави стверджувати, що для ефективного використання UML досить вивчити цілком осяжну підмножину мови. Така основа дає ***принцип Парето***, також відомий як *принцип 80/20.* Принцип свідчить, що **невелика доля (20%) причин, засобів, що вкладаються, або зусиль, що докладаються, відповідає за велику частку (80%) результатів, отримуваної продукції або зароблені винагороди.**

Виявлення тих 20% конструкцій UML, яких виявиться досить в 80% випадків, є привабливим завданням. У його розв'язанні зацікавлені як розробники програмного забезпечення, що використовують UML за службовим обов'язком, так і викладачі вищих навчальних закладів, що читають студентам

лекції з UML. Далі наведений огляд трьох досліджень, спрямованих на виділення «ядра» UML, і деякі висновки з цього питання.

Одна з перших спроб зрозуміти, як використовується UML на практиці, і тим самим хоч трохи навести порядок у великій кількості конструкцій мови, проводилася шляхом опитування професійних розробників про їх досвід використання UML1 [45]. Питання стосувалися канонічних діаграм UML.

Опитування показало, що **найчастіше розробники використовують діаграми класів.** 73% респондентів постійно використовували діаграми класів. **Найрідше використовуються діаграми кооперації**. Тільки 20% опитаних постійно стикаються у своїй роботі з цим типом діаграм.

Список діаграм, впорядкований за інтенсивністю використання, згідно з результатами дослідження, виглядає таким чином:

- діаграма класів;
- діаграма використання;
- діаграма послідовності;
- діаграма діяльності;
- діаграма станів;
- діаграма кооперації.

Інші діаграми UML1 – діаграми об’єктів, компонентів і розгортання – виявилися за рамками дослідження як «менш релевантні питанню», що вивчалося. Мабуть, вони вважалися найменш корисними.

Ці перші опитування показали значний розкид в даних, що свідчить про те, що в різних сферах застосування одні і ті ж конструкції мови мають різну вагу і різну цінність. Різні люди застосовують UML по-різному.

Вичерпним чином охарактеризувати предметні сфери застосування UML не так просто. Мабуть, сказати: «за таких-то умов застосовувати UML треба обов’язково, а за інших умов не можна не застосовувати UML ні в якому разі» – було б неправильно. Немає таких об’єктивних умов і критеріїв. Проте, є виразне суб’єктивне відчуття, що в одних випадках застосування UML виявляється зручним, приємним і ефективним, а в інших – важким, обтяжливим і даремним. Але чию суб’єктивну думку слід взяти до уваги в першу чергу? Звичайно, думка авторів мови!

Аналіз творів авторів UML, що відносяться до UML, показав, що автори вибирають у своїх книгах приклади з трьох класів предметних областей.

По-перше, це ***інформаційні системи управління підприємством***.

По-друге, ***додатки реального часу і вбудовані системи*** – наприклад, банкомат, бортова система автомобіля і тому подібне

По-третє, ***клієнт-серверні системи масового обслуговування, або веб-додатка***.

Порядок перерахування відповідає частоті прикладів, що зустрічаються. Чи означає це, що UML неможливо застосувати, наприклад, в програмному забезпеченні інженерних розрахунків? З одного боку, напевно, можна, а з іншого боку, таких прикладів або немає, або їх мало.

У статті «Чи може UML стати простим?» [46] її автори розповідають про своє дослідження з виявлення ядра UML, мінімального за об'ємом, але такого, що вирішує основні завдання користувачів. До дослідження були залучені експерти з різних областей індустрії розробки програмного забезпечення. Учасників просили оцінити важливість кожного типу діаграм UML1 для їх роботи. При цьому диференціювалася предметна область: інформаційні системи масштабу підприємства, системи реального часу і веб-додатка.

У таблиці. 6.1 наведені списки канонічних діаграм, впорядкованих за їх важливістю в різних предметних областях: від найважливішої діаграми до самої «неважливої».

Таблиця 6.1 – Важливість діаграм за предметними областями

Системи управління підприємством	**Системи реального часу**
1) діаграма класів; **2) діаграма використання;** **3) діаграма послідовності;** **4) діаграма діяльності;** 5) діаграма розміщення; 6) діаграма компонентів; 7) діаграма станів; 8) діаграма кооперації; 9) діаграма об'єктів.	**1) діаграма класів;** **2) діаграма станів;** **3) діаграма послідовності;** **4) діаграма використання;** 5) діаграма компонентів; 6) діаграма діяльності; 7) діаграма розміщення; 8) діаграма об'єктів; 9) діаграма кооперації.
Веб-додатки	**Усі системи**
1) діаграма класів; **2) діаграма використання;** **3) діаграма послідовності;** **4) діаграма станів;** 5) діаграма компонентів; 6) діаграма розміщення; 7) діаграма діяльності; 8) діаграма кооперації; 9) діаграма об'єктів.	**1) діаграма класів;** **2) діаграма використання;** **3) діаграма послідовності;** **4) діаграма станів;** 5) діаграма компонентів; 6) діаграма діяльності; 7) діаграма розміщення; 8) діаграма кооперації; 9) діаграма об'єктів.

Перші чотири діаграми в кожному з приведених списків, що виділені жирним шрифтом, віднесені до ядра UML у відповідній предметній області.

Можна помітити, що *діаграми класів, використання і послідовності* увійшли до всіх чотирьох ядер, що говорить про їх універсальну важливість. Це спостереження дало авторам статті підставу запропонувати в якості ядра UML1 три перераховані діаграми і пов'язані з ними конструкції.

У статті Станіслава Врича і Бартоша Марчинковськи [47] пропонується інший підхід до виділення ядра UML, що називається авторами «UML Light». У своєму дослідженні автори спираються на досвід викладання UML студентам.

У результаті було виявлено, що, на думку 57% респондентів, в UML надто багато типів діаграм (у UML2 їх 13).

Чотирма найкориснішими діаграмами, за результатами опитування, виявилися:

– діаграма класів;
– діаграма використання;
– діаграма діяльності;
– діаграма послідовності.

Саме вони і запропоновані авторами в якості UML Light.

В той же час діаграма послідовності була визнана студентами найобтяженішою різними елементами і позначеннями.

Найзрозумілішими і простішими у використанні визнані ***діаграми використання, класів і діяльності***. Найзаплутанішими і складнішими у використанні показалися оглядова діаграма взаємодії, діаграма розміщення і діаграма внутрішньої структури.

Як показують ці і інші дослідження, діаграма класів і діаграма використання однозначно ідентифікуються в якості основних і найкорисніших діаграм UML. Щодо включення в ядро діаграми автомата, діаграми послідовності або діаграми діяльності існують розбіжності, але одна з цих діаграм обов'язково включається.

Перераховані діаграми відбивають усі три принципові аспекти моделювання (***використання, структура, поведінка),*** і у більшості випадків можна обійтися тільки трьома діаграмами: ***використання, класів*** і якою-небудь ***діаграмою поведінки***.

Три діаграми якраз і складають 20% UML (3/13=0,23)**, достатні для 80% випадків.** Причому висновок стосовно вибору *діаграми поведінки* такий:

– у додатках реального часу – ***діаграма автомата***;
– у автоматизованих системах – ***діаграма діяльності***;
– у веб-додатках – ***діаграма послідовності***.

Виділення ядра UML не означає, що інші діаграми і пов'язані конструкції потрібно виключити з мови – їх можна вважати спеціалізованими розширеннями мови і використати при необхідності.

6.3. Вплив UML на процес розробки

UML не є моделлю процесу розробки, але це не означає, що UML ніяк не пов'язана з процесом розробки.

У попередньому розділі ми непрямим чином порушили питання про те, як впливає систематичне застосування UML на процес розробки. Дійсно, UML дозволяє наочно описувати зразки проектування, застосування зразків проектування підвищує якість проектування архітектури, що, на загальну думку, прискорює реалізацію.

У цьому розділі ми сформулюємо і інші думки з приводу того, в чому полягає основний чинник впливу UML на процес розробки. Але для цього нам знадобиться ввести в розгляд деякі терміни і поняття, окремі з яких розглядалися детально раніше, після чого ми сформулюємо основну тезу впливу застосування UML на процес розробки.

6.3.1. Технологія програмування

Технологія програмування – дисципліна, яка вивчає процеси розробки програмного забезпечення, англійською мовою називається Software Engineering. Для позначення цієї дисципліни використовується сталий вітчизняний термін «***технологія програмування».***

Технологія програмування є інженерною дисципліною, що входить в обов'язковий набір знань і умінь кожного інженера, причетного до створення і експлуатації програмного забезпечення комп'ютерів.

Технологія програмування має чітко виділений об'єкт вивчення – процеси розробки і супроводу програмного забезпечення, але нині не має єдиного методу і загальноприйнятого способу побудови.

Технологія програмування не є строгою математичною дисципліною, яку можна викласти послідовно, починаючи із засадничих понять і застосовуючи дедуктивні докази. Навпаки, технологія програмування є збіркою різнорідних і часто неузгоджених одна з одною моделей, методик і засобів. Дамо (нагадаємо) декілька визначень.

Програмування – це процес створення програмістом (людиною) програми (інформаційної структури), призначеної для подальшого виконання (комп'ютером).

Таким чином, у процесі програмування є присутніми явно суб'єкт, об'єкт і мета. У типовому (і звичному) випадку суб'єктом є людина, яка веде процес усвідомлено, об'єктом є текст на формальній мові, а метою є таке виконання програми, яке у свою чергу має явно позначений результат.

Технологія програмування – це сукупність методів і засобів, що дозволяють налагодити виробничий процес створення програмного забезпечення.

У цьому визначенні особливо слід підкреслити слово «виробничий», яке відбиває найважливішу особливість технології програмування.

Зазвичай технологію програмування доцільно розглядати в трьох аспектах.

- **Модель процесу**, тобто порядок проведення типового проекту з розробки програмного забезпечення. Сюди належать поняття життєвого циклу програмного забезпечення, визначення моделі процесу – виділення в ньому фаз, потоків робіт і інших складових.
- **Модель команди**, тобто стосунки між людьми в процесі розробки. Сюди належать визначення обов'язків працівників – учасників процесу, регламенти їх взаємодії, робочі процедури тощо. Характерним для цього аспекту є розгляд на рівні групи (команди) або проекту.
- **Дисципліна програмування**, тобто методи створення конкретних артефактів, що входять до складу програмного забезпечення. Сюди належать опис і застосування зразків проектування, стандарти кодування, методи тестування і відладки тощо. Характерним для цього аспекту є розгляд на рівні окремого працівника.

Усі три складові частини технології програмування зазнали бурхливого і нерівномірного розвитку і продовжують постійно мінятися. Для розуміння сучасних віянь в технології програмування необхідно хоч би коротко торкнутися її історії. У історії технології програмування відбувалося багато подій, висувалася і постійно висувається маса нових ідей. Щоб якось систематизувати ці факти, пропонуємо таку класифікацію.

Усі факти історії технології програмування ділимо на три класи, які приблизно відповідають трьом періодам.

Дореволюційний період. З моменту початку промислової розробки програмного забезпечення і до середини шестидесятих років XX століття питання власне технології програмування розглядалися, як правило, не окремо, а в зв'язку і в сукупності з іншими питаннями програмування. Зрозуміло, що технологічні проблеми існували, і пропонувалися методи їх рішення, але це не було предметом публічних громадських дискусій.

Слід підкреслити, що в той час програмування не було масовою професією, і було, в основному, долею талановитих і високоосвічених одинаків, фахівців в тій предметній області, де застосовувалися комп'ютери. Можна сказати, що, хоча програмування і було професійним, воно не було промисловим.

З основних технологічних ідей, що з'явилися в той період, слід зазначити появу мов програмування і компіляторів, а також явне усвідомлення важливості модульного програмування як основи для накопичення бібліотек програм і їх повторного використання.

Революція в програмуванні. До середини шестидесятих років ситуація змінилася. Комп'ютери стали дешевшими, компактнішими і продуктивнішими. Сфера застосування істотно розширилася, вони стали в масовому порядку застосовуватися в промисловості, науці, освіті. Разом із складними і відповідальними програмами з'явилися, і в набагато більшій кількості, звичайні програми для автоматизації виробничої і іншої повсякденної діяльності. Експлуатація програмного забезпечення перестала бути таїнством, доступним тільки посвяченим, програмування стало масовою професією.

Традиційно прийнято вважати, що «першою ластівкою», що започаткувала лавиноподібний процес створення технології програмування, був лист Е. Дейкстри в журнал Communications of the ACM у 1968 році. Очевидно, що лист Дейкстри подіяв як каталізатор, як маніфест – за декілька років були опубліковані, обговорені і практично впроваджені такі фундаментальні ідеї технології програмування:

1. Конструювання програм методом покрокового уточнення.
2. Проектування зверху вниз і від низу до верху.
3. Структурне програмування.
4. Метод хірургічної бригади.
5. Водоспадна модель процесу розробки.
6. Життєвий цикл програмного продукту.

За кожним напрямом традиція називає основоположників (авторів найпомітніших з ранніх публікацій), але фактично ці технологічні ідеї стали

плодами спільних зусиль, сотні людей внесли вагомий внесок в цей стрімкий розвиток.

Післяреволюційний період. Революція була успішною – в історично найкоротші терміни технологія програмування оформилася як інженерна дисципліна, і деякі її положення всюди були впроваджені в практику. Результати проявилися негайно – середня продуктивність праці в програмуванні збільшилася у декілька разів. Підвищилася і надійність, за рахунок розвитку практики систематичного тестування.

Розвиток технології програмування тривав, але вже не революційним, а еволюційним шляхом. Узяті за основу ідеї 60-х років розвивалися і удосконалювалися.

Структурне програмування переросло в об'єктно-орієнтоване, на зміну водоспадним моделям процесу прийшли ітераційні, а метод хірургічної бригади дав початок цілому спектру моделей команди розробників. Але ці підходи не були революційно новими, вони покращували і удосконалювали вже відоме.

Проте час не стоїть на місці. Апаратне забезпечення нестримно прогресує. Сфера застосування комп'ютерів розширюється все більше. Вже зараз без них ніде не можна обійтися, комп'ютери застосовуються всюди.

Найближчим часом комп'ютери з пристроїв, які використовуються дуже часто і у багатьох місцях, перетворяться на пристрої, які використовуються скрізь і завжди. Йдеться про те, що зараз називається вбудованими системами.

Важливо, що при проведенні проектів з розробки програм різних типів доцільно використовувати різні технології. Програму управління ракетою розробляють не так, як інформаційну систему відділу кадрів.

Таким чином, при описі технологій програмування необхідно вказувати типи програмних проектів, для яких ця технологія застосовна, тобто вказувати призначення і сферу застосування технології. Це ж стосується і мови UML, якщо розглядати застосування UML як технологію (частину технології) розробки програмного забезпечення.

6.3.2. Поради з впровадження UML в організації

Дамо декілька простих порад, що ґрунтуються на висновках із зроблених помилок, і які можуть виявитися корисними.

Організація повинна поставити вимірну мету впровадження UML. Багато хто недооцінює важливість постановки саме вимірної мети, і це груба помилка. Не варто впроваджувати UML заради впровадження, тільки заради того, щоб хвалитися потім цим перед колегами. Це нічого хорошого не дасть, наслідки впровадження будуть негативними, а не позитивними. Проектуючи систему рівня інформаційної системи відділу кадрів або вище, корисно зробити приблизний розрахунок трудомісткості.

Треба взяти загальну кількість варіантів використання, помножити на емпіричний коефіцієнт, і вийде приблизна (апріорна, заздалегідь відома) оцінка трудомісткості проекту. Коли проект буде завершений, треба порівняти фактичну трудомісткість з первинною оцінкою.

Цей показник називається точністю апріорної оцінки. Він дуже важливий для планування і управління діяльністю програмуючої організації.

Знати і застосовувати UML повинні усі учасники процесу. Якщо в програмуючій організації є один або два ентузіасти, які намагаються описувати свої рішення на UML, а більшість над ними поблажливо сміється, то таке застосування ні на що не вплине.

Позитивний вплив UML виявляється значним, тільки якщо мова застосовується масовим і систематичним чином. Особливо критичним є чинник залученості вищого керівництва організації в процес впровадження UML. Якщо усе вище керівництво підтримує впровадження особистим застосуванням UML у відповідних випадках – відмінно, мета, напевно, буде досягнута. А якщо серед керівництва є люди, які відмовляються візувати специфікації з діаграмами, то, швидше за все, нічого не вийде.

Інструмент, що підтримує UML, має бути легким у використанні. Саме ця якість, яка англійською мовою називається usability, грає первинну роль при вибиранні інструментальних засобів. Інші якості: ціна, краса оформлення діаграм, потужність алгоритмів генерації коду, відповідність стандартам – усі також важливі, але легкість використання є вирішальною.

Підводячи підсумок цього розділу, можна услід за Фредеріком Бруксом молодшим підтверджувати, що «**срібної кулі» все ще немає** [7], і мова UML не є зараз, і не стане в осяжному майбутньому «срібною кулею». Але UML вже зараз став найпопулярнішим, широко поширеним і дешевим, можна сказати «народним» засобом, що допомагає при специфікації, візуалізації, проектуванні і документуванні усіх артефактів, що виникають при розробці програмних систем, а в майбутньому і вплив UML на процес розробки програмного забезпечення тільки зростатиме.

«Срібної кулі немає» – це Брукс у своїй статті стверджує, що «не в одній технології або в управлінській техніці не існує універсального методу, що збільшує на порядок продуктивність, надійність і простоту». Він також стверджує, що «ми не можемо чекати збільшення прибутку в два рази кожні два роки» при розробці програмного забезпечення, як це відбувається з розробкою апаратного забезпечення.

Контрольні питання до розділу 6

1. Які ви знаєте рівні моделювання за допомогою UML?
2. Про що свідчить *принцип Парето*?
3. Перерахуйте 5 діаграм за інтенсивністю їх використання.
4. Перерахуйте 4 найкорисніші діаграмами, за результатами опитування С. Врича.
5. Назвіть три основні стратегії конструювання моделі процесу.

РОЗДІЛ 7. ОГЛЯД CASE-ЗАСОБІВ ДЛЯ ПОБУДОВИ ДІАГРАМ UML

Вибір CASE-засобу – особиста справа кожного читача. Ми лише спробуємо надати йому цей вибір, розглянувши деякі найбгідніші уваги, з точки зору авторів, CASE-засоби для побудови UML-діаграм. Причому постараємося розповісти і про визнаних лідерів ринку, і про його «аутсайдерів», і про комерційних «монстрів», і про «легкі» програми з відкритим початковим кодом. І почнемо з пакету, що є фактичним стандартом в області UML-проектування.

7.1. IBM Rational Rose

Rational Rose – сучасний і потужний засіб аналізу, моделювання і розробки програмних систем. Rational Rose згодиться при рішенні практично будь-яких завдань проектування інформаційних систем: від аналізу бізнес-процесів до *кодогенерації* на певній мові програмування. Такий арсенал дозволить не лише спроектувати нову систему, але і допрацювати стару, зробивши процес *зворотного проектування*.

Для того щоб якнайповніше покрити увесь сегмент ринку засобів проектування і розробки, випускається декілька версій продукту.

1. Rational Rose Modeler. Ця версія дозволить аналітикам і проектувальникам проводити аналіз бізнес-процесів і проектувати систему. Ця редакція не підтримує *кодогенерацію*.

2. Rational Rose Professional. Як видно з назви, це професійна редакція продукту. Залежно від вибраної мови програмування дозволяє виконувати пряме і *зворотне проектування*. Rose Professional замовляється тільки в певній конфігурації (наприклад, Rose Professional C++ або Rose Professional C++ DataModeler). Rational Rose Professional, звичайно, не створює 100 % виконуваного коду. На виході розробник отримує каркасний код інформаційної системи на певній (замовленій) мові програмування, яку згодом треба ще програмувати і програмувати. Продукт націлений і на аналітиків, і на розробників.

3. Rational Rose RealTime. Версія продукту, створена спеціально для отримання 100% виконуваного коду в реальному масштабі часу. Звичайно, RealTime дозволяє проводити пряме і *зворотне проектування* на мовах C або C++. За завіреннями розробників, на виході модель автоматично компілюється і збирається у виконуваний файл. Продукт призначений саме для розробників.

4. Rational Rose Enterprise. Абсолютно повна версія. Підтримуються усі функції інших редакцій, за винятком можливості 100% *кодогенерацій.* Таким чином, ця версія продукту покриває увесь спектр завдань з проектування, аналізу і кодогенерації. Це програмний пакет для усіх учасників проекту.

5. Rational Rose DataModeler. Це не конкретний варіант продукту, а функціональність з проектування баз даних. Функції DataModeler входять до складу Rose Enterprise або Professional.

На жаль, немає безкоштовної версії продукту, але для освітніх установ усе програмне забезпечення IBM доступно безкоштовно (для використання з навчальною метою) у рамках програми IBM Academic Initiative.

Залежно від постачання, в Rational Rose може бути розширений або звужений набір візуальних компонент (можливих діаграм). Втім, Rational Rose і так досить функціональний. Ось основні можливості продукту:

1. Пряме і *зворотне проектування* на мовах: ADA, Java, C, C++, Basic.
2. Підтримка технологій COM, DDL, XML.
3. Можливість генерації схем БД Oracle і SQL.

Інформацію про продукти Rational можна знайти на сайті: http://www-306.ibm.com/software/rational/ Це офіційний сайт Rational, де ви зможете знайти інформацію про Rational Rose та інші продукти Rational.

7.2. Borland Together

Borland Together ControlCenter – це інтегрована платформа розробки, що дозволяє спростити і прискорити *аналіз, дизайн, розробку* і *розгортання* комплексних корпоративних застосувань. Ці можливості поєднуються в одному інтегрованому рішенні з підтримкою UML, що допомагає командно розробляти високоякісні системи швидше і ефективніше. Ось деякі особливості Borland Together:

1. Підтримка XP («екстремальне програмування»). Together підтримує «гнучкі» процеси моделювання, а також надає інтерактивні можливості моделювання і підтримує усі види діаграм UML.

2. Прискорення процесів розробки шляхом застосування патернів. Ще одна модна тенденція в програмній інженерії: використання патернів, або шаблонів проектування, – деяких стандартних рішень, зразків в області проектування. Використовуючи ці зразки, експерт або розробник можуть швидко створити модель і привести її у відповідність з корпоративними стандартами і кращими практиками кодування.

3. Розгортання додатків на декілька серверів виконується швидко, без перекодування. Додаток можна розгорнути на декілька серверів додатків, просто написавши декілька рядків. З Together ControlCenter додаток може бути побудований для одного сервера додатків і легко перемкнутий на інший, розгорнутий на складній інфраструктурі.

4. Функція контролю якості полегшує життя розробників. Вбудоване *функціональне тестування* допомагає виявити проблеми ще в процесі розробки, що дійсно дуже важливо, оскільки вартість виправлення помилок тим вище, чим пізніше вони виявлені.

З вищесказаного стає ясно, що Borland Together – це щось набагато більше, ніж просто пакет для малювання «картинок в стилі UML».

Виглядає продукт непогано, в кращих традиціях Borland, чиї продукти завжди були ближчі вітчизняним розробникам. Інформацію про продукти Borland Together можна знайти на сайті: http://www.borland.com/together/

7.3. Microsoft Visio

Visio – рішення для побудови діаграм від Microsoft. За словами розробників, Visio допомагає перетворити технічні і бізнес-концепції у візуальну форму.

Образотворчі можливості Visio дійсно дуже широкі.Використовуючи зумовлені фігури Visio Professional, drag – and – drop і майстри, ви можете швидко і просто створювати зрозумілі і інформативні діаграми.

Можливості Visio можна легко розширювати, використовуючи нові шаблони бізнес-діаграм. Ви можете включати зовнішні джерела даних, сховища або колекції шаблонів, що зберігаються.

Visio Professional тісно з Microsoft Office Project, що дозволяє, наприклад, імпортувати звідти інтегрується завдання для членів команди.

За допомогою шаблонів UML ви можете створювати UML-диаграммы статичної структури ПЗ або проводити *зворотне проектування* за допомогою Visio 2003 Reverse Engineer Wizard.

Інформацію про продукти Microsoft Visio можна знайти на сайті: http://office.microsoft.com/en-us/FX 010857981033.aspx

7.4. StarUML, Dia, Draw.io

І нарешті, чудові (і до того ж **абсолютно безкоштовні**) засоби UML-моделювання.

StarUML – це пакет з відкритим програмним кодом, написаний на Delphi і працюючий під управлінням ОС сімейства Windows.

StarUML підтримує UML 2.0 (плюс його профайлы) і MDA (Model Driven Architecture). Функціонал пакету можна розширити за рахунок використання плагінів, так що кожен охочий може створити свій власний модуль для StarUML на будь-якому COM-сумісній мові (C++, Delphi, C#, ...).

Dia – вільний кроссплатформенный редактор діаграм, частина GNOME Office, але може бути встановлений незалежно. Підтримує українську і російську мови. Він може бути використаний для створення різних видів діаграм: блок-схем алгоритмів програм, статичних структур UML, баз даних, діаграм суть-зв'язок, радіоелектронних елементів, потокових діаграм, мережевих діаграм і інших.

На http://younglinux.info/book/export/html/169 є Підручник Dia російською мовою.

Draw.io - інструмент для створення діаграм UML і блок-схем **онлайн.** Нагадує MS Visio і можливо зроблений під нього, але додаток від Microsoft - платна, а онлайн сервіс Draw.io - безкоштовний. Сервіс підтримує декілька мов. Є версія з російським інтерфейсом. Сервіс дуже простий і зручний, інтерфейс приємний і зрозумілий.

З допомогою онлайн сервісу Draw.io можна створювати: Діаграми; Моделювання на UML; Графіки; Блок-схеми; Форми; Вставляти в діаграму зображення. Усі створені діаграми ви можете зберегти на комп'ютер, в Google Drive або Dropbox.

7.5. Основні відомості про CASE-засоби Rational Rose

7.5.1. Введення в Rational Rose

Rational Rose – сімейство об'єктно-орієнтованих CASE-засобів фірми Rational Software Corporation (у 2003 році її поглинула корпорація IBM) – призначене для автоматизації процесів аналізу і проектування програмного хабезпечення, а також для генерації кодів на різних мовах і випуску проектної документації. Rational Rose використовує метод об'єктно-орієнтованого аналізу і проектування, заснований на мові UML.

Поточна версія Rational Rose реалізує генерацію кодів програм для C++, Visual C++, Visual Basic, Java, PowerBuilder, CORBA Interface Definition Language (IDL), генерацію описів баз даних для ANSI SQL, Oracle, MS SQL Server, IBM DB2, Sybase, а також дозволяє розробляти проектну документацію у вигляді діаграм і специфікацій. Крім того, Rational Rose містить засоби реверсного інжинірингу програм і баз даних.

Структура і функції. У основі роботи Rational Rose лежить побудова діаграм і специфікацій UML, що визначають архітектуру системи, її статичні і динамічні аспекти. У складі Rational Rose можна виділити шість основних структурних компонентів: репозиторій, графічний інтерфейс користувача, засоби перегляду проекту (браузер), засоби контролю проекту, засоби збору статистики і генератор документів. До них додаються генератор кодів (індивідуальний для кожної мови) і аналізатор для C++, що забезпечує реверсный інжиніринг.

Засоби автоматичної генерації кодів програм на мові C++, використовуючи інформацію, що міститься в діаграмах класів і компонентів, формують файли заголовків і файли описів класів і об'єктів. Створюваний таким чином скелет програми може бути уточнений шляхом прямого програмування на мові C++.

Аналізатор кодів C++ реалізований у вигляді окремого програмного модуля. Його призначення – створювати модулі проектів Rational Rose на основі інформації, що міститься у визначуваних користувачем початкових текстах на C++. В процесі роботи аналізатор здійснює контроль правильності початкових текстів і діагностику помилок.

Аналізатор має широкі можливості налаштування по входу і виходу. Наприклад, можна визначити типи початкових файлів, базовий компілятор, задати, яка інформація має бути включена у формовану модель, і які елементи вихідної моделі слід виводити на екран. Таким чином, Rational Rose/C++ забезпечує можливість повторного використання програмних компонентів.

У результаті розробки проекту за допомогою CASE-засобу Rational Rose формуються такі документи:

- діаграми UML, що в сукупності є моделлю програмної системи, що розробляється;

- специфікації класів, об’єктів, атрибутів і операцій;
- заготовки текстів програм.

Тексти програм є заготовками для подальшої роботи програмістів. Склад інформації, що включається в програмні файли, визначається або за умовчанням, або на розсуд користувача.

Взаємодія з іншими засобами і організація групової роботи. Для підтримки командної роботи над проектом на кожній стадії життєвого циклу ПЗ є інтегрований набір продуктів Rational Suite. Rational Suite існує в таких варіантах:

- Rational Suite AnalystStudio – призначений для визначення і управління повним набором вимог до системи, що розробляється;
- Rational Suite DevelopmentStudio – призначений для проектування і реалізації ПЗ;
- Rational Suite TestStudio – є набір продуктів, призначених для автоматичного тестування додатків;
- Rational Suite Enterprise – забезпечує підтримку повного життєвого циклу ПЗ і призначений як для менеджерів проекту, так і для окремих розробників, що виконують декілька функціональних ролей.

7.5.2. Робота в середовищі Rational Rose

Елементи екрану.

П’ять основних елементів інтерфейсу Rose – це браузер, вікно документації, панелі інструментів, вікно діаграми і журнал (log) [39].

Їх призначення полягає в наступному:

- браузер (browser) – використовується для швидкої навігації по моделі;
- вікно документації (documentation window) – застосовується для роботи з текстовим описом елементів моделі;
- панелі інструментів (toolbars) – застосовуються для швидкого доступу до найпоширеніших команд;
- вікно діаграми (diagram window) – використовується для перегляду і редагування однієї або декількох діаграм UML;
- журнал (log) – застосовується для перегляду помилок і звітів про результати виконання різних команд.

На рис. 7.1 показані різні частини інтерфейсу Rose.

Браузер. Браузер – це ієрархічна структура, що дозволяє здійснювати навігацію по моделі. Усе, що додається в модель – дійові особи, варіанти використання, класи, компоненти – буде показано у вікні браузера. За допомогою браузера можна:

- додавати в модель елементи (дійові особи, варіанти використання, класи, компоненти, діаграми і так далі);
- переглядати існуючі елементи і зв’язки між елементами моделі;
- переміщати й перейменовувати елементи моделі;
- додавати елементи моделі до діаграми;

групувати елементи в пакети.

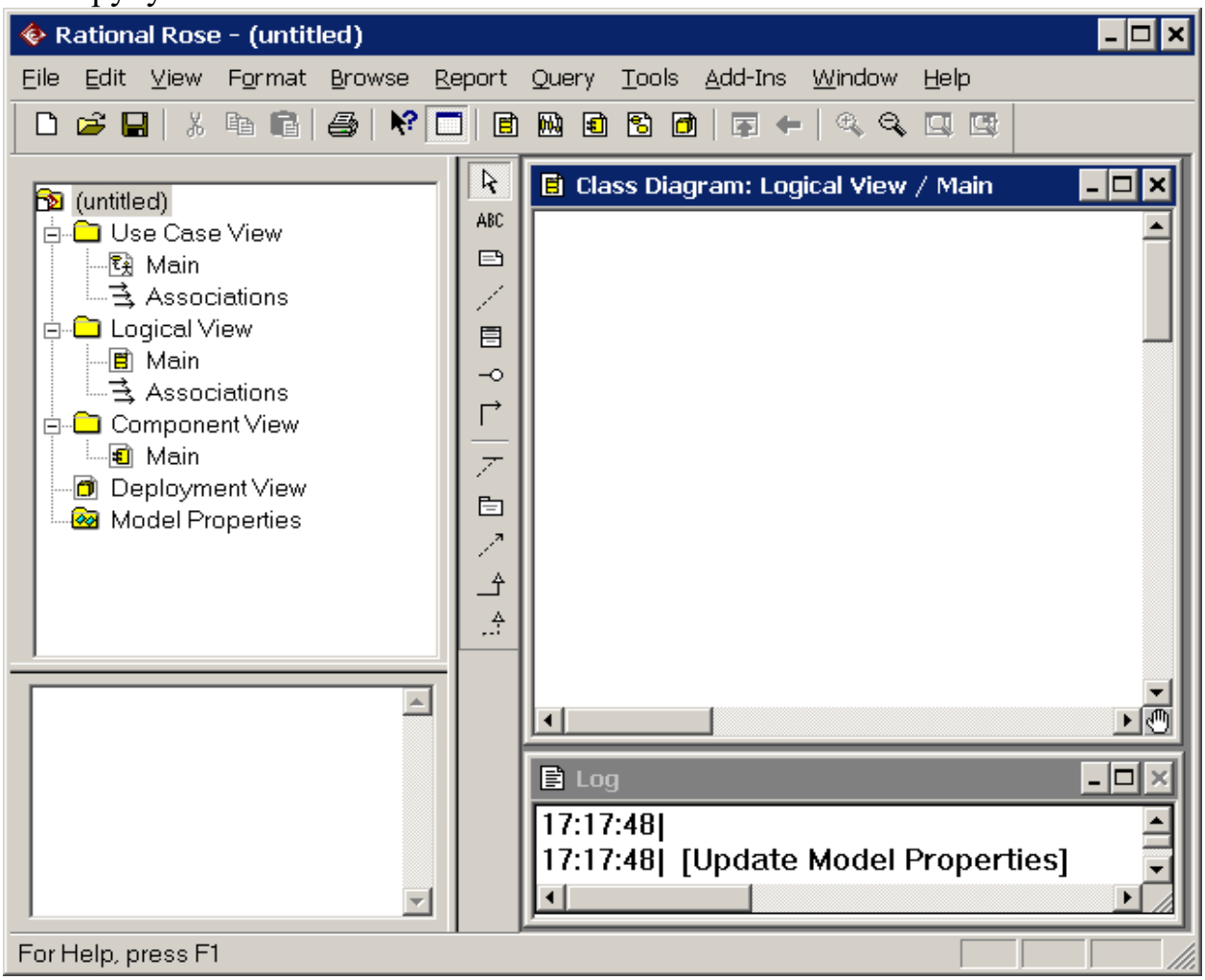


Рисунок 7.1 – Інтерфейс Rational Rose

Браузер підтримує чотири представлення (view): представлення варіантів використання, компонентів, розміщення і логічне представлення. Усі вони і елементи моделі, що містяться в них, описані нижче.

Браузер організований в деревовидному стилі. Кожен елемент моделі може містити інші елементи, що знаходяться нижче його в ієрархії. Знак «-» біля елементу означає, що його гілка повністю розкрита. Знак «+» – що його гілка згорнута.

Вікно документації. За його допомогою можна документувати елементи моделі Rose. Наприклад, можна зробити короткий опис кожної дійової особи. При документуванні класу усе, що буде написано у вікні документації, з'явиться потім у вигляді коментаря в згенерованому коді, що позбавляє від необхідності згодом вносити ці коментарі вручну. Документація виводитиметься також у звітах, що створюються в середовищі Rose.

Панелі інструментів. Панелі інструментів Rose забезпечують швидкий доступ до найпоширеніших команд. У цьому середовищі існує два типи панелей інструментів: стандартна панель і панель діаграми. Стандартна панель видно завжди, її кнопки відповідають командам, які можуть використовуватися

для роботи з будь-якою діаграмою. Панель діаграми своя для кожного типу діаграм UML.

Усі панелі інструментів можуть бути змінені і налагоджені користувачем. Для цього виберіть пункт меню Tools > Options, потім виберіть вкладку Toolbars.

Щоб показати або приховати стандартну панель інструментів (чи панель інструментів діаграми):

1. Виберіть пункт Tools > Options.
2. Виберіть вкладку Toolbars.
3. Щоб зробити видимою або невидимою стандартну панель інструментів, помітьте (чи зніміть позначку) контрольний перемикач Show Standard ToolBar (чи Show Diagram ToolBar).

Щоб збільшити розмір кнопок на панелі інструментів:

1. Клацніть правою кнопкою миші на необхідній панелі.
2. Виберіть в спливаючому меню пункт Use Large Buttons (Використати великі кнопки).

Щоб настроїти панель інструментів:

1. Клацніть правою кнопкою миші на необхідній панелі.
2. Виберіть пункт Customize (настроїти)
3. Щоб додати або видалити кнопки, виберіть відповідну кнопку і потім клацніть мишею на кнопці Add (додати) або Remove (видалити), як показано на рис. 7.2.

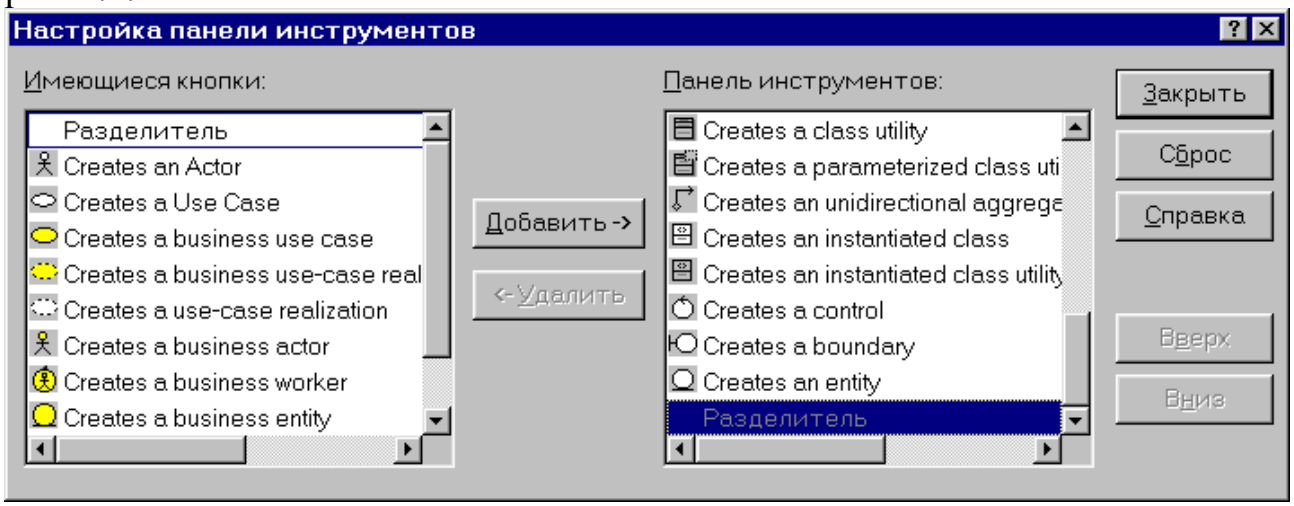


Рисунок 7.2 – Налаштування стандартної панелі інструментів

Вікно діаграми. У вікні діаграми видно, як виглядає одна або декілька діаграм UML моделі. При внесенні в елементи діаграми змін Rose автоматично відновить браузер. Аналогічно, при внесенні змін до елементу за допомогою браузера Rose автоматично відновить відповідні діаграми. Це допомагає підтримувати модель в несуперечливому стані.

Журнал. У міру роботи над вашою моделлю певна інформація спрямовуватиметься у вікно журналу. Наприклад, туди поміщаються повідомлення про помилки, що виникають при генерації коду. Не існує способу закрити журнал зовсім, але його вікно може бути мінімізоване.

Чотири представлення моделі Rose.

У моделі Rose підтримується чотири представлення (views) – представлення варіантів використання, логічне представлення, представлення компонентів і представлення розміщення. Кожне з них призначене для своїх цілей і для відповідної аудиторії. У подальших підрозділах цього розділу ми коротко розглянемо кожне з вказаних представлень, а в частині книги, що залишилася, детально обговоримо елементи моделі, що містяться в них.

Представлення варіантів використання. Це представлення містить усіх дійових осіб, усі варіанти використання і їх діаграми для конкретної системи. Воно може також містити деякі діаграми послідовності і кооперативні діаграми. На рис. 2.3 показано, як виглядає представлення варіантів використання у браузері Rose.

Представлення варіантів використання містить:

1. Дійових осіб.
2. Варіанти використання.
3. Документацію по варіантах використання, що деталізує процеси (потоки подій), що відбуваються в них, включаючи обробку помилок. Піктограмами зображаються зовнішні файли, що прикріплені до моделі Rose. Вид піктограми, залежить від додатка, використовуваного для документування потоку подій. В даному випадку (рис. 7.3) застосовувався Microsoft Word.
4. Діаграми варіантів використання. Зазвичай у системи буває декілька таких діаграм, кожна з яких показує підмножину дійових осіб і/або варіантів використання.
5. Пакети, що є групами варіантів використання і/або діючих осіб.

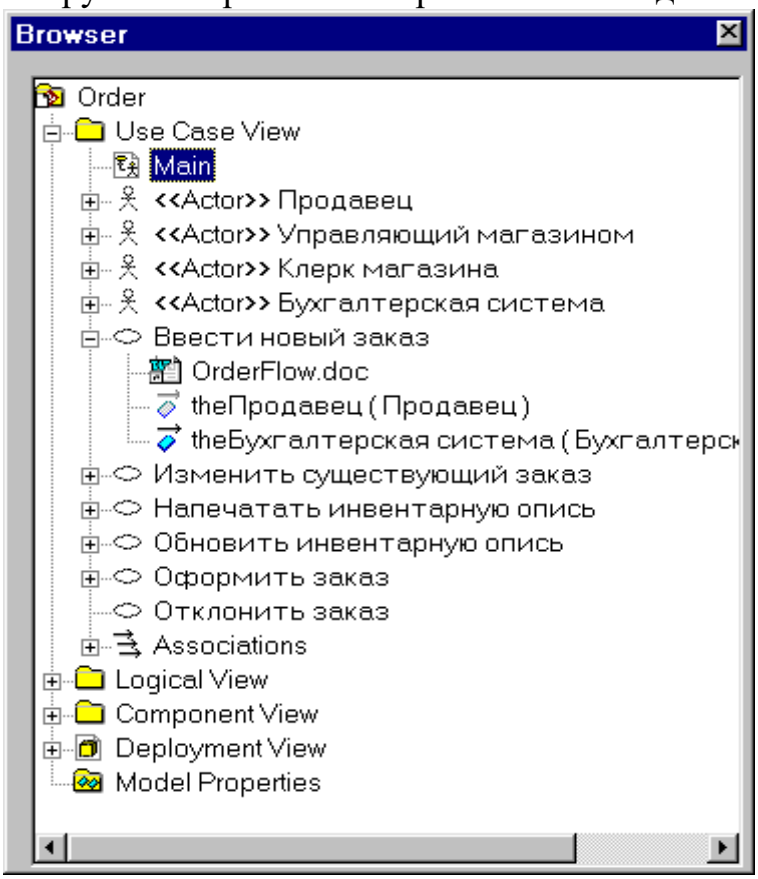


Рисунок 7.3 – Представлення варіантів використання

Логічне представлення. Логічне представлення, що показане на рис. 7.4, концентрується на тому, як система реалізовуватиме поведінку, описану у варіантах використання. Воно дає детальну картину складових частин системи і описує взаємодію цих частин. Логічне представлення включає, окрім іншого, конкретні необхідні класи, діаграми класів і діаграми станів. За їх допомогою конструюється детальний проект створюваної системи.

Логічне представлення містить:

1. Класи.
2. Діаграми класів. Як правило, для опису системи використовується декілька діаграм класів, кожна з яких відображає деяку підмножину усіх класів системи.
3. Діаграми взаємодії, вживані для відображення об’єктів, що беруть участь в одному потоці подій варіанту використання.
4. Діаграми станів.
5. Пакети, що є групами взаємозв’язаних класів.

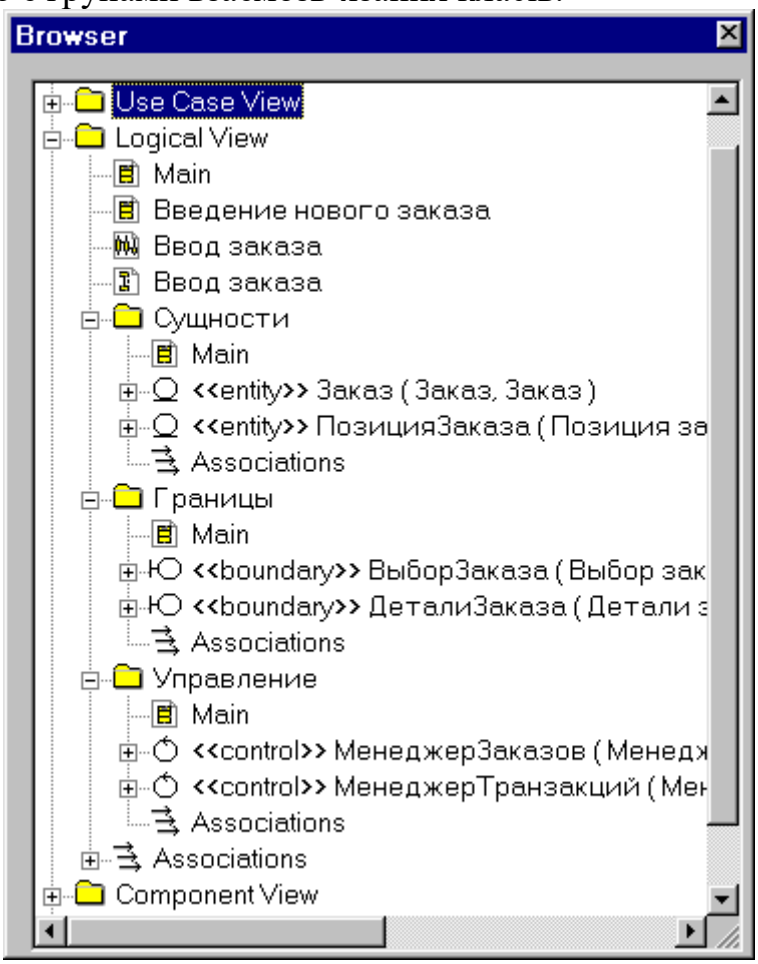


Рисунок 7.4 – Логічне представлення системи

Представлення компонентів. Представлення компонентів містить:

1. Компоненти, що є фізичними модулями коду.
2. Діаграми компонентів.
3. Пакети, що є групами пов’язаних компонентів.

Представлення розміщення. Останнє представлення Rose – це представлення розміщення. Воно відповідає фізичному розміщенню системи, яке може відрізнятися від її логічної архітектури.

У представлення розміщення входять:

1. Процеси, що є потоками (threads), що виконуються у відведеній для них області пам'яті.
2. Процесори, що включають будь-які комп'ютери, здатні обробляти дані. Будь-який процес виконується на одному або декількох процесорах.
3. Пристрої, тобто будь-яка апаратура, не здатна обробляти дані. До таких пристроїв належать, наприклад, термінали введення-виводу і принтери.
4. Діаграма розміщення.

Параметри налаштування відображення.

Зображення атрибутів і операцій на діаграмах класів. У Rose є можливість налаштувати діаграми класів так, щоб:

1. Показувати усі атрибути і операції.
2. Приховати операції.
3. Приховати атрибути.
4. Показувати тільки деякі атрибути або операції.
5. Показувати операції разом з їх повними сигнатурами або тільки їх імена.
6. Показувати або не показувати видимість атрибутів і операцій.
7. Показувати або не показувати стереотипи атрибутів і операцій.

Значення кожного параметра за умовчанням можна задати за допомогою вікна, що відкривається при виборі пункту меню Tools > Options.

У цього класу на діаграмі можна:

- показати усі атрибути;
- приховати усі атрибути;
- показати тільки вибрані вами атрибути;
- подавити виведення атрибутів.

Подавлення виведення атрибутів призведе не лише до зникнення атрибутів з діаграми, але і до видалення лінії, що показує місце розташування атрибутів у класі.

Існує два способи зміни параметрів представлення атрибутів на діаграмі. Можна встановити потрібні значення у кожного класу індивідуально. Можна також змінити значення потрібних параметрів за умовчанням до початку створення діаграми класів. Внесені таким чином зміни вплинуть тільки на новостворювані діаграми.

Щоб показати усі атрибути класу:

1. Виділіть на діаграмі потрібний клас.
2. Клацніть на нім правою кнопкою миші, щоб відкрити контекстно-залежне меню.
3. У ньому виберіть Options > Show All Attributes.

Щоб показати у класу тільки обрані атрибути:

1. Виділіть на діаграмі потрібний вам клас.
2. Клацніть на ньому правою кнопкою миші, щоб відкрити контекстно-залежне меню.

3. У ньому виберіть Options > Select Compartment Items.
4. Вкажіть потрібні вам атрибути у вікні Edit Compartment.

Щоб подавити виведення усіх атрибутів класу діаграми:

1. Виділіть на діаграмі потрібний вам клас.
2. Клацніть на ньому правою кнопкою миші, щоб відкритиконтекстно-залежне меню.
3. У ньому виберіть Options > Suppress Attributes.

Щоб змінити вигляд набраного за умовчанням атрибуту:

1. У меню моделі виберіть пункт Tools > Options.
2. Перейдіть на вкладку Diagram.
3. Для установки значень параметрів відображення атрибутів за умовчанням скористайтеся контрольними перемикачами Suppress Attributes і Show All Attributes. Зміна цих значень за умовчанням вплине тільки на нові діаграми. Вид існуючих діаграм класів не зміниться.

Як і у разі атрибутів, є декілька варіантів представлення операцій на діаграмах.

1. Показати усі операції.
2. Показати тільки деякі операції.
3. Приховати усі операції.
4. Подавити виведення операцій.

Крім того, можна:

1. Показати тільки ім’я операції. Це означає, що на діаграмі буде представлено тільки ім’я операції, але не аргументи або тип значення, що повертається.
2. Показати повну сигнатуру операції. На діаграмі буде представлено не лише ім’я операції, але і усі її параметри, типи даних параметрів і тип значення операції, що повертається.

Щоб показати усі операції класу:

1. Виділіть на діаграмі потрібний вам клас.
2. Клацніть на ньому правою кнопкою миші, щоб відкрити контекстно-залежне меню.
3. У ньому виберіть Options > Show All Operations.

Щоб показати тільки обрані операції класу:

1. Виділіть на діаграмі потрібний вам клас.
2. Клацніть на ньому правою кнопкою миші, щоб відкрити контекстно-залежне меню.
3. У ньому виберіть Options > Select Compartment Items.
4. Вкажіть потрібні вам операції у вікні Edit Compartment.

Щоб подавити виведення усіх операцій класу діаграми:

1. Виділіть на діаграмі потрібний вам клас.
2. Клацніть на ньому правою кнопкою миші, щоб відкрити контекстно-залежне меню.
3. У ньому виберіть Options > Suppress Operations.

Щоб показати на діаграмі класів сигнатуру операції:

1. Виділіть на діаграмі потрібний вам клас.

2. Клацніть на ньому правою кнопкою миші, щоб відкрити контекстно-залежне меню.
3. У ньому виберіть Options > Show Operation Signature.

Щоб змінити набраного за умовчанням вигляду операції:

1. У меню моделі виберіть пункт Tools > Options.
2. Перейдіть на вкладку Diagram.
3. Для установки значень параметрів відображення операцій за умовчанням скористайтеся контрольними перемикачами Suppress Operations, Show All Operations і Show Operation Signatures.

Щоб показати видимість атрибуту або операції класу:

1. Виділіть на діаграмі потрібний вам клас.
2. Клацніть на ньому правою кнопкою миші, щоб відкрити контекстно-залежне меню.
3. У ньому виберіть Options > Show Visibility.

Щоб змінити набутого за умовчанням значення параметра показу видимості:

1. У меню моделі виберіть пункт Tools > Options.
2. Перейдіть на вкладку Diagram.
3. Для установки параметрів відображення видимості за умовчанням скористайтеся контрольним перемикачем Show Visibility.

Для перемикання між нотаціями видимості Rose і UML:

1. У меню моделі виберіть пункт Tools > Options.
2. Перейдіть на вкладку Notation.
3. Для перемикання між нотаціями скористайтеся перемикачем Visibility as Icons. Якщо цей перемикач помічений, використовуватиметься нотація Rose. Якщо ні – то нотація UML. Зміна цього параметра вплине тільки на нові діаграми. Існуючі діаграми класів залишаться попередніми.

Контрольні питання до розділу 7

1. Призначення CASE-засобу *IBM Rational Rose*.
2. Які версії продукту *Rational Rose* ви знаєте?
3. Структура і функції *Rational Rose*.
4. Які п’ять основних елементів інтерфейсу *Rose* ви знаєте?
5. Призначення панелі інструментів *Rose*.
6. Назвіть основне призначення *Браузера* в *Rose*.

ЧАСТИНА 3. СТРУКТУРНИЙ АНАЛІЗ І ПРОЕКТУВАННЯ ІС

Висока динаміка зміни ситуації на ринку пред'являє жорсткі вимоги, як до функціональності ІС, так і до процесу їх створення.

Сучасні засоби дозволяють досить швидко створювати (впроваджувати) ІС за готовими вимогами. Але дуже часто виявляється, що ці системи не задовольняють замовників. Головною причиною такого положення є неправильне, неточне або неповне визначення вимог до ІС. Проблема формування вимог до ІС залишається дотепер однією з таких, що найважче формалізуються, і найдорожчих і найважчих для виправлення у разі помилки. Саме тому така велика роль початкових етапів ЖЦ створення ІС, коли ці вимоги мають бути виявлені і формалізовані, в отриманні кінцевого результату.

Тому для створення адекватної інформаційної системи потрібна технологія, яка б допомогла сформувати вимоги до ІС, спроектувати і розробити систему, що відповідає цим вимогам. Наявність такої технології є вирішальним чинником успіху при створенні інформаційних систем.

Нині використовується велика кількість підходів, які дозволяють, так або інакше, створювати моделі бізнес-процесів підприємств. Найважливішими з підходів є структурний (функціональний) і об'єктно-орієнтований підхід.

Суть структурного підходу до розробки ІС полягає в її декомпозиції (розбитті) на функції, що автоматизуються: система розбивається на функціональні підсистеми, які у свою чергу діляться на підфункції, що підрозділяються на завдання і так далі.

Об'єктно-орієнтований підхід використовує об'єктну декомпозицію, при цьому статична структура системи описується в термінах об'єктів і зв'язків між ними, а поведінка системи описується в термінах обміну повідомлень між об'єктами.

Яка декомпозиція інформаційної системи правильніша – алгоритмічна чи за об'єктами? У цьому запитанні є правильна відповідь на нього: важливі обидва аспекти. Але досвід показує, що корисно починати з об'єктної декомпозиції. Такий початок допоможе нам краще впоратися з організацією складності програмних систем. Це особливо вірно, коли ми розглядаємо світ з об'єкгно-орієнтованого погляду, оскільки об'єкти як абстракції реального світу є окремими насиченими зв'язаними інформаційними одиницями. Класифікуючи об'єкти за групами родинних абстракцій, ми чітко розділяємо загальні й унікальні властивості різних об'єктів, що допомагає нам потім справлятися з властивою їм складністю.

РОЗДІЛ 8. ЗАСТОСУВАННЯ СТРУКТУРНОГО ПІДХОДУ В АНАЛІЗІ ВИМОГ І ВИЗНАЧЕННІ СПЕЦИФІКАЦІЙ ПЗ

8.1. Основні відомості

Розробка будь-якого програмного забезпечення розпочинається з аналізу вимог до майбутнього програмного продукту. У результаті аналізу отримують специфікації програмного забезпечення, що розробляється: виконують декомпозицію і змістовну постановку вирішуваних завдань, уточнюють їх взаємодію і визначають експлуатаційні обмеження. У процесі визначення специфікацій будують загальну модель предметної області як деякої частини реального світу, з якою буде тим або іншим способом взаємодіяти програмне забезпечення, що розробляється, і конкретизують його основні функції.

Специфікації є повним і точним описом функцій і обмежень програмного забезпечення, що розробляється. При цьому функціональні специфікації описують функції програмного забезпечення, що розробляється, а експлуатаційні специфікації визначають вимоги до технічних засобів, надійності, безпеки тощо.

Стосовно функціональних специфікацій вимога повноти означає, що специфікації повинні містити усю істотну інформацію, щоб ніщо важливе не було упущене, і не повинні містити несуттєвої інформації, наприклад, деталей реалізації, щоб не перешкоджати розробникові у виборі найефективніших рішень. Вимога точності означає, що специфікації повинні однозначно сприйматися як замовником, так і розробником [38; 42-44].

Останню вимогу виконати досить складно, оскільки природна мова для опису специфікацій не підходить: детальні специфікації на природній мові не забезпечують необхідної точності. Точні специфікації програмного забезпечення, що розробляється, можна визначити тільки розробивши деяку формальну модель цього програмного забезпечення.

Усі функціональні специфікації програмного забезпечення, що розробляється, описують перелік функцій і склад оброблюваних даних. Вони розрізняються тільки системою пріоритетів (акцентів), яка використовується розробником у процесі аналізу вимог і визначення специфікацій. Так, діаграми переходів станів визначають деякі аспекти поведінки програмного забезпечення в часі, діаграми потоків даних – напрям і структуру потоків даних, а концептуальні діаграми класів – відношення між основними поняттями предметної області. На рис. 8.1 показана класифікація моделей, що використовуються як специфікації програмного забезпечення, що розробляється [38].

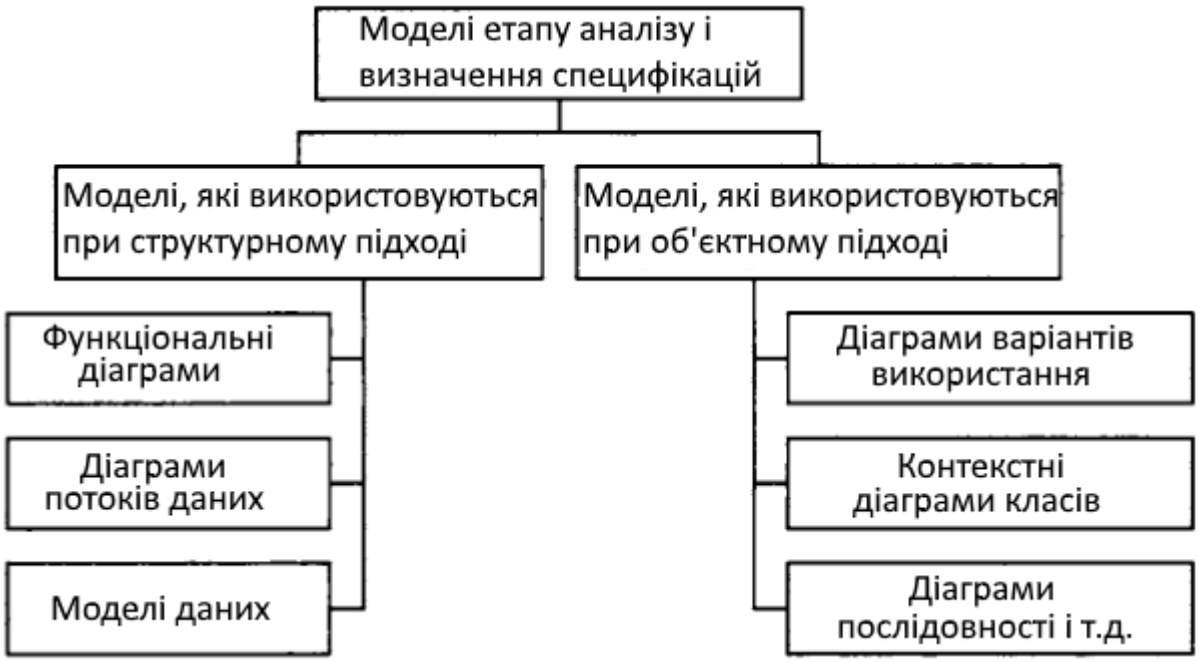


Рисунок 8.1 – Класифікація моделей програмного забезпечення

У рамках структурного підходу на етапі аналізу і визначення специфікацій використовують три типи моделей: орієнтовані на функції, орієнтовані на дані і орієнтовані на потоки даних.

Кожен тип моделі доцільно використати для свого специфічного класу програмних розробок. Різні моделі описують проектоване програмне забезпечення з різних сторін. Методології структурного аналізу і проектування, засновані на моделюванні потоків даних зазвичай використовують комплексне представлення програмного забезпечення, що проектується, у вигляді сукупності таких моделей:

- діаграми переходів станів (SDT – State Transition Diagrams), що характеризують поведінку системи в часі;
- функціональні діаграми (SADT – Structured Analisis and Design Technique);
- діаграми потоків даних (DFD – Data Flow Diagrams), що описують взаємодію джерел і споживачів інформації через процеси, які мають бути реалізовані в системі;
- діаграми «сутність-зв’язок» (ERD – Entity – Relationship Diagrams), що описують бази даних системи, що розробляється.

Компоненти специфікації методологій структурного аналізу наведені на рис. 8.2 [38].

Специфікацію процесу зазвичай представляють у вигляді короткого текстового опису, схем алгоритмів, псевдокодів.

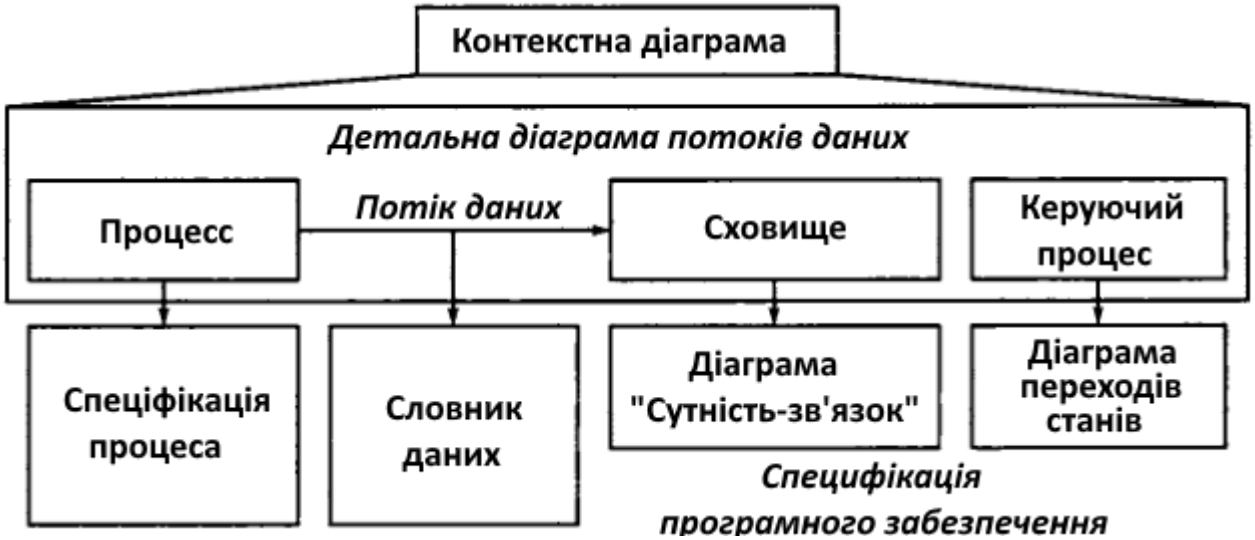


Рисунок 8.2 – Компоненти повної специфікації методологій структурного аналізу

Словник даних – цей короткий опис основних понять, що використовуються при складанні специфікацій. Словник повинен включати визначення основних понять предметної області, опис структур елементів даних, їх типів і форматів, а також усіх скорочень і умовних позначень. Словник призначений для підвищення міри розуміння предметної області і виключення ризику виникнення розбіжностей при обговоренні моделей між замовниками і розробниками.

Головний недолік структурного підходу полягає в такому: процеси і дані існують окремо один від одного, причому проектування ведеться від процесів до даних. Таким чином, окрім функціональної декомпозиції, існує також структура даних, що знаходиться на другому плані.

8.2. Діаграми переходів станів

Діаграми переходів станів (SDT) демонструють поведінку програмної системи, що розробляється, при отриманні управляючих дій. Під діями, що управляють, або сигналами, в даному випадку розуміють інформацію, що управляє, отримується системою ззовні. Наприклад, діями, що управляють, вважають команди користувача і сигнали датчиків, підключених до комп’ютерної системи. Отримавши дію, що управляє, система, що розробляється, повинна виконати певні дії, а потім або залишитися в тому ж стані, або перейти в інший стан, зафіксувавши деякі зміни в системі.

Головне призначення цієї діаграми – описати можливі послідовності станів і переходів, які в сукупності характеризують поведінку елементу моделі впродовж його життєвого циклу. Діаграма переходів станів представляє динамічну поведінку сутностей на основі специфікації їх реакції на сприйняття деяких конкретних подій. Системи, які реагують на зовнішні дії інших систем або користувачів, іноді називають ***реактивними***. Якщо такі дії ініціюються в довільні випадкові моменти часу, то говорять про асинхронну поведінку моделі.

Попри те, що діаграми найчастіше використовуються для опису поведінки окремих екземплярів класів (об'єктів), вони також можуть застосовуватися для специфікації функціональності інших компонентів моделей, таких як варіанти використання, дійові особи, підсистеми, операції і методи.

Для побудови діаграми переходів станів необхідно відповідно до теорії кінцевих автоматів визначити основні стани, дії (чи умови переходу), що управляють, виконувані дії і можливі переходи програмного забезпечення, що розробляється. Умовні позначення, що використовуються при побудові діаграм переходів станів, показані на рис. 8.3 [38].

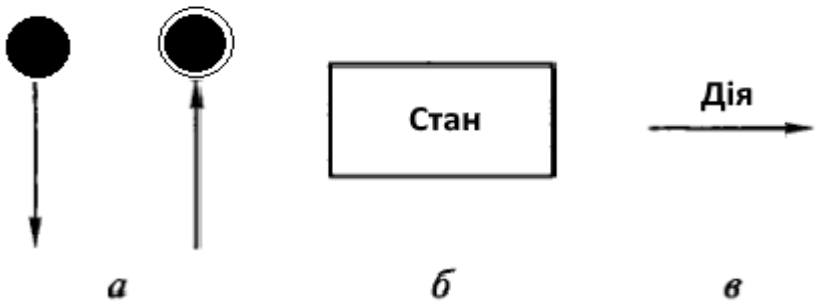


Рисунок 8.3 – Умовні позначення діаграм переходів станів:
а – початкове і кінцеве стани; *б* – проміжний стан; *в* – перехід

Діаграма переходів станів, по суті, є граф спеціального виду, який означає деякий автомат. Поняття «автомат» в контексті UML має досить специфічну семантику, засновану на теорії автоматів. Вершинами цього графа є стани і деякі інші типи елементів автомата (псевдостани), які зображаються відповідними графічними символами. Дуги графа служать для позначення переходів із стану в стан.

Простим прикладом візуального представлення станів і переходів на основі формалізму автоматів може служити ситуація із справністю технічного пристрою, такого як комп'ютер. В цьому випадку вводяться в розгляд два найзагальніші стани: «справний» і «несправний» і два переходи: «вихід з ладу» і «ремонт». Графічно ця інформація може бути представлена у вигляді діаграми станів комп'ютера (рис. 8.4) [38].

Рисунок 8.4 – Діаграми переходів станів для пристрою типу комп'ютер

Головна відмінність між станом і переходом полягає в тому, що час знаходження системи в окремому стані істотно перевищує час, який витрачається на перехід з одного стану в інший. Передбачається, що в межі час

переходу з одного стану в інший дорівнює нулю (якщо додатково нічого не сказано). Іншими словами, перехід об'єкту із стану в стан відбувається миттєво.

Для інтерактивного програмного забезпечення з розвиненим призначеним для користувача інтерфейсом основними діями, що управляють, виступають команди користувача, для програмного забезпечення реального часу – сигнали від датчиків і (чи) оператора виробничого процесу. Загальною для цих типів програмного забезпечення є наявність стану очікування, коли система призупиняє роботу до отримання чергової дії, що управляє.

На відміну від інтерактивних систем для систем реального часу зазвичай встановлено жорсткіше обмеження на час обробки отриманого сигналу. Таке обмеження часто вимагає виконання додаткових досліджень поведінки системи в часі.

До програмного забезпечення, при розробці якого потрібно уточнення особливостей поведінки за допомогою побудови діаграми переходів станів, належить і програмне забезпечення, що орієнтоване на роботу в мережі. При цьому, зазвичай, окремо будують моделі поведінки сервера і клієнта, представляючи повідомлення, що передаються між ними, у вигляді дій, що управляють. Наведемо складніший приклад побудови діаграми переходів станів (рис. 8.5) [38].

Рисунок 8.5 – Діаграма переходів станів об'єкту «Замовлення»

На пропонованій діаграмі (див. рис. 8.5) описуються можливі послідовності станів і переходів, які в сукупності характеризують поведінку об'єкта «Замовлення» автоматизованої інформаційної системи «Склад оптової торгівлі» впродовж його існування (вступ, обробка, формування постачання).

На діаграмі відображаються функції, які виконуються об’єктом «Замовлення» в певному стані. Мітки діяльності мають такий синтаксис: *виконати*/< *діяльність* > (наприклад, виконати/перевірити рядок). Переходи мають мітки, які синтаксично складаються з трьох необов’язкових частин: <*Подія*> <[*Умова*]> </*Дія*> (наприклад, [не усі рядки проведені]/отримати наступний рядок).

8.3. Функціональні діаграми

Функціональні діаграми (SADT) відбивають взаємні зв’язки функцій програмного забезпечення, що розробляється. Вони створюються на ранніх стадіях проектування інформаційних систем для того, щоб допомогти проектувальникові виявити основні функції і складові частини проектованої програмної системи і, по можливості, виявити і усунути істотні помилки.

Для створення функціональних діаграм пропонується використати методологію SADT, запропоновану Д. Россом. Застосування методології SADT дозволяє побудувати модель, що складається з діаграм, фрагментів текстів і глосарію, що мають посилання один на одного. Діаграми – головні компоненти моделі. Функції системи і інтерфейси представлені на діаграмах у вигляді блоків і дуг. Місце з’єднання дуги з блоком визначає тип інтерфейсу. Інформація, що управляє, входить у блок згори, тоді як інформація, яка піддається обробці, показана з лівого боку блоку, а результати виходу даються з правого боку. Механізм (людина або автоматизована система), що здійснює операцію, представлений дугою, що входить у блок знизу (рис. 8.6) [38].

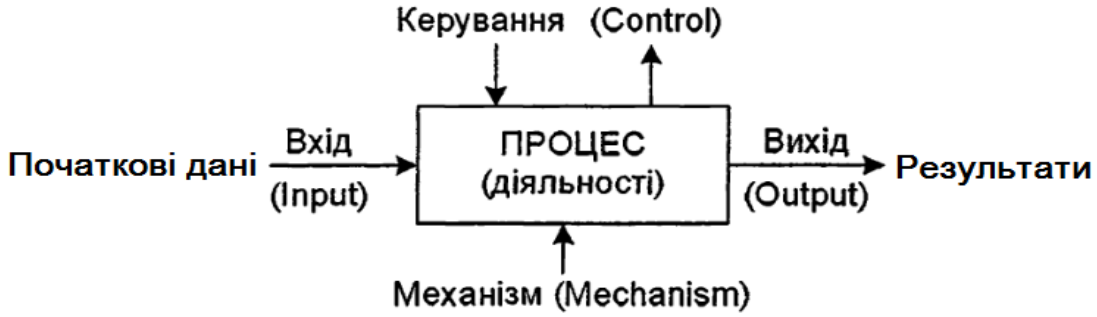


Рисунок 8.6 – Функціональний блок і інтерфейсні дуги

Одна з найважливіших особливостей методології SADT – поступове введення все більших рівнів деталізації у міру створення діаграм, що відображають модель. Побудова SADT-моделі розпочинається з представлення усієї системи у вигляді простої компоненти – одного блоку і дуг, що зображують інтерфейси з функціями поза системою. Оскільки єдиний блок представляє усю систему як єдине ціле, ім’я, вказане у блоці, є загальним. Це вірно і для інтерфейсних дуг – вони також є повним набором зовнішніх інтерфейсів системи в цілому.

Потім блок, в якому система представлена в якості єдиного модуля, деталізується на іншій діаграмі за допомогою декількох блоків, сполучених інтерфейсними дугами. Ці блоки представляють основні підфункції початкової функції. Ця декомпозиція виявляє повний набір підфункцій, кожна з яких представлена як блок, межі якого визначені інтерфейсними дугами. Кожна з

цих підфункцій може бути декомпонована так само для детальнішого представлення. У усіх випадках кожна підфункція може містити тільки ті елементи, які входять в початкову функцію. Крім того, модель не може опустити які-небудь елементи, тобто, як вже відзначалося, батьківський блок і його інтерфейси забезпечують контекст. До нього не можна нічого додати, і з нього не може бути нічого видалено.

Модель SADT є серією діаграм з супровідною документацією, що розбивають складний об'єкт на складові частини, представлені у вигляді блоків. Деталі кожного з основних блоків показані у вигляді блоків на інших діаграмах. Кожна детальна діаграма є декомпозицією блоку із загальнішої діаграми. На кожному кроці декомпозиції загальніша діаграма називається батьківською для детальнішої діаграми.

Дуги, що входять у блок і виходять з нього на діаграмі верхнього рівня, є точно тими ж самими, що і дуги, що входять в діаграму нижнього рівня і виходять з неї, тому що блок і діаграма представляють одну і ту ж частину системи [38].

Стрілки, що приходять з батьківської діаграми або йдуть на неї, нумерують, використовуючи символи і числа. Символ означає тип зв'язку: I – вхідні, C – управляючі, M – механізми, R – результати. Число – номер зв'язку по відповідній стороні батьківського блоку, рахуючи зверху вниз і зліва направо.

Усі діаграми зв'язують одна з одною ієрархічною нумерацією блоків: початковий рівень – A0, наступний – A1, A2 тощо, наступний, – A11, A12, A13 і так далі, де перша цифра – номер батьківського блоку, а остання – номер конкретного субблоку батьківського блоку. Деталізацію завершують при отриманні функцій, призначення яких добре зрозуміло як замовникові, так і розробникові. Ці функції описують, використовуючи природну мову або псевдокоди. В процесі побудови ієрархії діаграм фіксують усю уточнювальну інформацію і будують словник даних, в якому визначають структури і елементи даних, показаних на діаграмах. Таким чином, у результаті отримують специфікацію, яка складається з ієрархії функціональних діаграм, описів функцій нижнього рівня і словника, що мають посилання один на одного.

На рис. 8.7 представлена декомпозиція трьох діаграм, що показує структуру SADT-моделі, загальне (рис. 8.7, *а)* і детальне представлення блоку A0 (рис. 8.7, *б)*, декомпозицію блоку A4 (рис. 8.7, *в)*. Не приєднувані дуги відповідають входам, управлінням і виходам батьківського блоку. Джерело або одержувач цих дуг може бути виявлене тільки на батьківській діаграмі. Не приєднувані кінці повинні відповідати дугам на початковій діаграмі. Усі граничні дуги мають бути продовжені на батьківській діаграмі, щоб вона була повною і несуперечливою.

Кожен компонент моделі може бути декомпонований на іншій діаграмі, тобто кожна діаграма ілюструє «внутрішню будову» блоку на батьківській діаграмі.

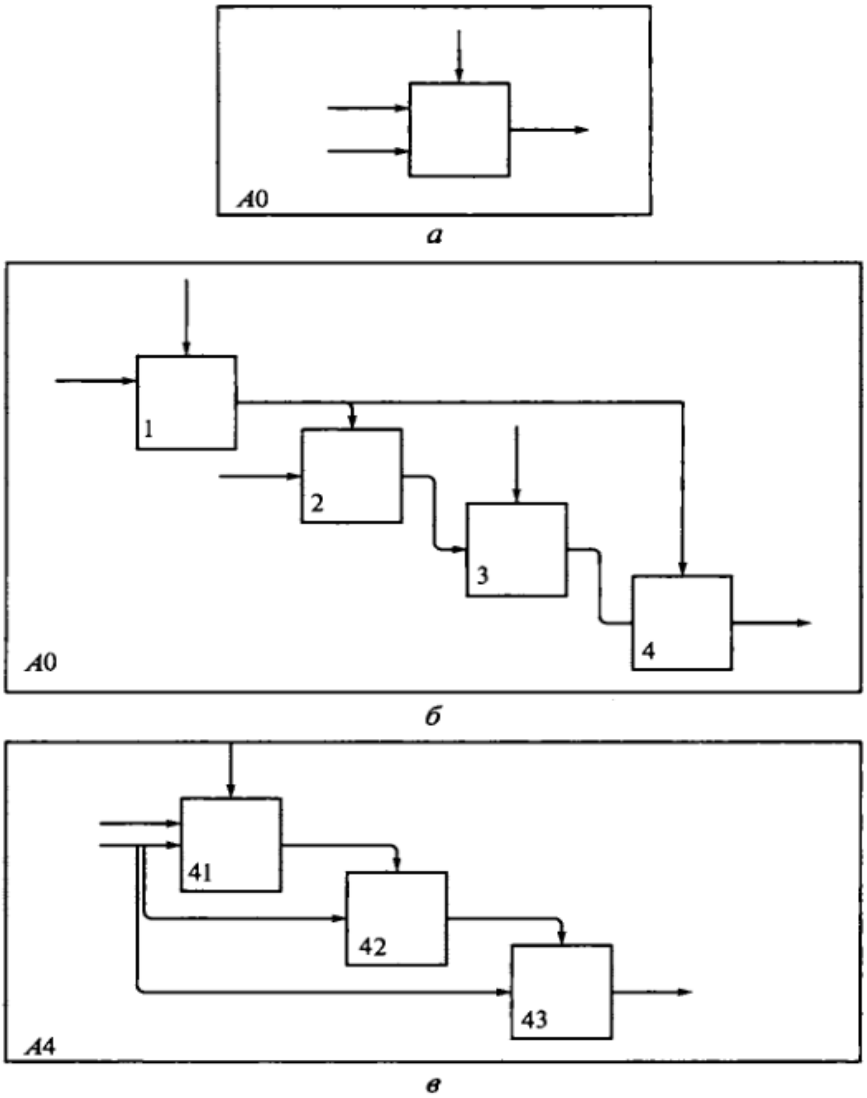


Рисунок 8.7 – Структура SADT-моделі. Декомпозиція діаграм

Одним з важливих моментів при моделюванні системи за допомогою методології SADT є точна узгодженість типів зв'язків між функціями. Розрізняють, принаймні, такі типи зв'язків: випадковий; логічний; тимчасовий; процедурний; комунікаційний; послідовний; функціональний [38].

Розробку функціональних діаграм продемонструємо на прикладі уточнення специфікацій програми побудови графіків і таблиць функцій однієї змінної [38].

На рис. 8.8, *а* показана діаграма верхнього рівня, на якій добре видно, що є початковими даними для програми і отримання яких результатів ми чекаємо.

Діаграма на рис. 8.8, *б* уточнює функції програми. На ній показані чотири блоки: введення/вибір функції і її розбір, додавання функції в список, побудову таблиці значень і побудову графіка функції.

Для кожного блоку визначені початкові дані, дії, що управляють, і результати. Згідно з правилами позначення входів-виходів, що мають продовження на батьківській діаграмі, на цій діаграмі використані такі позначення: *I1* – функція; *I2* – відрізок; *I3* – крок; *C1* – вид графік-таблиця; *R1* – графік функції на відрізку; *R2* – таблиця значень функції на відрізку.

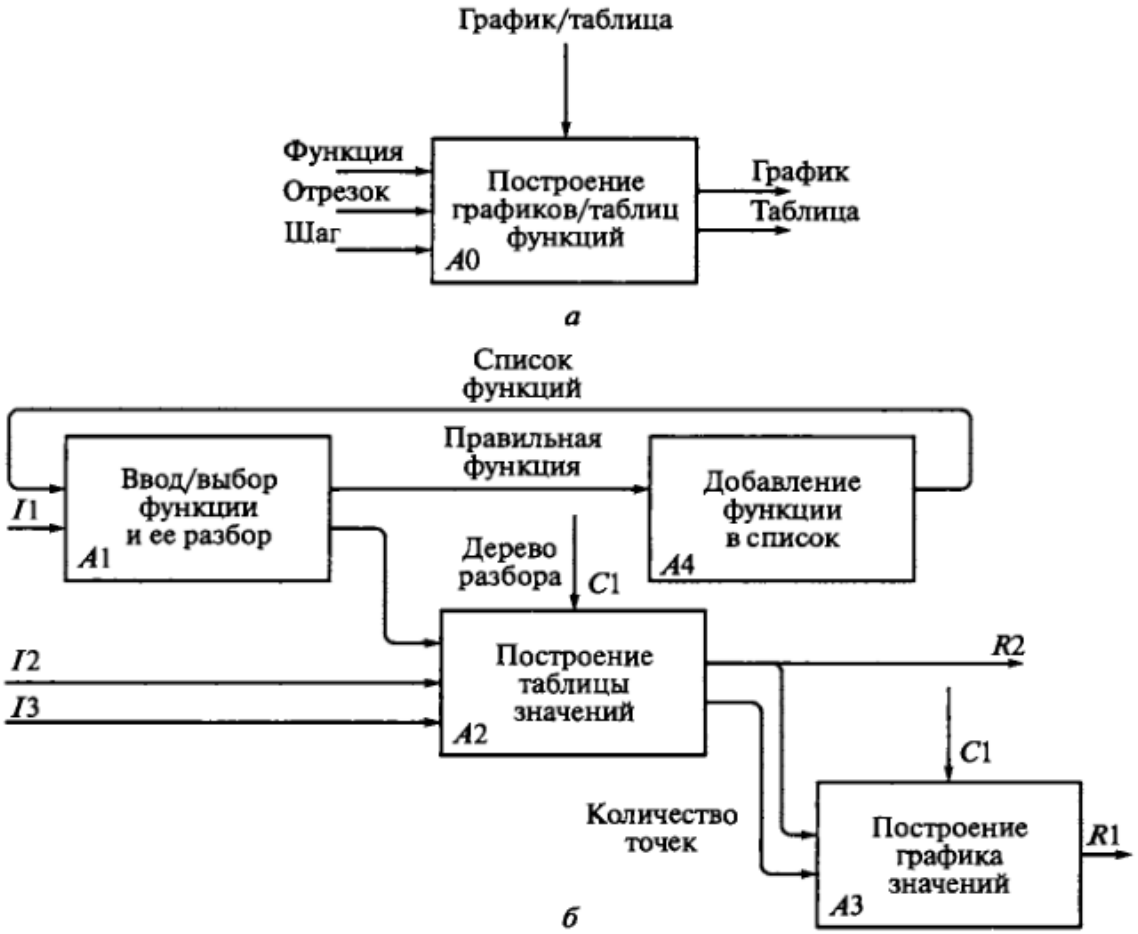


Рисунок 8.8 – Функціональні діаграми для системи дослідження функцій: *а* – діаграма верхнього рівня; *б* – уточнювальна діаграма

Функціональну модель доцільно застосовувати для визначення специфікацій програмного забезпечення, що не передбачає роботу із складними структурами даних, оскільки вона орієнтована на декомпозицію функцій.

8.4. Діаграми потоків даних

Діаграма потоків даних (DFD) складається з вузлів обробки даних, засобів їх зберігання і зовнішніх стосовно використовуваної діаграми джерел або споживачів даних.

Діаграма потоків даних – основний засіб моделювання функціональних вимог до системи, що проектується або реально існує. У основі моделі лежать поняття зовнішньої сутності, процесу, сховища (накопичувача) даних потоку даних. Джерела інформації (зовнішні сутності) породжують інформаційні потоки (потоки даних), що переносять інформацію до підсистем або процесів; ті, у свою чергу, перетворюють інформацію і породжують нові потоки, які переносять інформацію до інших процесів або підсистем, накопичувачів даних або зовнішніх сутностей – споживачів інформації.

Для зображення діаграм потоків даних традиційно використовують два види нотацій – Йордана і Гейна-Сарсона, які представлені в таблиці. 8.1.

Таблиця 8.1 – Види нотацій

Поняття	Нотація Йордана	Нотація Гейна — Сарсона
Зовнішня сутність	**Найменування**	**Номер** **Найменування**
Процес, система, підсистема	**Найменування** **Номер**	**Номер** **Найменування** **Механізм**
Накопичувач даних	**Найменування**	№ **Найменування**
Потік даних	**Найменування** →	**Найменування** →

Зовнішня сутність – це матеріальний предмет або фізична особа, що є джерелом або приймачем інформації, наприклад, замовники, персонал, постачальники, клієнти, склад. Визначення деякого об’єкту або системи в якості зовнішньої сутності вказує на те, що цей об’єкт або ця система знаходяться за межами кордонів аналізованої інформаційної системи.

У процесі аналізу деякі зовнішні сутності можуть бути перенесені всередину діаграми аналізованої інформаційної системи, якщо це необхідно, або, навпаки, частина процесів інформаційної системи може бути винесена за межі діаграми і представлена як зовнішня сутність.

При побудові моделі складної інформаційної системи вона може бути представлена в найзагальнішому вигляді на так званій ***контекстній діаграмі***. На такій діаграмі показують як система, що розробляється, взаємодіятиме із споживачами і джерелами інформації, тобто описують інтерфейс системи із зовнішнім світом. Система може бути представлена як єдине ціле або може бути декомпонована на ряд підсистем. Номер підсистеми служить для її ідентифікації. У поле імені вводиться найменування підсистеми у вигляді речення з підметом і відповідними визначниками і доповненнями.

Процес є перетворенням вхідних потоків даних у вихідні відповідно до певного алгоритму. Фізично процес може бути реалізований різними способами: це може бути підрозділ організації (відділ), що виконує обробку вхідних документів і випуск звітів, програма, апаратно реалізований логічний пристрій тощо. Номер процесу служить для його ідентифікації. У поле імені вводиться найменування процесу, при цьому використання таких дієслів, як обробити, модернізувати або відредагувати означає, як правило, недостатньо

глибоке розуміння цього процесу і вимагає подальшого аналізу. Інформація в полі фізичної реалізації показує, який підрозділ організації, програма або апаратний пристрій виконують цей процес.

Накопичувач даних є абстрактним пристроєм для зберігання інформації, яку можна у будь-який момент помістити в накопичувач і через деякий час витягнути; причому способи поміщення і витягання можуть бути будь-якими. Накопичувач даних може бути реалізований фізично у вигляді ящика в картотеці, таблиці в оперативній пам'яті, файлу на магнітному носії тощо.

У нотації Гейна-Сарсона накопичувач даних ідентифікується буквою *D* і довільним числом. Ім'я накопичувача вибирається з міркування найбільшої інформативності для проектувальника. Накопичувач даних в загальному випадку виступає прообразом майбутньої бази даних, і опис даних, що зберігаються в ньому, має бути пов'язаний з інформаційною моделлю.

Потік даних визначає інформацію, що передається через деяке з'єднання від джерела до приймача. Реальний потік даних може бути інформацією, що передається кабелем між двома пристроями, листами, що пересилаються поштою, магнітними стрічками або дискетами, що переносяться з одного комп'ютера на інший і т.д. Кожен потік даних має ім'я, що відбиває його зміст.

Побудову ієрархії діаграм потоків даних розпочинають з контекстної діаграми (діаграми нульового рівня), яка визначає найзагальніший вигляд системи. Зазвичай початкова контекстна діаграма має форму зірки. Якщо проектована система містить велику кількість зовнішніх сутностей (більше 10), має розподілену природу або включає вже існуючі підсистеми, то будують ієрархії контекстних діаграм.

Наведемо приклад побудови діаграми потоків даних АІС «Склад оптової торгівлі» [38]. Побудову ієрархії діаграм потоків даних розпочнемо з контекстної діаграми, яка визначає найзагальніший вигляд системи. Таким чином, визначимо, як система, що розробляється, взаємодіятиме з приймачами і джерелами інформації (рис. 8.9).

Автоматизована інформаційна система «Склад оптової торгівлі» призначена для отримання даних про рух і наявність товарів, придбаних для оптового продажу. Первинні документи з приходу товарів фіксуються в журналі надходження товарів. Оформлення і облік реалізації товарів залежать від способу розрахунку між покупцем і продавцем за придбані товари. Менеджер складу веде журнал обліку закупівель і відпуску товарів. Дані первинних документів зберігаються у відповідних накопичувачах.

Зовнішніми сутностями для системи, що розробляється, є постачальники, покупці, менеджер складу, відділ обліку і контролю, відділ прийому і оформлення замовлень. Відомості про них зберігаються у відповідних таблицях (довідниках). Постачальник передає товар на склад, документи на постачання товару (накладні, рахунки-фактури) вводяться у базу даних. Покупець подає замовлення на придбання товарів, у відділі прийому і оформлення замовлень перевіряється кожен рядок замовлення, що надійшло.

Рисунок 8.9 – Початкова контекстна діаграма в нотації Йордана для АИС «Склад оптової торгівлі»

За відсутності на складі певної позиції оформляється заявка постачальникові, тобто ініціюється постачання необхідного товару. На підставі замовлення заповнюються документи на реалізацію товару, які зберігаються у базі даних, роздруковуються і видаються покупцеві. У кінці кожного дня усі первинні документи передаються менеджерові відділу обліку (бухгалтерові). На підставі відомостей про прихід і реалізацію товару менеджери складу і працівники відділу обліку формують звіти по оборотах і залишках товару на складі.

Розглянемо приклад побудови діаграми потоків даних підсистеми розподілу студентів [38]. Розробимо діаграму потоків даних автоматизованої системи, призначеної для збору і зберігання інформації про студентів, що навчаються на старших курсах навчального закладу. Система призначена для використання в освітніх установах, з метою надання через інтернет потенційним працедавцям інформації про випускників. Початкова контекстна діаграма потоків даних в нотації Гейна-Сарсона наведена на рис. 8.10. Зовнішніми сутностями в ній є працедавець, адміністратор і студент.

Працедавець реєструється в системі, вводить дані для пошуку кандидатів і отримує від системи результати пошуку. Адміністратор вводить інформацію про студентів, які зберігаються у базі даних. Студент може змінити свою контактну інформацію, зміни також зберігаються у базі даних. Вибраним студентам працедавець посилає повідомлення з пропозицією про роботу. Діаграма потоків даних системи розподілу студентів наведена на рис. 8.11.

Можна надалі ще деталізувати процеси, проте стає ясно, що повна специфікація цієї розробки повинна включать опис бази даних у вигляді діаграми «сутність-зв’язок».

Рисунок 8.10 – Початкова контекстна діаграма в нотації Гейна-Сарсона для системи розподілу студентів

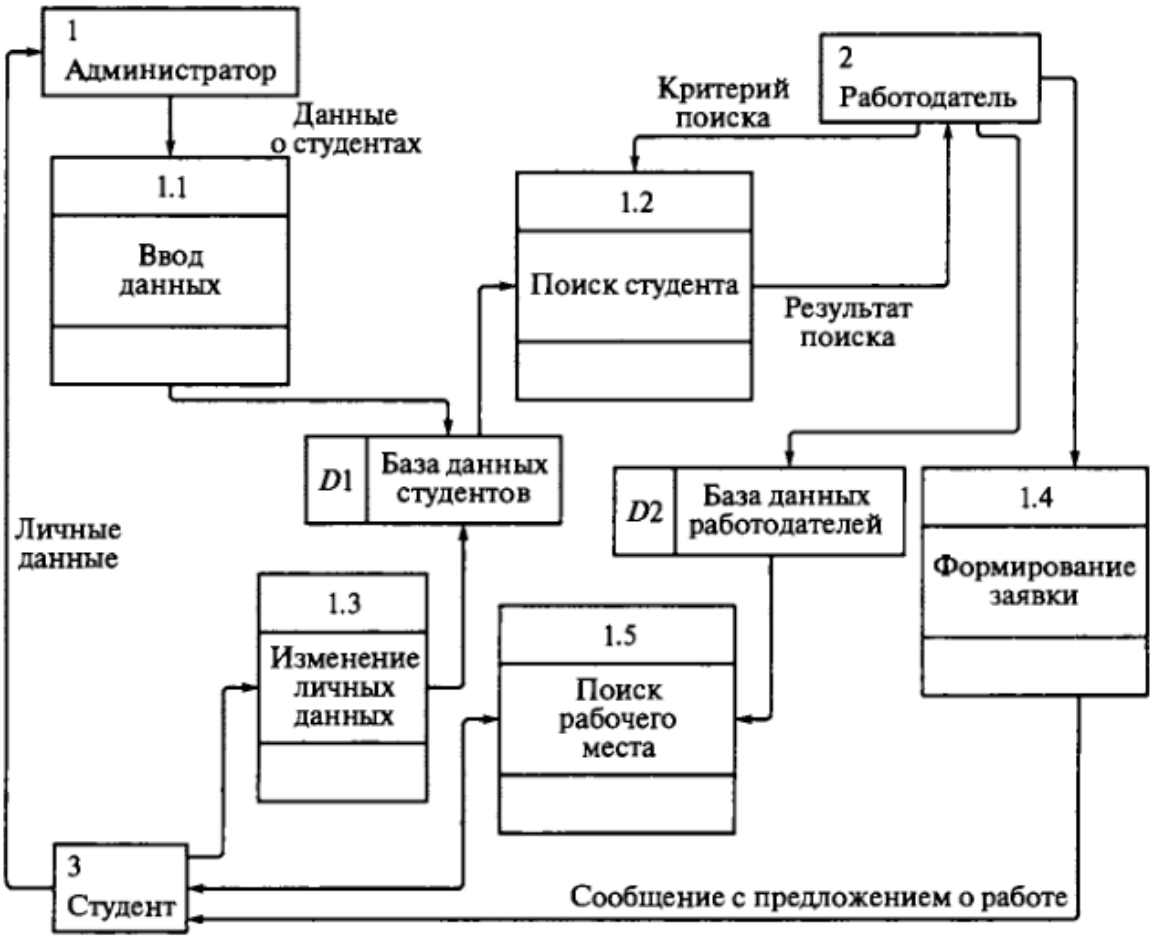


Рисунок 8.11 – Діаграма потоків даних системи розподілу студентів

8.5. Діаграма «сутність-зв’язок»

Одними з найважливіших елементів інформаційної системи є бази даних. Найвідповідальніший момент в створенні бази даних – це побудова її структури. Якщо структура бази даних побудована стратегічно правильно, то це негайно позитивно відобразиться на характеристиці усієї БД, особливо на надійності і швидкості вибірки даних. Тому в цьому розділі ставиться мета розповісти про методику створення самого структурного каркаса бази даних як ключової ланки в проектуванні інформаційних систем [41].

Діаграма «сутність-зв’язок» (ER-модель даних) забезпечує стандартний спосіб визначення даних і відношень між ними. Вона включає сутності і

взаємозв’язки, що відбивають основні бізнес-правила предметної області. Діаграми «сутність-зв’язок» на відміну від функціональних діаграм визначають специфікації структур даних програмного забезпечення. Перший варіант моделі «сутність-зв’язок» був запропонований П. Ченом. Надалі багатьма авторами були розроблені свої варіанти подібних моделей.

Усі варіанти діаграм «сутність-зв’язок» виходять з однієї ідеї – графічне зображення наочніше такі діаграми використовують графічне зображення сутностей предметної області, їх властивостей (атрибутів) і взаємозв’язків між сутностями. Найбільш текстового опису. Усі поширеною є нотація Баркера, її і дотримуватимемося.

Базовими поняттями ER-моделі даних (ER – Entity-Relationship) є сутність, атрибут і зв’язок.

Сутність – це клас однотипних реальних або абстрактних об’єктів (людей, подій, станів, предметів тощо), інформація про яких має істотне значення для даної предметної області. Структурою даних називають сукупність правил і обмежень, які відбивають зв’язки, що існують між окремими частинами (елементами) даних.

Кожна сутність повинна мати:

- унікальне ім’я;
- один або декілька атрибутів, які або належать сутності, або наслідують через зв’язок;
- один або декілька атрибутів, які однозначно ідентифікують кожен екземпляр сутності.

Екземпляр сутності – це конкретний представник цієї сутності. Ім’я сутності повинне відбивати тип або клас об’єкту, а не його конкретний екземпляр (студент, а не Іванов).

На діаграмі в нотації Баркера сутність зображається прямокутником, іноді із закругленими кутами (рис. 8.12, *а).* Кожна сутність має один або декілька атрибутів.

Атрибут – будь-яка характеристика сутності, значима для даної предметної області і призначена для кваліфікації, ідентифікації, класифікації, кількісної характеристики або вираження стану сутності (рис. 8.12, *б).* Атрибут, таким чином, є деяким типом характеристик або властивостей, що асоціюються з множиною реальних або абстрактних об’єктів. Екземпляр атрибуту – певна характеристика конкретного екземпляра сутності.

Рисунок 8.12 – Позначення сутності в нотації Баркера:
а – без атрибутів; *б* – з вказівкою атрибутів; *в* – з уточненням атрибутів і їх типів (# – ключовий, * – обов’язковий, ° – необов’язковий)

Атрибути діляться на ключові, тобто такі, що входять до складу унікального ідентифікатора ключа, і описові – інші.

Первинний ключ – це атрибут або сукупність атрибутів і (чи) зв'язків, призначена для унікальної ідентифікації кожного екземпляра суті (сукупність ознак, що дозволяють ідентифікувати об'єкт). Ключові атрибути поміщають в початок списку і позначають символом «#» (рис. 8.12, *в).*

Описові атрибути можуть бути обов'язковими або необов'язковими. Обов'язкові атрибути для кожної сутності завжди мають конкретне значення, необов'язкові можуть бути не визначені. Обов'язкові і необов'язкові описові атрибути позначають символами «*» і «°» відповідно.

Зв'язок – це відношення однієї сутності до іншої або до самої собі. Кожен зв'язок може мати один з двох модальностей зв'язків. Якщо будь-який екземпляр однієї сутності пов'язаний хоч би з одним екземпляром іншої сутності, то зв'язок є обов'язковим (рис. 8.13, *а).* Необов'язковий зв'язок є умовним відношенням між сутностями (рис. 8.13, *б).* Зв'язок може мати різну модальність з різних кінців. Кожен зв'язок може бути прочитаний як зліва направо, так і справа наліво. Кожна сутність може мати будь-яку кількість зв'язків з іншими сутностями моделі. Розрізняють три типи відношень (рис. 8.14): «один-до-одного»; «один-до-багатьох»; «багато-до-багатьох».

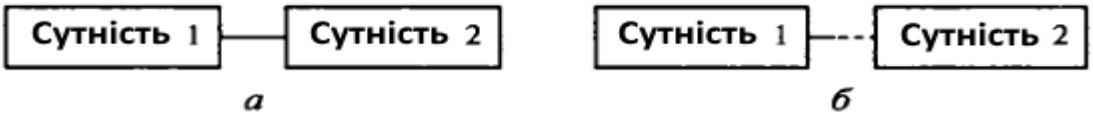


Рисунок 8.13 – Модальність зв'язку:
а – обов'язкова; *б* – необов'язкова (пунктир до середини лінії)

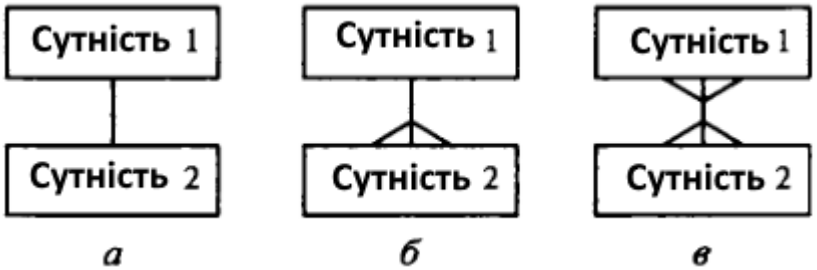


Рисунок 8.14 – Позначення відношень в нотації Баркера:
а – «один-до-одного»; *б* – «один-до-багатьохм»; *в* – «багато-до-багатьох»

Зв'язок «один-до-одного» означає, що один екземпляр першої сутності пов'язаний тільки з одним екземпляром другої сутності. До такого типу зв'язків удаються у разі економії пам'яті або, якщо є необхідність «засекречення» частини даних.

При зв'язку «один-до-багатьох» кожен екземпляр першої сутності пов'язаний з декількома екземплярами другої сутності.

Зв'язок «багато-до-багатьох» показує, що кожен екземпляр першої сутності може бути пов'язаний з декількома екземплярами другої сутності, і навпаки. Такий тип зв'язку є тимчасовим. Він допустимий на ранніх етапах

розробки моделі. Надалі такий зв'язок необхідно замінити двома зв'язками «один-до-багатьох» шляхом створення проміжної сутності.

Незалежна сутність представляє незалежні дані, які завжди є присутніми в системі. Вони можуть бути як пов'язані з іншими сутностями, так і не пов'язані.

Залежна сутність представляє дані, залежні від інших сутностей системи, тому вона завжди має бути пов'язана з іншими сутностями.

Асоційована сутність представляє дані, які асоціюються із відношеннями між двома і більше сутностями. Зазвичай цей вид сутностей з'являється в моделі для дозволу відношення «багато-до-багатьох» (рис. 8.15).

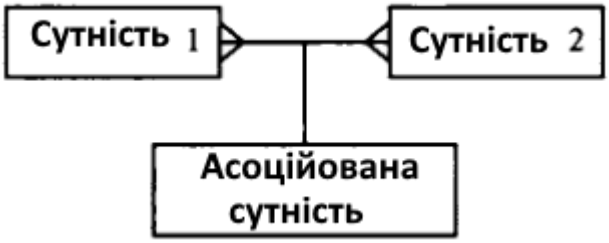


Рисунок 8.15 – Позначення асоційованої сутності в нотації Баркера

Якщо екземпляр сутності повністю ідентифікується своїми ключовими атрибутами, то говорять про повну ідентифікацію сутності. Інакше ідентифікація сутності здійснюється з використанням атрибутів пов'язаної сутності, що вказується рискою на лінії зв'язку (рис. 8.16).

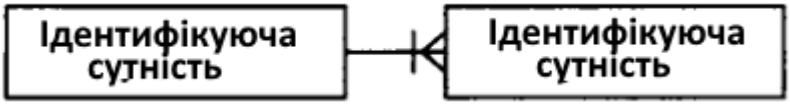


Рисунок 8.16 – Позначення ідентифікації за допомогою іншої сутності в нотації Баркера

Приклад розробки діаграми «сутність – зв'язок» АІС «Склад оптової торгівлі». Основними сутностями для виконання вказаного завдання є: постачальник, покупець, товар (рис. 8.17) [38].

Відразу виникає очевидний зв'язок між сутностями – «покупці можуть купувати багато товарів», «товари можуть отримуватися багатьма покупцями». Відношення між ними належить до типу «багато-до-багатьох» (рис. 8.17, *а).* Для реалізації цього відношення введемо асоційовану сутність «Накладна», яка відбиває придбання (продаж) товару покупцем (постачальником) (рис. 8.17, *б).*

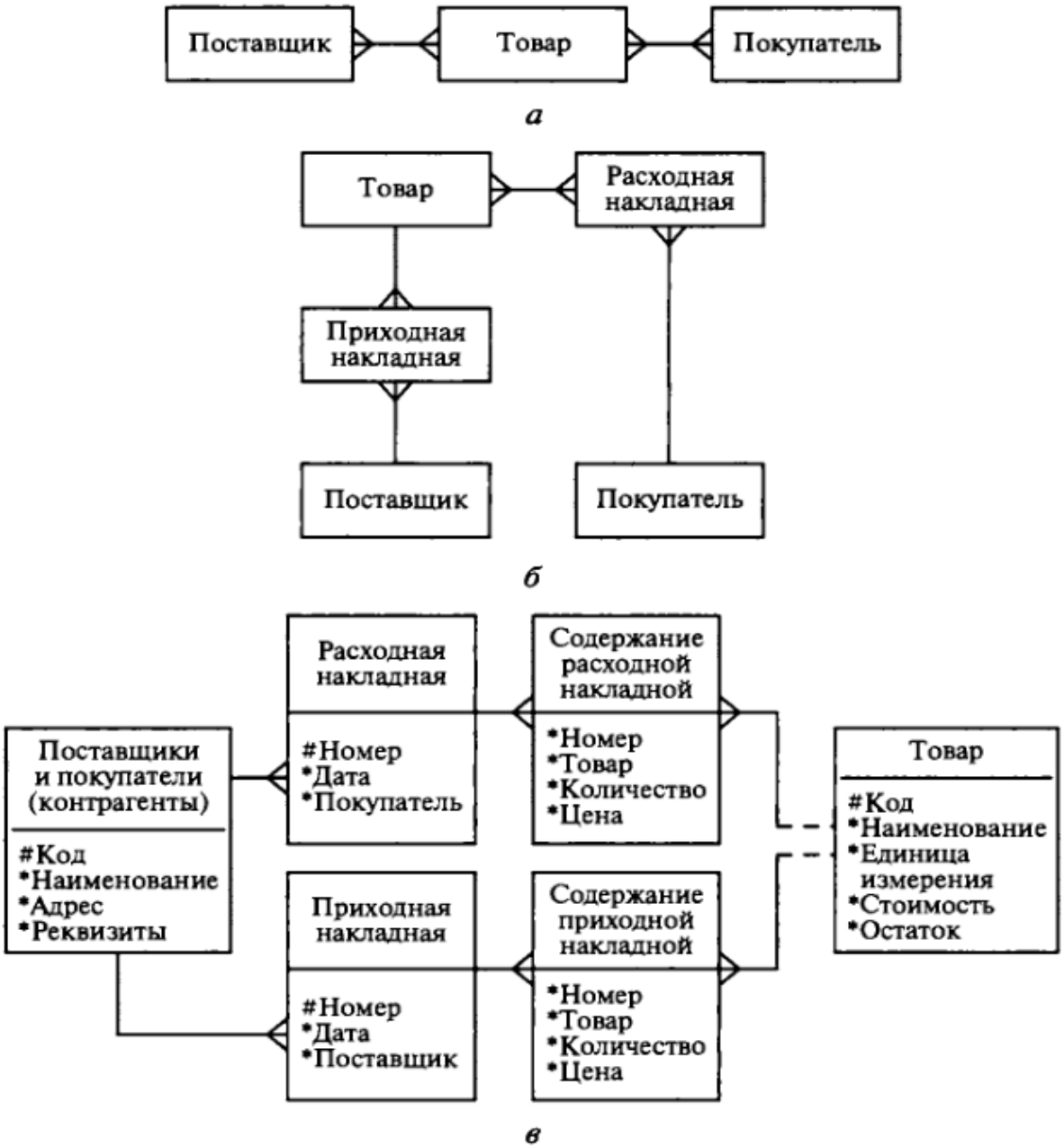


Рисунок 8.17 – Варіанти ER-диаграммы:
а – перший; *б* – проміжний; *в* – остаточний

Проаналізуємо атрибути сутностей. Кожен постачальник і покупець є юридичною особою і має найменування, адресу, банківські реквізити. Кожен товар має найменування, ціну, характеризується одиницею виміру. Кожна накладна має унікальний номер, дату виписки, список товарів з кількостями і цінами, а також загальну суму. Покупці купують товари, отримуючи при цьому витратні накладні, в які внесені дані про кількість і ціну придбаного товару. Кожен покупець може отримати декілька накладних. Кожна накладна повинна виписуватися на одного покупця. Кожна накладна повинна містити не менше одного товару (не може бути «порожньої» накладної). Кожен товар, у свою чергу, може бути проданий декільком покупцям по декількох накладних.

Аналогічний ланцюг міркувань можна збудувати для визначення зв’язків між сутностями «Товар» і «Постачальник». Покупець може бути одночасно і

постачальником, тому ці дві сутності об’єднані в одну – «Контрагент». Тепер можна усе це внести в діаграму, як показано на рис. 8.17, *в*.

Уточнена діаграма має бути перевірена з точки зору можливості отримання усіх вихідних даних (звітів), вказаних в технічному завданні або показаних на діаграмі потоків даних системи, що розробляється.

Контрольні питання до розділу 8

1. Що таке структурний аналіз?
2. Перерахуйте принципи створення діаграм. Поясніть призначення кожного виду діаграми.
3. У чому полягає суть методу структурного аналізу?
4. У чому полягають основні принципи структурного підходу?
5. Що визначається на етапі структурного аналізу?
6. Що загального і в чому відмінності між функціональною моделлю SADT і діаграмою потоків даних?
7. Назвіть переваги та недоліки структурного підходу.

РОЗДІЛ 9. ПРОЕКТУВАННЯ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ ПРИ СТРУКТУРНОМУ ПІДХОДІ

9.1. Структурна схема

Процес проектування програмного забезпечення включає визначення структурних компонентів програмної системи і зв'язків між ними. Результат уточнення структури може бути представлений у вигляді структурної схеми, яка дає досить повне уявлення про проектоване програмне забезпечення.

Структурна схема розробляється на початкових стадіях проектування і передує розробці схем інших типів. ***Структурна схема*** визначає основні функціональні частини системи, їх призначення і взаємозв'язки між ними. Схема відображає принцип дії системи в найзагальнішому вигляді.

На рис. 9.1 наведена структурна схема програмного забезпечення автоматизованої інформаційної системи «Склад оптової торгівлі» [38].

Рисунок 9.1 – Структурна схема програмного забезпечення АИС «Склад оптової торгівлі»

9.2. Функціональна схема

Функціональна схема – це схема взаємодії компонентів програмного забезпечення з описом інформаційних потоків, складу даних в потоках і вказівкою використовуваних файлів і пристроїв. Функціональні схеми інформативніші, ніж структурні.

Схеми можуть використовуватися на різних рівнях деталізації, при цьому число рівнів залежить від розмірів і складності завдання обробки даних. Рівень деталізації має бути таким, щоб різні частини і взаємозв'язок між ними були зрозумілі в цілому.

Схеми даних відображають шлях данних при розв'язанні задач і визначають етапи обробки, а також різні вживані носії даних. Схема даних складається з таких символів:

- символи даних (символи даних можуть також вказувати вид носія даних);
- символи процесу, що виконується з даними (символи процесу можуть також вказувати функції, що дотримуватися з урахуванням логічних умов);
- лінійні символи, що вказують потік управління;
- спеціальні символи, що використовуються для полегшення написання і читання схеми.

Схеми роботи системи відображають управління операціями і потік даних в системі. Схема роботи системи складається з таких символів:

- символи даних, що вказують на наявність даних (символи даних можуть також вказувати вид носія даних);
- символи процесу, що вказують, які операції слід виконати над даними, а також визначають логічний шлях, якого слід дотримуватися;
- лінійні символи, що вказують потоки даних між процесами і (чи) носіями даних, а також потік управління між процесами;
- спеціальні символи, що використовуються для полегшення написання і читання блок-схеми.

Для зображення функціональних схем використовують спеціальні позначення, встановлені ГОСТ 19.701-90.

Контрольні питання до розділу 9

1. Наведіть приклад структурної схеми ПЗ.
2. Для чого використовують функціональні схеми?
3. Опишіть основні елементи функціональних схем ПО.
4. У чому полягають достоїнства і недоліки використання функціональних схем?
5. Що відображають схеми програм?
6. Чтр відображають схеми роботи системи?

ЧАСТИНА 4. ОБ'ЄКТНО-ОРІЄНТОВАНИЙ АНАЛІЗ І ПРОЕКТУВАННЯ ІС

Об'єктна модель принципово відрізняється від моделей, які пов'язані з традиційними методами структурного аналізу, проектування і програмування. Але це не означає, що об'єктна модель вимагає відмови від усіх раніше знайдених і випробуваних часом методів і прийомів. У об'єктно-орієнтованому підході основна категорія об'єктної моделі – клас, який об'єднує в собі на елементарному рівні як дані, так і операції, які над ними виконуються (методи). Саме з цієї точки зору зміни, пов'язані з переходом від структурного до об'єктно-орієнтованого підходу, є найпомітнішими.

До недоліків об'єктно-орієнтованого підходу належать деяке зниження продуктивності функціонування ПЗ і високі початкові витрати. Об'єктна декомпозиція істотно відрізняється від функціональної, тому перехід на нову технологію пов'язаний як з подоланням психологічних труднощів, так і додатковими фінансовими витратами.

Розробка програмного забезпечення ІС є послідовністю чітко певних етапів, на кожному з яких виконується певне завдання [40].

1. Концептуалізація системи. Придумується додаток і формулюються пробні вимоги, будується модель предметної області.

2. Формування та аналіз вимог. Поглиблене розуміння вимог досягається за допомогою побудови моделей. Мета аналізу полягає у вказівці того, ***що повинно бути зроблено*** (а не того, як це повинно бути досягнуто).

3. Проектування системи. Пропонується високорівнева стратегія розв'язання задачі створення додатка (архітектура системи), що визначає основи для подальшого проектування класів. Моделі реального світу, що отримані на етапах аналізу, розширюються і коригуються так, щоб їх можна було реалізувати на комп'ютері.

4. Реалізація. Проект перетворюється в програмний код і структури баз даних.

5. Тестування. Перевіряється, що додаток придатний для практичного використання і дійсно задовольняє поставленим вимогам.

6. Навчання. Користувачам допомагають освоїтися з додатком.

7. Розгортання. Додаток розгортається там, де планується його застосування. Здійснюється перехід з успадкованих додатків.

8. Підтримка. Забезпечується довгострокова життєздатність додатка.

Об'єктно-орієнтований підхід переносить основні зусилля із розробки програмного забезпечення на етапи аналізу і проектування. Іноді здається неправильним витрачати багато часу на аналіз і проектування системи, проте ці додаткові витрати з лишком окупаються швидкою і простою реалізацією.

Перші три етапи процесу розробки і частково четвертий етап розглядаються в цій частині посібника. Інші етапи теж важливі, але вони не належатья до предмета нашого посібника.

РОЗДІЛ 10. ОБ'ЄКТНО-ОРІЄНТОВАНА РОЗРОБКА ВИМОГ

У цьому розділі розглядається інструментарій мови UML для виконання двох пов'язаних завдань: формування початкових вимог замовника і аналізу детальних вимог [18; 21; 29; 33; 43]. Мається на увазі, що вимоги задаються у формі бажаної поведінки системи. Саме тому повинні будуватися динамічні моделі, що відбивають поведінку системи в часі. Засоби мови UML для створення динамічних моделей численні і різноманітні. В якості діаграм для формування вимог обговорюються діаграми Use Case (діаграми варіантів використання або діаграми прецедентів) і діаграма діяльності, а в якості діаграм для аналізу вимог – діаграма комунікації і діаграма послідовності.

Додатково описуються діаграми кінцевих автоматів (діаграми станів) – потужний інструмент для моделювання об'єктів, керованих подіями.

10.1. Моделювання предметної області

Щоб створити якісну ІС, не досить зрозуміти бізнес-процеси і потреби замовника. Важливо розуміти, якою саме інформацією система повинна управляти. А для цього треба знати, які об'єкти потрапляють в предметну область ІС і які логічні зв'язки між ними існують.

Модель предметної області – це найважливіша модель об'єктно-орієнтованого аналізу. Вона відображає основні (з точки зору того, хто моделює) класи понять (концептуальні класи) предметної області. У цьому розділі наводяться початкові відомості про створення моделей предметної області. Докладніше питання моделювання предметної області висвітлюються в роботі [18].

Метою побудови моделі предметної області є отримання графічного представлення логічної структури досліджувальної предметної області. Логічна модель предметної області ілюструє сутності, а також їх взаємовідносини між собою.

Якщо використати ітеративний процес розробки ПЗ (наприклад, Rational Unified Process – RUP або дуже компактну методологію екстремального програмування – eXtreme, XP-процесс), то кожній ітерації відповідатиме своя модель предметної області, що відображає сценарії прецедентів, які реалізуються на цьому етапі. Таким чином, модель предметної області еволюціонує в процесі розробки системи.

10.1.1. Поняття предметної області

У об'єктно-орієнтованих системах структура коду визначається класами. Тому, перш ніж приступати до кодування, треба з'ясувати, які класи знадобляться. З цією метою слід намалювати одну або декілька діаграм класів. Для кожного класу необхідно описати усі атрибути, тобто дані-члени, і операції, які представляють собою функції програми (методи). Іншими

словами, ми повинні ідентифікувати усі функції і упевнитися, що у нас є дані, з якими ці функції повинні працювати.

Для ілюстрації способів організації і взаємодії класів використовуватимуться діаграми класів. Отже ми маємо намір отримати дуже детальні діаграми класів рівня проектування. Говорячи про рівень проектування, ми маємо на увазі такий рівень деталізації, при якому діаграма класу може служити шаблоном для створення коду; вона повинна точно відбивати організацію майбутньої програми.

Одне з найскладніших питань при розробці об'єктно-орієнтованих програм полягає в розподілі поведінки, що має на увазі визначення функцій, з яких складатиметься програма. Треба також вирішити, якому класу повинна належати кожна функція.

Вимагається розподілити поведінку усієї системи, тобто віднести кожну функцію до певного проектованого класу. У UML для цієї мети краще всього підходять **діаграми послідовності і взаємодії** – ідеальний засіб для ухвалення рішень про розподіл поведінки. Так, наприклад, діаграми послідовності розробляються окремо для кожного сценарію і показують, який об'єкт відповідає за ту або іншу функцію. На діаграмі послідовності видно, що екземпляри об'єктів взаємодіють шляхом обміну повідомленнями. Кожне повідомлення призводить до виклику тієї або іншої функції об'єкта-одержувача. Саме тому вона прекрасно виконує завдання візуалізації розподілу поведінки.

Отже, вся складність в тому, як перейти від прецедентів до діаграм послідовності. У більшості випадків це непросте завдання, оскільки прецеденти описують систему на рівні вимог, а діаграми послідовності дають детальне представлення з точки зору проектування.

Наступний етап – перехід від нечітких формулювань прецеденту до дуже точних і детальних діаграм послідовності. Для цього робиться детальний аналіз вимог.

У верхній частині діаграми послідовності розміщується множину об'єктів, що беруть участь у цьому сценарії. Перш ніж малювати діаграму послідовності, нам потрібно, зокрема, обдумати, які об'єкти знадобляться. Непогано було б також мати уявлення про виконувані ними функції. Працюючи над діаграмою послідовності, ми роздумуватимемо про те, як відобразити безліч функцій, що реалізовують необхідну поведінку, на безліч об'єктів, що беруть участь в сценарії.

Аналіз вимог займає місце між тим, що система повинна робити, і тим, як цього досягти. Результати аналізу вимог використовуються не лише для поліпшення тексту прецеденту, але і для постійного уточнення статичної моделі. Нові об'єкти, які виявляються у міру малювання діаграм, повинні увійти до діаграм класів. На цій стадії основні атрибути доречно включити в деякі з найбільш важливих класів.

Але одне питання доки залишається відкритим. І належить воно якраз до проблеми виявлення об'єктів, забутих при першій спробі спроектувати систему.

Модель предметної області **–** це візуальне подання концептуальних класових об'єктів реального світу в термінах предметної області, а не програмні компоненти [18]. Такі моделі інакше називають концептуальними моделями, моделями об'єктів предметної області, або об'єктними моделями аналізу. Це свого роду словник основних абстракцій, тобто найважливіших іменників у просторі завдання. Іменники, які описують поняття з предметної області, називають ***доменними об'єктами***. На самому початку аналізу і проектування необхідно створити модель предметної області, в якій усі доменні об'єкти будуть зображені на одній великій діаграмі класів.

У термінології UML модель предметної області – це, по суті справи, **діаграма класів.** Зазвичай в цій моделі опускається велика частина деталей, зокрема атрибути і операції класів. Ось чому можна вважати, що модель предметної області є зведеною діаграмою класів, як візуальний словник важливих абстракцій або словник предметної області.

Словник предметної області – часто згадуване поняття об'єктно-орієнтованого аналізу і проектування. Усі теоретики об'єктно-орієнтованого підходу одностайно стверджують, що словник предметної області – найважливіший артефакт, що лежить в основі аналізу і проектування.

З цим не можна не погодитися, оскільки правильний науковий підхід до будь-якого завдання вимагає спочатку домовитися про терміни. Якщо словник виділений, то дійсно, далі усе йде відмінно. Але хто виділяє словник?

Якщо розробник, то він має бути експертом в конкретній предметній області і знати її досконально, інакше ніхто не може гарантувати повноту і адекватність словника, а помилки при виділенні базових класів належать до найважчих проектних помилок.

Якщо ж замовник, то він має бути зразком акуратності і далекоглядності, оскільки помилку замовника розробник не помітить і не виправить – помилка перейде в проект, потім в додаток і проявиться вже при експлуатації найнеприємнішим чином. Найкращий варіант – коли словник складається спільно і за декілька ітерацій.

Насправді це перший крок до отримання справжньої діаграми класів, при якому ми намагаємося представити систему в цілому. Потім у процесі роботи над прецедентами ми поступово уточнюватимемо цю діаграму і врешті-решт отримаємо детальну статичну модель системи.

Модель предметної області треба побудувати до складання варіантів використання, щоб уникнути дублювання понять. Якщо схожі назви потраплять в тексти варіантів використання, то знайти їх там набагато складніше, ніж в словнику проекту.

Необхідно відштовхуватися від моделі предметної області, яка є діаграмою класів аналітичного рівня, – перше наближення до статичної структури системи. Потім можна поступово її збагачувати, прагнучі до отримання детального проекту. Діаграма класів, яка зображена в нижній половині рис. 10.1, – це статичний опис організації коду, тоді як прецеденти і, наприклад, діаграма послідовності, описують динамічну поведінку системи під час виконання.

Свої перші уявлення про систему ми зафіксуємо у вигляді статичної моделі предметної області, після чого можна зайнятися дослідженням різних прецедентів. При опрацюванні кожного з них ми щось додаватимемо в діаграму класів. Після розгляду усіх сценаріїв, які повинна підтримувати система, додавання необхідних деталей і повторного рецензування результату повинен вийти проект, що задовольняє усім вимогам.

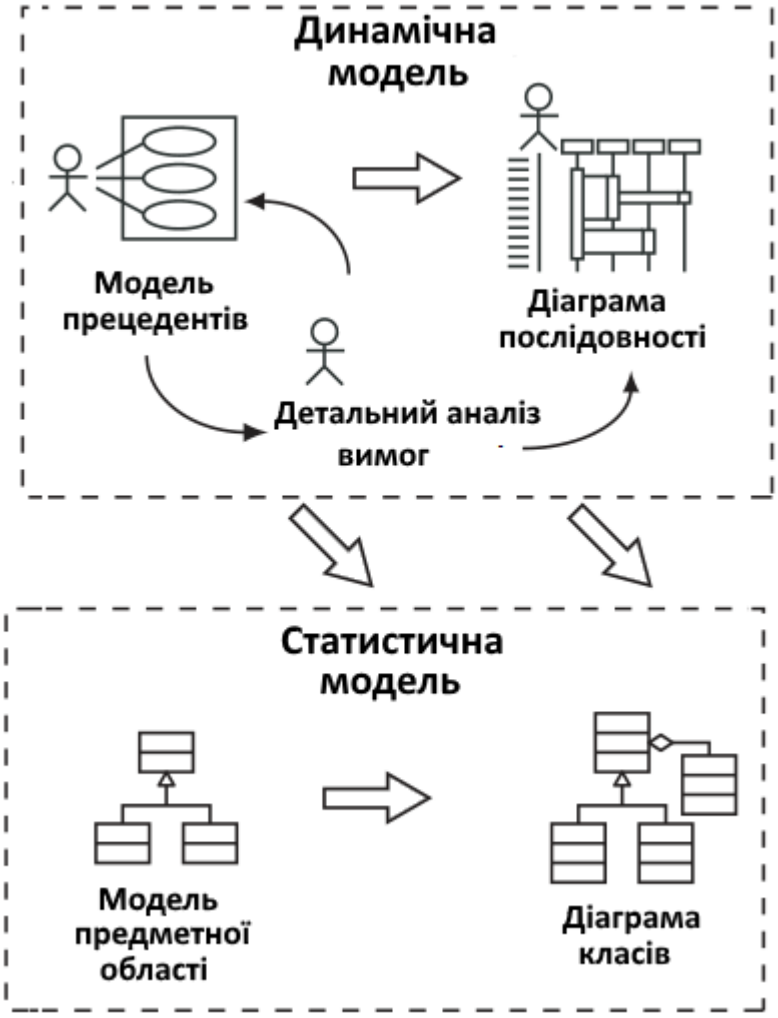


Рисунок 10.1 – Простий підхід до моделювання за допомогою UML

Для побудови моделі предметної області розглянемо реальний приклад під назвою «Книжковий Internet-магазин» з продажу книг. Ті прецеденти і класи, з якими ми зустрінемося, покликані задовольнити вимоги замовника (власника книжкового магазину).

Вимоги до системи зазвичай виглядають як фрази, побудовані у вигляді тверджень: система повинна робити те і не повинна робити це. Для побудови моделі предметної області ці вимоги видимі, з них виділяються іменники і характеризуючи їх прикметники. По них будується словник предметної області.

Виділимо **сутності предметної області** з наступних вимог [30]:

1. **Інтернет-магазин** повинен мати веб-інтерфейс, але він повинен мати можливість підключення через інші інтерфейси (веб-сервіси і т.п.).
2. Інтернет-магазин призначений для продажу **книг**, з оплатою **замовлень** через **інтернет**.

3. Користувач повинний мати можливість додати книги в онлайн **кошик**, після чого зробити **оплату.** Користувач може прибрати **предмети** з кошику.
4. Користувач повинен мати можливість вести **списки бажаних покупок**, тобто книг, які він хоче купити пізніше.
5. Користувач повинен мати можливість відмінити замовлення до того, як він відправлений поштою.
6. Користувач повинен мати можливість сплатити замовлення **кредитною карткою** або по **рахунку на оплату**.
7. Інтернет-магазин повинен вбудовуватися на сайти **партнерів** у вигляді **міні-каталогу**, який складається по **основному каталогу**, що зберігається в центральній **базі даних**.
8. Користувач повинний мати можливість створити **обліковий запис клієнта**, щоб система зберегла дані користувача (ім'я, адреси, дані банківської картки худе) в центральній базі даних і відновлювала їх при вході.
9. При вході користувача його пароль повинен звірятися з **паролем** в **основному списку паролів**, збереженим у базі даних.
10. Користувач повинен мати можливість шукати книги різними **способами пошуку –** по **заголовку**, по **автору**, **ключовому слову** або **категорії** і після пошуку переглядати **детальний опис книги**.
11. Користувач повинен мати можливість залишати відгуки на вподобані книги. Відгук повинен включати виставлений клієнтом **рейтинг** (1-5), який повинен показуватися разом із заголовком книги в **списку книг**.
12. Інтернет-магазин повинен дозволяти стороннім **продавцям** додавати свої **каталоги книг** в **основний каталог книг**, так щоб книги цих продавців були присутніми в **результатах пошуку**.

У цьому списку зустрічаються повтори, деякі з них не входять в предметну область, деякі – тільки здаються іменниками. Клієнт і Продавець – учасники взаємодії з системою (актори) і мають бути розміщені на діаграмах варіантів використання.

На термін Каталог у нас наступні кандидати: Каталог книг, Список книг, Міні-каталог, Основний каталог книг. Каталог і список є різними поняттями, тому якщо виникають сумніви, то треба уточнювати у замовника.

Інтернет – занадто загальне поняття і нічого по суті до моделі не додає. Поняття Пароль – занадто маленьке, щоб бути об'єктом і повинно відповідати елементу інтерфейсу, тому ми прибираємо його з моделі. Те ж відноситься до Заголовка і Ключового слова.

Книга і Детальний опис книги – це повтор. Залишимо Книгу, оскільки це коротша назва без втрати сенсу.

У модель предметної області не можна поміщати назви екранів або інші класи, що відносяться до інтерфейсу користувача.

Цей розділ у нас передує обговоренню прецедентів. Річ у тому, що, приступаючи до їх запису, ми можемо не викладати прецеденти з чисто призначеної для користувача точки зору. Замість цього можна буде

формулювати їх в контексті **об'єктної моделі.** Це дозволить зв'язати статичні і динамічні частини моделі, що дуже важливо, якщо ми хочемо займатися проектуванням на базі аналізу прецедентів. Модель предметної області може також представляти **словник термінів**, яким автори прецедентів можуть користуватися на пізніших етапах.

10.1.2. Основні елементи моделювання предметної області

Перше, що треба зробити при побудові статичної моделі системи, – відшукати класи, які адекватно відбивають абстракції предметної області.

Кращими джерелами класів, мабуть, є високорівневий опис завдання, низькорівневі вимоги, описи прецедентів, і експертна оцінка завдання. У міру уточнення цього переліку повинно відбуватися таке:

- іменники і іменні групи стають об'єктами і атрибутами;
- дієслова і дієслівні групи стають операціями і асоціаціями;
- родовий відмінок показує, що іменник має бути атрибутом, а не об'єктом.

Далі слід відсіяти зі списку кандидатів на звання класу непотрібні (надмірні або несуттєві) і непридатні (занадто розпливчаті або представляючі концепції, що лежать за рамками моделі) елементи.

У ході виявлення об'єктів з предметної області необхідно встановити, які зв'язки існують між ними. Особливий інтерес представляють два види відношень: ***узагальнення*** (відношення між підкласом і суперкласом, відношення виду «***є***», <|—) і ***агрегація*** (відношення між цілим і частиною, відношення виду «***має***», <>—). Між класами предметної області можуть існувати і інші відношення, у тому числі прості ***асоціації*** (——), але ці два виключно важливі. Класи, що відображають модель предметної області, потім увійдуть до основи статичної моделі діаграми класів.

У результаті модель предметної області, доповнена асоціаціями (статичними відношеннями) між парами класів, повинна адекватно описувати аспекти завдання, не залежні від часу (статичні), і в цьому нагадувати діаграми сутність-зв'язок, вживані в моделюванні даних.

Необхідно також відвести фіксований час на побудову початкової моделі предметної області. Не потрібно домагатися її досконалості, просто постарайтеся закінчити її швидше, а потім модифікуйте у міру просування вперед. Не забувайте вносити необхідні коригування в модель класів аналітичного рівня на стадії виконання аналізу придатності і на пізніших етапах процесу.

Класи в UML – це місце для розміщення атрибутів (тобто даних-членів) і операцій (тобто функцій, що виконуються об'єктами). Проте, починаючи моделювати предметну область, не потрібно витрачати багато часу на ідентифікацію атрибутів і операцій; цим ми займемося пізніше, при уточненні і наповненні статичної частини моделі. Зараз же слід сконцентрувати увагу на виявленні власне об'єктів і відношень між ними.

На рис. 10.2 наведена повна модель предметної області для книжкового Internet-магазину.

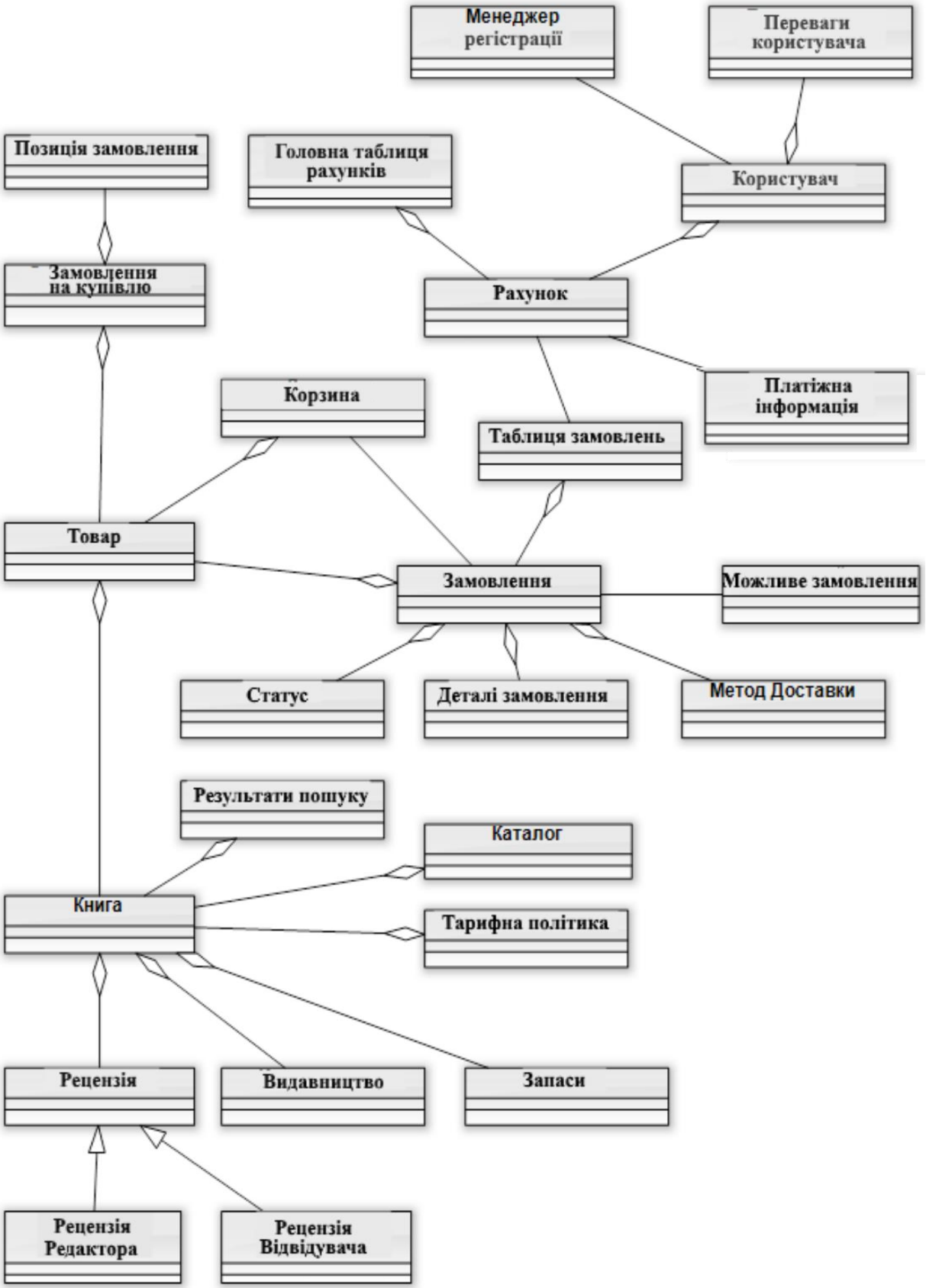


Рисунок 10.2 – Модель предметної області Internet-магазину

Для відображення класів в діаграмі класів аналізу, окрім стандартного позначення класу (прямокутник), можна використати й інші варіанти відображення класів аналізу (табл. 10.1) [17].

Таблиця 10.1 – Варіанти відображення класів аналізу

Варіанти відображення	Граничний клас	Управляючий клас	Клас сутності
Графічний стереотип	Діалогове вікно «Нормативи»	Розрахунок Vдоп	План
Стандартне позначення з рядком-стереотипом (прямокутник)	«boundary» Діалогове вікно «Нориативи»	«control» Розрахунок Vдоп	«entity» План

Назначение классов анализа.

Граничний клас – використовується для моделювання взаємодії між системою і акторами. Взаємодія часто включає отримання або передачу інформації, запити на надання послуг тощо. Граничні класи є абстракціями діалогових вікон, форм, панелей, комунікаційних інтерфейсів, інтерфейсів периферійних пристроїв, інтерфейсів API і т.п. Кожен граничний клас має бути пов'язаний як мінімум з одним актором.

Управляючий клас – відповідає за координацію, взаємодію і управління іншими об'єктами, виконує складні обчислення, управляє безпекою, транзакціями і т.п.

Клас сутності – використовується для моделювання довгоживучої, нерідко такої, що зберігається інформації. Класи суті є абстракціями основних понять предметної області – людей, об'єктів, документів і т. д., як правило, що зберігаються в табличному або іншому виді.

Наведемо приклади зв'язків між класами аналізу цього типу з використанням выдношень трьох типів.

Відношення агрегації вказує на відношення «частина-ціле» і відображається суцільною лінією з незакращенным ромбом з боку «цілого». Це відношення, як і асоціація, означає, що «об'єкт-ціле» містить посилання на «об'єкт-частину». «Об'єкт-частина» також може містити посилання на «об'єкт-ціле» (рис. 10.3). Агрегація може вказуватися тільки між класами одного типу.

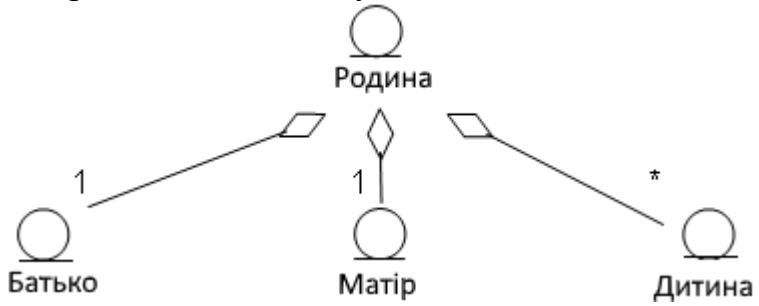


Рисунок 10.3 – Приклад агрегації

Відношення узагальнення є відношенням між загальнішим (абстрактним) класом (батьком або предком) і його окремим випадком (дочірнім класом або нащадком). Графічно це відношення позначається суцільною лінією із

стрілкою, у вигляді незакращенного трикутника, від нащадка до батька (рис. 10.4). Відношення узагальнення може бути тільки між класами одного виду.

Рисунок 10.4 – Приклад узагальнення

Відношення композиції аналогічно агрегації, в якій «частини» не можуть існувати окремо від «цілого». Стосовно класів (об'єктам) це означає, що при знищенні «об'єкту-цілого» мають бути знищені усі пов'язані з ним «об'єкти-частини». При цьому допускається створення «об'єктів-частин» набагато пізніше або знищення набагато раніше «об'єкту-цілого» (рис. 10.5).

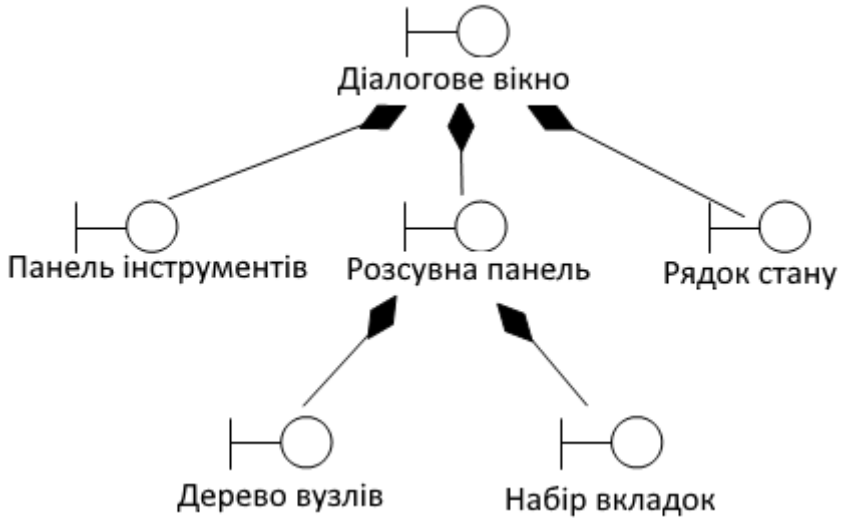


Рисунок 10.5 – Приклад композиції

Відношення залежності стосовно діаграми класів аналізу означає, що в специфікації або тілі методів об' єктів одного класу(покладу) виконується звернення до атрибутів, методам або безпосередньо до об' єктів іншого класу(непокладу). Графічно це відношення позначається штриховою стрілкою від залежного класу до незалежного. Например, в теле метода инициализации init() класса UserPUT выполняется обращение к методу getProperties(), определенному в классе System.

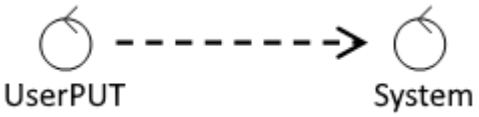


Рисунок 10.6 – Приклад залежності

Перерахуємо **типові помилки**, які початківці допускають при моделюванні предметної області:

1. *Відразу починають призначати кратності асоціаціям, стараючись, щоб у кожної асоціації була кратність*. При моделюванні предметної області не слід звертати увагу на кратності, оскільки це тільки віднімає час.
2. *Так ретельно виконують аналіз іменників і дієслів, що забувають про усе інше. Є* ризик спуститися на занадто низький рівень абстракції. Користуйтеся цією методикою, щоб почати виявлення об'єктів, але не занадто захоплюйтеся.
3. *Включають в класи операції, не вивчивши як слід прецеденти і діаграми послідовності*. Не варто приділяти надто багато уваги визначенню операцій на етапі моделювання предметної області. У цей момент ще бракує інформації для ухвалення обгрунтованих рішень.
4. *З приводу кожної асоціації виду «є частиною» сперечаються, що використати – агрегацію або композицію*. Оригінальний опис відношення «має по посиланню», трансформувалося в UML в поняття агрегації. А відношення «має за значенням» стало «сильною» формою агрегації – композицією (мається на увазі, що батьківський клас володіє классом-»частиною»: якщо батько знищується, то автоматично ліквідовуються і усі екземпляри об'єктів, що входять в нього). Спроба розрізнити ці випадки на етапі моделювання предметної області – вірний спосіб збитися з шляху. На цій стадії необхідно говорити просто про агрегацію. Точніший вибір слід відкласти до етапу детального проектування.
5. *Уникайте спроб побудувати повну модель предметної області на початкових етапах розробки.* Такий стиль властивий каскадному процесу розробки, коли модель предметної області будувалася на етапі аналізу без урахування зворотного зв'язку. Виділяйте на побудову моделі предметної області не більше декількох годин на кожній ітерації.

10.2. Формування та аналіз вимог

У цьому розділі розглядаються різновиди вимог, що пред'являються до програмних систем, обговорюються їх характеристики і пояснюються основні процеси для роботи з вимогами: формування вимог, аналіз вимог, управління змінами вимог [21; 36].

Щоб побудувати що-небудь, ми, передусім, повинні зрозуміти, чим *це повинно бути.* Процес розуміння і документування цього і називається ***аналізом вимог***.

Вимога – це умова чи можливість, якої повинна відповідати система. Основне завдання етапу визначення вимог – знайти, обговорити і зафіксувати, що дійсно вимагається від системи у формі, зрозумілій і клієнтам і членам команди розробників. Результатом аналізу вимог є документ, який зазвичай називають *специфікацією вимог, або специфікацією вимог до програмного забезпечення*.

10.2.1. Види вимог до програмного забезпечення

Вимогами називають опис функціональних можливостей і обмежень, що накладаються на створювану програмну систему. Зазвичай вимоги виражають, що система повинна робити. Тут не намагаються сформулювати, як добитися виконання цих функцій.

Наприклад, можлива така вимога до банківської системи:

Система повинна надати клієнтові можливості виконання таких операцій над його рахунком: перегляд, зняття грошей, додавання грошей.

А ось такий запис вимогою не є:

Інформація банківського рахунку повинна зберігатися у вигляді трьох таблиць СУБД MySQL.

Тут вказується як має бути побудована система, бо, що вона повинна робити. Втім, можуть бути і виключення з цього правила. Наприклад, у замовника можуть бути особливі причини для такої реалізації.

Розрізняють дві категорії представлення вимог: вимоги замовника (первинні вимоги) і вимоги розробника (детальні вимоги). Відрізняються вони одна від одної мірою опрацювання описів.

Первинні вимоги документують бажання і потреби замовника і пишуться мовою, зрозумілою замовникові. ***Детальні вимоги*** документують вимоги в спеціальній, структурованій формі, вони деталізовані стосовно до первинних вимог.

Роботу із створення первинних вимог називають ***збором*** або ***формуванням вимог***. Проводиться вона на етапі підготовки життєвого циклу розробки.

Роботу із створення детальних вимог називають ***аналізом вимог***. Проводиться вона на етапі моделювання життєвого циклу розробки.

Деякі проблеми можуть бути породжені відсутністю чіткого розуміння відмінності між цими категоріями вимог. Пояснимо відмінність між ними.

Наприклад, вимога замовника має вигляд:

ВИМОГИ ЗАМОВНИКА
1. ПЗ повинно забезпечити засоби для введення і збереження різноманітних даних абонента-користувача

а вимога розробника, що відповідає їй, може записуватися в структурованій формі:

ВИМОГИ РОЗРОБНИКА
1.1. Користувач повинен мати можливість визначати тип даних, що вводяться. *1.2. Для кожного типу даних має бути відповідний засіб, що забезпечує введення і збереження елементу даних цього типу.* *1.3. Кожен тип даних повинен представлятися відповідною піктограмою*

на дисплеї користувача.
1.4. Користувачеві повинна пропонуватися піктограма для кожного типу даних. Крім того, повинна пропонуватися можливість самостійного вибору піктограми для кожного типу даних.
1.5. При виборі користувачем піктограми типу даних до елементу даних має бути застосований засіб, що асоціюється з вказаним типом.

Таким чином, вимоги замовника є первинним описом природною мовою функцій, що виконуються системою, і обмежень, що накладаються на неї. Додатково до них можуть прикладатися пояснюючі діаграми. Вимоги замовника поміщаються в системну специфікацію.

Вимоги розробника містять деталізований опис функцій і обмежень системи, що оформляється у вигляді специфікації аналізу. Вона служить основою для укладення контракту між покупцем і розробниками. Дуже часто стандарти програмної інженерії інтегрують обидві специфікації (замовника і розробника) в єдиний документ (рис. 10.7).

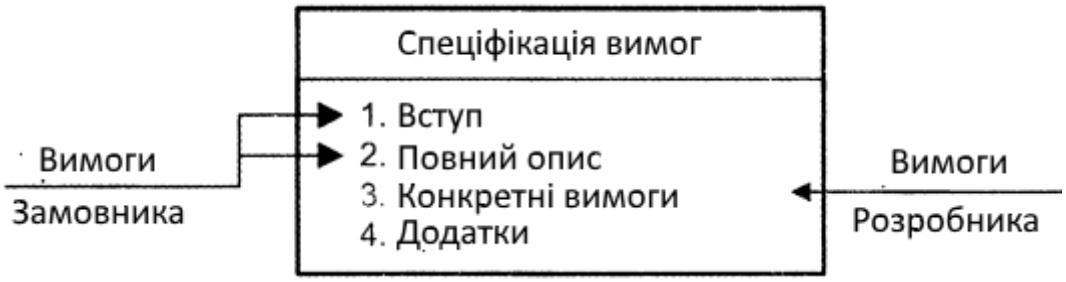


Рисунок 10.7 – Вимоги замовника і розробника

Розрізняють два види вимог:

1. ***Функціональні вимоги***. Вони описують поведінку системи і сервіси (функції), які вона повинна виконувати. При цьому виходять з усебічного аналізу проблемної (предметної) області. Розглядаються різноманітні варіанти поведінки, що визначаються різними даними і станами зовнішнього середовища.
2. ***Нефункціональні вимоги***. Ці вимоги належать до характеристик системи і її зовнішнього оточення, критеріїв ефективності і безпеки. Додатково можуть перераховуватися обмеження, що накладаються на дії і функції системи, а також на умови розробки.

Функціональні вимоги задають роботу, яку повинна виконувати програмна система, наприклад:

ПО обчислюватиме вартість товарів, що знаходяться в кошику користувача.

З іншого боку, така вимога, як

обчислення вартості товарів повинне робитися менш ніж за одну секунду

не є функціональним, оскільки воно не задає конкретну роботу. Замість цього воно оцінює роботу, тобто визначає деяке твердження про роботу.

Приклад 10.1. Функціональні вимоги для ПЗ системи управління літального апарату мають вигляд:

1. Система повинна вимірювати проекції швидкості (трьома каналами), обчислювати відхилення від програмних значень вектора швидкості і радіус-вектора, а потім формувати дії, що управляють, посилаючи їх на виконавчі органи літального апарату (ЛА).
2. Система повинна забезпечувати кутову стабілізацію руху ЛА: вимір кутів (трьома каналами), обчислення відхилень кутів від програмних значень, формування дій, що управляють, розподіл дій, що управляють, між виконавчими органами.
3. Кожна програма автопілота повинна забезпечуватися унікальним ідентифікатором (Program_ID), який записується в «чорний ящик» разом з параметрами польоту.

Перша вимога описує цикл поведінки системи. Друга вимога визначає набір функцій системи, а третя вимога вводить комбінацію поведінки і конкретного параметра системи.

Очевидно, що форми запису вимог можуть бути найрізноманітнішими. Проте вимоги мають бути:

- *ясними* (не допускати двоякого тлумачення, що призводить до спотворення змісту побажань замовника);
- *погодженими* (не містити суперечливих і взаємовиключних тверджень);
- *повними* (визначати усю необхідну функціональність системи).

Нефункціональні вимоги не пов’язані безпосередньо з функціями системи. Багато нефункціональних вимог стосуються системи в цілому, а не окремих її елементів. Це означає, що вони можуть бути критичнішими, ніж одиничні функціональні вимоги. Помилка у функціональній вимозі може знизити функціональні можливості системи, а помилка в нефункціональній вимозі може призвести до відмови усієї системи. І. Соммервілл запропонував виділяти три групи нефункціональних вимог [13]:

1. Вимоги до програмної системи. Описують властивості і характеристики системи. Сюди належать вимоги до швидкості роботи, продуктивності, місткості необхідної пам’яті, надійності, переносимості системи на різні комп’ютерні платформи і зручності експлуатації.
2. Організаційні вимоги. Відображають питання роботи і організації взаємодії замовника і розробника. Вони включають стандарти розробки програмної системи, вимоги до методів і засобів розробки, вказують терміни створення і набір документації.
3. Зовнішні вимоги. Враховують чинники зовнішнього середовища. Вони визначають вимоги по взаємодії цієї системи із зовнішнім оточенням, юридичні зобов’язання, а також етичні вимоги, що гарантують прийнятність системи для користувачів.

Приклад 10.2. Нефункціональні вимоги до продуктивності.

Вимоги до продуктивності визначають часові обмеження, які мають бути виконані в системі. Обговорюють обмеження за часом обчислення, періодичності обчислень, використанню оперативної пам'яті, використанню зовнішніх пристроїв тощо:

Цикл регулювання швидкості літального апарату повинен укладатися в 64 мс.

Вимоги до продуктивності є важливою частиною систем, що працюють в реальному часі, в яких дії повинні укластися в певні часові рамки. Прикладами систем реального часу можуть бути системи запобігання зіткненням, управління польотом.

Приклад 10.3. Нефункціональних вимог до надійності і доступності. Вимоги надійності задають рівень надійності, визначаючи межі (чи величину) неідеальної роботи системи:

Система управління мікрокліматом оранжереї повинна давати не більше двох помилок на місяць.

Доступність, близька за змістом до надійності, оцінює міру доступності системи користувачам. Наприклад:

Система продажу авіаквитків має бути доступна користувачам 24 години на добу. Вона може бути недоступна (знаходитися в стані профілактики) впродовж 10 хвилин за 30-денний період.

Приклад 10.4. Нефункціональні вимоги обмежень. Обмеження описують межі характеристик або умов роботи системи, втім, можуть обмежуватися і умови розробки:

Система управління крилатої ракети (СУ КР) повинна розраховувати координати цілі з точністю до трьох метрів.

Часто накладаються обмеження за інструментами і мовами. Вони обумовлені традиціями організації, що склалися, досвідом програмістів:

СУ КР має бути розроблена мовою Ada 2005.
Управління конфігурацією розробки повинно проводитися в середовищі утиліти IBM Rational ClearCase.

10.2.2. Формування вимог

Мета цієї роботи – сформувати (визначити) вимоги замовника.

Вимоги замовника представляються в такій формі, що вони зрозумілі будь-якому користувачеві, що не володіє спеціальними технічними знаннями, і членам команди розробників. Первинні вимоги повинні визначати тільки зовнішню поведінку ПЗ, без деталізації структурної організації системи. Вони записуються природною мовою з використанням простих таблиць, а також наочних діаграм, рисунків.

У 1960-1970-і роки було поширено загальну помилку про те, що повний аналіз вимог необхідно виконувати на початку роботи над проектом із створення системи (передбачався каскадний процес розробки). На початку 1980-х років стало зрозуміло, що так вчиняти неправильно. Такий підхід призвів до краху багатьох проектів. Тому будь-які методи, які припускають повне визначення і фіксацію вимог на початку розробки, приречені на невдачу, оскільки базуються на неправильних припущеннях, що виключають неминучу зміну вимог в процесі розробки.

Однією з головних причин провалу проектів з розробки програмних систем є недостатнє залучення користувачів до процесу розробки. Тому важливість участі користувачів у процесі формування вимог і аналізу системи важко переоцінити.

Опишемо кроки процесу формування вимог.

Крок 1. Визначення представників замовника. Важливо виявити таке коло осіб, яке дозволить скласти комплексне уявлення про портрет майбутньої системи: її функціональності і повному переліку характеристик. Іноді це зробити досить складно. У будь-якому випадку потрібно розібратися в предметній області системи, виявити коло користувачів і інших зацікавлених осіб, до складу яких може увійти керівництво замовника (і навіть керівництво розробників), обслуговуючий персонал тощо.

Крок 2. Проведення опитування представників замовника. Оскільки зазвичай є декілька зацікавлених осіб, перше питання – вирішити, в якому порядку їх опитувати. Очевидно, що треба розставити пріоритети в списку опитуваних (згідно з вашими припущеннями про важливість отримуваних відомостей). Далі планується час і тривалість опитувань, на яких мають бути присутніми як мінімум два члени команди розробників із засобами запису розмов (диктофони, відеокамери тощо).

Під час інтерв'ю слід сконцентруватися на слуханні, брати участь в діалозі (потрібно ставити питання і мотивувати співрозмовника, уточнювати побажання і потреби, пропонувати варіанти поведінки і використання системи), робити детальні замітки. У кінці опитування слід запланувати наступну зустріч.

Пара розробників, що беруть участь в опитуванні і підтримують один одного, забезпечують можливість оперативного формулювання «важких» вимог із слів замовника. Часто замовник сам формулює вимоги по ходу розмови, але іноді потребує допомоги. У цих випадках розробник і замовник спільно «шліфують» концепцію вимоги. Іншими словами, замовникові бувають потрібні підказки для формування концепції.

Ефективним способом отримання і формулювання вимог є приклади. Ці приклади пропонуються розробниками, як правило, в графічній формі.

Крок 3. Документування результатів опитування. Після кожного опитування створюється чернетка – письмова форма набору вимог. Вона відсилається замовникам для коментарів і корекції. Як правило, потім проводиться серія повторних опитувань. Завершується серія опитувань

зборами. За підсумками зборів готується документ, що містить усі вимоги і представляється у форматі стандарту, вибраного для специфікації вимог. Цей документ затверджується замовником.

Крок 4. Перевірка вимог. Мета перевірки специфікації вимог полягає в оцінці правильності визначень, які в ній містяться. Перевірка гарантує, що усі положення вимог коректні, відбивають бажані характеристики і задовольняють потребам замовника. Може виявитися, що вимоги, які в специфікації виглядали чудово, при реалізації багаті проблемами. Розробники випробовують серйозний дискомфорт при реалізації неясних або неповних вимог. Доведено, що виправлення помилок у вимогах, робота над якими вже завершена, вимагає дуже багато зусиль. Дослідження показали: виправляти помилки вимог у кінці розробки системи в 100 разів дорожче, ніж в ході формування цих вимог.

Перевірка вимог виконується замовником і розробником спільно, вона засвідчує що:

- предметна область проекту описана коректно;
- розробник і замовник мають однакові уявлення про цілі системи;
- специфікація вимог вірно описує бажану функціональність і характеристики системи, які відповідають потребам замовника і інших зацікавлених осіб:
- вимоги повні і якісні;
- усі вимоги погоджені один з одним, не містять протиріч.

Перевірка повинна підтвердити, що в специфікації є присутніми тільки якісні вимоги як за змістом (коректні, повні, погоджені), так і за формою запису (ясні, такі, що легко модифікуються).

Зрозуміло, що перевірити можна тільки задокументовані, а не уявні вимоги. Річ у тому, що багато організацій не справляються з формуванням вимог. Це не означає, що вони не користуються вимогами – просто вимоги існують лише в головах конкретних розробників. У результаті крах програмного проекту стає майже неминучим.

10.2.3. Аналіз вимог

Формування вимог є лише початковою фазою роботи з вимогами.

Аналіз вимог розглядає вимоги замовника як початкові дані, на виході аналізу – вимоги розробника, які справедливо називають ***детальними вимогами***. Аналіз вимог служить мостом між підготовкою, плануванням і проектуванням ПЗ. Тут відбувається перехід зі світу замовника у світ розробника. Міняється мова запису вимог. Тепер це не природна мова людини, а мова формалізованих моделей. Ці моделі не зрозумілі замовникові, але близькі розробникові. За допомогою цих моделей можна добитися точнішого відображення безлічі деталей, властивих програмній системі.

Звичайно, що ці моделі теж відповідають принципам побудови вимог: показувати ***«що»*** потрібно робити, а не ***«як»*** це робиться. Але, з іншого боку,

для повного опису системи потрібна така деталізація, яка повинна включати інформацію про організацію системи на рівні архітектури.

Розробникам програмних систем потрібний базис для проектування і конструювання. Цей базис і утворюється набором детальних вимог, які детально, повно і погоджено описують властивості і функціональність системи. Кожна з цих вимог нумерується і відстежується по ходу розробки. Реалізація кожної вимоги тестується. Ясно, що детальні вимоги повинні «виникати» з первинних вимог.

Аналіз вимог є необхідною і дуже важливою процедурою, яка надалі допоможе оптимізувати роботу команди і уникнути нерозумінь, а також дозволяє зрозуміти, чи можна в принципі виконати ці вимоги – з точки зору часу, ресурсів і бюджету. Ще одним аргументом на користь детального аналізу вимог є те, що, за різними оцінками, в них зароджується від ½ до ¾ усіх проблем з програмним забезпеченням.

Зазвичай детально аналізуються такі вимоги:

– вимоги, що описують функціональність проекту;
– інтерфейс користувача,
– апаратний і програмний інтерфейси;
– критерії ефективності;
– критерії безпеки і коректності системи.

Оскільки більшість помилок тягнуться саме з вимог до ПЗ, то треба з цим якось боротися, тобто зробити так, щоб не було таких проблем:

– незрозумілість вимог;
– часта змінність;
– зміни, що вносяться в останню мить;
– невірний аналіз вимог.

Із-за всього переліченого вище може статися таке:

– зрив терміну проекту;
– буде зроблено не те і не так як треба;
– зміни не контролюються, і команда не знає, що робити.

Щоб уникнути усього цього слід зробити так, щоб вимоги були:

– завершеними;
– несуперечливими;
– коректними;
– недвозначними.

У процесі аналізу вимог перевіряється їх відповідність певному набору властивостей.

Завершеність. Вимога повинна бути повною і закінченою з точки зору представлення в ній усієї необхідній інформації. Вимога повинна містити усю інформацію, необхідну для розробників, ніщо не пропущене з міркувань «це і так всім зрозуміло».

Типові проблеми із завершеністю:

1. Відсутні нефункціональні складові вимоги або посилання на відповідні нефункціональні вимоги. Наприклад, «*паролі повинні зберігатися в зашифрованому вигляді»* – який алгоритм шифрування?

2. Вказана лише частина деякого перерахування. Наприклад, «*експорт здійснюється у формати PDF, PNG тощо»*. – що ми повинні розуміти під «тощо».?
3. Приведені посилання неоднозначні. Наприклад, «*див. вище»* замість «*див. підрозділ 2.4»*.

Атомарність, одиничність. Вимога є атомарною, якщо її не можна розбити на окремі вимоги без втрати завершеності, і вона описує одну і тільки одну ситуацію.

Типові проблеми з атомарністю:

1. В одній вимозі, фактично, міститься декілька незалежних. Наприклад, «*кнопка «Restart» не повинна відображатися при зупиненому сервісі, вікно «Log» повинне вміщувати не менше 20-ти записів про останні дії користувача»* – тут навіщось в одній пропозиції описані абсолютно різні елементи інтерфейсу в абсолютно різних контекстах.
2. Вимога допускає різночитання в силу граматичних особливостей мови. Наприклад, «*якщо користувач підтверджує замовлення і редагує замовлення або відкладає замовлення, то повинен видаватися запит на оплату»* – тут описані три різні випадки, і цю вимогу варто розбити на три окремих щоб уникнути плутанини. Таке порушення атомарності часто спричиняє за собою виникнення суперечності.
3. В одній вимозі об'єднаний опис декількох незалежних ситуацій. Наприклад, «*коли користувач входить в систему, йому повинне відображатися вітання; коли користувач увійшов до системи, повинне відображатися ім'я користувача; коли користувач виходить з системи, повинне відображатися прощання»* – усі ці три ситуації заслуговують того, щоб бути описаними окремими і куди детальнішими вимогами.

Несуперечність, послідовність. Вимога не повинна містити внутрішніх протиріч і протиріч іншим вимогам і документам, вимоги мають бути зрозумілими.

Типові проблеми з несуперечністю:

1. Протиріччя усередині однієї вимоги. Наприклад, «*після успішного входу в систему користувача, що не має права входити в систему».* – тоді як він успішно увійшов до системи, якщо не мав такого права?
2. Протиріччя між двома і більше вимогами, між таблицею і текстом, малюнком і текстом, вимогою і прототипом тощо. Наприклад, «*Кнопка «Close» завжди має бути червоною»* і «*Кнопка «Close» завжди має бути синьою»* – так все ж червоною чи синьою?
3. Використання невірної термінології або використання різних термінів для позначення одного і того ж об'єкту або явища. Наприклад, «*у разі, якщо роздільна здатність вікна складає менше 800x600»*. – роздільна здатність є у екрану, а у вікна є розмір.

Недвозначність. Вимога описана без використання жаргону, неочевидних абревіатур і розпливчатих формулювань, і допускає тільки однозначне об'єктивне розуміння. Вимога атомарна в плані неможливості різного трактування окремих фраз.

Типові проблеми з недвозначністю:

1. Використання термінів або фраз, що допускають суб'єктивне тлумачення. Наприклад, «*додаток повинен підтримувати передачу великих об'ємів даних»* – наскільки «великих?
2. Використання неочевидних або двозначних абревіатур без розшифровки.

Здійсненність. Вимога технологічно здійсненна і може бути реалізована у рамках бюджету і термінів розробки проекту.

Типові проблеми із здійсненністю:

1. Вимоги технічно не реалізовуються на сучасному рівні розвитку технологій. Наприклад, «*аналіз договорів повинен виконуватися із застосуванням штучного інтелекту, який виноситиме однозначне коректне заключення про міру вигоди від укладення договору».*
2. Вимоги, що в принципі не реалізовуються. Наприклад, «*система пошуку повинна заздалегідь передбачати усі можливі варіанти пошукових запитів і кешувати їх результати».*

Обов'язковість, потрібність і актуальність. Якщо вимога не є обов'язковою до реалізації, вона повинна бути просто виключена з набору вимог. Якщо вимога потрібна, але «не дуже важлива», для вказівки цього факту використовується вказівка пріоритету (див. «Упорядкованість за …».). Також мають бути виключені (або перероблені) вимоги, що втратили актуальність.

Типові проблеми з обов'язковістю і актуальністю :

1. Вимога була додана «про всяк випадок», хоча реальної потреби в ній не було і немає.
2. Вимозі виставлені невірні значення пріоритету за критеріями важливості і/або терміновості.
3. Вимога застаріла, але не було перероблена або видалена.

Відстеженість. Відстеження буває вертикальним і горизонтальним. Вертикальне відстеження дозволяє співвідносити між собою вимоги на різних рівнях вимог. Горизонтальне відстеження дозволяє співвідносити вимогу з тест-планом, тест-кейсами, архітектурними рішеннями тощо.

Для забезпечення відстеження часто використовуються спеціальні інструменти з управління вимогами і/або матриці відстеження.

Типові проблеми з відстеженістю:

1. Вимоги не пронумеровані, не структуровані, не мають змісту, не мають працюючих перехресних посилань.
2. При розробці вимог не були використані інструменти і техніка управління вимогами.
3. Набір вимог неповний, носить уривчастий характер з явними «пропусками».

Модифікованість. Ця властивість характеризує простоту внесення змін до окремих вимог і в набір вимог. Можна говорити про наявність модифікованості у тому випадку, коли при доопрацюванні вимог шукану інформацію легко знайти, а її зміна не призводить до порушення інших описаних в цьому переліку властивостей.

Типові проблеми модифікованістю:

1. Вимоги неатомарні (див. «Атомарність») і не відстежені (див. «Відстеженість»), а тому їх зміна з високою ймовірністю породжує суперечність (див. «Несуперечність»).
2. Вимоги від самого початку суперечливі (див. «Несуперечність»). У такій ситуації внесення змін (не пов'язаних з усуненням суперечності) тільки посилює ситуацію, збільшуючи суперечність і знижуючи відстеженість.
3. Вимоги представлені в незручній для обробки формі. Наприклад, не використані інструменти управління вимогами, і у результаті команді доводиться працювати з десятками величезних текстових документів.

Упорядкованість за важливості, стабільності, терміновості. Важливість характеризує залежність успіху проекту від успіху реалізації вимоги. Стабільність характеризує вірогідність того, що в осяжному майбутньому у вимогу не буде внесено ніяких змін. Терміновість визначає розподіл в часі зусиль проектної команди по реалізації тієї або іншої вимоги.

Типові проблеми з упорядкованістю полягають в її відсутності або невірній реалізації і призводять до показаних нижче наслідків.

1. Проблеми з упорядкованістю по важливості підвищують ризик невірного розподілу зусиль проектної команди, спрямування зусиль на другорядні завдання і ведуть до кінцевого провалу проекту із-за нездатності продукту виконувати ключові завдання з дотриманням ключових умов.
2. Проблеми з упорядкованістю по стабільності підвищують ризик виконання безглуздої роботи з удосконалення, реалізації і тестуванню вимог, які в найближчий час можуть зазнати кардинальних змін (аж до повної втрати актуальності).
3. Проблеми з упорядкованістю по стабільності підвищують ризик порушення бажаної замовником послідовності реалізації функціональності і введення цієї функціональності в експлуатацію.

Коректність і верифікованість. Фактично ці властивості витікають з дотримання усіх вище перелічених характеристик. Тобто, можна сказати, що вони не виконуються, якщо порушена хоч би одна з вище перелічених характеристик. На додаток можна відмітити, що верифікованість має на увазі спосіб однозначної перевірки «виконані вимоги або ні», можливість створення об'єктивного тест-кейсу (тест-кейсів), що однозначно показує, що вимога реалізована вірно і поведінка додатка в точності відповідає вимозі.

До типових проблем з коректністю можна віднести:

1. Друкарські помилки. Особливо небезпечні друкарські помилки в абревіатурах, що перетворюють одну осмислену абревіатуру на іншу

також осмислену, але таку, що не має відношення до деякого контексту; такі друкарські помилки украй складно помітити.

2. Наявність неаргументованих вимог до дизайну і архітектури.
3. Погане оформлення тексту і супутньої графічної інформації, граматичні, пунктуаційні та інші помилки в тексті.
4. Невірний рівень деталізації. Наприклад, занадто глибока деталізація вимоги на рівні бізнес-вимог, або недостатня деталізація на рівні вимог до продукту.
5. Вимоги до користувача, а не до додатка. Наприклад, «*користувач має бути в змозі відправити повідомлення»* – на жаль, розробники не можуть впливати на стан користувача.

Детальні вимоги орієнтовані на читання розробниками, написані вони їх «рідною» мовою. Правда, замовники теж можуть їх оцінити (за допомогою розробників).

Розглянемо типові кроки аналізу вимог.

Крок 1. Організація первинних вимог. Необхідність цієї роботи обумовлена великою кількістю вимог. У міру їх розростання неорганізований список швидко перетворюється на некерований. Стандарти рекомендують декілька способів організації:

1. *За режимом*. Деякі системи міняють поведінку залежно від режиму роботи. Наприклад, система управління може мати різні набори функцій залежно від режиму: навчання, нормальний режим або аварійний режим.
2. *За категоріями користувачів*. Деякі системи забезпечують різні набори функцій для різних категорій користувачів. Наприклад, система управління ліфтом надає різні можливості для пасажирів, обслуговуючого персоналу і пожежників.
3. *За об'єктами*. Об'єкти – це програмні сутності системи, які можуть мати фізичні аналоги в зовнішньому середовищі. Наприклад, в системі контролю за пацієнтом об'єкти включають пацієнтів, датчики, медсестер, приміщення, лікарів, ліки. Кожен об'єкт несе в собі набір даних і функцій. Функції об'єктів також називають послугами, методами, операціями.
4. *За властивостями*. Властивість – сервіс, що надається зовнішньому середовищу, визначається за допомогою пар «вхідна дія – реакція». Наприклад, у телефонній системі властивості включають локальний виклик, переадресацію викликів і циркулярний виклик. Зверніть увагу, що «реакція» може бути розосереджена по різних частинах програмної системи.
5. *За відгуками*. Деякі системи організовуються за допомогою опису усіх функцій, що підтримують генерацію різних відгуків. Наприклад, функції системи обліку персоналу можуть бути організовані в розділи, що

відповідають усім функціям для складання чека по оплаті, усім функціям для складання списку службовців і т.д.

Крок 2. Перетворення первинних вимог в детальні вимоги. Звичайна одна вимога замовника перетвориться в декілька детальних вимог, хоча можливо і відображення «один в один». Рекомендації по роботі з детальною вимогою:

1. *Первинна оцінка*. Можливі варіанти: 1) ця функціональна вимога – відповідає методу реалізації; 2) занадто велике – важко управляти, слід розділити на частини: 3) занадто маленьке – немає сенсу розглядати окремо, потрібно приєднати до іншої вимоги.
2. *Забезпечення тестируемости вимоги*. Написання тестів для перевірки реалізації вимоги. Продумуються варіанти як позитивного, так і негативного результату тестів.
3. *Аналіз однозначності тлумачення вимоги.*
4. *Призначення пріоритету вимоги*. Вибираються варіанти: істотне, бажане або необов'язкове.
5. *Перевірка повноти вимоги*. Слід переконатися в наявності усіх «забезпечуючих» вимог.
6. *Перевірка узгодженості вимоги* з іншими вимогами. Аналізуються і усуваються можливі протиріччя.
7. Вимога заноситься в специфікацію аналізу (специфікацію вимог).

Крок 3. Атестація детальних вимог. Атестація (валідація) повинна підтвердити, що вимоги дійсно визначають ту систему, яка потрібна замовникові. Атестація дуже важлива, оскільки помилки вимог можуть призвести до істотної переробки системи і великих витрат, якщо будуть виявлені при розробці або при експлуатації системи. Змістовно до складу атестації входять:

1. *Перевірка правильності вимог*. Замовник вважає, що система потрібна для виконання заданих ним функцій. Проте обговорення із зацікавленими особами і подальший аналіз можуть виявити додаткові функції, і їх теж потрібно врахувати.
2. *Перевірка на несуперечність*. Вимоги не повинні визначати суперечливі факти. Інакше кажучи, у вимогах не повинно бути обмежень, що суперечать одне одному, або різних визначень однієї і тієї ж функції.
3. *Перевірка на повноту*. Вимоги повинні описувати усі необхідні функції і обмеження системи.
4. *Перевірка на здійснимість*. Аналізуються ті вимоги, що реалізовуються у рамках тимчасових і бюджетних обмежень проекту.

Атестація вимог – дороге і складне заняття, що вимагає високої кваліфікації, хорошого кругозору і розвиненої уяви. Рідко вдається виявити усі проблеми вимог, тому повертатися до атестації доводиться багаторазово.

10.2.4. Специфікація вимог

Специфікація вимог – це документ, що є офіційним приписом для розробників ПЗ. Він містить опис вимог замовника (первинних вимог) і розробника (детальних вимог). Первинні вимоги документуються при формуванні вимог, а детальні вимоги – при виконанні аналізу вимог.

Багато організацій розробляли стандарти документування вимог. Найповнішим і найавторитетнішим вважають стандарт Інституту інженерів з електротехніки і радіоелектроніки IEEE Std 830-1998, який повинен допомогти:

- замовникам ПЗ точно описати, що вони хочуть отримати;
- розробникам ПЗ точно зрозуміти, що хоче замовник.

Цей стандарт стверджує, що якісно складена специфікація програмного забезпечення (Software Requirements Specification, SRS) повинна принести замовникам, розробникам і іншим особам певні вигоди:

1. *Створити основу для угоди між замовниками і розробниками з приводу функцій, які повинен виконувати програмний продукт.* Повний опис функцій ПЗ, приведений в SRS, допоможе потенційним користувачам визначити, чи відповідає ПЗ їх потребам або як необхідно змінити ПЗ, щоб задовольнити ці потреби.
2. *Зменшити об'єм робіт з розробки*. Підготовка SRS змушує замовника розглянути вагу вимоги до початку проекту і скорочує подальше повторне проектування, кодування і тестування. Ретельний аналіз вимог, вказаних в SRS, може розкрити упущення, неправильне розуміння і протиріччя на ранніх стадіях циклу розробки, коли ці проблеми простіше виправити.
3. *Забезпечити основу для оцінки витрат і графіків робіт*. Опис продукту, що розробляється відповідно до SRS, є практичною основою для оцінки витрат на проект і може використовуватися для формування контракту і бюджетних обмежень.
4. *Забезпечити основу для атестації (валідації) та верифікації.* При використанні якісної SRS організації можуть підвищити ефективність планів атестації та верифікації.
5. *Полегшити передачу користувачам*. SRS робить простішою передачу програмного виробу новим користувачам або його установку на нових комп'ютерах. Таким чином, для замовників спрощується передача ПЗ іншим підрозділам їх організації, а для розробників спрощується передача ПЗ новим замовникам.
6. *Служити в якості основи для розширення*. Оскільки в SRS обговорюється продукт, а не проект, в якому він розроблений, то SRS служить основою для подальшого розширення готового продукту. SRS може дещо змінитися, але залишиться базисом для подальшої еволюції продукту.

10.2.5. Управління вимогами

Вимоги в життєвому циклі розробки програмної системи постійно змінюються. Після створення початкової версії вимог приходить нове, глибше розуміння предметної області ПЗ, промальовуються нові деталі, які раніше були непомітні. Звідси новий виток роботи над вимогами. Постійність вимог – це, швидше, виключення із загального правила. Щоб зміни вимог не поховали проект завчасно, процесом зміни вимог потрібно управляти.

В ході управління вимогами треба вирішити низку питань:

1. *Розпізнавання і облік вимог*. Кожна вимога має бути індивідуально врахована, оскільки воно може перетинатися з іншими вимогами.
2. *Управління внесенням змін*. Повинна передбачатися послідовність захисних дій для оцінки дії зміни і вартості зміни.
3. *Стратегія трасування*. Існують залежності між вимогами, а також між вимогами і проектними рішеннями системи. Трасування повинне виявляти залежні вимоги, запам’ятовувати ці залежності і відстежувати вплив вимог один на одного і на проектні рішення.

Управління вимогами потребує автоматизованої підтримки, яку забезпечують програмні утиліти. Утиліти підтримують такі дії:

1. *Зберігання вимог*. Інформація про вимоги повинна зберігатися в захищеній керованій пам’яті, доступній для учасників процесу розробки вимог.
2. *Реалізація циклу зміни вимоги*. Зміна вимоги відбувається в інтерактивному режимі: організовується діалог співробітника з програмною утилітою.
3. *Управління трасуванням*. Утиліта виявляє залежні вимоги, виконуючи дії трасування.

10.3. Формування вимог за допомогою діаграми Use Case

Досвід показує, що моделі формування та аналізу вимог повинні доповнювати, а не замінювати специфікації вимог на природній мові (тобто вимоги в текстовому уявленні).

У цьому розділі обговорюється основне питання, яке необхідно задати на початку розробки будь-якої системи: «*Що хочуть зробити її користувачі»?*. Ми спробуємо як можна детальніше представити дії користувачів і реакцію системи, оскільки її поведінка обумовлена їх вимогами. Іншими словами, робота системи залежить від того, як до неї звертаються користувачі і чого вони хочуть від неї добитися.

Впродовж досить тривалого періоду часу в процесі як об’єктно-орієнтованого, так і традиційного структурного проектування розробники використали типові сценарії, що допомагають краще зрозуміти вимоги до системи. Ці сценарії трактувалися дуже неформально – вони майже завжди використовувалися і украй рідко документувалися. Івар Якобсон уперше ввів

поняття «**Варіант використання**» (use case) і надав йому таку значущість, що він перетворився на основний елемент розробки і планування проекту [17].

Діаграми варіантів використання дозволяють отримати відмінне візуальне уявлення про вимоги користувачів. Діаграма Use Case визначає поведінку системи з точки зору користувача. Діаграма Use Case розглядається як головний засіб для первинного моделювання динаміки системи, використовується для з'ясування вимог замовника до системи, що розробляється, фіксації цих вимог у формі, яка дозволить проводити подальшу розробку [29].

У російській (українській) літературі діаграми Use Case часто називають ***діаграмами варіантів використання, діаграмами використання*** або ***діаграмами прецедентів***.

Прецеденти (Варіанти використання, Use Case) **–** це розповідні історії про використання системи, які широко використовуються для осмислення і формулювання вимог. Вони роблять вплив на безліч аспектів проекту, включаючи ООАП. ***Прецеденти – це не діаграми, а текстові описи.*** Оскільки опис варіантів використання легко зрозуміти, варіанти використання є особливо корисним способом комунікації із замовником.

Частенько прецеденти треба продумувати набагато детальніше. Але основна ідея полягає в дослідженні і формулюванні функціональних вимог шляхом написання історій «з життя системи». Ці історії допомагають сформулювати різні завдання і є ***сценаріями*** використання системи – це спеціальна послідовність дій або взаємодій між виконавцями і системою. На перший погляд, описати прецеденти не складно, хоча частенько досить важко визначити, що вимагається від системи і описати це на потрібному рівні деталізації.

Важливою особливістю опису прецедентів є їх орієнтація на цілі і завдання користувача. В процесі опису необхідно ставити питання: «Хто є користувачем системи? Які типові сценарії використання системи? Які цілі і завдання користувачів»?. Такий погляд на систему значно більше орієнтований на користувача, чим звичайний список системних вимог.

Не слід плутати варіант використання з конкретними операціями майбутньої системи. Кожен варіант використання пов'язаний з деякою метою, що має самостійне значення. Наприклад, для текстового редактора «Формування змісту» – це варіант використання, а «Зв'язування заголовків» із спеціальними стилями – операція, яку необхідно виконати, щоб стала можлива автоматична побудова змісту.

Спробуємо перерахувати ті переваги, які дає цей підхід в порівнянні з іншими підходами.

Прості твердження. Моделювання використання фактично дозволяє переписати початкове технічне завдання в строгій і формальній, але, в той же час, дуже простій і наочній графічній формі, як сукупність простих тверджень стосовно того, що робить система для користувачів.

Абстрагування від реалізації. Моделювання використання припускає формулювання вимог до системи абсолютно незалежно від її реалізації. Іншими словами, представлення використання описує тільки, **що** робить система (але не як це робиться і не **навіщо** це треба робити).

Відмітимо, що інші підходи, використовуючи на перших кроках терміни і поняття реалізації (структура програми, структура даних, структура взаємодіючих об'єктів), накладають мимовільні обмеження на реалізацію, які не витікають з сутності завдання, тобто можуть служити джерелом неефективності і помилок.

Декларативний опис. Кожен варіант використання описує (а вірніше сказати, іменує) деяку множину послідовностей дій, що доставляють суттєвий для користувача результат. Проте в моделі немає вказівок на те, який варіант використання повинен виконуватися раніше, а який пізніше, тобто немає опису алгоритму, тобто немає алгоритмічних помилок.

Виявлення меж. Представлення використання визначає межі системи і постулює існування у зовнішньому світі агентів (дійових осіб), що використовують її. Опис системи у вигляді чорного ящика з певними інтерфейсами здається дуже схожим на представлення використання, але тут є важлива відмінність, яка часто упускається з виду.

Якщо обмежитися тільки описом інтерфейсів, то дуже легко допустити помилки такого типу: передбачити інтерфейс, який не потрібний, тому що ним ніхто не користується. Чи, аналогічно, забути інтерфейс, який потрібний певній категорії користувачів. На діаграмі використання самотні і покинуті дійові особи і варіанти використання виявляються з першого погляду.

Таким чином, висновок такий: моделювання використання безпечніше і надійніше за альтернативні методи, тобто за інших рівних умов дозволяє вчинити менше грубих проектних помилок на ранніх стадіях проектування. У цьому полягає основна перевага цього методу.

Але не слід думати, що моделювання використання – це панацея від усіх хвороб розробки програмного забезпечення. Існують області, де моделювання використання практично нічого не дає. Наприклад: розробка автономного компілятора певної мови програмування.

До складу діаграм варіантів використання входять елементи Use Case, актори, а також відношення залежності, узагальнення і асоціації. Як і інші діаграми, діаграми Use Case можуть включати примітки і обмеження.

Залежно від мети виконання процедури розрізняють такі варіанти використання:

- основні (базові) – забезпечують необхідну функціональність ПЗ, що розробляється;
- допоміжні – забезпечують виконання необхідних налаштувань системи і її обслуговування (наприклад, архівація інформації і тому подібне);
- додаткові – забезпечують додаткові зручності для користувача (як правило, реалізуються, якщо не вимагають серйозних витрат яких-небудь ресурсів ні при розробці, ні при експлуатації).

10.3.1. Актори і елементи Use Case

Питання про виділення (чи ідентифікацію) дійових осіб при складанні моделі – одне з найболючіших. Невдалий вибір дійових осіб може негативно вплинути на усю модель у цілому. Тут легко вдатися до крайності: оголосити, що є одна дійова особа (зовнішній світ), що взаємодіє з усіма варіантами використання або, навпаки, придумати штучних дійових осіб для кожного варіанту використання. Обидва екстремальні варіанти являються моделлю чорного ящика і зводять до нуля переваги моделювання використання.

Формального методу ідентифікації дійових осіб не існує. Тут перерахуємо деякі прийоми, які корисно мати на увазі при виділенні дійових осіб. Спершу вкажемо детальніше визначення дійової особи.

З синтаксичної точки зору *дійова особа* (актор, actor **–** користувачі, зовнішні системи, пристрої) **–** це стереотип класифікатора, який позначається спеціальним значком. Для дійової особи вказується тільки ім'я, що ідентифікує її в системі. **Семантично *дійова особа*** – це множина логічно взаємозв'язаних ролей.

Тут доречно дати визначення класифікатора. ***Класифікатор*** **(**від лат. *classis – розряд* і *facere – робити)* – систематизований перелік найменованих об'єктів, кожному з яких у відповідність даний унікальний код (ім'я). Класифікація об'єктів робиться згідно з правилами розподілу заданої множини об'єктів на підмножини (*класифікаційні угрупування*) відповідно до встановлених ознак їх відмінності або схожості.

Роль **(role)** в UML **–** це ***інтерфейс*** **(interface),** підтримуваний цим ***класифікатором*** **(classifier)** в цій ***асоціації*** **(assocation).**

З **прагматичної точки зору** головним є те, що дійові особи знаходяться поза системою, що проектується (або в частині системи, що розглядається).

У типових випадках різні дійові особи призначаються для категорій користувачів (якщо їх вдається виділити природним чином), зовнішніх програмних і апаратних засобів (якщо система взаємодіє з такими).

Виділення категорій користувачів відбувається, як правило, неформально: з міркувань здорового глузду і власного досвіду. Проте, дамо декілька порад. Має сенс віднести користувачів до різних категорій, якщо спостерігаються такі ознаки:

- користувачі беруть участь у різних (незалежних) бізнес-процесах;
- користувачі мають різні права на виконання дій і доступ до інформації;
- користувачі взаємодіють з системою в різних режимах: від випадку до випадку, регулярно, постійно.

Виділення варіантів використання – ключ до всього подальшого моделювання. На цьому етапі визначається функціональність системи, тобто, що корисного система повинна робити у зовнішньому світі.

Нотація для **варіанту (елементу) використання** дуже мізерна – це просто ім'я, поміщене в овал (чи поміщене під овалом – такий варіант теж допустимий). Іншими словами, функції, що виконуються системою, на рівні моделювання використання ніяк не розкриваються – їм тільки даються імена.

Семантично *варіант використання* (use case) – це опис множини можливих послідовностей дій (подій), що призводять до значимого для дійової особи результату. Кожна конкретна послідовність дій називається ***сценарієм*.**

Прагматика (дія) варіанту використання полягає в тому, що серед усіх послідовностей дій, які можуть статися при роботі додатка, виділяються такі, в результаті яких виходить явно видимий і досить важливий для дійової особи (зокрема, для користувача) результат.

Сказане для дійових осіб доречно повторити і для варіантів використання: вибір варіантів використання сильно впливає на якість моделі, а формальні методи запропонувати важко – допомагають тільки досвід і чуття.

Якщо у вашому розпорядженні є технічне завдання, пункти якого природним чином переводяться у варіанти використання, знайте, що вам сильно повезло. Якщо технічне завдання є сумішшю очевидних побажань користувача, смутних тверджень і конкретних вимог (як завжди буває), то спробуйте пошукати в тексті віддієслівні іменники і дієслова з прямим доповненням: може трапитися, що в них зашифровані варіанти використання. Якщо у вас зовсім немає технічного завдання, складіть його, користуючись виключно простими твердженнями.

Третім типом сутності, вживаним на діаграмі використання, є **коментар**. Відмітимо, що коментарі є дуже важливим засобом UML, значення якого часто недооцінюється початкуючими користувачами. Коментарі можна і треба вживати на усіх типах діаграм, а не тільки на діаграмах використання.

UML є уніфікованою, але ніяк не універсальною мовою – при моделюванні проектувальник часто може сказати про модельовану систему більше, ніж це дозволяє зробити строга, але обмежена нотація UML. У таких випадках найвідповідальнішим засобом для внесення в модель додаткової інформації є коментар.

На відміну від більшості мов програмування коментарі в UML синтаксично оформлені за допомогою спеціальної нотації і виступають на тих же правах, що і інші сутності. А саме, коментар має свою графічну нотацію – прямокутник із загнутим куточком («собаче вухо»), в якому знаходиться текст коментаря.

Коментарі можуть знаходитися у відношенні відповідності з іншими сутностями – ці відношення зображаються пунктирною лінією без стрілок. Якщо пунктирна лінія відсутня, то коментар належить до усієї діаграми.

Коментар містить текст, який вводить користувач – творець моделі. Це може бути текст у довільному форматі: природною мовою, мовою програмування, формальною логічною мовою і т.д. Більше того, якщо можливості інструменту це дозволяють, в примітках можна зберігати гіперпосилання, вкладені файли і інші артефакти, зовнішні стосовно до моделі.

У UML послідовно проводиться такий важливий принцип: **уся інформація, яку користувач вносить в модель, має бути збережена інструментом у внутрішньому уявленні моделі** і пред'явлена на першу вимогу, навіть якщо інструмент не уміє обробляти цю інформацію. Коментарі є найважливішим прикладом реалізації цього принципу.

Коментарі можуть мати стереотипи. У UML визначені два стандартні стереотипи для коментарів:

- «requirement» – описує загальну вимогу до системи;
- «responsibility» – описує відповідальність сутності (класифікатора).

Коментарі першого стереотипу часто є присутніми на діаграмах використання, а коментарі другого стереотипу – на діаграмах класів.

Вершинами в діаграмі Use Case є актори (виконавці) і елементи Use Case (варіанти використання). Їх позначення показані на рис. 10.8.

Рисунок 10.8 – Позначення актора і елементу Use Case

Актори представляють зовнішній світ, що потребує роботи системи. Елементи Use Case представляють дії, що виконуються системою в інтересах акторів.

Актор (дійова особа) – це роль поза системою об'єкта, який прямо взаємодіє з її частиною – конкретним елементом (елементом Use Case). Розрізняють акторів і користувачів. Користувач це фізичний об'єкт, який використовує систему (це може бути навіть не людина, а організація або машина). Він може грати декілька ролей і тому може моделюватися декількома акторами. Справедливо і зворотне – актором можуть бути різні користувачі.

Ідентифікацію прецедентів краще всього розпочати зі списку акторів, а потім розглянути, як кожен актор збирається використати систему. За допомогою такої стратегії можна отримати список потенційних прецедентів. Кожному прецеденту має бути присвоєне коротке описове ім'я, що є дієслівною групою (прецедент означає «виконати» що-небудь).

Наприклад, для комерційного літального апарату можна виділити двох акторів: пілота і касира. Сидоров – користувач, який іноді діє як пілот, а іноді як касир. Як зображено на рис. 10.9, залежно від ролі Сидоров взаємодіє з різними елементами Use Case.

Рисунок 10.9 – Модель Use Case

Елемент Use Case – це опис послідовності дій (чи декількох послідовностей), які виконуються системою і роблять для окремого актора видимий результат.

Один актор може використати декілька елементів Use Case і навпаки: один елемент Use Case може мати декілька акторів, що використовують його. Кожен елемент Use Case задає певний шлях використання системи. Набір усіх елементів Use Case визначає повні функціональні можливості системи.

Під час ідентифікації прецедентів можуть виявитися деякі нові актори. Це нормально. Моделювання прецедентів – ітеративний процес, здійснюваний шляхом поетапного уточнення. Усе розпочинається з імені прецеденту, а деталі доповнюються пізніше.

10.3.2. Відношення в діаграмах Use Case (використання)

На діаграмах використання застосовуються такі основні типи відношень:

- асоціація між дійовою особою і варіантом використання;
- узагальнення між дійовими особами;
- узагальнення між варіантами використання;
- залежності між варіантами використання.

Відношення між дійовою особою і варіантом використання. Між актором і елементом Use Case можливий тільки один вид відношень – ***асоціація***, що відображає їх взаємодію (рис. 10.10). Як і будь-яка інша асоціація, вона може бути помічена ім’ям, ролями, потужністю.

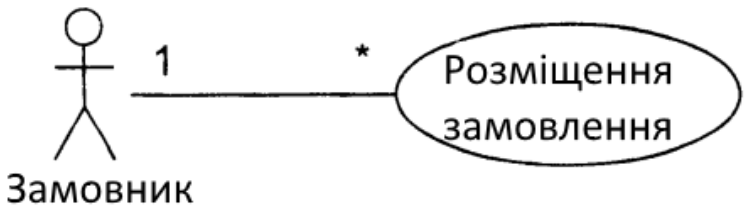


Рисунок 10.10 – Відношення асоціації

Іншими словами, ця асоціація означає, що дійова особа обов’язково безпосередньо бере участь у виконанні кожного зі сценаріїв, що описуються варіантом використання. Асоціація є найважливішим і, фактично, обов’язковим відношенням на діаграмі використання. Дійсно, якщо на діаграмі використання немає асоціацій між дійовими особами і варіантами використання, то це означає, що система не взаємодіє із зовнішнім світом. Такі системи, так само як і їх моделі, не мають практичного сенсу.

Узагальнення між дійовими особами. Між акторами допустиме відношення узагальнення (рис. 10.11), що означає, що екземпляр нащадка може взаємодіяти з такими ж різновидами екземплярів елементів Use Case, що і екземпляр батька. Узагальнення акторів виносить поведінку, загальну для двох або більше за акторів, в актора-батька.

Між елементами Use Case визначено відношення ***узагальнення і два різновиди відношення залежності – включення і розширення***.

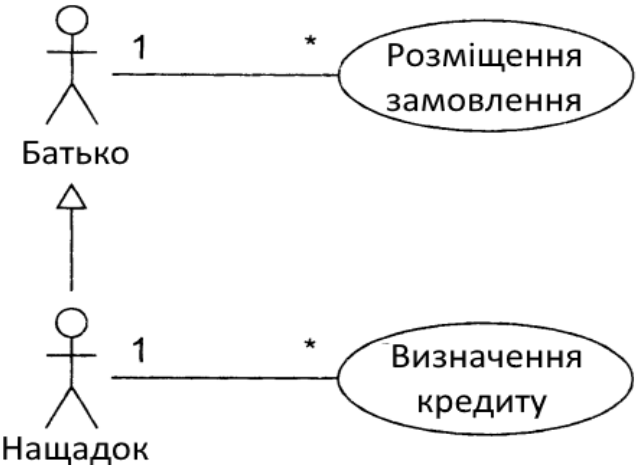


Рисунок 10.11 – Відношення узагальнення між акторами

Узагальнення між варіантами використання. Відношення узагальнення (рис. 10.12) фіксує, що нащадок наслідує поведінку батька. Крім того, нащадок може доповнити або перевизначити поведінку батька. Елемент Use Case, що є нащадком, може заміщати елемент Use Case, що є батьком, у будь-якому місці діаграми. Тобто, узагальнення прецедентів виносить поведінку, загальну для одного або більше за прецеденти, у батьківський прецедент.

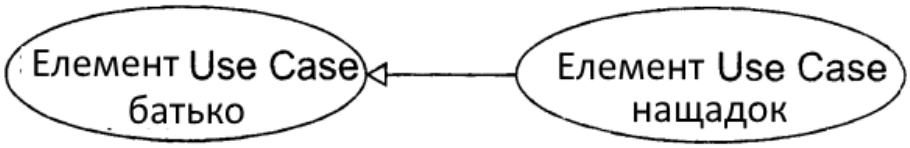


Рисунок 10.12 – Відношення узагальнення між елементами Use Case

Відношення включення між елементами Use Case («include»). Відношення включення (рис. 10.13) означає, що базовий елемент Use Case (прецедент) явно включає поведінку іншого елементу Use Case в точці, яка визначена у базі. Елемент Use Case, що включається, ніколи не використовується самостійно – його зміст може бути тільки частиною іншого, більшого елементу Use Case. Тобто, відношення «include» виносить кроки, загальні для декількох прецедентів, в окремий прецедент, який потім включається в інші.

Відношення включення є прикладом відношення делегації. При цьому в окреме місце (елемент Use Case, що включається) поміщається певний набір обов'язків системи. Далі інші частини системи можуть агрегувати в себе ці обов'язки (за необхідності).

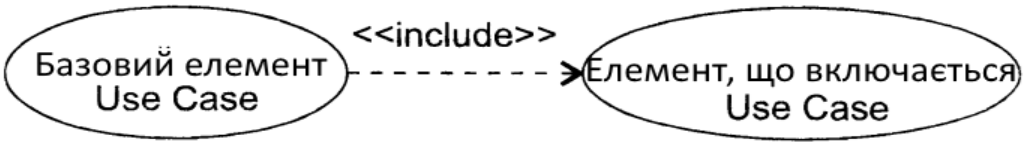


Рисунок 10.13 – Відношення включення між елементами Use Case

Відношення розширення між елементами Use Case («extend»). Відношення розширення (рис. 10.14) означає, що базовий елемент Use Case неявно включає поведінку іншого елементу Use Case в точці, яка визначається побічно розширеним елементом Use Case.

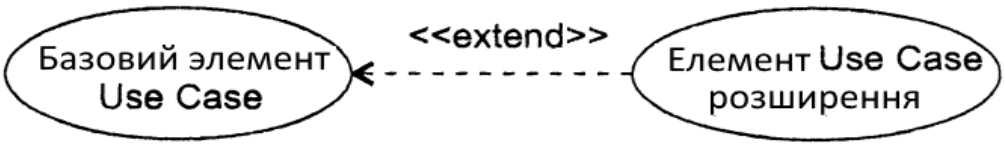


Рисунок 10.14 – Відношення розширення між елементами Use Case

Базовий елемент Use Case може бути автономним, але за певних умов його поведінка може розширюватися поведінкою іншого елементу Use Case. Базовий елемент Use Case може розширюватися тільки в певних точках – точках розширення. Відношення розширення застосовується для моделювання більш детальної поведінки системи. У такий спосіб можна відокремити обов'язкову поведінку від необов'язкової поведінки.

Наприклад, можна використати відношення розширення для окремого потоку, який виконується тільки за певних умов, що знаходяться поза увагою базового елементу Use Case. Нарешті, можна моделювати окремі потоки, вставка яких в певну точку управляється актором.

Багато розробників рекомендують уникати відношення розширення, а більше уваги приділяти текстовому опису прецеденту [36].

Приклад найпростішої діаграми Use Case, в якій використані відношення включення і розширення, наведений на рис. 10.15.

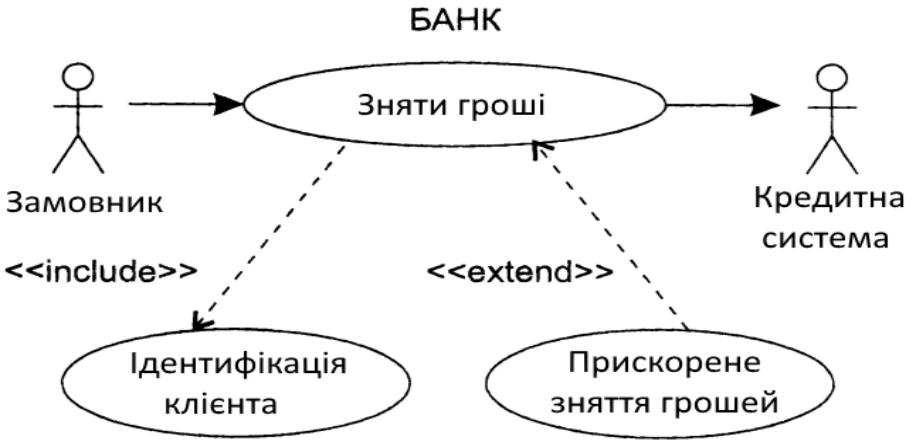


Рисунок 10.15 – Проста діаграма Use Case для банку

Як показано на рис. 10.16, усередині елемента Use Case може бути додаткова секція із заголовком Extention points. У цій області перераховуються точки розширення. У вказану тут точку «*додаткові запити*» вставляється послідовність дій від розширеного елементу Use Case «Запит каталогу». Для довідки, відзначимо, що точка розширення розміщена після дій, що забезпечують створення замовлення. На цьому ж рисунку відображені відношення спадкоємства між елементами Use Case. Видно, що елементи Use Case Оплата готівкою і Оплата в кредит наслідують поведінку елемента Use Case Зробити Оплату і є його спеціалізаціями.

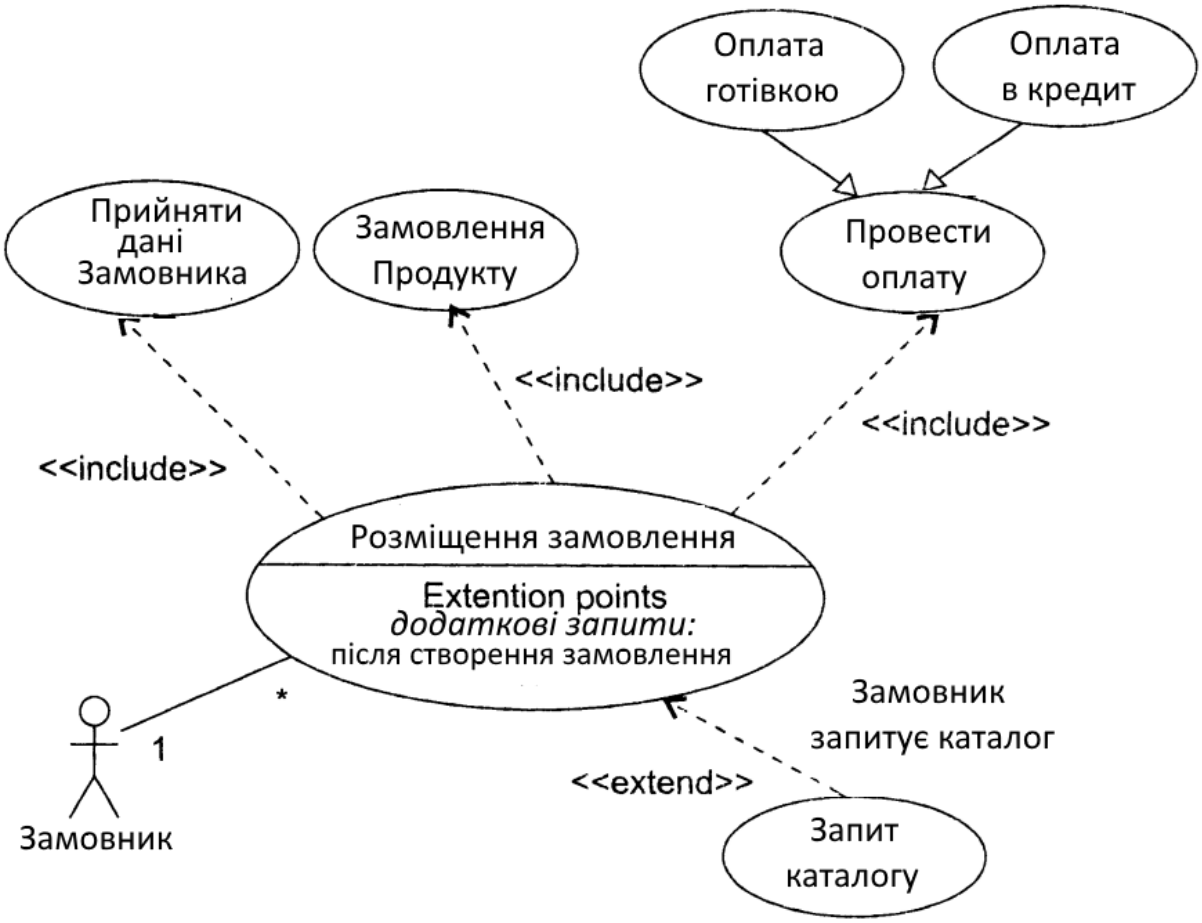


Рисунок 10.16 – Діаграма Use Case для обслуговування замовника

10.3.3. Робота з елементами Use Case

Елемент Use Case описує, ***що*** повинна робити система, але не визначає, ***як*** вона повинна це робити. При моделюванні це дозволяє відділяти зовнішнє представлення системи від її внутрішнього представлення.

Поведінка елементу Use Case описується потоком подій. Початковий опис виконується в текстовій формі, прозорій для користувача системи. У потоці подій виділяють:

- основний потік і альтернативні потоки поведінки;
- як і коли стартує і закінчується елемент Use Case;
- коли елемент Use Case взаємодіє з акторами;
- якими даними обмінюються актор і система.

Для уточнення і формалізації потоків подій використовують діаграми послідовності. Зазвичай одна діаграма послідовності визначає головний потік в елементі Use Case, а інші діаграми – потоки виключень.

У загальному випадку один елемент Use Case описує набір послідовностей, в якому кожна послідовність представляє можливий потік подій. Кожна послідовність називається сценарієм.

Сценарій – конкретна послідовність дій, яка ілюструє поведінку елемента Use Case. Сценарії є для елементу Use Case майже тим же, чим є екземпляри для класу. Вважають, що сценарій – це екземпляр елементу Use Case.

10.3.4. Специфікація елементів Use Case

Специфікація елементу Use Case – основне джерело інформації для виконання аналізу вимог і проектування системи. Дуже важливо, щоб зміст специфікації був представлений в повній і конструктивній формі. У загальному випадку специфікація включає головний потік, підпотоки і альтернативні потоки поведінки. В якості шаблону специфікації представимо опис елементу Use Case «*Купувати авіаквиток»* для моделі інформаційної системи авіакаси.

Передумова. Перед початком цього елементу Use Case має бути виконаний елемент Use Case «Заповнити базу даних авіарейсів».

Головний потік. Цей елемент Use Case починається, коли покупець реєструється в системі і вводить свій пароль. Система перевіряє – чи правильний пароль (E-1), і пропонує покупцеві вибрати одну з дій: СТВОРИТИ, ВИДАЛИТИ, ПЕРЕВІРИТИ, ВИКОНАТИ, ВИХІД.

1. Якщо вибрана дія СТВОРИТИ, виконується підпотік S-1: створити замовлення авіаквитка.
2. Якщо вибрана дія ВИДАЛИТИ, виконується підпотік S-2: видалити замовлення авіаквитка.
3. Якщо вибрана дія ПЕРЕВІРИТИ, виконується підпотік S-3: перевірити замовлення авіаквитка.
4. Якщо вибрана дія ВИКОНАТИ, виконується підпотік S-4: реалізувати замовлення авіаквитка.
5. Якщо вибрана дія ВИХІД, елемент Use Case закінчується.

Підпотоки.

S-1: створити замовлення авіаквитка. Система відображає діалогове вікно, що містить поля для пункту призначення і дати польоту. Покупець вводить пункт призначення і дату польоту (E-2). Система відображає параметри авіарейсів (E-3). Покупець вибирає авіарейс. Система зв’язує покупця з вибраним авіарейсом (E-4). Повернення до початку елементу Use Case.

S-2: видалити замовлення авіаквитка. Система відображає параметри замовлення. Покупець підтверджує рішення про ліквідацію замовлення (E-5). Система видаляє зв’язок з покупцем (E-6). Повернення до початку елементу Use Case.

S-3: перевірити замовлення авіаквитка. Система виводить (E-7) і відображає параметри замовлення авіаквитка: номер рейсу, пункт призначення, дата, час, місце, ціна. Коли покупець вказує, що він закінчив перевірку, виконується повернення до початку елементу Use Case.

S-4: реалізувати замовлення авіаквитка. Система просить параметри кредитної карти покупця. Покупець вводить параметри своєї кредитної карти (E-8). Повернення до початку елементу Use Case.

Альтернативні потоки.

E-1: введений неправильний ID-номер покупця. Покупець може повторити введення ID-номера або припинити елемент Use Case.

E-2: введений неправильний пункт призначення/дата польоту. Покупець може повторити введення пункту призначення/дати польоту або припинити елемент Use Case.

E-3: немає відповідних авіарейсів. Покупець інформується, що зараз такий політ неможливий. Повернення до початку елементу Use Case.

E-4: не може бути створений зв'язок між покупцем і авіарейсом. Інформація зберігається, система створить цей зв'язок пізніше. Елемент Use Case триває.

E-5: введений неправильний номер замовлення. Покупець може повторити введення правильного номера замовлення або припинити елемент Use Case.

E-6: не може бути видалений зв'язок між покупцем і авіарейсом. Інформація зберігається, система видалятиме цей зв'язок пізніше. Елемент Use Case триває.

E-7: система не може вивести інформацію замовлення. Повернення до початку елементу Use Case.

E-8: некоректні параметри кредитної карти. Покупець може повторити введення параметрів карти або припинити елемент Use Case.

Таким чином, в цій специфікації зафіксовано, що всередині елементу Use Case знаходиться один основний і дванадцять допоміжних потоків дій. Надалі розробник може прийняти рішення про виділення і цього елементу Use Case самостійних елементів Use Case. Очевидно, що якщо самостійний елемент Use Case містить підпотік, то його слід підключати до базового елементу Use Case відношенням ***include.*** У свою чергу, самостійний елемент Use Case з альтернативним потоком підключається до базового елементу Use Case відношенням ***extend***.

10.3.5. Банкомат – приклад діаграми Use Case

Найбільші труднощі при побудові діаграм Use Case викликає застосування *відношення включення і розширення*. Дуже важливо розібратися у відмітних особливостях цих відношень, специфіці взаємодії елементів Use Case, що з'єднуються за їх допомогою.

Приклад діаграми Use Case для банкомату, в якій використані відношення включення і розширення, наведений на рис. 10.17 [29].

У цій діаграмі один базовий елемент Use Case **Сеанс Банкомату,** який взаємодіє з актором **Клієнт**. До базового елементу Use Case підключені два розширенні елементи Use Case **(Стан, Зняти)** і два елементи Use Case **(Ідентифікація клієнта, Перевірка рахунку),** що включаються.

У свою чергу, до елементу Use Case **Ідентифікація клієнта** підключений елемент Use Case, що включається, **Перевірити достовірність,** а до елементу Use Case **Зняти –** розширювальний елемент Use Case **Захоплення карти** (він же розширює елемент Use Case **Перевірити достовірність**).

Рисунок 10.17 – Діаграма Use Case для банкомату (з використанням *включення* і *розширення*)

Бачимо, що елемент Use Case **Сеанс Банкомату** має дві точки розширення **(діалог можливий, видача квитанції),** а елементи Use Case **Зняти** і **Перевірити достовірність –** по одній точці розширення **(перевірка зняття і перевірка відповідності**).

У точки розширення можлива вставка поведінки з розширювального елементу Use Case. Вставка відбувається, якщо виконується умова розширення:

- для розширювального елементу Use Case **Стан** це умова – **запит стану**;
- для розширювального елементу Use Case **Зняти** це умова **– запит зняття**;
- для розширювального елементу Use Case **Захоплення карти** це умова **– список підозр**.

Для розширювального (базового) елементу Use Case ці умови є зовнішніми, тобто формованими поза ним. Іншими словами, елементу Use Case **Сеанс Банкомату** нічого невідомо про умови **запит стану** і **запит зняття,** а елементам Use Case **Зняти** і **Перевірити достовірність –** про умову **список**

підозр. Умови розширення є наслідками подій, що відбуваються в зовнішньому середовищі.

Стрілки розширення в діаграмі підписані. Окрім стереотипу тут вказані:

- у круглих дужках – імена точок розширення;
- у квадратних дужках – умова розширення.

Опис розширювального елементу Use Case розділений на сегменти, кожен сегмент обслуговує одну точку розширення базового елементу Use Case.

Кількість сегментів розширювального елементу Use Case дорівнює кількості точок розширення базового елементу Use Case. Перший сегмент розширювального елементу Use Case розпочинається з умови розширення, умова записується тільки один раз, його дія поширюється і на усі інші сегменти.

Поведінка базового елементу Use Case задається внутрішнім потоком подій аж до точки розширення. У точці розширення можливе виконання розширяльного елементу Use Case, після чого поновлюється робота внутрішнього потоку. Специфікації елементів Use Case даної діаграми мають такий вигляд.

Елемент Use Case **Сеанс Банкомату**

include (Ідентифікація клієнта)	//включення
include (Перевірка рахунку)	//включення
(діалог можливий)	//перша точка розширення
надрукувати заголовок квитанції	
(видача квитанції)	//друга точка розширення
кінець сеансу	

Розширювальний елемент Use Case **Стан**

сегмент	//початок першого сегменту
прийняти запит зняття	//умова розширення
визначити суму	
(перевірка зняття)	//точка розширення
сегмент	//початок другого сегменту
надрукувати суму, що знімається	
видати готівку	

Розширювальний элемента елемент Use Case **Зняти**

сегмент	//початок першого сегменту
прийняти запит зняття	//умова розширення
визначити суму	
(перевірка зняття)	//точка розширення
сегмент	//початок другого сегменту
надрукувати суму, що знімається	
видати готівку	

Розширювальний елемент Use Case **Захоплення карти**

сегмент	//початок єдиного сегменту
прийняти список підозр	//умова розширення
проковтнути карту	

кінець сеансу	
сегмент	

Елемент Use Case, що включається, **Ідентифікація клієнта**

отримати ім’я клієнта	
include (Перевірити достовірність)	//включення
отримати номер рахунку клієнта	

Елемент Use, що включається, Case **Перевірка рахунку**

встановити з’єднання з Базою цих рахунків	
отримати багатство і обмеження рахунку	

Елемент Use, що включається, Case **Перевірити достовірність**

встановити з’єднання з Базою цих клієнтів	
отримати параметри клієнта	
(перевірка)	//точка розширення

Опишемо можливу поведінку моделі, що задається цією діаграмою.

Актор **Клієнт** ініціює дії базового елементу Use Case **Сеанс Банкомату.** На першому кроці запускається елемент Use Case, що включається, **Ідентифікація клієнта**. Цей елемент Use Case отримує ім’я клієнта і запускає елемент Use Case **Перевірити достовірність**, внаслідок чого встановлюється з’єднання з базою цих клієнтів і виходять параметри клієнта.

Якщо до цього моменту виконується умова розширення **список підозр,** то «спрацьовує» розширювальний елемент Use Case **Захоплення карти**, карта заарештовується і робота системи припиняється.

Інакше відбувається повернення до елементу Use Case **Ідентифікація клієнта**, який отримує номер рахунку клієнта і повертає управління базовому елементу Use Case.

Базовий елемент Use Case переходить до другого кроку роботи – викликає елемент Use Case, що включається, **Перевірка рахунку**, який встановлює з’єднання з базою цих рахунків і отримує стан і обмеження рахунку.

Управління знову повертається до базового елементу Use Case. Базовий елемент Use Case переходить до першої точки розширення **діалог можливий**. У цій точці можливе підключення одного з двох розширяльних елементів Use Case.

Допустимо, що до цього моменту виконується умова розширення **запит стану,** тому запускається перший сегмент елементу Use Case **Стан**. В результаті відображається інформація про стан рахунку і управління передається базовому елементу Use Case. У базовому елементі Use Case друкується заголовок квитанції і забезпечується перехід до другої точки розширення **видача квитанції.**

Оскільки в активному стані продовжує знаходитися розширювальний елемент Use Case **Стан**, запускається його другий сегмент – в квитанції друкується інформація про стан рахунку.

Востаннє управління повертається до базового елементу Use Case – завершується сеанс роботи банкомату.

За правилами мови UML повну діаграму Use Case можна поміщати в рамку з п'ятикутником в лівому верхньому кутку. Ім'я діаграми записується в п'ятикутнику услід за позначкою use case. Рамка вважається додатковою презентаційною можливістю, тобто не є обов'язковим елементом діаграми.

Помітимо, що по суті кожен елемент Use Case задає деякий набір вимог і відповідає деякому набору понять. Надалі кожен елемент Use Case буде реалізований певним набором класів. Якщо деякий елемент Use Case представлений як такий, що розширює, то це означає, що набір понять, що задається ним, перетинає набір понять іншого, базового елементу Use Case.

10.3.6. Побудова моделі вимог

Якщо подивитися на модель використання з найзагальнішої точки зору, то неважко помітити, що в моделі є присутніми:

- внутрішня система, що моделюється (на рис. 10.18 – це Машина утилізації), у формі набору варіантів використання, можливо пов'язаних залежностями і узагальненнями;
- зовнішнє оточення, у формі набору дійових осіб, можливо пов'язаних узагальненнями;
- зв'язок між системою, що моделюється, і зовнішнім оточенням у формі асоціацій між дійовими особами і варіантами використання.

Зазвичай абсолютно ясно, що знаходиться усередині системи, що моделюється, а що зовні. Якщо це з якоїсь причини неясно, або ж вимагається збільшити наочність діаграм, то можна скористатися спеціальною конструкцією, яка називається «Межі системи».

Межі системи – це графічний коментар у формі прямокутної рамки, що вживається на діаграмах використання і відділяє внутрішню частину системи від її зовнішнього оточення (рис. 10.13).

Внутрішня частина, що виділяється межами, має в UML конкретну назву – суб'єкт («Машина утилізації»).

Суб'єкт **(subject) –** це класифікатор, який реалізує поведінку, що декларується варіантами використання.

Якщо межі системи використовуються на діаграмі, то можна вказати ім'я (і стереотип), які будуть належати до суб'єкта.

Варіанти використання описують внутрішність системи, а дійові особи – її оточення. Так от, системою може виступати будь-яка сутність, для якої можна визначити функціональні або нефункціональні вимоги. Це може бути і підсистема головної системи, окремий компонент і просто клас.

Якщо ми розглядаємо модель використання деякої підсистеми, то інші підсистеми (що взаємодіють з тією, що розглядається) будуть дійовими особами для даної підсистеми. Якщо ми розглядаємо модель використання деякого класу, то інші класи (що взаємодіють з тим, що розглядається) будуть дійовими особами для даного класу.

Дійовою особою може виступати інша система. На рис. 10.13 в якості такої системи може виступати, наприклад, майстерня з ремонту машини утилізації або, як у нашому випадку, оператор з налаштування і ремонту.

Нагадаємо, що основне призначення діаграм Use Case – формування вимог замовника до майбутнього програмного додатку. Наведемо приклад розробки ПЗ для машини утилізації, яка приймає використані пляшки, банки, ящики. Для визначення елементів Use Case, які повинні виконуватися в системі, спочатку визначають акторів.

Вибір акторів. Пошук акторів – велика робота. Спочатку виділяють первинних акторів (споживачів, користувачів), що використовують систему за прямомим призначенням. Кожен з первинних акторів бере участь у виконанні одного або декількох головних завдань системи. У нашому прикладі первинним актором є **Споживач**. Споживач кладе в машину пляшки та отримує квитанцію від машини.

Окрім первинних, існують і вторинні актори. Вони спостерігають і обслуговують систему. Вторинні актори існують тільки для того, щоб первинні актори могли використати систему. У нашому прикладі вторинним актором є **Оператор**. Оператор обслуговує машину і отримує денні звіти про її роботу. Ми не потребуватимемо оператора, якщо не буде споживачів. Таким чином, зовнішнє середовище машини утилізації має вигляд, представлений на рис. 10.18.

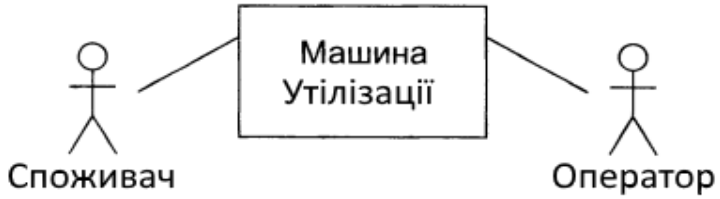


Рисунок 10. 18 – Зовнішнє середовище машини утилізації

Ділення акторів на первинних і вторинних полегшує вибір системної архітектури в термінах основного функціонального призначення. Системну структуру визначають в основному первинні актори. Саме від них в систему приходять головні зміни. Тому повне виділення первинних акторів гарантує, що архітектура системи буде налаштована на більшість важливих користувачів.

Визначення елементів Use Case. Після вибору зовнішнього середовища можна виявити внутрішні функціональні можливості системи. Для цього визначаються елементи Use Case.

Кожен елемент Use Case задає деякий шлях використання системи, виконання деякої частини функціональних можливостей. Повна сукупність елементів Use Case визначає усі існуючі шляхи використання системи.

Елемент Use Case – це послідовність взаємодій в діалозі, що виконується актором і системою. Запускається елемент Use Case актором, тому зручно виявляти елементи Use Case за допомогою акторів.

Розглядаючи кожного актора, ми вирішуємо, які елементи Use Case він може виконувати. Для цього вивчається опис системи (з точки зору актора) або проводиться обговорення з тими, хто діятиме як актор.

Перейдемо до прикладу. **Споживач –** первинний актор, тому розпочнемо з цієї ролі. Цей актор повинен виконувати повернення утилізованих елементів. Так формується елемент Use Case **Повернення Елементу**. Наведемо його текстовий опис:

Use Case починається, коли споживач починає повертати банки, пляшки, ящики. Для кожного елементу, поміщеного в машину утилізації, система збільшує кількість елементів, прийнятих від ***Споживача****, і загальну кількість елементів цього типу за день. Після здачі усіх елементів* ***Споживач*** *натискає кнопку видачі квитанції, щоб отримати квитанцію, на якій надруковані назви повернених елементів і загальна сума повернення.*

Наступний актор – **Оператор.** Він отримує денний звіт про елементи, здані за день. Це утворює елемент Use Case **Створення Денного Звіту.** Його опис:

Починається оператором, коли він хоче отримати інформацію про елементи, здані за день. Система друкує кількість елементів кожного типу і загальну кількість елементів, отриманих за день.

Для підготовки до створення нового денного звіту скидається в нуль параметр ***Загальна кількість****.*

Крім того, актор **Оператор** може змінювати параметри елементів, що здаються. Назвемо відповідний елемент Use Case **Зміна Елементу**. Його опис:

Можуть змінюватися ціна і розмір кожного повертаного елементу. Можуть додаватися нові типи елементів.

Після виявлення усіх елементів діаграма Use Case системи приймає виглядд, показаний на рис. 10.19.

Рисунок 10.19 – Діаграма Use Case для машини утилізації

Найчастіше повні описи елементів Use Case формуються за декілька ітерацій. На кожному кроці в опис вводяться додаткові деталі. Наприклад, остаточний опис **Повернення Елемента** може мати вигляд:

Коли споживач повертає елемент, що здається, елемент вимірюється системою. Виміри дозволяють визначити тип елементу. Якщо тип

допустимий, то збільшується кількість елементів цього типу, прийнятих від ***Споживача****, і загальна кількість елементів цього типу за день.*

Якщо тип недопустимий, то на панелі машини висвічується «недійсний».

Коли ***Споживач*** *натискає кнопку квитанції, принтер друкує дату. Робляться обчислення. За кожним типом прийнятих елементів друкується інформація: назва, прийнята кількість, ціна, разом для типу. У кінці друкується сума, яку повинен отримати споживач.*

Не завжди очевидно, як розподілити функціональні можливості по окремих елементах Use Case, і що є варіантом одного і того ж елементу Use Case. Основний критерій вибору – складність елементу Use Case. При аналізі варіантів поведінки розглядають їх відмінності. Якщо відмінності малі, варіанти вбудовують в один елемент Use Case. Якщо відмінності великі, то варіанти описуються як окремі елементи Use Case.

Зазвичай елемент Use Case задає одну основну і декілька альтернативних послідовностей подій.

Кожен елемент Use Case виділяє приватний аспект функціональних можливостей системи. Тому елементи Use Case забезпечують інкрементну схему аналізу функцій системи. Можна незалежно розробляти елементи Use Case для різних функціональних областей, а пізніше об'єднати їх разом (для формування повної моделі вимог). На основі елементів Use Case в кожен момент часу можна концентрувати увагу на одній частковій проблемі, що дозволяє вести паралельну розробку.

Розширення функціональних можливостей. Для додавання в елемент Use Case нових дій зручно застосовувати відношення розширення. За його допомогою базовий елемент Use Case може бути розширений новим елементом Use Case. У нашому прикладі поведінка системи не визначена для випадку, коли елемент, що здається, застряг. Введемо елемент Use Case **Елемент Застряг,** який розширюватиме базовий елемент Use Case **Повернення Елемента** (рис. 10.20).

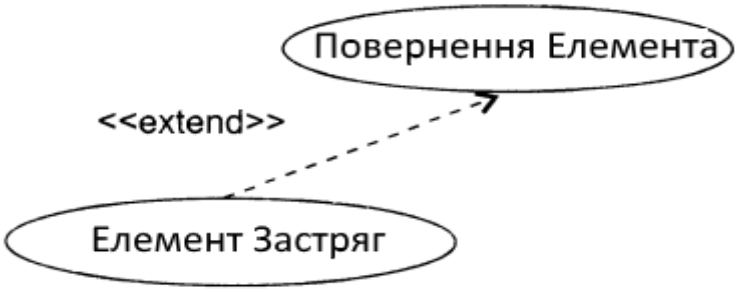


Рисунок 10.20 – Розширення елементу Use Case **Повернення Елемента**

Таким чином, опис базового елементу залишається тим самим, простим. Ще один приклад наведений на рис. 10.21.

Тут ми бачимо тільки один базовий елемент Use Case **Сеанс роботи**. Усі інші елементи Use Case можуть додаватися як розширення. Базовий елемент Use Case при цьому залишається майже без змін.

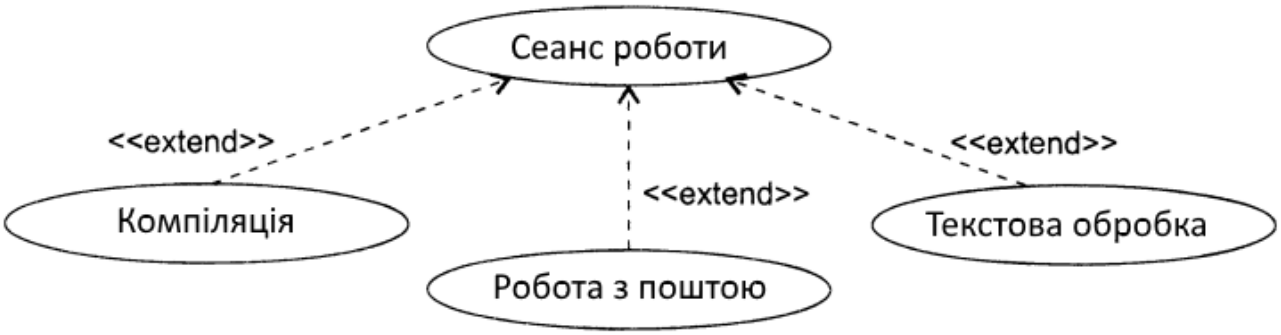


Рисунок 10.21 – Застосування відношення розширення

Відношення розширення визначає переривання базового елементу Use Case, яке відбувається для вставки іншого елементу Use Case. Базовий елемент Use Case не знає, чи буде виконуватись переривання, чи ні. Обчислення умов переривання знаходиться поза компетенцією базового елементу Use Case.

У розширювальному елементі Use Case вказується посилання на те місце базового елементу Use Case, куди він вставлятиметься (при перериванні). Після виконання розширювального елементу Use Case триває виконання базового елементу Use Case.

Зазвичай розширення використовують:

- для моделювання варіантних частин елементів Use Case;
- для моделювання складних і рідко виконуваних альтернативних послідовностей;
- для моделювання підпорядкованих послідовностей, які виконуються тільки в певних випадках;
- для моделювання систем з вибором на основі меню.

Головне, що слід пам’ятати: рішення про вибір, підключення варіанту на основі розширення приймається поза базовим елементом Use Case. Якщо ж ви вводите у базовий елемент Use Case умовну конструкцію, конструкцію вибору, то доведеться застосовувати відношення включення.

Уточнення моделі вимог. Уточнення моделі зводиться до виявлення однакових частин в елементах Use Case і витягання цих частин. Будь-які зміни в такій частині, виділеній в окремий елемент Use Case, автоматично впливатимуть на усі елементи Use Case, які використовують її спільно.

Витягнуті елементи Use Case називають ***абстрактними***. Вони не можуть бути конкретизовані самі по собі, застосовуються для опису однакових частин в інших, конкретних елементах Use Case. Таким чином, описи абстрактних елементів Use Case використовуються в описах конкретних елементів Use Case. Говорять, що конкретний елемент Use Case знаходиться у відношенні «включає» з абстрактним елементом Use Case.

Повернемося до нашого прикладу. В даному прикладі два конкретні елементи Use Case **Повернення Елемента і Створення Денного Звіту** мають загальну частину – дії, що забезпечують друк квитанції. Тому, як показано на рис. 10.22, можна виділити абстрактний елемент Use Case Друк. Цей елемент Use Case спеціалізуватиметься на виконанні роздруківок.

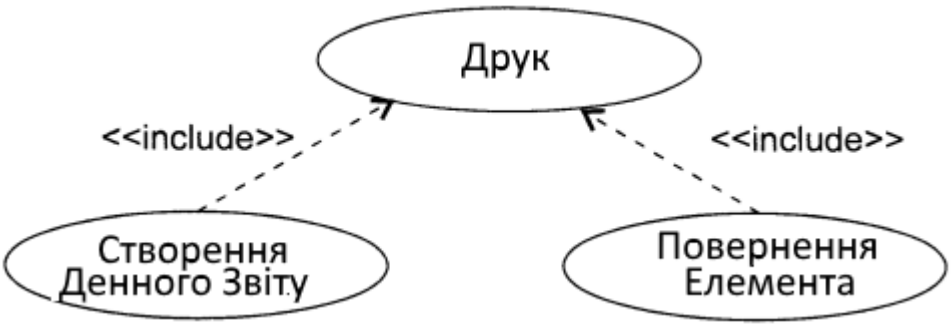


Рисунок 10. 22 – Застосування відношення включення

У свою чергу, абстрактні елементи Use Case можуть використовуватися іншими абстрактними елементами Use Case. Так утворюється ієрархія. При побудові ієрархії абстрактних елементів Use Case керуються правилом: виділення елементів Use Case припиняється досягнувши рівня окремих операцій над об'єктами.

Виділення абстрактних елементів Use Case можна спростити за допомогою абстрактних акторів.

Абстрактний актор – це загальний фрагмент ролі в декількох конкретних акторах. Абстрактний актор виражає подібності в елементах Use Case. Конкретні актори знаходяться ву відношенні спадкоємства з абстрактним актором. Так, в машині утилізації конкретні актори мають одну загальну поведінку: вони можуть отримувати квитанцію. Тому можна визначити одного абстрактного актора – **Одержувача Квитанції.** Як показано на рис. 10.23, спадкоємцями цього актора є **Споживач** і **Оператор**.

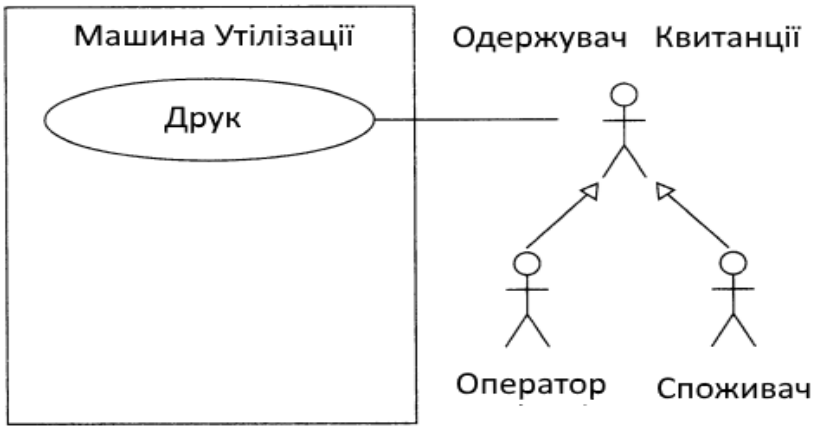


Рисунок 10.23 – Виділення абстрактного актора

10.3.7. Реалізація варіантів використання

Після того як побудовано представлення використання (результат моделювання використання), тобто, виділені дійові особи, варіанти використання і встановлені відношення між ними, встає природне питання: що далі? Тобто, як далі слід продовжувати моделювання засобами UML?

Представлення використання, якщо воно ретельно продумане і детально промальоване, є формою технічного завдання (***специфікацією функціональних вимог***), яка містить достатньо інформації для подальшого проектування і

реалізації будь-яким іншим методом. Може трапитися, що в колективах розробників, що досконало володіють якою-небудь іншою методикою проектування і реалізації, таке обмежене використання UML виявиться цілком виправданим.

Проте UML містить засоби не лише для моделювання використання, але і для підтримки усіх інших фаз процесу розробки.

Таким чином, UML корисно знати, навіть якщо вона не використовується в процесі розробки або використовується тільки частково. Тому ми розглянемо інші засоби UML, тобто можливі шляхи переходу від моделювання використання до інших видів моделювання [16; 23; 24; 29].

Дійові особи знаходяться поза системою – з ними нічого робити не треба. Можна сказати, що дійові особи вже виконали своє завдання, просто з'явившись в моделі системи. Таким чином, перехід від моделювання використання до інших видів моделювання полягає в **уточненні, деталізації** і **конкретизації варіантів використання**.

В уявленні використання ми показали, **що робить система,** тепер слід визначити, **як це робиться**. Назвемо це реалізацією варіантів використання.

Реалізація варіанту використання (use case realization) – цей опис усіх або деяких сценаріїв, що становлять варіант використання.

Повторимо ще раз: варіант використання – це опис множини послідовностей подій або дій (сценаріїв), що доставляють значимий для дійової особи результат. Існує багато способів опису алгоритмів, більш менш формальних. Ми розглянемо тільки ті, які застосовуються саме при реалізації варіантів використання.

10.3.8. Рекомендації до розробки діаграм варіантів використання

Як було відмічено раніше, одне з головних призначень діаграми варіантів використання полягає у формалізації функціональних вимог до системи. Діаграма варіантів використання може служити основою для узгодження із замовником функціональних вимог до системи на ранній стадії проектування. Будь-який з базових варіантів використання в подальшому може бути підданий декомпозиції на часткові варіанти використання. При цьому рекомендується, щоб загальна кількість акторів в моделі не перевищувала 20, а варіантів використання – 50. Інакше модель втрачає свою наочність і, можливо, замінює собою одну з деяких інших діаграм.

Для розробки діаграми варіантів використання рекомендується дотримуватися деякої послідовності дій [38]:

- визначити головних, або первинних, і другорядних акторів;
- визначити цілі головних акторів стосовно системи;
- сформулювати основні варіанти використання, які специфікують функціональні вимоги до системи;
- упорядкувати варіанти використання по мірі убування ризику їх реалізації;

- виділити учасників, інтереси, передумови і постумови виконання вибраного варіанту використання;
- написати успішний сценарій реалізації вибраного варіанту використання;
- визначити виключення або неуспіх у виконанні сценарію варіанту використання;
- написати сценарії для усіх виключень;
- виділити загальні варіанти використання і зобразити їх взаємозв'язки з базовими із стереотипом uses (include);
- виділити варіанти використання для виключень і зобразити їх взаємозв'язки з базовими із стереотипом extend;
- перевірити діаграму на відсутність дублювання варіантів використання і акторів.

Варіанти використання можуть бути додатково специфіковані примітками з текстом, які в подальшому можуть стати прототипами операцій і методів спільно з атрибутами.

10.4. Аналіз вимог за допомогою діаграм діяльності

Опис функціональних вимог у вигляді варіантів використання, поза сумнівом, є дуже важливим етапом в життєвому циклі розробки програмного забезпечення. Адже саме на цьому етапі відбувається узгодження із замовником того, як виглядатиме і функціонуватиме система.

Але ось узгодження позаду і тепер на підставі затверджених вимог необхідно систему спроектувати і реалізувати. Завдання аналітика на цьому етапі – перетворити опис варіантів використання в технічно грамотний опис структури і поведінки системи, зрозумілий архітекторам і розробникам. Хорошою практикою для такого опису є візуальне моделювання.

UML дозволяє представити усі аспекти системи, що проектується, з будь-якою необхідною точністю. При цьому для кожного варіанту використання може бути побудовано декілька діаграм різних видів – кожен вид діаграми описує свій аспект. Уся сукупність діаграм, що розкривають «внутрішню кухню» варіантів використання, складає так звану *концептуальну або аналітичну модель системи*. Саме на основі цієї моделі надалі проектується архітектура системи, що розробляється, реалізуються її функції і створюються структури даних.

Головна відмінність моделювання від інших методів вивчення складних систем – можливість оптимізації системи до її реалізації. Процес моделювання складається з трьох стадій: формалізації (перехід від реального об'єкта до моделі), моделювання (аналіз і оптимізація моделі), інтерпретації (переклад результатів моделювання в область реальності).

Аналітичне моделювання полягає в побудові моделі, заснованої на описі поведінки об'єкту або системи об'єктів.

Після того, як побудовано представлення використання (результат моделювання використання), тобто виділені дійові особи, варіанти

використання і встановлені відношення між ними, постає природне питання: що далі? Тобто, як далі слід продовжувати моделювання засобами UML?

Взагалі кажучи, відповідь може бути такою: нічого більше не робити засобами UML. Представлення використання, якщо воно ретельно продумане і детально промальоване, є ***формою технічного завдання (специфікацією функціональних вимог***), що містить достатньо інформації для подальшого проектування і реалізації будь-яким іншим методом. Може трапитися, що в колективах розробників, що досконало володіють якою-небудь іншою методикою проектування і реалізації, таке обмежене використання UML виявиться цілком виправданим.

Проте UML містить засоби не лише для моделювання використання, але і для підтримки усіх інших фаз процесу розробки, і ці засоби, щонайменше, не гірші альтернативних.

Таким чином, UML корисно знати, навіть якщо вона не використовується в процесі розробки або використовується тільки частково. Тому ми розглянемо інші засоби UML нижче. Мета цього розділу – викласти наше бачення можливих шляхів переходу від моделювання використання до інших видів моделювання.

Дійові особи знаходяться поза системою – з ними нічого робити не треба. Можна сказати, що дійові особи вже виконали своє завдання, просто з'явившись в моделі системи. Таким чином, перехід від моделювання використання до інших видів моделювання полягає в **уточненні, деталізації і конкретизації варіантів використання**.

У представленні використання ми показали, **що робить система**, тепер треба визначити, **як вона це робить**. Назвемо це поведінкою або реалізацією варіантів використання.

Реалізація варіанту використання (use case realization) – **це опис усіх або деяких сценаріїв, що становлять варіант використання.**

Нагадаємо: варіант використання – це опис множини послідовностей подій або дій (сценаріїв), що дають значимий для дійової особи результат. Найчастіше метод опису множини послідовностей дій, що використовується, полягає в указанніі алгоритму, виконання якого представляє послідовність дій з необхідної множини. Існує багато способів опису алгоритмів, більш менш формальних. Ми розглянемо чотири, що часто вживаються саме при реалізації варіантів використання.

1.Текстові описи. Історично найзаслуженіший і досі один з найпопулярніших способів: скласти текстовий опис типового сценарію варіанту використання.

Розглянемо наведений нижче текст як приклад одного з можливих сценаріїв ІС відділу кадрів з розділу 16.

Сценарій варіанту використання Звільнення за власним бажанням:

1. Співробітник пише заяву.
2. Начальник підписує заяву.
3. **Якщо** є невикористана відпустка, **то** бухгалтерія розраховує компенсацію.

4. Бухгалтерія розраховує вихідну допомогу.
5. Системний адміністратор видаляє (чи блокує) обліковий запис.
6. Менеджер персоналу оновлює базу даних.

Здавалося б, що тут неясного? А неясно, наприклад, ось що: як повинна поводитися система, якщо на кроці 2 начальник **не** підписує заяву. З тексту сценарію не лише не ясна відповідь, але, гірше за те, при неуважному читанні можна і не помітити, що є питання.

Текстові описи сценаріїв хороші: прості, усім зрозумілі, легко і швидко складаються. Погані вони тим, що можуть бути неповні і неточні, і ці недоліки непомітні.

2. Реалізація програми на псевдокоді. Розробники найчастіше описують алгоритми за допомогою *програм* – текстів на деякій мові.

Якщо програма призначена для виконання комп'ютером, то вона має бути записана на суто формальній мові, яку називають у цьому випадку ***мовою програмування***.

Якщо ж програма призначена виключно для читання і, можливо, виконана людиною, то можна застосувати менш формальну (і зручнішу) мову, яку в цьому випадку називають ***псевдокодом***.

Зазвичай у псевдокод включають суміш загальновідомих ключових слів мов програмування і неформальні вирази на природній мові, що означають виконувані дії. Ці вирази мають бути зрозумілі людині, яка пише (чи читає) програму на псевдокоді, але зовсім не зобов'язані бути допустимими виразами мови програмування. Текст на псевдокоді схожий на код програми на мові програмування, але таким не є – звідси і назва.

Другий спосіб реалізації варіанту використання – записати алгоритм на псевдокоді, що розглядається нами, хороший тим, що зрозумілий, звичний і доступний будь-якому розробникові. Проте нині навряд чи можна рекомендувати такий спосіб реалізації, як основний, з таких причин.

1. Реалізація на псевдокоді погано узгоджується з сучасною парадигмою об'єктно-орієнтованого програмування.
2. При використанні псевдокоду втрачаються усі переваги використання UML: наочна візуалізація за допомогою картинки, суворість і точність мови проектування і реалізації, підтримка поширеними інструментальними засобами.
3. Рішення на псевдокоді практично неможливо використати повторно.

3. Діаграма діяльності. Третій спосіб реалізації варіанту використання – описати алгоритм за допомогою діаграми діяльності. З одного боку, діаграма діяльності – це повноцінна діаграма UML, з іншого боку, діаграма діяльності незначно відрізняється від блок-схеми (а тим самим і від псевдокоду).

Таким чином, реалізація варіанту використання діаграмою діяльності є компромісним способом ведення розробки. **По суті, це проектування зверху вниз в термінах і позначеннях UML**.

Якщо діаграма варіантів використання дає «вигляд зверху» на функціональність системи, то діаграма діяльності, навпаки, дозволяє детально ілюструвати окремий варіант використання і його сценарії. Залежно від

ступеня деталізації діаграми діяльностей можуть використовуватися на різних етапах розробки. На етапі аналізу вимог і уточнення специфікацій діаграми діяльностей дозволяють конкретизувати основні функції програмного забезпечення, що розробляється .

Під діяльністю в даному випадку розуміють завдання (операцію), яке необхідно виконати вручну або за допомогою засобів автоматизації. Кожному варіанту використання відповідає своя послідовність завдань. У теоретичному плані діаграма діяльності – це узагальнене представлення алгоритму, що реалізовує варіант використання, що аналізується.

4. Діаграми взаємодії. Четвертий з основних способів реалізації варіанту використання – створити одну або декілька діаграм взаємодії у формі ***діаграм комунікації*** або ***діаграм послідовності***, які описують один або декілька сценаріїв цього варіанту використання.

10.4.1. Характеристика діаграм діяльності

Діаграми діяльності (видів діяльності) представляють бажану поведінку системи у виді послідовно і паралельно виконуваних кроків, що сполучаються потоками управління. Кроками є конкретні дії. Іноді тут показують і потоки даних. По суті це один із способів реалізації варіанту використання – описати алгоритм за допомогою діаграми діяльності. Словом, діаграми діяльності дуже схожі на блок-схеми алгоритмів, але принципова різниця між діаграмами діяльності і нотацією блок-схем полягає в тому, що перші підтримують паралельне процеси.

У UML *діаграми видів діяльності* забезпечують систему позначень для послідовності видів діяльності. Ці позначення можна використати в різних цілях (у тому числі для візуалізації кроків комп'ютерного алгоритму), проте вони особливо корисні для побудови діаграм прецедентів і графіків виконання проекту.

Оскільки прецеденти, представлені в текстовому виді, є складовою частиною аналізу процесів, для них теж корисно будувати діаграми видів діяльності, виконання, що наочно представляють складні потоки, з багатьма учасниками і паралельними процесами, що дозволяють заощадити тисячі слів.

Діаграма діяльності піддавалася найбільшим змінам при зміні версій мови UML, тому недивно, що вона були знову змінена і істотно розширена в UML 2. У UML 1 діаграма діяльності розглядалася як особливий випадок діаграм станів. Це викликало немало труднощів у фахівців, що моделюють потоки робіт, для яких добре підходять діаграми діяльності. У UML 2 це обмеження було ліквідоване.

На рис. 10.24 показаний приклад простої **діаграми діяльності.** Стартує вона з початкового вузла, а потім виконуємо операцію **Receive Order** (Прийняти замовлення). Потім йде галуження, яке має один вхідний потік і декілька вихідних паралельних потоків.

З рис. 10.24 видно, що операції **Fill Order** (Заповнити заявку), **Send Invoice** (Послати рахунок) і що йдуть за ними виконуються паралельно. По суті, в даному випадку це означає, що послідовність операцій не має значення. Можна заповнити заявку, послати рахунок, доставити товар (Delivery), а потім отримати оплату (**Receive Payment**); чи можна послати рахунок, отримати оплату, заповнити заявку, а потім доставити товар.

Можна також виконувати операції по черзі. Узяти із складу першу позицію замовлення, друкувати рахунок, узяти другу позицію замовлення, покласти рахунок в конверт і т. д. Чи можна щось робити одночасно: друкувати рахунок однією рукою, а іншою рукою брати що-небудь із складу. Згідно з діаграмою, будь-яка з цих послідовностей дій допустима.

Діаграма діяльності дозволяє будь-кому, хто виконує цей процес, вибирати порядок дій. Іншими словами, діаграма тільки встановлює правила обов'язкової послідовності дій, яким ви повинні слідувати. Це важливо для моделювання бізнес-процесів, оскільки ці процеси часто виконуються паралельно. Такі діаграми також корисні при розробці паралельних алгоритмів, в яких незалежні потоки можуть виконувати роботу паралельно.

За наявності паралелізму потрібна синхронізація. Ми не закриваємо замовлення, поки він не сплачений і не доставлений. Це показується за допомогою **об'єднання (join)** перед операцією **Close Order** (Закрити замовлення). Потік, що виходить з об'єднання, виконується тільки у тому випадку, коли усі потоки, що входять, досягли об'єднання. Тому ви можете закрити замовлення, тільки коли прийнята оплата і замовлення доставлений.

Умовна поведінка схематично позначається за допомогою рішень і злиття**.** Рішення, яке в UML 1 називалося гілкою, має один потік, що входить, і декілька захищених вихідних потоків. Кожен вихідний потік має захист – умовне вираження, поміщене в квадратні дужки. Кожного разу досягши рішення вибирається тільки один з вихідних потоків, тому захисту мають бути такими, що взаємно виключають. Застосування [else] в якості захисту означає, що потік [else] використовується у тому випадку, коли інші захисту цього рішення набувають неправдивого значення.

На рис. 10.19 рішення розташовується після операції заповнення заявки. Якщо у вас термінове замовлення, то він виконується протягом доби **(Overnight Delivery)***;* інакше робиться звичайна доставка **(Regular Delivery)**.

Злиття має декілька вхідних потоків і один вихідний. Злиття означає завершення умовної поведінки, яку було почато рішенням.

Застосування діаграм діяльності для реалізації варіантів використання не занадто наближає до появи цільового артефакту – програмного коду, проте може привести до глибшого розуміння істоти завдання і навіть відкрити несподівані можливості поліпшення додатку, які було важко угледіти в первинній постановці завдання.

Основною вершиною в діаграмі діяльності є ***вузол дії***, який зображається як прямокутник із закругленими кутами.

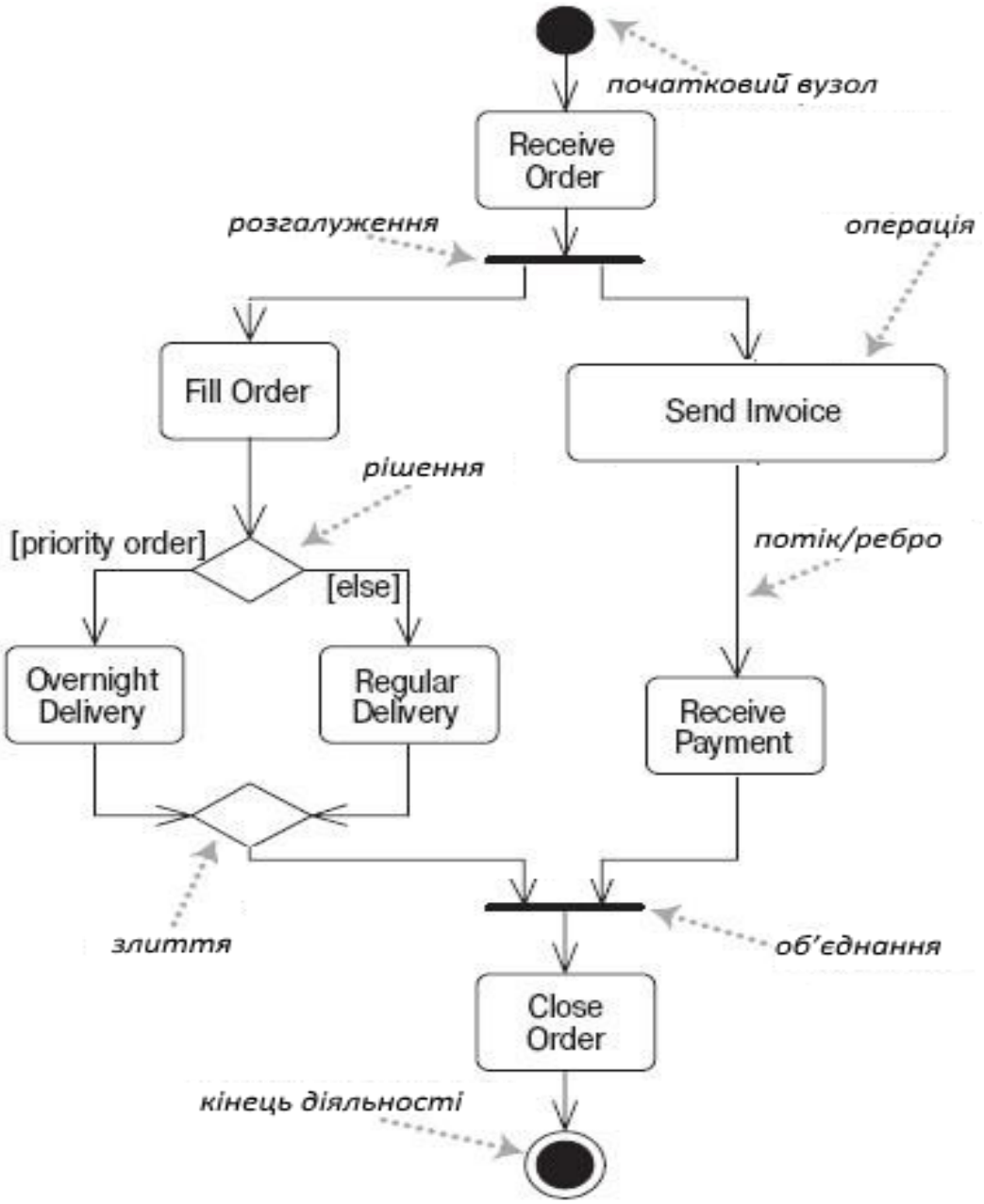


Рисунок 10.24 – Проста діаграма діяльності

Дія вважається атомарною (дію не можна перервати) і виконується за один квант часу, його не можна піддати декомпозиції. Якщо треба представити складну дію, яку можна піддати подальшій декомпозиції (розбити на ряд простіших дій), то використовують виклик іншої діяльності. Зображення виклику діяльності містить піктограму тризубця («граблі») в правому нижньому кутку (рис. 10.25).

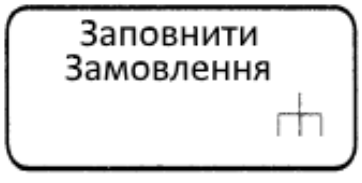


Рисунок 10.25 – Вузол виклику діяльності

Фактично в цю вершину вписується ім’я іншої діаграми діяльності, що має внутрішню структуру. Переходи між вершинами – вузлами дій – зображаються у вигляді стрілок. Переходи виконуються після закінчення дій.

Крім того, в діаграмах діяльності використовуються вузли управління і вузли об’єктів:

- *розгалуження* (ромб з одним входом і декілька витікаючих стрілок);
- *об’єднання* (жирна горизонтальна лінія з декількома входями і однією витікаючою стрілкою);
- *лінійка синхронізації – злиття* (ромб, який має декілька вхідних потоків і один вихідний);
- початковий вузол діяльності (чорний круг);
- *кінцевий вузол діяльності* (незафарбований круг, в якому розміщений чорний круг меншого розміру);
- *кінцевий вузол потоку* (незакрашений круг, в якому розміщений чорний хрест Х);
- *об’єкт* (звичайний прямокутник, ім’я якого підкреслюється).

Вершина «*розгалуження»* дозволяє відобразити розгалуження обчислювального процесу, стрілки, що виходять з нього, позначаються сторожовими умовами галуження. Вершина *«об’єднання»* відмічає точку об’єднання альтернативних потоків дій. *Лінійки синхронізації* дозволяють показати паралельні потоки дій, відмічаючи точки їх синхронізації при запуску (момент розділення) і при завершенні (момент злиття).

10.4.2. Декомпозиція операції в діаграмі діяльності

Операції можуть бути розбиті на вкладені діяльності. Можна узяти алгоритм доставки, свідчень на мал. 10.24, і визначити його як самостійну діяльність (рис. 10.26), а потім викликати його як операцію (рис. 10.27).

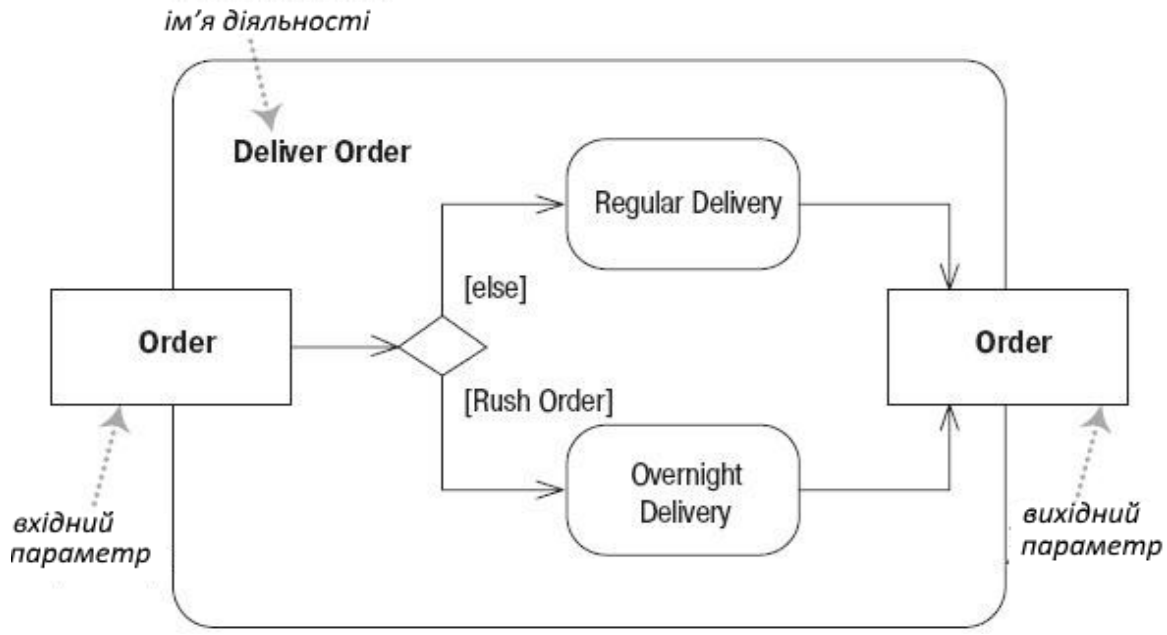


Рисунок 10.26 – Додаткова діаграма діяльності

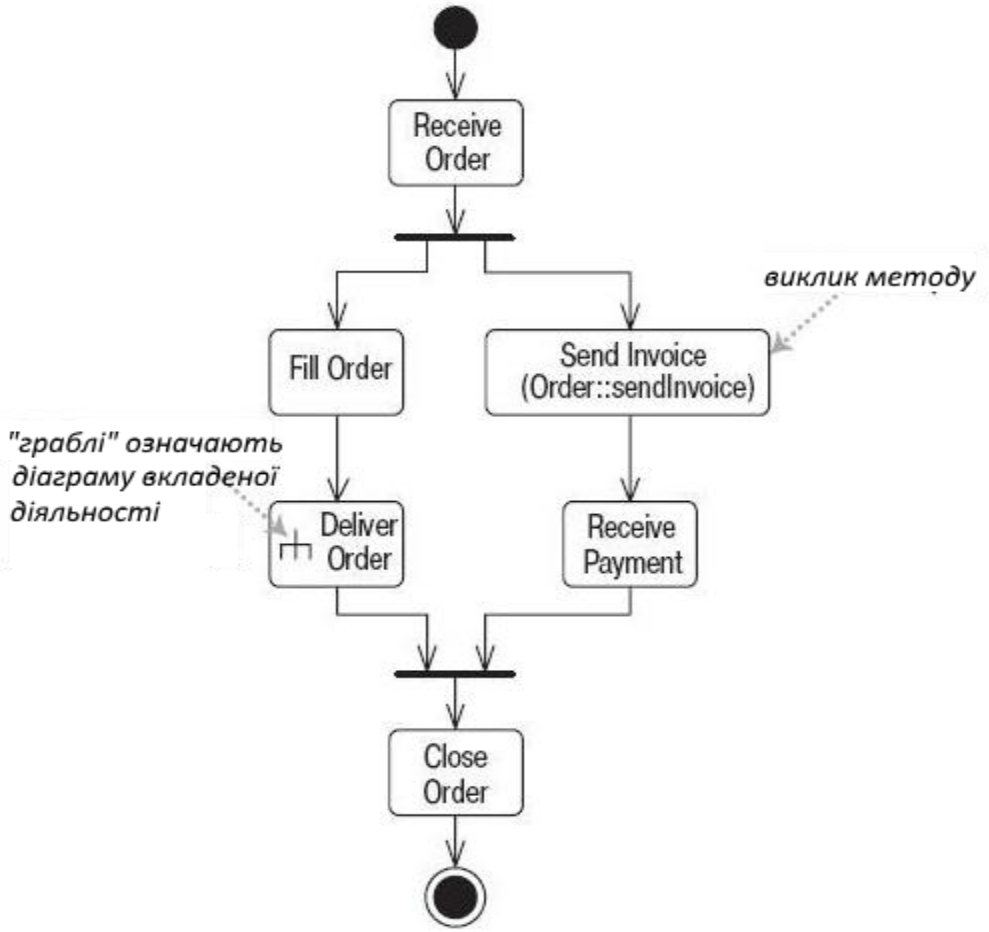


Рисунок 10.27 – Діяльність з рис. 10.24 модифікована для виклику діяльності з рис. 10.26

Операції можуть бути реалізовані або як вкладені діяльності або як методи класів. Вкладену діяльність можна позначити за допомогою символу «грабель». Виклик методу відображається за допомогою синтаксису **ім'я-класу:: ім'я-методу**. Можна також вставити в символ операції фрагмент коду, якщо поведінка представлена не єдиним викликом методу.

10.4.3. Розділи діаграми діяльності

Діаграми діяльності розповідають про те, що відбувається, але нічого не говорять про того, хто які дії виконує. У програмуванні це означає, що діаграма не відображає, який клас є відповідальним за ту або іншу операцію. У моделюванні бізнес процесів це означає, що не відображений розподіл обов'язків між підрозділами фірми. Це не завжди є труднощами. Часто має сенс сконцентруватися на тому, що відбувається, а не на тому, хто яку роль грає в цьому сценарії.

Можна розбити діаграму діяльності на розділи, щоб показати, хто що робить, тобто які операції виконує той або інший клас або підрозділ підприємства. На рис. 10.28 наведений простий приклад, що показує, як операції по обробці замовлення можуть бути розподілені між різними підрозділами.

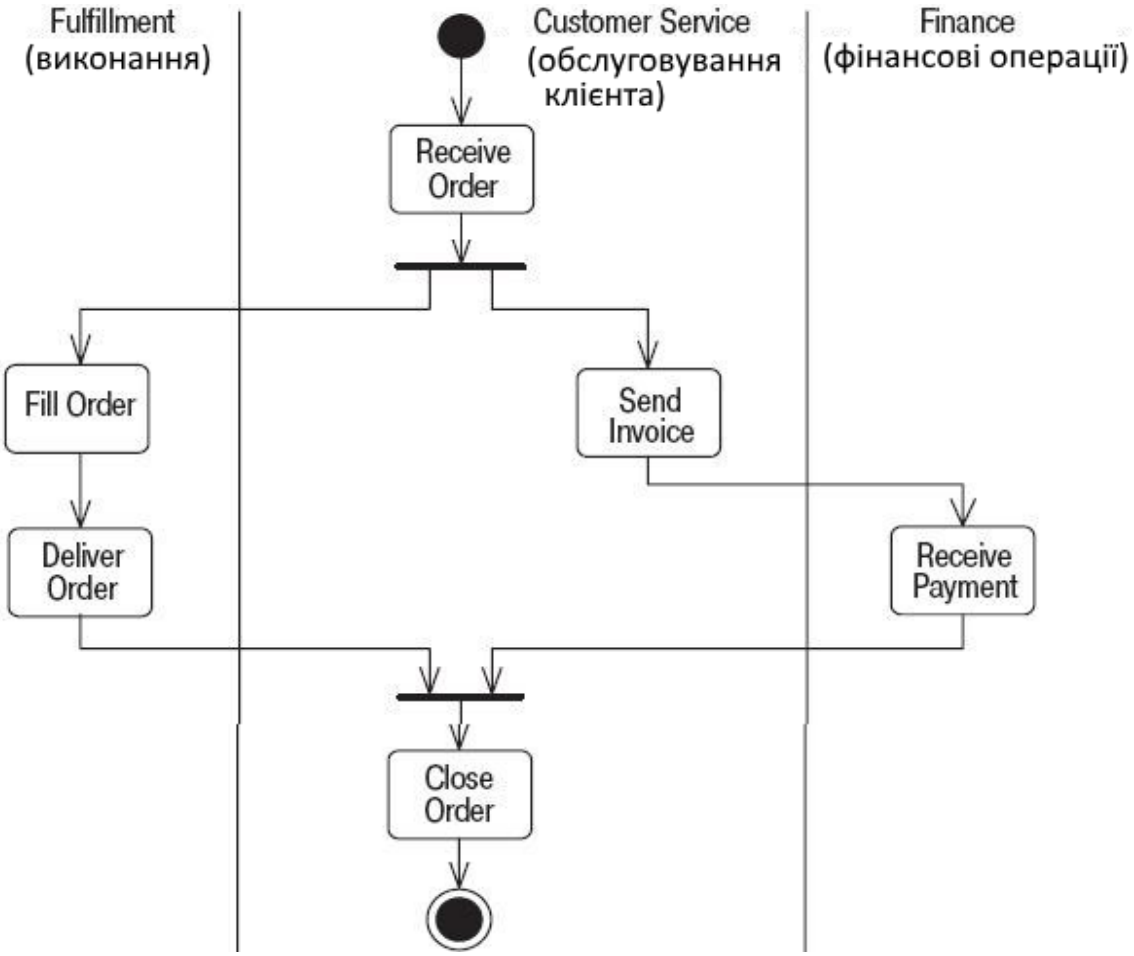


Рисунок 10.28 – Розбиття діаграми діяльності на розділи

На рис. 10.28 представлено просте одновимірне розбиття. Цей спосіб часто називають ***плавальними доріжками***, і така форма була єдиною в UML 1. У UML 2 сітка може бути двовимірною. Уся сукупність дій розподілена по трьох **розділах**, кожен з яких має своє ім’я і відповідає окремій зацікавленій особі, що показує, хто що робить. Розділи відокремлені один від одного вертикальними лініями.

10.4.4. Рекомендації до побудови діаграм діяльності

Діаграми діяльності відіграють важливу роль в розумінні процесів реалізації алгоритмів виконання операцій класів і потоків керування в модельованій системі. Використовувані для цієї мети традиційні блок-схеми алгоритмів мають серйозні обмеження у відображенні паралельних процесів і їх синхронізації. Застосування доріжок і об'єктів відкриває додаткові можливості для наочного зображення бізнес-процесів.

Діаграма діяльності багато в чому нагадує діаграму станів, хоча й не тотожня їй. Тому багато рекомендацій з побудови діаграм станів виявляються справедливими для діаграми діяльності. Зокрема, ця діаграма будується для окремого класу, варіанту використання, окремої операції класу або підсистеми.

10.5. Аналіз вимог за допомогою діаграм взаємодії

Ще один з основних способів реалізації варіанту використання – створити одну або декілька *діаграм взаємодії* у формі ***діаграм послідовності*** або ***діаграм комунікації,*** які описують один або декілька сценаріїв цього варіанту використання. Вони ілюструють спосіб взаємодії об'єктів за допомогою повідомлень.

Цей спосіб найбільшою мірою відповідає ідеології UML і рекомендується авторами мови як основний і переважний.

Розглянемо основні достоїнства і недоліки реалізації варіанту використання діаграмами взаємодії.

Розпочнемо з позитивного. На діаграмах взаємодії представлена взаємодія об'єктів, т. е. екземплярів деяких класів. Тим самим, побудова такої діаграми з необхідністю призводить до виявлення деяких класів, які повинні існувати в моделі і деяких операцій цих класів (а саме тих, які з'являться у формі повідомлень типу «виклик операції» на діаграмі взаємодії).

Більше того, оскільки повідомлення передаються від об'єкту до об'єкту уздовж зв'язків (в основному, екземплярів асоціацій), то виявляються виявленими і деякі необхідні асоціації. Таким чином, реалізація варіанту використання якою-небудь діаграмою взаємодії забезпечує органічний перехід від моделювання використання до моделювання структури і моделювання поведінки.

При складанні діаграм взаємодії для декількох варіантів використання можуть бути задіяні одні і ті ж класи, що грають різні ролі в різних кооперациях. Однакова поведінка і структура, використання, що проявляються в різних варіантах, виявляються реалізованими одним класом. Такий стиль моделювання повністю відповідає об'єктно-орієнтованому підходу і забезпечує концептуальну цілісність проектованої системи.

При реалізації варіантів використання діаграмами взаємодії на ранніх етапах моделювання з'являються супутні фрагменти діаграм класів. Ці діаграми набагато ближче до цільового артефакту – програмному коду – ніж діаграми використання і навіть діаграми діяльності. Практично усе CASE-инструменты підтримують генерацію коду по діаграмах класів, а деякі – і по діаграмах взаємодії.

Таким чином, з'являється можливість автоматизованої побудови прототипів (версій системи, що забезпечують часткову функціональність), що повністю відповідає ідеології програмування – инкрементальной розробки.

Проте у діаграм взаємодії UML є істотне обмеження. В цілому ці діаграми формулюються на рівні об'єктів, а тому дозволяють реалізувати тільки окремий сценарій (умовно кажучи, екземпляр варіанту використання). Іншими словами, діаграми взаємодії дозволяють описати протокол виконання алгоритму, але не сам алгоритм.

Якщо алгоритм виконання варіанту використання лінійний (просто послідовність дій, без галужень і циклів), то проблеми немає – лінійні алгоритми суть протоколи свого виконання. Для галужень і циклів діаграми

взаємодії містять деякі засоби моделювання. Іноді цих засобів може виявитися досить.

Що ж робити, якщо все-таки ніяк не вдається побудувати вичерпну діаграму взаємодії для варіанту використання? В цьому випадку рекомендується застосовувати наступні методи.

По-перше, реалізувати варіант використання декількома діаграмами взаємодії. Кожна діаграма описує окремий сценарій, а всі разом вони дають достатнє уявлення про реалізацію варіанту використання.

По-друге, можна переглянути представлення використання так, щоб виключити варіанти використання, що важко реалізовуються. Наприклад, за допомогою узагальнення виділити варіанти використання, які описують однорідніше безліч сценаріїв і легше реалізуються діаграмами взаємодії. Зазвичай ітеративно застосовують обидва прийоми, поки не буде досягнутий задовільний результат.

Взаємодії визначають поведінку системи у вигляді комунікацій між його частинами (об'єктами), представляючи систему як співтовариство спільно працюючих об'єктів. Саме тому **взаємодії вважають основним апаратом для аналізу, тобто створення детальних вимог до програмної системи**.

На відміну від узагальнених вимог замовника, в яких реалізація програмної системи не видно, вони прямо спираються на елементи програмної реалізації. Початковими даними для створення взаємодій є діаграми, що відбивають вимоги замовника (***діаграми Use Case і діаграми діяльності***).

У програмній системі об'єкти повинні взаємодіяти один з одним, посилаючи повідомлення. ***Взаємодія*** – ця така поведінка на основі обміну повідомленнями між набором об'єктів, яке забезпечує реалізацію вимог до системи.

10.5.1. Об'єкти і ролі

Учасники взаємодії можуть бути або конкретними, або узагальненими сутностями. Як конкретні сутності вони виражають цілком певні об'єкти.

Наприклад, **першокурсник,** екземпляр класу **Студент,** може описувати конкретна особа (Оля Кулик). З іншого боку, як узагальнена сутність **першокурсник** може представляти будь-який екземпляр класу **Студент**.

При розробці моделей детальних вимог зручніше використати узагальнені об'єкти, оскільки вони підвищують міру універсальності моделей.

Узагальнений об'єкт називається роллю і позначається так:

имяРоли:ИмяКласса.

На відміну від імені ролі ім'я конкретного об'єкту завжди підкреслюється:

имяОбъекта:ИмяКласса.

Приклад 10.1. На рис. 10.29, а показані класи **Інститут** і **Студент,** між якими існує асоціація «многие-ко-многим», на рис. 10.29, *б* – пересилка узагальненого повідомлення між відповідними ролями **а:Інститут** і **першокурсник:Студент,** на рис. 10.29, *в* – пересилка конкретного

повідомлення між конкретними об’єктами **чну:Інститут** і **оляКулик:Студент** (тут ім’я Оля, з маленької букви – такі правила іменування об’єктів).

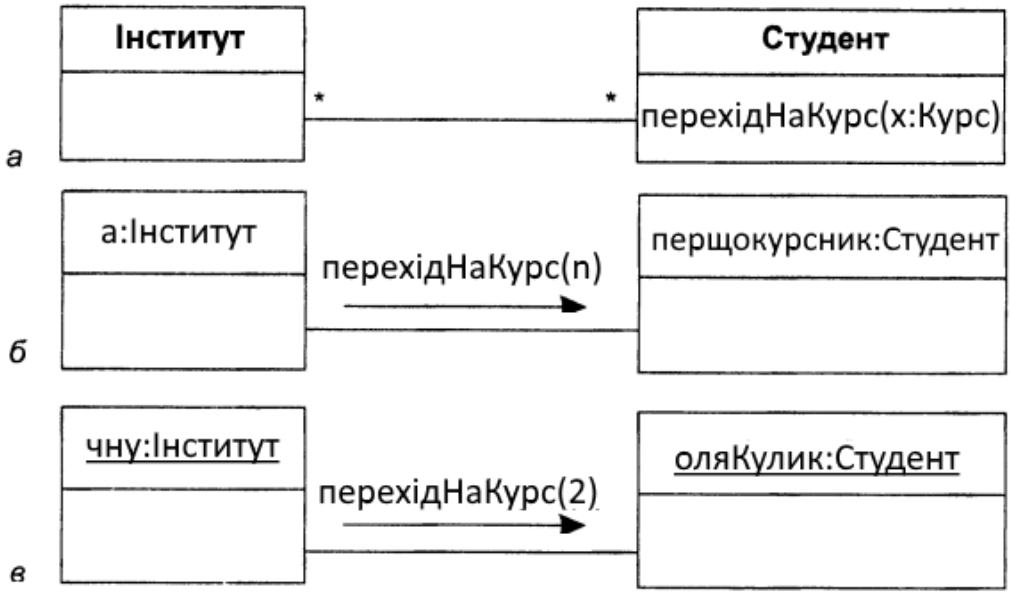


Рисунок 10.29 – Класи, ролі і об’єкти**:**
а – класи; б – ролі; *в* – конкретні об’єкти

10.5.2. Діаграми взаємодії

Частенько на етапі специфікації вимог необхідно показати не лише алгоритм дій або зміну стану об’єкту, але і обмін повідомленнями між окремими об’єктами системи. Цю задачу вирішує діаграма взаємодії.

Діаграми взаємодії призначені для моделювання динамічних можливостей системи. Діаграма взаємодії показує взаємодію, що включає набір учасників (ролей, узагальнених об’єктів) і їх відношень, а також повідомлення, що пересилаються між учасниками. На відміну від діаграми діяльності, яка показує тільки послідовність (алгоритм) роботи системи, діаграми взаємодії акцентують увагу розробників на повідомленнях, що ініціюють виклик певних операцій об’єкту (класу) або є результатом виконання операції.

Існує два різновиди діаграми взаємодії – діаграма послідовності і діаграма комунікації.

Діаграма послідовності – це діаграма взаємодії, яка виділяє впорядкування повідомлень за часом.

Діаграма комунікації – це діаграма взаємодії, яка виділяє структурну організацію узагальнених об’єктів, що посилають і приймають повідомлення.

10.5.3. Діаграми послідовностей

Діаграма послідовності – одна з різновидів діаграм взаємодії. Відбиваючи сценарій поведінки в системі, ця діаграма забезпечує наочніше представлення порядку передачі повідомлень. Правда, вона не дозволяє показати такі деталі, які видно на діаграмі комунікації (структурні характеристики об’єктів і зв’язків).

Мета діаграми послідовності – зображувати об’єкти, що беруть участь у взаємодії, і послідовність повідомлень (вхідні і вихідні події), якими вони обмінюються, часовий аспект поведінки яких може мати істотне значення при моделюванні синхронних процесів [18; 25; 29; 30; 38].

Графічно діаграма послідовності – двовимірна схема, яка показує узагальнені об’єкти (ролі), розміщені уздовж горизонтальної осі, і повідомлення, впорядковані за часом, уздовж вертикальної осі (лінія життя). Прецеденти визначають, як зовнішні виконавці взаємодіють з програмною системою. В процесі цієї взаємодії виконавцем генеруються системні події, які є запитами на виконання деякої системної операції.

Для побудови діаграми послідовності системи необхідно:

- ідентифікувати кожну дійову особу (об’єкт) і зображувати для неї лінію життя;
- з опису варіанту використання визначити безліч системних подій і їх послідовність;
- зображувати системні події у вигляді ліній із стрілкою на кінці між лініями життя дійових осіб і системи, а також вказати імена подій і списки передаваних значень.

На діаграмі послідовності зображаються виключно ті об’єкти, які безпосередньо беруть участь у взаємодії і не показуються можливі статичні асоціації з іншими об’єктами. Для діаграми послідовності ключовим моментом є саме динаміка взаємодії об’єктів в часі. При цьому діаграма послідовності має як би два виміри.

Один вимір – зліва направо у вигляді вертикальних ліній, кожна з яких зображує лінію життя окремого об’єкту, що бере участь у взаємодії. Графічно кожен об’єкт зображається прямокутником і розташовується у верхній частині своєї лінії життя.

Другий вимір діаграми послідовності – вертикальна тимчасова вісь, спрямована зверху вниз. Початковому моменту часу відповідає сама верхня частина діаграми. При цьому взаємодії об’єктів реалізуються за допомогою повідомлень, які посилаються одними об’єктами іншим. Повідомлення зображаються у вигляді горизонтальних стрілок з ім’ям повідомлення і також утворюють порядок за часом свого виникнення. Іншими словами, повідомлення, розташовані на діаграмі послідовності вище, ініціюються раніше тих, які розташовані нижче. При цьому масштаб на осі часу не вказується, оскільки діаграма послідовності моделює лише тимчасову впорядкованість взаємодій типу «раніше-пізніше».

Припустимо, що у нас є замовлення, і ми збираємося викликати команду для визначення його вартості. При цьому об’єкту замовлення (**Order)** необхідно проглянути усі позиції замовлення і визначити їх ціни, засновані на правилах побудови ціни продукції в рядку замовлення **(Order Line**). Виконавши це для усіх позицій замовлення, об’єкт замовлення повинен вичислити загальну знижку, яка визначається індивідуально для кожного клієнта. На рис. 10.24 приведена діаграма, що представляє реалізацію цього сценарію.

Як показано на рис. 10.30, учасники взаємодії (ролі або об'єкти) збожеволіють на вершині діаграми, уздовж горизонтальної осі. Зазвичай ліворуч розміщується учасник, що ініціює взаємодію, а справа – учасники за збільшенням підлеглості.

Повідомлення, що посилаються і приймаються учасниками, показуються уздовж вертикальної осі (лінії життя), в порядку зростання часу від вершини до основи діаграми. Використовується той же синтаксис і позначення синхронізації, що і в діаграмах комунікації.

Одна з переваг діаграми послідовності полягає в тому, що майже не доведеться пояснювати її нотацію. Можна бачити, що екземпляр замовлення посилає рядку замовлення **повідомлення *getQuantity*** і ***getProduct.*** Можна також бачити, як замовлення застосовує метод до самого собі і як цей метод посилає **повідомлення getDiscountInfo** екземпляра клієнта.

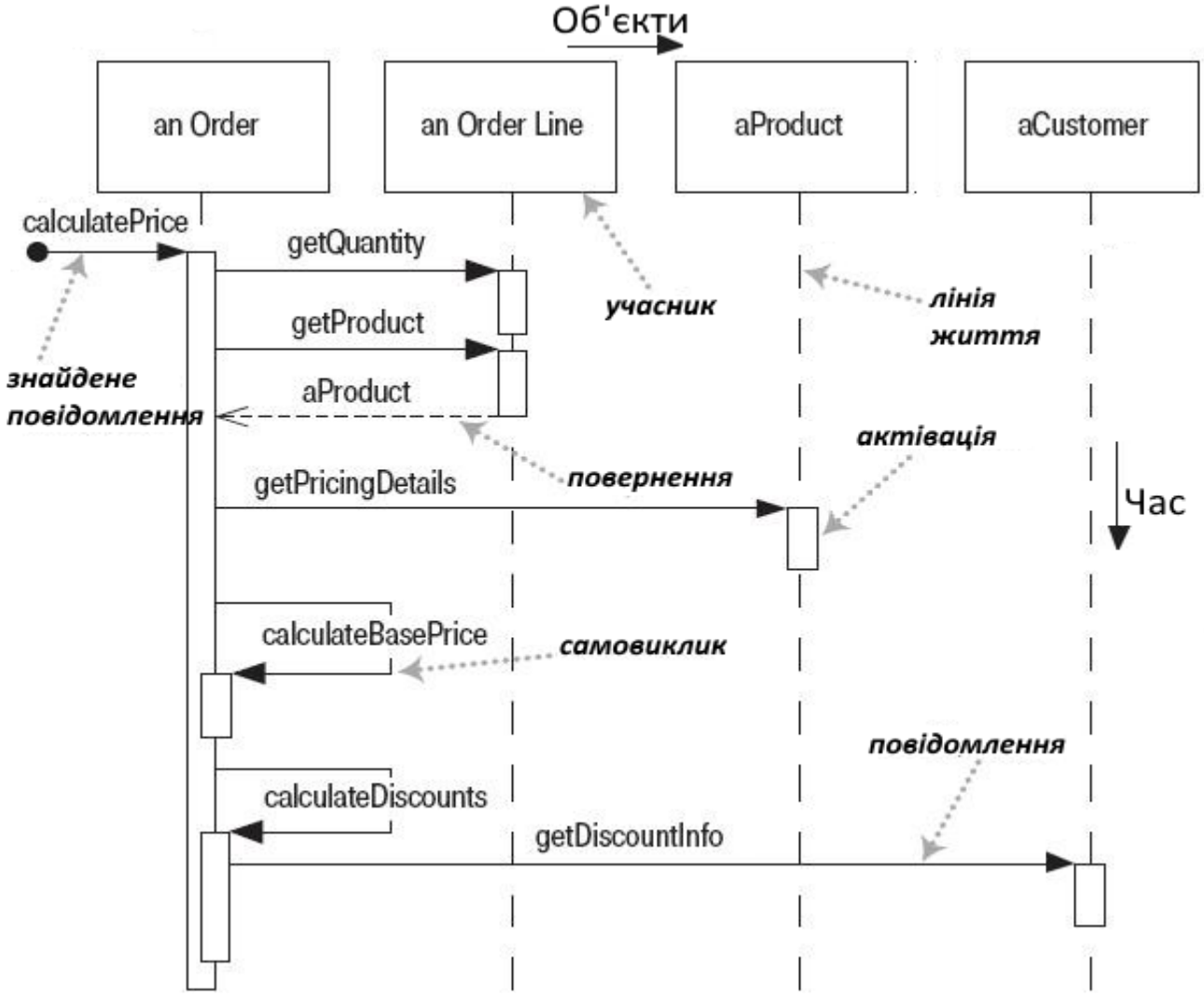


Рисунок 10.30 – Діаграма послідовності оформлення вартості замовлення

Проте діаграма не усе показує так добре. Послідовність повідомлень ***getQuantity***, ***getProduct***, ***getPricingDetails*** *і* ***calculateBasePrice*** має бути реалізована для кожного рядка замовлення, тоді як метод ***calculateDiscounts*** викликається лише одного разу.

Від діаграм комунікації діаграми послідовності відрізняють дві важливі характеристики.

Перша характеристика – *лінія життя* учасника взаємодії.

Лінія життя учасника взаємодії – це вертикальна пунктирна лінія, яка означає період існування учасника взаємодії.

Повідомлення виглядають на діаграмі як горизонтальні стрілки, що йдуть від лінії життя одного учасника (посилача) до лінії життя іншого учасника (одержувача). Хоча учасник може послати повідомлення і самому собі. Якщо повідомленню потрібно деякий час на пересилку, то стрілка повідомлення зображається з нахилом вниз – це означає, що отримання повідомлення відбувається із затримкою. При цьому у обох кінців стрілки можуть розташовуватися мітки з вказівкою часу відправки і отримання повідомлення.

У першого повідомлення (див. рис. 10.30) немає учасника, що послав його, оскільки воно приходить з невідомого джерела. Воно називається ***знайденим повідомленням***.

Більшість узагальнених об'єктів існують упродовж усієї взаємодії, їх лінії життя тягнуться від вершини до основи діаграми. Втім, узагальнені об'єкти можуть створюватися в ході взаємодії. Їх лінії життя починаються з моменту прийому повідомлення <<create>> (new, створити). Крім того, узагальнені об'єкти можуть знищуватися в ході взаємодії. Їх лінії життя закінчуються з моменту прийому повідомлення <<destroy>> (знищити). Як представлено на рис. 10.31, знищення лінії життя відзначається позначкою **X** у кінці лінії.

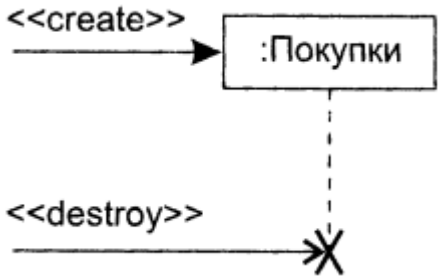


Рисунок 10.31 – Створення і знищення узагальненого об'єкту

Друга характеристика – *специфікація виконання (execution specification) або активація* (activation).

Специфікація виконання (активація, смуга активності) – це високий, тонкий прямокутник, що відображає період часу (інтервал активності учасника), впродовж якого узагальнений об'єкт виконує дію (свою операцію). Вершина прямокутника відмічає початок дії (до неї підходить стрілка повідомлення, що ініціює цю дію), а основу – його завершення. Момент завершення може маркіруватися повідомленням повернення, яке показується пунктирною стрілкою (вона спрямована до лінії життя об'єкту, що викликав). У процедурному потоці управління стрілки повернення можна опустити, оскільки їх наявність мається на увазі, але вказівка їх на діаграмі надає їй велику ясність. Можна показати вкладення активації (наприклад, рекурсивний виклик власної операції). Для цього друга активація малюється трохи правіше за першу (рис. 10.32).

У мові UML смуги активності не обов’язкові.

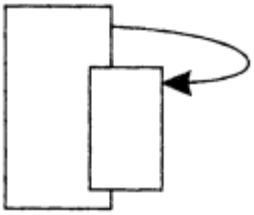


Рисунок 10.32 – Вкладення специфікацій виконання (активацій)

Звичайно, основне призначення діаграми послідовності – зображувати лінійний потік управління, але іноді виникає необхідність у використанні складніших потоків управління. Для цієї мети в діаграму впроваджують комбіновані фрагменти.

Комбінований фрагмент складається з ключового слова, підмножини ліній життя і одного або декількох

субфрагментів, що називаються операндами взаємодії. Кількість і значення субфрагментів залежить від ключового слова. Фрагмент поміщається в рамку. Найбільш популярні наступні комбіновані фрагменти:

1. Використання взаємодії (interaction use) – це посилання на іншу взаємодію, яка описується окремою діаграмою послідовності. Воно позначається ключовим словом **ref**.
2. Цикл (ключове слово **loop**) – має один субфрагмент із сторожовою умовою. Сторожова умова може містити мінімальну і максимальну кількість повторень, а також логічну умову. Субфрагмент повторюється до тих пір, поки сторожова умова істинна, але не менше, чим вказана мінімальна кількість разів, і не більше, ніж вказана максимальна кількість разів. Якщо сторожова умова відсутня, воно вважається істинним, і виконання циклу повністю визначається вказаною кількістю повторень.
3. Умовний фрагмент (ключове слово **alt**) – включає два або більше за субфрагмент, кожен з яких має сторожову умову. Субфрагменти відділяються один від одного за допомогою горизонтальних пунктирних ліній. Коли потік управління досягає умовного фрагмента, виконується той з його субфрагментів, сторожова умова якого є істинною. Якщо сторожова умова істинно більш ніж у одного субфрагмента, вибір одного з таких субфрагментів здійснюється випадковим чином. Якщо жодна із сторожових умов не є істинною, то жоден субфрагмент не виконується.
4. Необов’язковий фрагмент (ключове слово **opt**) – є часткою випадком умовного фрагмента. У нього входить один субфрагмент, який виконується у разі, якщо його сторожова умова істинна, і не виконується, якщо воно неправдиве.
5. Паралельний фрагмент (ключове слово **par**) – має два або більше субфрагмеитов. Коли потік управління досягає паралельного фрагмента, то усі його субфрагменти виконуються паралельно. Відносний порядок дотримання повідомлень в паралельних субфрагментах не визначений, і порядок виконання окремих елементів може бути будь-яким. Коли

виконання усіх субфрагментів завершується, потік управління наново зливається воєдино.

Крім того, в мові UML визначені ще вісім спеціальних комбінованих фрагментів.

Як приклад на рис. 10.33 показана діаграма послідовності для обробки замовлення квитків у філармонію.

У цю діаграму вкладено «використання взаємодії», що визначає параметри замовника, і фрагмент-цикл, причому в цикл, у свою чергу, вкладений умовний фрагмент з двома субфрагментами. У кожній ітерації циклу обробляється один елемент замовлення.

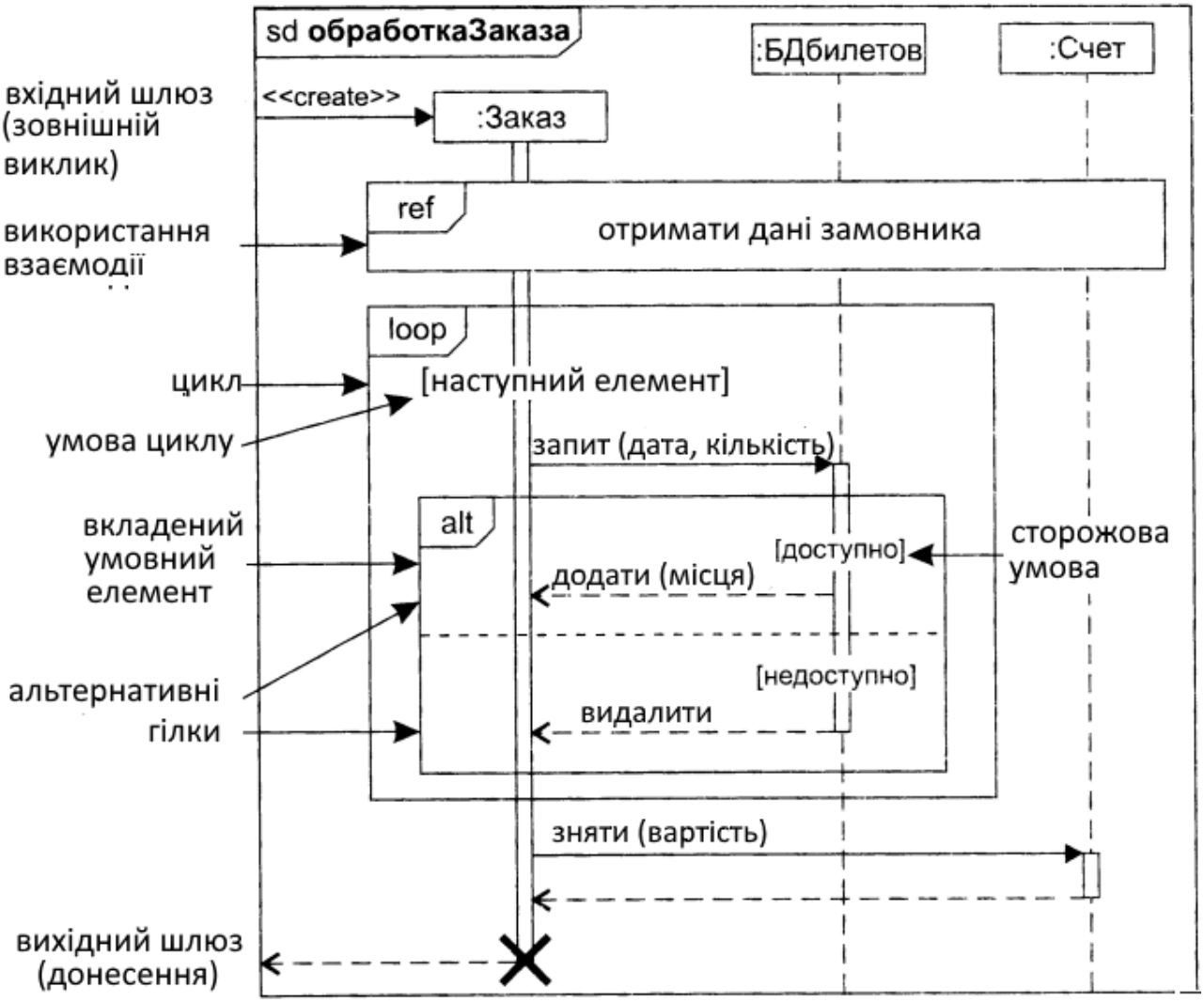


Рисунок 10.33 – Діаграма послідовності з вкладеними фрагментами

В ході обробки запрошується база цих квитків. Параметрами запиту є дата концерту і необхідна кількість квитків. При позитивному результаті (істинною є сторожова умова **доступно** першого умовного субфрагмента) в квитках вказуються місця. При негативному результаті (істинною є сторожова умова **недоступно** другого умовного субфрагмента) в квитках відмовляють. Цикл завершується після обробки усіх елементів замовлення. За підсумками обробки з рахунку клієнта знімається певна сума.

Обмеження на кількість ітерацій фрагмента-циклу вказуються в дужках після ключового слова **loop**:

loop – мінімум = 0, максимум необмежений

1oop(кількість) – мінімум = максимум = кількість

1oop(мін, макс) – явне завдання мінімальної і максимальної меж.

Окрім меж на лінії життя фрагмента може бути записаний логічний вираз сторожової умови. Цикл триває, поки умова істинна, але виконуватися він буде не менш мінімальної кількості разів і не більше максимальної, незалежно від сторожової умови.

На рис. 10.34 показаний цикл з межами і сторожовою умовою.

Рисунок 10.34 – Межі числа ітерацій

Діаграма послідовності дозволяє показати стани моделі. Стан (чи умова) зображається символом стану (прямокутник з заокругленими кутами), який поміщається на лінію життя. Цей стан (умова) зберігається в моделі до моменту настання наступного стану на лінії життя.

На рис. 10.35 показані стани в життєвому циклі квитка у філармонію. Коли лінія життя уривається символом стану, стан об'єкту міняється. До символу стану зазвичай примикає стрілка повідомлення, що викликало зміну стану.

Діаграма послідовності може взаємодіяти з іншими діаграмами. В цьому випадку в ній передбачаються вхідні і вихідні шлюзи.

Вхідний шлюз **–** це точка входу зовнішніх повідомлень в діаграму послідовності (див. рис. 10.33). Відповідно ***вихідний шлюз*** забезпечує видачу повідомлень в зовнішнє середовище.

Діаграми послідовності вважаються найпопулярнішим засобом представлення детальних вимог на етапі аналізу. Початковими даними для їх створення є словесні описи вимог замовника, що містяться в специфікаціях елементів Use Case (сценаріях). Ці описи і перетворяться в діаграми послідовності. Мета перетворення – підвищити рівень точності, формалізувати вимоги, привести їх до виду, необхідного для розв'язання завдань проектування.

Для побудови діаграм послідовності проводиться граматичний розбір кожного сценарію елементу Use Case: значущі іменники перетворюються на узагальнені об'єкти і їх атрибути, а значущі дієслова – в повідомлення, що пересилаються між об'єктами.

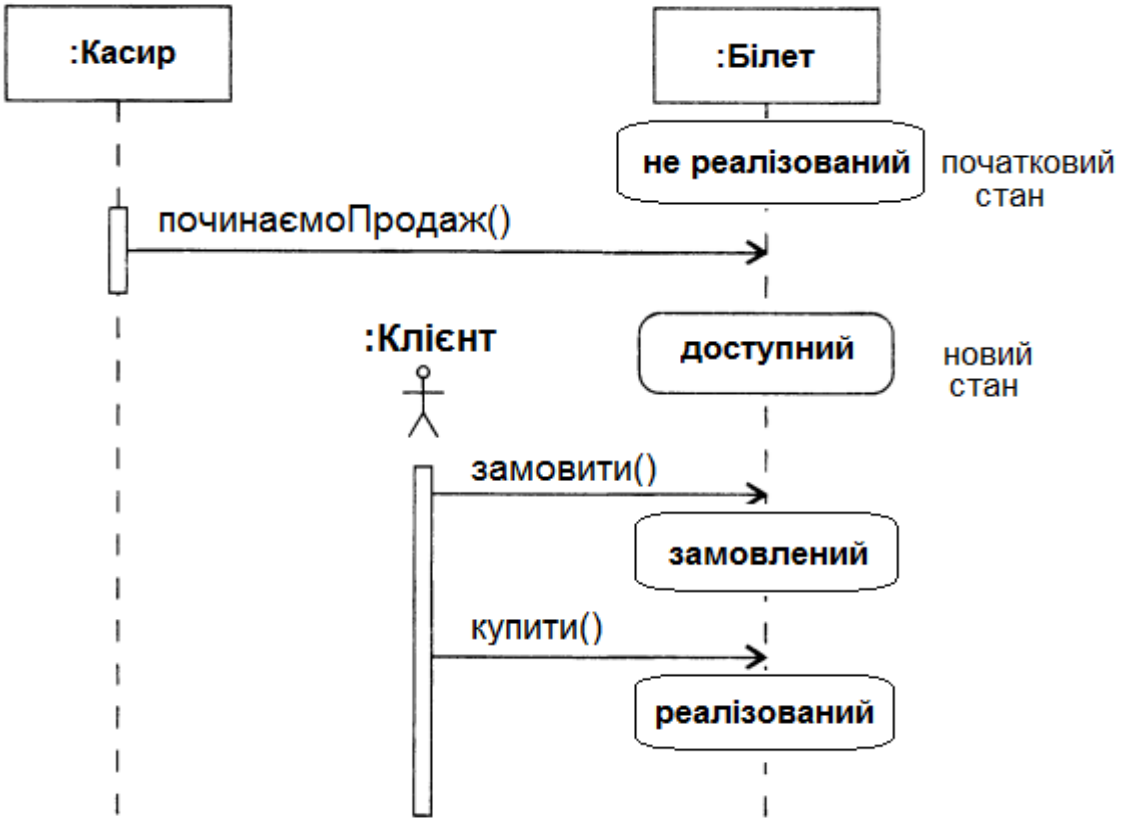


Рисунок 10.35 – Стан об’єкту на діаграмі послідовності

На цій стадії моделювання список операцій вказаних класів ще далеко неповний (а атрибути доки зовсім відсутні), проте сам факт появи класів в моделі є важливим рішенням, що істотно наближає нас до реалізації системи.

Більшість CASE-інструментів підтримують такий режим роботи. Складаючи деяку діаграму, (наприклад, в нашому випадку діаграму послідовності), можна визначити в моделі деякі сутності (наприклад, класи) потрібні для моделювання в даний момент, не відбиваючи їх поки що ні на якій діаграмі.

Можна уявити собі такий стиль моделювання, коли ніякі діаграми взагалі не складаються, а, користуючись засобами інструменту, в модель послідовно додаються сутності і відношення між ними. Звичайно, такий стиль ігнорує один з найважливіших аспектів UML – наочність моделі – і на практиці не застосовується. Проте, приводячи це зауваження, хотілося б ще раз підкреслити найважливішу обставину, що часто вислизає від початкуючих користувачів UML: **модель не є простою сумою діаграм, навпаки, кожна діаграма є не більше ніж обмеженою проекцією незалежно існуючої моделі**.

Обидва типи діаграм мають свої переваги і недоліки, тому не можна однозначно віддати перевагу одному з видів діаграм. Діаграми послідовності слід застосовувати тоді, коли вимагається подивитися на поведінку декількох об’єктів у рамках одного прецеденту. Діаграми послідовності хороші для представлення взаємодії об’єктів, але не дуже підходять для точного визначення поведінки.

З іншого боку, діаграми комунікації мають свої переваги при використанні діаграм великих розмірів, оскільки нові об’єкти до цих діаграм можна додавати як по горизонталі, так і по вертикалі.

Якщо ви хочете подивитися на поведінку одного об’єкта в декількох прецедентах, то застосуйте ***діаграму стану.*** Якщо ж потрібно вивчити поведінку декількох об’єктів в декількох прецедентах або потоках, не забудьте про ***діаграму діяльності***.

10.5.4. Діаграми комунікації

Діаграми комунікації відображають взаємодію ролей або об’єктів в процесі функціонування системи, описують обмін даними (повідомленнями) між різними учасниками взаємодії. Такі діаграми моделюють сценарії поведінки системи. Замість того щоб малювати кожного учасника у вигляді лінії життя і показувати послідовність повідомлень, розташовуючи їх по вертикалі, як це робиться в діаграмах послідовності, комунікаційні діаграми допускають довільне розміщення учасників, дозволяючи малювати зв’язки, що показують відношення учасників, і використати нумерацію для представлення послідовності повідомлень. Позначення об’єкту показане па рис. 10.36.

ім’я
атрибути

Рисунок 10.36 – Позначення об’єкту

Ім’я об’єкту підкреслюється і вказується завжди, атрибути вказуються вибірково. І ім’я об’єкту, і імена атрибутів записуються з маленької букви.

Синтаксис представлення імені має вигляд: **ім’яОб’єкту: Ім’яКласу**

Приклади запису імені.

адам: Людина	Ім’я об’єкту і класу
: Користувач	Тільки ім’я класу (анонімний об’єкт)
мойКомпьютер	Тільки ім’я об’єкту (мається на увазі, що ім’я класу відоме)
агент:	Об’єкт – сирота (мається на увазі, що ім’я класу невідоме)

Синтаксис представлення атрибуту має вигляд: **ім’я: Тип = Значення**

Приклади записи атрибуту.

номер: Телефон = ‘736-10-520’	Ім’я, тип, значення
активний = Тгис	Ім’я і значення

Об’єкти взаємодіють один з одним за допомогою зв’язків – каналів для передачі повідомлень. Зв’язок між парою об’єктів розглядається як екземпляр асоціації між їх класами. Іншими словами, зв’язок між двома об’єктами існує тільки тоді, коли є асоціація між їх класами. Неявно усі класи мають асоціацію самі з собою, отже, об’єкт може послати повідомлення самому собі.

Отже, зв'язок – це шлях для пересилки повідомлення. ***Повідомлення*** – це специфікація передачі інформації між об'єктами в очікуванні того, що буде забезпечена необхідна діяльність. Прийом повідомлення розглядається як подія. Результатом обробки повідомлення зазвичай є дія. У мові UML моделюються такі різновиди дій.

виклик (call)	У об'єкті запускається операція
повернення (return)	Повернення значення в об'єкт, що викликає
посилка (send)	У об'єкт посилається сигнал
створення	Створення об'єкта, виконується по стандартному повідомленню <<create>>
знищення	Знищення об'єкта, виконується по стандартному повідомленню <<destroy>>

Для запису повідомлень в мові UML прийнятий такий синтаксис:

имяАтрибута = имяСообщения (Аргументи):ВозвращаемоеЗначение

де **имяАтрибута** задає атрибут, куди поміщається значення, що повертається. Приклади запису повідомлень.

коорд = текущПоложенне(самолетТ1)	Виклик операції, повернення значення
сповіщення()	Посилка сигналу
установитьМаршрут(х)	Виклик операції з дійсним параметром
<<create>>	Стандартне повідомлення для створення об'єкта

Коли об'єкт посилає повідомлення в інший об'єкт (делегуючи деяку дію одержувачеві), об'єкт-одержувач, у свою чергу, може послати повідомлення в третій об'єкт і т.д. Так формується потік повідомлень – послідовність управління. Очевидно, що повідомлення в послідовності мають бути пронумеровані. Номери записуються перед іменами повідомлень, напрями повідомлень вказуються стрілками (розміщуються над лініями зв'язків). Найзагальнішу форму управління задає процедурний потік (потоік з вкладеннями) – потік синхронних повідомлень. Як показано на рис. 10.37, процедурний потік малюється стрілками із зафарбованими наконечниками.

Тут повідомлення **2.1: напій = виготовити(суміш№3)** визначено як перше повідомлення, вкладене в друге повідомлення **2: замовити(Суміш№3)** послідовності, а повідомлення **2.2: принести(напій)** – як друге вкладене повідомлення.

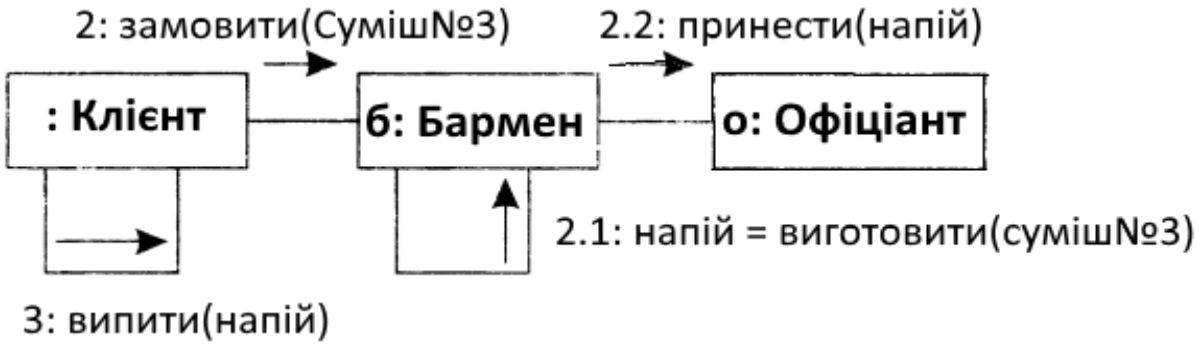


Рисунок 10.37 – Потік синхронних повідомлень

Усі повідомлення процедурної послідовності вважаються синхронними. Робота з синхронним повідомленням підпорядковується такому правилу: передавач чекає до тих нір, поки одержувач не прийме і не обробить повідомлення. У нашому прикладі це означає, що третє повідомлення буде послано тільки після обробки повідомлень 2.1 і 2.2.

Відмітимо, що міра вкладеності повідомлень може бути будь-якою. Головне, щоб дотримувалося правило – послідовність повідомлень зовнішнього рівня поновлюється тільки після завершення вкладеної послідовності.

Менш загальну форму управління задає асинхронний потік управління. Як показано на рис. 10.38, асинхронний потік малюється звичайними стрілками. Тут усі повідомлення вважаються асинхронними, при яких передавач не чекає реакції від одержувача повідомлення. Такий вид комунікації має семантику поштової скриньки – одержувач приймає повідомлення у міру готовності.

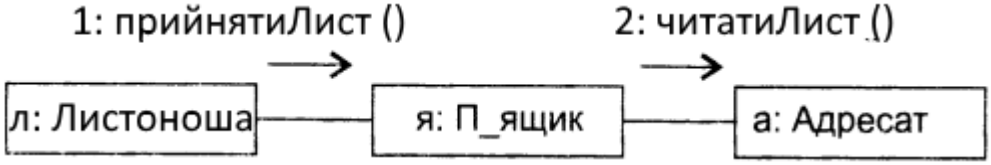


Рисунок 10.38 – Потік асинхронних повідомлень

Іншими словами, передавач і одержувач не синхронізують свою роботу, швидше один об’єкт «позбавляєтьсявід повідомлення для іншого об’єкту. У нашому прикладі повідомлення **читатиЛист()** визначене як друге повідомлення в потоці управління.

Помітимо, що учасниками взаємодій на рис. 10.32-10.33 оголошені ролі, тобто узагальнені об’єкти. Для посилань на звичайні об’єкти усі імена у вершинах треба було б підкреслити.

Окрім розглянутих лінійних потоків управління можна моделювати і складніші форми – ітерації і галуження.

Ітерація представляє послідовність повідомлень, що повторюється. Після номера повідомлення ітерації додається вираз: ***[i:= 1 . n]**

Він означає, що повідомлення ітерації повторюватиметься задану кількість разів. Наприклад, чотирикратне повторення першого повідомлення **мадюватиСторонуПрямокутника** можна задати вираженням: **1*[i:= 1 4]: мадюватиСторонуПрямокутника(i)**

Для моделювання галуження після номера повідомлення додається вираз умови, наприклад: [x>0]. Повідомлення альтернативної гілки позначається таким же номером, але з іншою умовою: [x<=0]. Приклад ітераційного потоку повідомлень, що розгалужується, наведений на рис. 10.39.

Тут перше повідомлення повторюється 4 рази, а в якості другого вибирається одне з двох повідомлень (залежно від значення змінної **x).** У результаті узагальнений об'єкт «рисователя» намалює на екрані прямокутне вікно, а узагальнений об'єкт «співрозмовника» виведе в нього відповідне донесення.

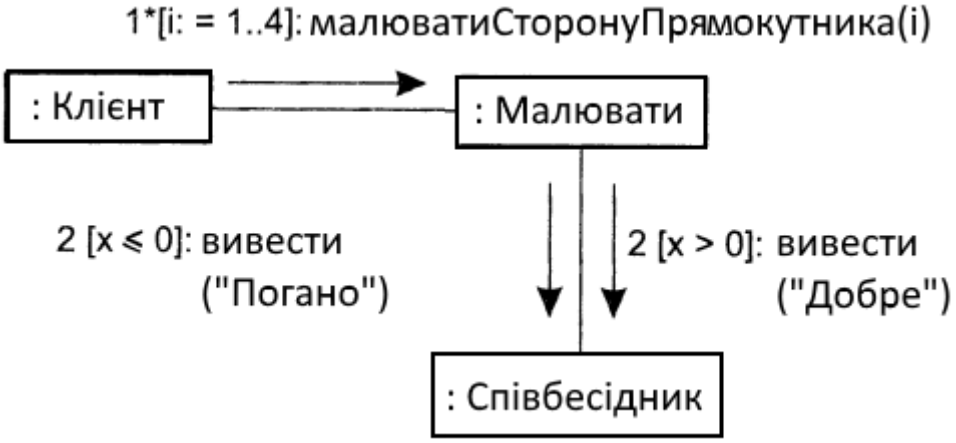


Рисунок 10.39 – Ітерація і галуження

Таким чином, для формування діаграми комунікації виконуються наступні дії:

- відображаються учасники
- цих учасників;
- зв'язки позначаються повідомленнями взаємодії;
- малюються зв'язки, що сполучають, які посилають і отримують певні учасники.

У результаті формується ясне візуальне представлення потоку управління (у контексті структурної організації співпрацюючих об'єктів).

Діаграма комунікації поміщається в рамку. Ім'я діаграми вказується вслід за позначкою **comm** в п'ятикутнику, що розміщується в лівому верхньому кутку рамки. Як приклад на рис. 10.40 приведена діаграма комунікації системи управління польотом літального апарату.

На цій діаграмі представлені п'ять узагальнених об'єктів системи. Потік управління в системі включає вісім повідомлень: чотири асинхронних і чотири синхронні повідомлення. Узагальнений екземпляр **Контроллера СУ** чекає прийому і обробки повідомлень:

вклРегСкор(); вклРегУгл(); текущСкор(); текущУгл()

Порядок дотримання повідомлень заданий їх номерами. Для п'ятого і сьомого повідомлень вказані умови:

- включення **Регулятора Швидкості** відбувається, якщо відносний час польоту ***T*** більше заданого періоду ***Тзад***;

– включення **Регулятора Кутів** забезпечується, якщо відносний час польоту менше або рівно заданому періоду.

Основне питання, пов’язане з **комунікаційними діаграмами,** полягає в тому, в яких випадках потрібно віддати перевагу їм, а не загальніші **діаграми послідовності**. Провідну роль в ухваленні такого рішення грають особисті вподобання.

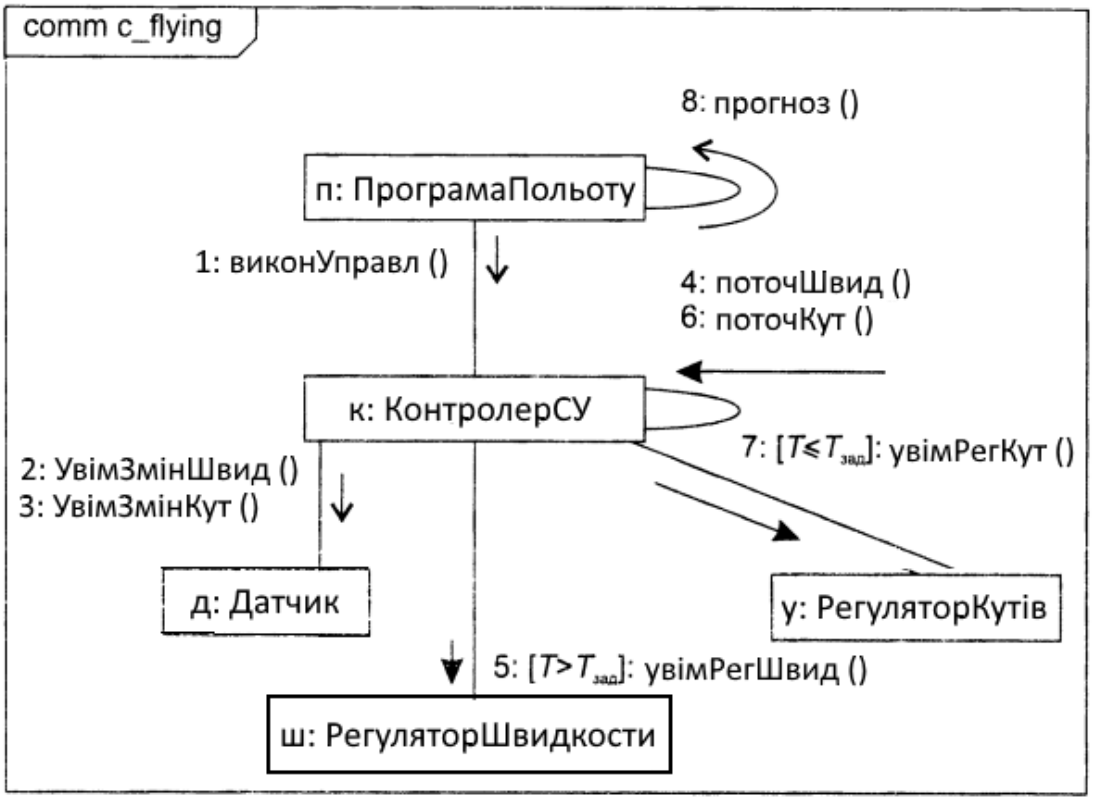


Рисунок 10.40 – Діаграма комунікації системи управління польотом

У зв'язку з тим, що діаграми послідовності і комунікації є різними поглядами на одні і ті ж процеси, більшість Case-засобів (наприклад, Rational Rose) дозволяють створювати з діаграми послідовності діаграму комунікації і навпаки, а також робити автоматичну синхронізацію цих діаграм.

10.5.5. Рекомендації з побудови діаграм послідовності і комунікації

Побудову діаграми послідовності доцільно починати з виділення і з всієї сукупності тих і тільки тих класів, об'єкти яких беруть участь у модельованій взаємодії. Після цього всі об'єкти наносяться на діаграму з дотриманням деякого порядку ініціалізації повідомлень. Тут необхідно встановити, які об'єкти будуть існувати постійно, а які тимчасово – тільки на період виконання ними необхідних дій.

Коли об'єкти візуальовані, можна розпочати специфікацію повідомлень. При цьому варто враховувати ті ролі, які відіграють повідомлення у системі. При необхідності для уточнення цих ролей треба використати їхні різновиди і стереотипи. Для знищення об'єктів, які

створюються на час виконання своїх дій, треба передбачити явне повідомлення.

Подальша деталізація діаграми послідовності пов'язана із введенням тимчасових обмежень до виконання окремих дій у системі. Для простих асинхронних повідомлень тимчасові обмеження можуть бути відсутні.

Діаграми комунікації також відображають взаємодію ролей або об’єктів в процесі функціонування системи, описують обмін даними (повідомленнями) між різними учасниками взаємодії як і діаграми послідовності. Але даграма послідовності не дозволяє показати такі деталі, які видно на діаграмі комунікації (структурні характеристики об’єктів і зв’язків).

Замість того щоб малювати кожного учасника у вигляді лінії життя і показувати послідовність повідомлень, розташовуючи їх по вертикалі, як це робиться в діаграмах послідовності, комунікаційні діаграми допускають довільне розміщення учасників, дозволяючи малювати зв’язки, що показують відношення учасників, і використати нумерацію для представлення послідовності повідомлень.

Контрольні питання до розділу 10

1. Визначте і опишіть чотири види вимог до ПЗ.
2. Які різновиди нефункціональних вимог ви знаєте?
3. У чому відмінності вимог замовника і вимог розробника?
4. У чому відмінності детальних вимог і вимог розробника?
5. Що потрібно зробити для забезпечення тестування вимоги?
6. Які способи організації детальних вимог ви знаєте?
7. З яких елементів складається діаграма Use Case?
8. Які відношення дозволені між елементами діаграми Use Case?
9. Для чого застосовують діаграми Use Case?
10. Що таке сценарій елементу Use Case?
11. Як документується відношення включення?
12. Як документується відношення розширення?
13. Який порядок побудови моделі вимог?
14. Охарактеризуйте засоби і можливості діаграми діяльності.
15. Коли не слід застосовувати діаграму діяльності?
16. Які засоби діаграми діяльності дозволяють відобразити паралельні дії?
17. Навіщо в діаграму діяльності введені «плавальні доріжки»?
18. Поясніть синтаксис представлення атрибуту в діаграмі комунікації.
19. Як вказується повторення повідомлень на діаграмі комунікації?

РОЗДІЛ 11. ОБ'ЄКТНО-ОРІЄНТОВАНЕ ПРОЕКТУВАННЯ ІС

Впродовж десятиліть розробники програмного забезпечення створювали свої проекти або з нуля, або використовуючи вже накопичений досвід. Нині інтенсивно розвивається дисципліна програмної архітектури і проектування. При аналізі моделюється те, **що повинна робити система**. При проектуванні моделюється те, як ця поведінка може бути реалізована.

У цьому розділі ми розглянемо коло питань проектування в об'єктно-орієнтованому стилі: принципи детального проектування, інструментарій мови UML, що підтримує архітектурне і детальне проектування [29; 33; 40].

Розглянемо способи відповіді на питання **з чого складається система**? У простих випадках однієї короткої відповіді на це питання буває достатньо для досягнення цілей моделювання. Наприклад: молоток складається з ударної частини і руків'я. У складніших випадках потрібні ієрархічні уточнення, наприклад: ударна частина столярного молотка має з одного боку бойок, а з інші обценьки і виготовляється із сталі.

У будь-якому випадку в центрі уваги знаходяться відношення «частина-ціле» і статичні властивості частин і цілого. При моделюванні структури не розглядаються такі відношення, як «причина-наслідок» або «раніше-пізніше». Тому план розділу полягає в тому, щоб спочатку обговорити загальні принципи моделювання структури, а потім розібрати різноманітні конкретні засоби моделювання структури, передбачені UML.

11.1. Принципи моделювання структури

Моделюючи структуру, ми описуємо складові частини системи і відношення між ними. UML у більшості випадків застосовується в якості об'єктно-орієнтованої мови моделювання, тому не дивно, що основним видом складових частин, з яких складається система при такому підході, є класи і відношення між ними.

У кожен конкретний момент функціонування системи можна вказати кінцевий набір конкретних об'єктів (екземплярів класів) і існуючих між ними зв'язків (екземплярів відношень). Проте в процесі роботи цей набір не залишається незмінним: об'єкти створюються і знищуються, зв'язки встановлюються і втрачаються.

Число можливих варіантів наборів об'єктів і зв'язків, які можуть мати місце в процесі функціонування системи, якщо і не нескінченно, то може бути неосяжно велике. Представити їх усі в моделі практично неможливо, а головне безглуздо, оскільки така модель із-за свого об'єму буде недоступна для розуміння людиною, а, значить, і даремна при розробці системи. Яким же чином можна будувати компактні (корисні) моделі неосяжних (потенційно нескінченних) систем? Відповідь на це питання викладена в наступному параграфі.

11.1.1. Дескриптори

Метод побудови кінцевих (і невеликих) моделей нескінченних (чи дуже великих) систем відомий людству споконвіку і пронизує усе сучасне знання в різних формах і під різними назвами. Наведемо декілька прикладів з предметних областей, близьких до цієї теми.

Перший приклад належить до звичайного програмування. Професійні програмісти так звикли працювати з текстами програм на мові програмування, що часто забувають, що код – це не більше ніж модель обчислювального процесу.

Зазвичай текст програми компактний, а описувані ним обчислення неосяжні. Розробники пишуть програми (складають точні описи послідовностей елементарних дій), які вони (розробники) самі виконати свідомо не в змозі.

Ще приклад. Розглянемо опис якої-небудь формальної мови, скажемо мови програмування Паскаль. Він складається з приблизно двох десятків синтаксичних правил, декількох десятків сторінок тексту і близько сотні невеликих прикладів. Відмітимо, що синтаксис мови Паскаль описаний Н. Віртом за допомогою синтаксичних діаграм.

Відмітимо, що в наведених прикладах, окрім найкомпактнішого опису, мається на увазі, що відомий набір правил інтерпретації опису, що дозволяють побудувати за описом великої кількості будь-який його елемент. Самі правила і спосіб їх завдання різні в різних випадках, але принцип один і той же.

У UML цей принцип формалізований у вигляді поняття дескриптора.

Дескриптор **–** це опис загальних властивостей множини об’єктів, включаючи їх структуру, відношення, поведінку, обмеження, призначення тощо. Дескриптор має дві сторони: це сам опис великої кількості (intent) і множини значень, що описуються дескриптором (extent). Антонімом для дескриптора є поняття *літерала* (literal).

Літерал описує сам себе. Наприклад, тип даних integer є дескриптором: він описує множину цілих чисел, потенційно нескінченну (чи кінцеву, але досить велику, якщо йдеться про машинну арифметику).

Зображення числа 1 описує саме число «один» і більше нічого – це літерал. Майже усі елементи моделей UML є дескрипторами – саме тому засобами UML вдається створювати представницькі моделі досить складних систем. Розглянуті раніше варіанти використання і дійові особи – дескриптори, класи, що розглядаються тут, асоціації, компоненти, вузли – також дескриптори. Коментар же є літералом – він описує сам себе. Пакет також є літералом.

11.1.2. Призначення структурного моделювання

Розглянемо детальніше, які саме структури треба моделювати і навіщо. Виділимо такі структури:

- структура зв’язків між об’єктами під час виконання програми;

- структура зберігання даних;
- структура програмного коду;
- структура компонентів в додатку;
- структура складних об’єктів, що складаються зі взаємодіючих частин;
- структура артефактів в проекті;
- структура використовуваних обчислювальних ресурсів.

Наша класифікація може бути не зовсім повна і вже зовсім не ортогональна (згадані структури не є незалежними, вони пов’язані один з одним), але в цілому відповідає практиці розробки додатків, що склалася, оскільки дозволяє фіксувати основні рішення, що приймаються в процесі проектування і реалізації.

У цьому розділі ми коротко обговоримо призначення перерахованих структур і вкажемо засоби UML, що призначені для їх моделювання.

Структура зв’язків між об’єктами під час виконання програми. У парадигмі об’єктно-орієнтованого програмування процес виконання програми полягає в тому, що програмні об’єкти взаємодіють один з одним, обмінюючись повідомленнями. Найбільш поширеним типом повідомлення є виклик методу об’єкту одного класу з методу об’єкту іншого класу.

Для того щоб викликати метод об’єкту, треба мати доступ до цього об’єкту. На рівні програмної реалізації цей доступ може бути забезпечений найрізноманітнішими механізмами. Наприклад, об’єкт, що викликає метод, може зберігати покажчик (посилання) на об’єкт, що містить метод, який викликається.

Якщо атрибути об’єктів представлені записами в таблиці бази даних, а методи (нерідкий варіант реалізації) – процедурами, що зберігаються, системи управління базами даних (СУБД), то ідентифікація об’єктів здійснюється за первинним ключом таблиці.

В усіх випадках має місце така ситуація: один об’єкт «знає» інші об’єкти і, значить, може викликати відкриті методи, використовувати і змінювати значення відкритих атрибутів і так далі. В цьому випадку, говорять, що об’єкти пов’язані, тобто між ними є зв’язок. Для моделювання структури зв’язків в UML використовуються відношення асоціації на діаграмі класів.

Структура зберігання даних. Програми обробляють дані, які зберігаються в пам’яті комп’ютера. У парадигмі об’єктно-орієнтованого програмування для зберігання даних під час виконання програми призначені атрибути класів. Проте велика частина додатків для автоматизації діловодства влаштована так, що певні дані (не всі) повинні зберігатися в пам’яті комп’ютера не лише під час сеансу роботи додатка, але постійно, тобто між сеансами.

Об’єкти, які зберігають значення своїх атрибутів навіть після того, як завершився потік управління, що породив їх, називають такими, що зберігаються. У UML для моделювання цього атрибуту об’єктів і їх складових застосовується стандартна властивість {**persistence**}, яка може бути призначена класифікаторові, асоціації або атрибуту і може приймати одне з двох значень:

– **persistent** – екземпляри класифікатора, асоціації або значення атрибуту, відповідно, мають бути такими, що зберігаються;
– **transient** – протилежно попередньому – зберігати екземпляри не вимагається (значення за умовчанням).

Нині найпоширенішим способом зберігання об'єктів є використання СУБД. При цьому класу, що зберігається, відповідає таблиця бази даних, а об'єкт (точніше кажучи, набір значень атрибутів), що зберігається, представляється записом в таблиці.

Питання структури зберігання даних є первинним для додатків баз даних. На щастя, відомі надійні методи вирішення цього питання – схеми баз даних, діаграми «сутність-зв'язок». Ці ж методи (з точністю до позначень) застосовуються і в UML у формі асоціацій з вказівкою кратності полюсів.

Структура програмного коду. Програми істотно відрізняються за величиною – бувають програми великі і маленькі. Наскільки великі ці відмінності: від сотень рядків коду (і менше) до сотень мільйонів рядків (і більше). Такі великі кількісні відмінності не можуть не проявлятися і на якісному рівні. Дійсно, для маленьких програм структура коду практично не має значення, для великих – навпаки, має чи не вирішальне значення.

Оскільки UML не є мовою програмування, модель не визначає структуру коду безпосередньо, проте непрямим чином структура моделі істотно впливає на структуру коду.

Більшість інструментів підтримують напівавтоматичну генерацію коду для однієї або декількох, частіше об'єктно-орієнтованих, мов програмування. У більшості випадків класи моделі транслюються в класи (чи еквівалентні їм конструкції) цільової мови. Крім того, багато інструментів враховують структуру пакетів в моделі і транслюють її у відповідні «надкласові» структури цільової системи програмування. Таким чином, якщо задіяний засіб автоматичної генерації коду, то структура класів і пакетів в моделі фактично повністю моделює структуру коду додатка.

Структура компонентів в додатку. Додаток, що складається з однієї компоненти, має тривіальну структуру компонентів, моделювати яку немає сенсу. Але більшість сучасних застосувань на етапі проектування є взаємозв'язком багатьох компонентів, навіть якщо і не є розподіленими.

Компонентна структура припускає опис двох аспектів: по-перше, як класи розподілені по компонентах, по-друге, як (через які інтерфейси) компоненти взаємодіють один з одним. Обидва ці аспекти моделюються діаграмами компонентів UML.

Структура складних об'єктів, що складаються зі взаємодіючих частин. Для моделювання цієї структури застосовується новий засіб UML 2 – діаграма внутрішньої структури класифікатора.

Ця діаграма використовується для опису внутрішньої структури класів і компонентів. Існує ще одна сутність, яка також дозволяє описати взаємодію множини частин. Це сутність називається кооперацією і служить для опису взаємодії в деякому контексті. З точки зору внутрішньої структури основна відмінність кооперації від класу і компонента полягає в тому, що кооперація не

є власником своїх частин, і з'єднувачі частин кооперації можуть не мати явного вираження у вигляді асоціації. Проте, як у класів і компонентів, у кооперації можуть бути екземпляри, які функціонують під час виконання.

Структура артефактів в проекті. Тільки найпростіші застосування складаються з одного артефакту – здійснимого коду програми. Більшість реальних застосувань налічують у своєму складі десятки і сотні різних компонентів: здійснимих двійкових файлів, файлів ресурсів, файлів початкового коду, різних супроводжуючих документів, довідкових файлів, файлів з даними і так далі

Для великого застосування важливо не лише мати точний і повний список усіх артефактів, але і вказати, які саме з них входять в конкретний екземпляр системи. Річ у тому, що для великих застосувань в проекті співіснують різні версії одного і того ж артефакту. Це вичерпним чином моделюється діаграмами компонентів і розміщення UML, де передбачені стандартні стереотипи для опису артефактів різних типів.

Структура використовуваних обчислювальних ресурсів. Додаток, що складається з багатьох артефактів, як правило, буває розподіленим, тобто різні артефакти розміщуються на різних комп'ютерах. Діаграми розміщення дозволяють включити в модель опис і цієї структури.

11.1.3. Класифікатори

Найважливішим типом дескрипторів є класифікатори.

Класифікатор **(classifier) –** це дескриптор множини однотипних об'єктів.

З цього визначення безпосередньо витікає основна і характеристична властивість класифікатора: **класифікатор (прямо або побічно) може мати екземпляри**.

У UML визначені досить багато класифікаторів. Ми розглядаємо їх частинами. Раніше ми детально розглядали тільки два з них, а саме:

- дійова особа (actor);
- варіант використання (use case).

Класифікатори, які розглянуті в цьому розділі, наведені нижче:

- артефакт (artifact);
- тип даних (data type);
- асоціація (association);
- клас асоціації (association clas);
- інтерфейс (interface);
- клас (clas);
- кооперація (collaboration);
- компонент (component);
- вузол (node).

Усі класифікатори мають деякі загальні властивості, які використовуються в подальшому викладі. У цьому параграфі опишемо сім найважливіших властивостей класифікаторів.

1. Класифікатори (як і усі елементи моделі) мають імена. Ім'я служить для ідентифікації елементу моделі і тому має бути унікальне в цьому просторі імен.

2. Класифікатор може мати екземпляри. Екземпляри бувають прямі і непрямі.

Якщо деякий об'єкт безпосередньо породжений за допомогою конструктора класифікатора **A,** то цей об'єкт називається ***прямим екземпляром*** (direct instance) класифікатора **A**.

Якщо класифікатор **A** є узагальненням класифікатора **B** або, що те ж саме, класифікатор **B** є спеціалізацією класифікатора **A,** то усі екземпляри класифікатора **B** є ***непрямими екземплярами*** класифікатора **A**.

Ця властивість є транзитивною: якщо класифікатор **A** є узагальненням класифікатора **B,** а класифікатор **B** є узагальненням класифікатора **C,** то усі екземпляри класифікатора **C** також є непрямими екземплярами **A**.

3. Класифікатор може бути абстрактним або конкретним.

Абстрактний (abstract) класифікатор не може мати прямих екземплярів і в цьому випадку його ім'я виділяється курсивом.

Конкретний (concrete) класифікатор може мати прямі екземпляри і в цьому випадку його ім'я записується прямим шрифтом.

Абстрактний класифікатор – це такий дескриптор множини об'єктів, в якій немає прямого опису елементів великої кількості, але цей класифікатор пов'язаний відношенням узагальнення з іншими класифікаторами і об'єднання множини їх екземплярів вважається множиною екземплярів цього абстрактного класифікатора.

Іншими словами, множина визначається не прямо, а через сукупність підмножин. Наприклад, інтерфейс, будучи абстрактним класом, не може мати безпосередніх екземплярів, але клас, що реалізовує його, може, отже, інтерфейс є класифікатором.

4. Класифікатор (як і інші елементи моделі) має видимість.

Видимість **(visibility)** визначає, чи може складова одного класифікатора (у тому числі ім'я) використовуватися в іншому класифікаторові.

Іншими словами, якщо в певному контексті щось доступне і може бути якось використано, то воно є видимим (у цьому контексті). Якщо ж воно не видиме, то і не може бути використано. Видимість є властивістю усіх елементів моделі (хоча не для усіх елементів ця властивість є істотною). Видимість може приймати одне з чотирьох значень:

1. *Відкритий (позначається знаком* **+** або ключовим словом **public**).
2. *Захищений (позначається знаком* **#** або ключовим словом **protected**).
3. *Закритий (позначається знаком* – або ключовим словом **private**).
4. *Пакетний (позначається знаком* **~** або ключовим словом **package**).

Відкритий елемент моделі є видимим скрізь, де є видимим елемент, що містить його. Наприклад, відкритий атрибут класу видно скрізь, де видно сам клас.

Захищений елемент моделі видно як в елементі, що його містить (контейнері), так і в усіх елементах, для яких контейнер є узагальненням.

Наприклад, захищений атрибут класу видно в класі, що містить його, і в усіх підкласах.

Закритий елемент моделі видно тільки в елементі, якому він належить. Наприклад, закритий атрибут класу видно тільки в цьому класі.

Елемент моделі зі значенням видимості пакетний, видимий елементам тільки того пакету, в якому він сам визначений.

Для видимості в UML немає значення за умовчанням. Наприклад, якщо для атрибуту класу не вказано значення видимості, то це не означає, що атрибут за умовчанням відкритий або, навпаки, закритий. Це означає, що видимість для цього атрибуту в моделі не вказана і в цьому аспекті модель не повна.

5. Усі складові класифікатора мають зону дії.

Зона дії (scope) визначає, як проявляє себе складова класифікатора в екземплярах, тобто мають екземпляри свої значення складової або спільно використовують одне значення.

Зона дії має два можливі значення:

1. **Екземпляр** (instance) – ніяк спеціально не позначається, оскільки мається на увазі за умовчанням.
2. **Класифікатор** (classifier) – опис складової класифікатора підкреслюється.

Якщо зоною дії складової є екземпляр, то кожен екземпляр класифікатора має своє значення складової.

Наприклад, зоною дії атрибуту за умовчанням є екземпляр. Це означає, що кожен об’єкт – екземпляр класу – має своє власне значення атрибуту, яке може мінятися незалежно від значень цього атрибуту інших об’єктів, екземплярів цього ж класу.

Якщо зоною дії складової є класифікатор, то усі екземпляри класифікатора спільно використовують одне значення складової. Наприклад, конструктор зазвичай має зоною дії класифікатор (клас), оскільки є процедурою, загальною для усіх екземплярів цього класу.

6. Класифікатор має *кратність*, тобто обмеження на кількість екземплярів класифікатора, як множини. Не слід плутати кратність з кількістю елементів (екземплярів). Множина, що вказана в моделі, під час виконання може мати різну кількість елементів, і кількість елементів може динамічно мінятися. Кратність визначає межі цих змін.

***Кратність* (multiplicity) *множини* –** це множина чисел, які задають усі допустимі значення потужності для цієї множини.

Синтаксично кратність задається виразом, який є непорожньою послідовністю елементів (розділених комами), кожен з яких має такий формат.

Нижня межа .. ВЕРХНЯ МЕЖА

У якості верхній і нижньої межі використовуються натуральні числа або нуль. Крім того, як верхня межа може використовуватися символ *. Якщо нижня межа не задана, то вона опускається разом з символом .. (дві точки). У таблиці. 11.1 наведені деякі приклади виразів кратності.

Таблиця 11.1 – Вирази кратності

Вирази кратності	**Вирази кратності**
0..* чи *	Довільне число елементів
1..*	Один або більше за елементи
0..1	Не більше за один елемент
1..10	Від одного до десяти елементів
1..3, 5, 7..10	Один, два, три, п’ять, сім, вісім, дев’ять або десять елементів
5..3	Некоректна кратність. Нижня межа більша за верхню
-1..3	Некоректна кратність. Негативні числа недопустимі

Зазвичай на практиці використовуються такі варіанти кратності класифікаторів.

Класифікатор не має екземплярів (кратність 0) – такий класифікатор називається ***службою*** (utility). Усі складові служби мають зоною дії класифікатор. Зберігання інформації, що обробляється службою, забезпечують об’єкти, що використовують службу. Типовий приклад – набір процедур загального користування, скажімо, бібліотека математичних функцій. Служби використовуються в додатках досить часто, тому є навіть стандартний стереотип *«***utility»**, що визначає класифікатор як службу.

Класифікатор має рівно один екземпляр (кратність 1). Такий класифікатор називається ***одинаком*** (singleton). По суті, між службою і одинаком відмінності незначні, але іноді одинака використати зручніше, наприклад, коли класифікатор є елементом реального світу, існуючим в єдиному екземплярі. Як приклад можна навести клавіатуру (один екземпляр), яка підключається до комп’ютера.

Класифікатор має фіксоване число екземплярів (наприклад, кратність 7**).** Такий варіант не часто, але зустрічається. Наприклад, моделювання портів в концентраторі.

Класифікатор має довільне число екземплярів (кратність ***).** Оскільки цей варіант зустрічається найчастіше, він ніяк спеціально не вказується і мається на увазі за умовчанням.

7. Класифікатори (і тільки вони) **можуть брати участь відносно *узагальнення.***

Сказане в цьому розділі закінчимо діаграмою класів з метамоделі UML (рис. 11.1), яка описує саме поняття класифікатора і містить ті класифікатори, які будуть розглянуті у рамках моделювання структури.

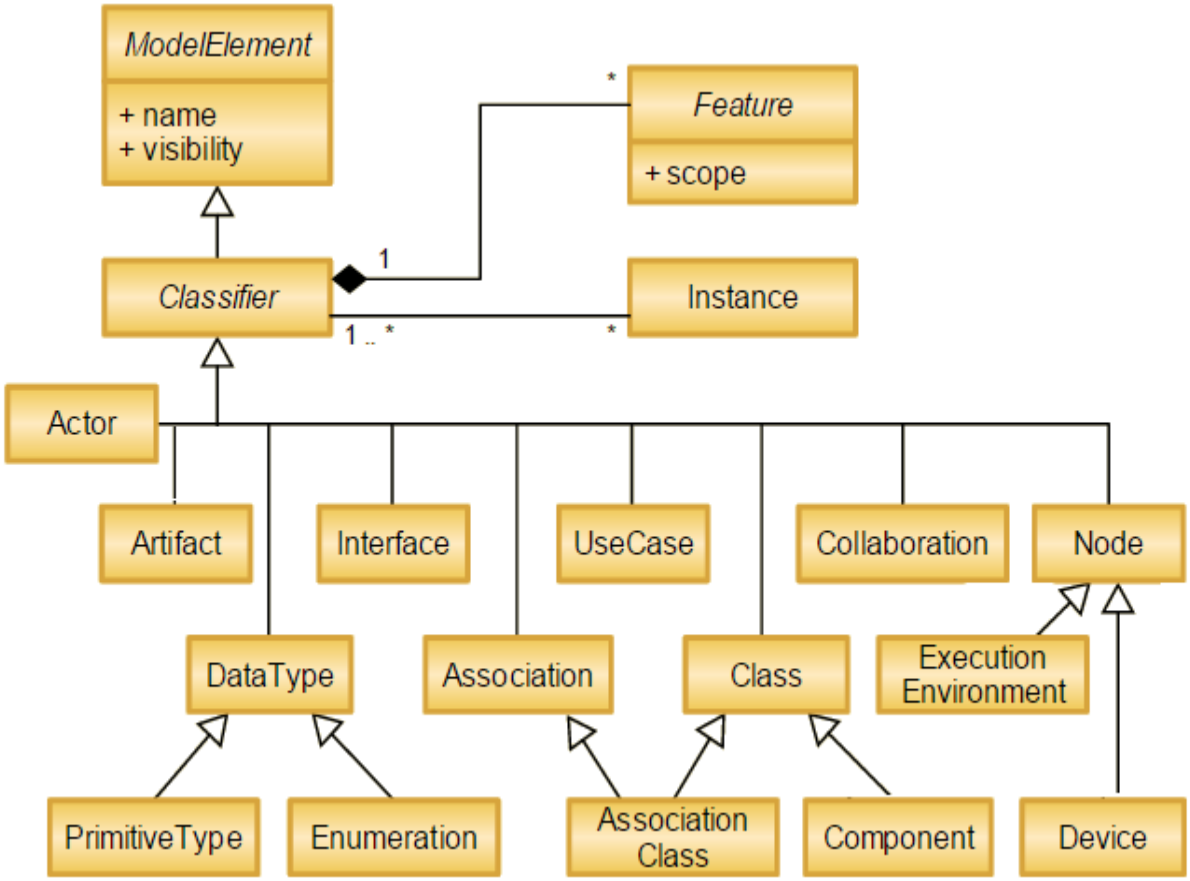


Рисунок 11.1 – Частина метамоделі класифікатора

11.1.4. Ідентифікація класів

Повторимо ще раз: опис класів і відношень між ними є основним засобом моделювання структури в UML. Але перш ніж переходити до техніки опису класів, корисно обговорити питання про те, як виділяються класи, що підлягають опису. З цього питання в літературі наводиться багато різних міркувань, порад і рекомендацій.

Сама різноманітність підходів свідчить про те, що серед них немає універсального і застосовного в усіх випадках. Виберемо з цих рекомендацій три прийоми виділення класів, найпростіших, а тому найдієвіших і широко застосовних:

- словник предметної області;
- реалізація варіантів використання;
- зразки проектування.

Словник предметної області **(domain dictionary) –** це набір основних понять (сутностей) цієї предметної області.

Точнішого визначення дати неможливо, зате можна дати корисну пораду. Розгляньте уважно текст технічного завдання (чи іншого документу, що лежить в основі проекту) і виділіть у змістовній частині іменники – усі вони є кандидатами на те, щоб бути назвами класів (чи атрибутів класів) проектованої системи. Зрозуміло, після цієї простої операції до отриманого списку треба

застосувати фільтр здорового глузду і досвіду, відсікаючи непотрібне або додаючи пропущене.

Розглянемо тепер, як виявляються класи в процесі реалізації варіантів використання. Якщо при реалізації варіантів використання застосовуються діаграми взаємодії, то в цьому процесі в якості побічного ефекту виділяються деякі класи безпосередньо, оскільки на діаграмах комунікації і послідовності основними сутностями є об'єкти, які з потреби треба віднести до певних класів.

Використання діаграм діяльності також може підказати, які класи треба визначити в системі, особливо якщо на діаграмі діяльності разом з ***потоком управління*** (control flow) є присутнім ***потік даних*** (data flow).

Проте, якщо сценарії варіантів використання описуються природною мовою або псевдокодом, то виділити класи значно важче.

Фактично, якщо варіанти використання реалізуються на псевдокоді або діаграмами діяльності без всякого зв'язку з об'єктами, то виявлення об'єктної структури системи просто відкладається «на потім». Іноді це може бути цілком виправдано. Наприклад, архітектор, що моделює систему, перш ніж почати проектування основної структури класів, хоче глибше вникнути в логіку бізнес-процесів незнайомої йому предметної області.

Підведемо підсумки. Класи в моделі ідентифікуються в результаті проведення трьох частково незалежних процесів: **аналізу предметної області, узгодження із вже побудованою моделлю і застосування теоретичних міркувань**. Жодним з цих методів не можна нехтувати, але одночасно можна стверджувати, що жоден з них не є універсальним і самодостатнім: вирішальним є здоровий глузд і досвід архітектора, який будує модель.

11.2. Архітектурне проектування

Формування архітектури – перший і засадничий крок у розв'язанні задачі проектування, який закладає фундамент представлення програмної системи, здатної виконувати весь спектр детальних вимог.

Створення архітектури – це ***проектування на найвищому*** рівні (логічна архітектура). *Логічна архітектура* описує систему в термінах її принципової організації у вигляді пакетів, програмних класів і підсистем. Вона називається логічною, оскільки не визначає способи розгортання цих елементів в різних операційних системах або на фізичних комп'ютерах в мережі (це стосується *архітектури розгортання*).

Частину процесу проектування, що залишилася, називають ***детальним проектуванням.*** Ясний опис архітектури дуже важливий для усіх застосувань і обов'язковий у тому випадку, коли до розробки залучається велика кількість людей. Причиною цього служить необхідність розбиття усього застосування на частини (модулі) з їх подальшим складанням. Вибір архітектури забезпечує необхідну модульність.

Після невеликої практики досить просто створювати маленькі програми. Великі застосування із-за їх складності ставлять перед розробниками дуже

важкі завдання, розв'язання яких досить важко досягається на практиці. Складність не в сенсі кількості рядків коду, а в сенсі їх взаємозв'язку. Дуже хороший спосіб боротьби із складністю – розбиття завдання на підзадачі, що мають характерні властивості невеликих програм.

Мова UML підтримує техніку об'єктно-орієнтованої декомпозиції і включає спеціальні засоби для запису структур архітектурного рівня, які гармонізовані як із засобами для фіксації вимог, так і із засобами для забезпечення детального проектування. Розглянемо ці засоби.

11.2.1. Діаграми пакетів

Діаграма пакетів служить, в першу чергу, для організації елементів в групи за якою-небудь ознакою з метою спрощення структури і організації роботи з моделлю системи.

Класи складають структурний кістяк об'єктно-орієнтованої системи. Хоча вони виключно корисні, але потрібне щось більше для структуризації великих систем, які можуть складатися з сотень класів.

Діаграма пакетів – це структурна діаграма, в якій основними елементами є пакети і залежності між ними.

Пакет – сховище елементів, зображається у вигляді прямокутника із закладкою в одному з кутів (зазвичай в лівому верхньому). Якщо вміст пакету не показується, ім'я пакету вказують усередині прямокутника (рис. 11.21).

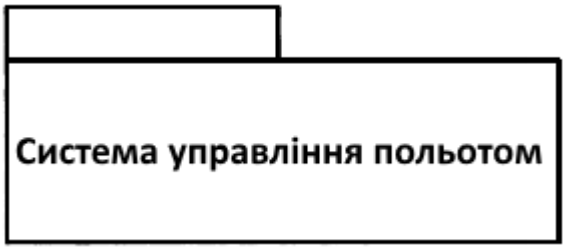


Рисунок 11.2 – Іменування пакету з прихованим вмістом

При відображенні утримуваного пакету в прямокутнику ім'я пакету вказують усередині закладки (рис. 11.3).

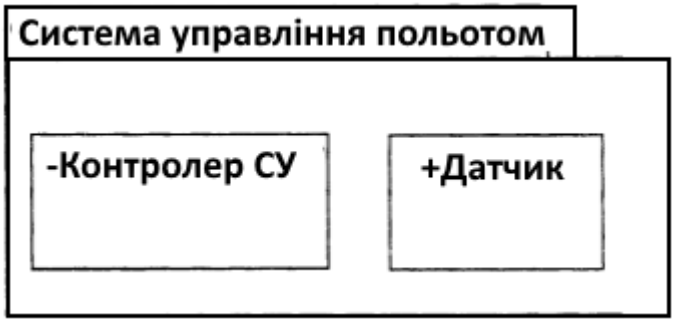


Рисунок 11.3 – Іменування пакету з відображенням його змісту

Вміст пакету можна показати і іншим способом, за допомогою відношення володіння (рис. 11.4). Відношення володіння (containment) означають лінією з невеликим хрестиком в гуртку, що примикає до контура

пакету. В даному випадку пакет **Система управління польотом** володіє двома класами: **Контроллер СУ** і **Датчик**.

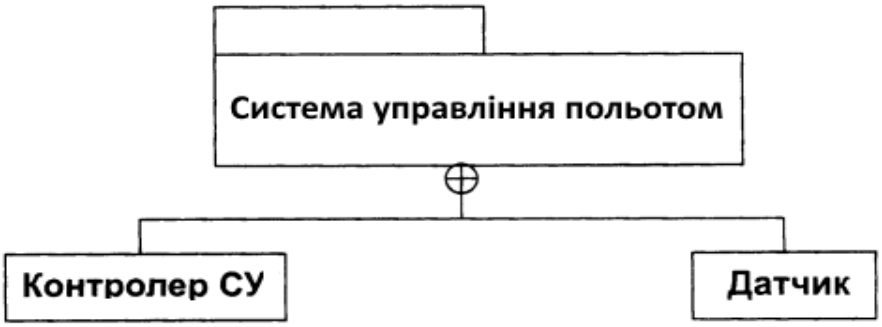


Рисунок 11.4 – Зовнішнє позначення утримуваного пакету

Пакет є засобом угрупування, всередині пакету можуть знаходитися вкладені пакети, класи тощо. Пакет задає видимість елементів, що знаходяться в ньому. Видимість характеризує доступність елементів (чи їх вмісту) іншим елементам моделі. Видимість елементів пакету може бути визначена як приватна або публічна. Публічні елементи доступні іншим елементам пакету-власника або вкладених в нього пакетів, а також пакетам, що імпортують цей пакет. Приватні елементи недоступні поза пакетом-власником.

Видимість елементу пакету вказується за допомогою символу видимості, який ставиться перед ім’ям елемента:

1. Плюс (+) означає публічну видимість.
2. Мінус (-) відповідає приватній видимості.

Наприклад, на рис. 11.2 клас **Контроллер СУ** є приватним (до нього немає доступу поза пакетом **Система управління польотом**), а клас **Датчик –** публічним елементом (він доступний для усіх). Можливість доступу до якогось елементу зображається за допомогою відношення **залежності**. Це відношення показує, що деякий елемент залежить від іншого елементу.

Залежності між елементами пакетів і пакетами показуються за допомогою пунктирних стрілок. Хвіст стрілки примикає до залежного елементу (клієнта), а вістря вказує на елемент, від якого залежать (сервер) (рис. 11.5).

Імена елементів в кожному пакеті утворюють власний простір імен, в межах якого працює схема прямої адресації, по простих адресах. На елементи з інших пакетів можна посилатися за їх кваліфікованими іменами, що мають формат **ім’я пакету::ім’я елементу**.

Для спрощення посилань на чужі імена слід застосовувати механізм імпортування. Пакет (клієнт), що імпортує елемент з іншого пакету (постачальника), звертається до елементів, що імпортуються, за їх простими іменами. Імпортувати можна тільки публічні (видимі) елементи. Приватні елементи пакетів імпортувати в інші пакети не можна.

Можлива як приватна, так і публічна видимість імпорту всередині імпортуючого пакету. Якщо вона публічна, імпортований елемент видно усім елементам, яким видно весь імпортуючий пакет в цілому. Якщо ж вибрана приватна видимість, імпортований елемент не буде видимий ззовні імпортуючого пакету.

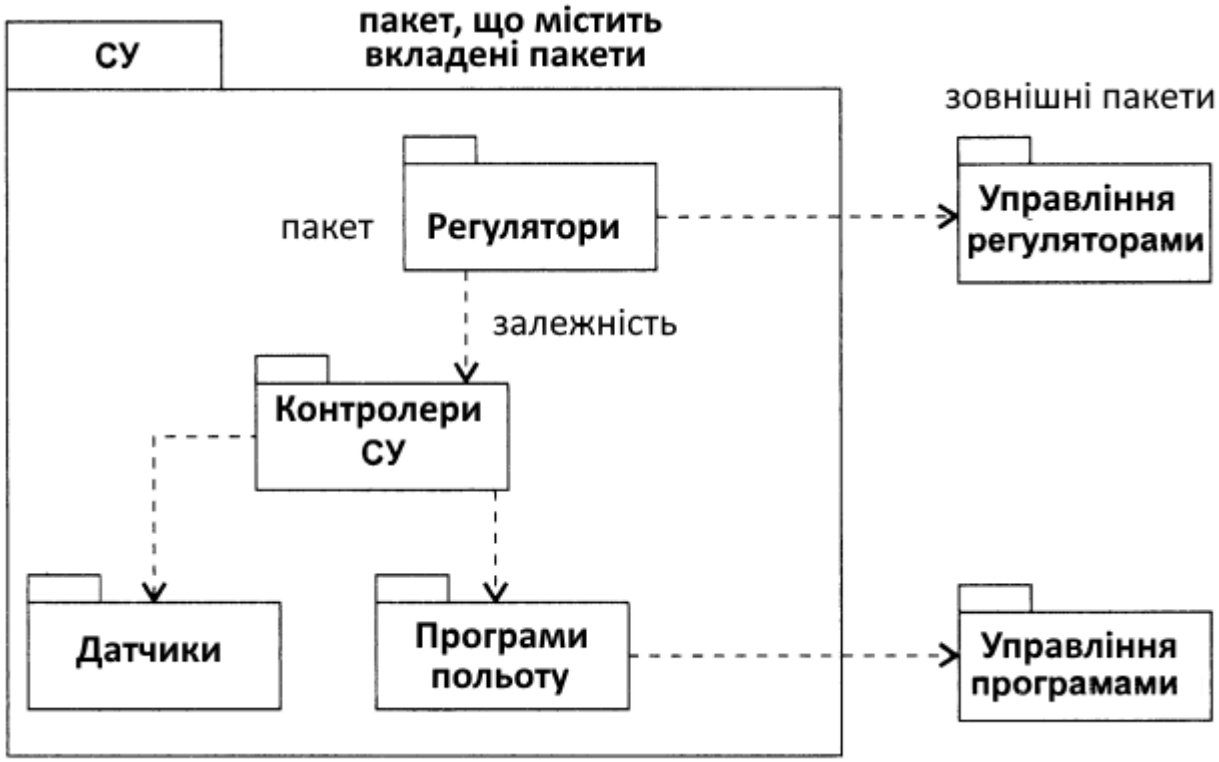


Рисунок 11.5 – Залежності між пакетами системи управління літальним апаратом

У загальному випадку працюють такі правила видимості елементів:

1. Визначений в пакеті елемент видно усередині цього пакету.
2. Якщо елемент видно усередині пакету, його видно і всім пакетам, вкладеним в даний.
3. Якщо пакет імпортує інший пакет з публічними елементами, то всі елементи імпортованого пакету виявляються видні всередині імпортуючого пакету.
4. Пакет не бачить елементів власних вкладених пакетів, поки він не імпортує їх (за умови, що ці елементи публічні).

Залежність імпортування відображається пунктирною стрілкою, хвіст якої знаходиться на пакеті-клієнтові, а вістря – на пакеті-постачальнику. В якості мітки на стрілці ставиться стереотип «import» при публічному імпорті і стереотип «access» – при приватному імпорті.

Розглянемо приклад публічного імпорту пакетів (рис. 11.6). Пакет **П1** імпортує пакет **П2.** Класи **К1** і **К2** пакету **П1** можуть застосувати просте ім’я **К3** для звернення до публічного класу **К3** пакету **П2,** але приватний клас **К4** для них невидимий. Класи **К3** і **К4** пакету **П2** можуть використати кваліфіковане ім’я **П1::К1** для звернення до публічного класу **К1** пакету **П1,** а приватний клас **К4** з пакету **П2** невидимий.

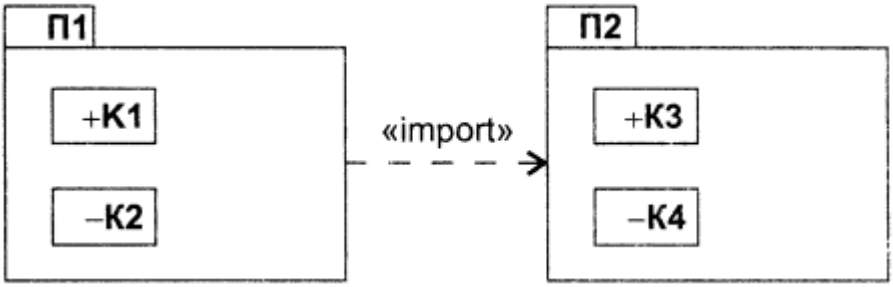


Рисунок 11.6 – Публічний імпорт пакетів

На рис. 11.7 представлені чотири рівні вкладеності.

На першому (зовнішньому) рівні знаходяться пакет **П1** і публічний клас **К6,** на другому – пакети **П2** і **П3,** на третьому рівні – пакети **П4, П5** і публічні класи **К4.. К5,** на четвертому (внутрішньому) рівні – публічні класи **К1, К2** і приватний клас **К3.** Пакет **П1** використовує публічне імпортування вкладеного пакету **П2** і пакету **П5,** вкладеного в пакет **П3.** Пакет **П2** застосовує приватне імпортування пакету **П3**.

Механізм імпорту може бути заданий як на рівні пакету, так і на рівні окремих елементів пакету.

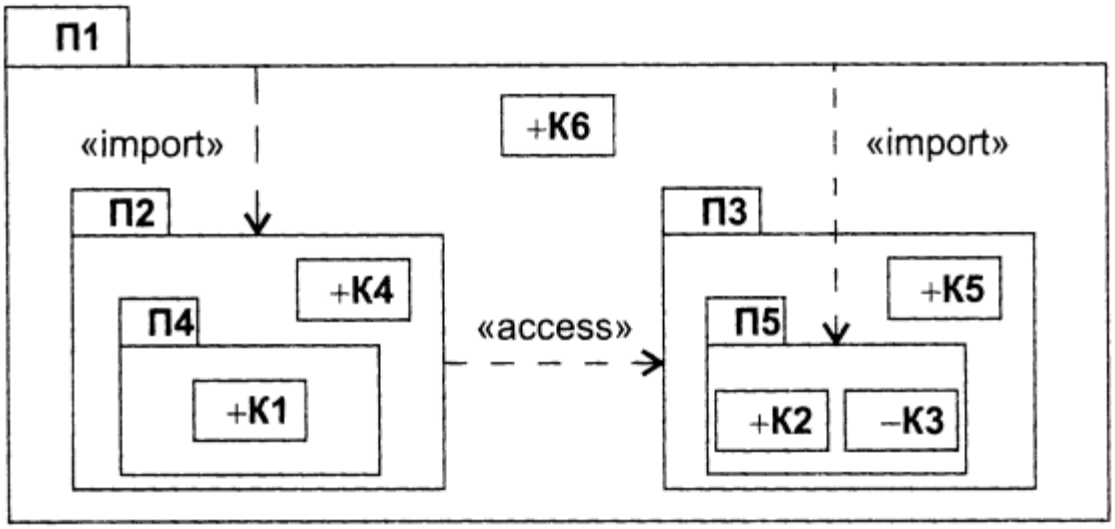


Рисунок 11.7 – Приватний імпорт пакетів

Наприклад, у випадку, показаному на рис. 11.8, в пакет **П1** імпортуються обидва класи **К3** і **К4** пакету **П2**.

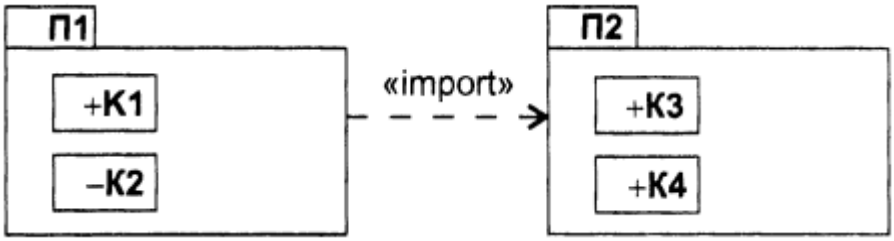


Рисунок 11.8 – Публічний імпорт на рівні пакету

У свою чергу, на рис. 11.9 представлений варіант, коли в пакет **П1** імпортується тільки один клас **К3** з пакету **П2.** У цьому варіанті до класу **К4** доведеться звертатися за кваліфікованим іменем **П2::К4**.

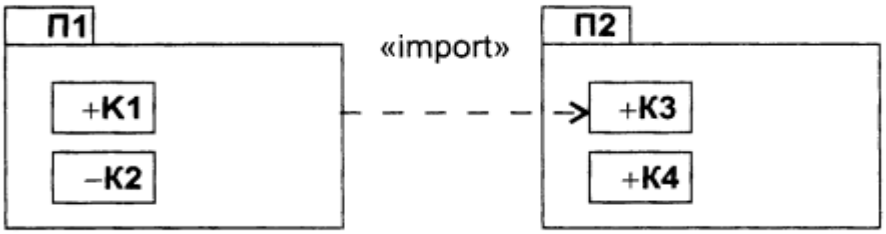


Рисунок 11.9 – Публічний імпорт на рівні елементу пакету

У контексті архітектурного проектування пакет є зручним засобом упаковки підсистеми, яка є частиною архітектурної організації, а діаграму пакетів – моделлю структури архітектурного рівня.

Діаграми пакетів зручні у великих за розмірами системах для представлення картини залежностей між основними елементами системи.

11.2.2. Діаграми компонентів

Ще одним будівельним блоком для створення архітектури об'єктно-орієнтованої системи вважається компонент. ***Діаграма компонентів*** показує визначення, внутрішню структуру і залежності набору компонентів. Фізичними компонентами можуть виступати файли, бібліотеки, модулі, виконувані файли, пакети тощо.

Діаграма компонентів, на відміну від раніше розглянутих діаграм, описує особливості фізичного представлення системи. Діаграма компонентів дозволяє визначити архітектуру системи, що розробляється, встановивши залежності між програмними компонентами, в ролі яких може виступати початковий, бінарний і виконуваний код. У багатьох середовищах розробки модуль або компонент відповідає файлу. Пунктирні стрілки, що сполучають модулі, показують відношення взаємозалежності, аналогічні тим, які мають місце при компіляції початкових текстів програм. Основними графічними елементами діаграми компонентів є компоненти, інтерфейси і залежності між ними.

На рис. 11.10 показаний приклад простої діаграми компонентів. В даному прикладі компонент **Till** (Каса) може взаємодіяти з компонентом **Sales Server** (Сервер продажів) за допомогою інтерфейсу **sales message** (Повідомлення про продажі). Оскільки мережа ненадійна, то компонент **Message Queue** (Черга повідомлень) встановлений так, щоб каса могла спілкуватися з сервером, коли мережа працює, і спілкуватися з чергою повідомлень, коли мережа відключена. Тоді черга повідомлень зможе спілкуватися з сервером, коли мережа знову стане доступною. В результаті черга повідомлень надає інтерфейс для спілкування з касою, і вимагає такий же інтерфейс для спілкування з сервером. Сервер розділений на два основні компоненти: **Transaction Processor** (Процесор транзакцій) реалізує інтерфейс повідомлень, а **Accounting Driver** (Драйвер рахунків) спілкується *з* **Accounting System** (Система ведення рахунків).

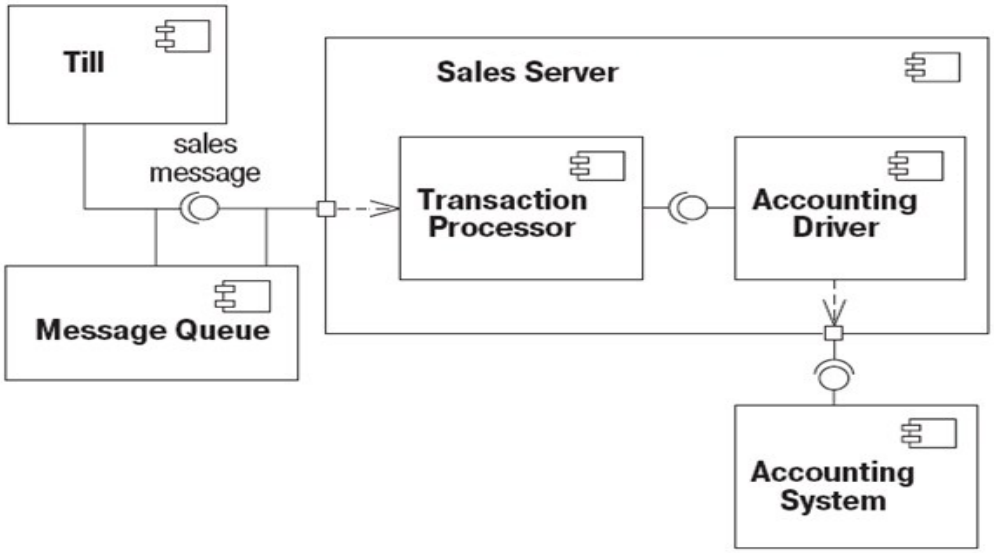


Рисунок 11.10 – Приклад діаграми компонентів

Компоненти. ***Компонент*** – модульна і замінювана частина системи, яка відповідає набору інтерфейсів і забезпечує реалізацію цього набору інтерфейсів. Реалізація компонента завжди прихована. Компоненти системи з однаковими наборами інтерфейсів є взаємозамінними.

Графічно компонент зображається як прямокутник із стереотипом «component». Замість вказівки стереотипу(чи на додаток до нього) можна помістити в правому верхньому куті прямокутника піктограму компонента. Ця піктограма має вигляд прямокутника, в одну із сторін якого врізані два менші прямокутники. Графічне зображення компонента веде своє походження від позначення модуля програми, що застосовувався для відображення особливостей інкапсуляції даних і процедур. Так, верхній маленький прямокутник концептуально асоціювався з даними, які реалізує цей компонент. Нижній маленький прямокутник асоціювався з операціями або методами, що реалізовуються компонентом.

Ім’я компонента вказується в зовнішньому прямокутнику (рис. 11.11). Компонент, подібно до класу або типу, є узагальненим описом, тому існують екземпляри компонента. Ім’я екземпляра компонента підкреслюється, формат його імені схожий на формат імені для об’єкту (**ім’я компонента: ім’я типу**).

Рисунок 11.11 – Нотація для компонентів

В якості власних імен компонентів прийнято використовувати імена виконуваних файлів (з розширенням EXE), імена динамічних бібліотек (DLL),

імена Web-сторінок (HTML), імена текстових файлів (TXT, DOC), файлів довідки (HLP), імена файлів баз даних (DB), імена файлів з початковими текстами програм (H, CPP, JAVA), скрипти (PL, ASP) та інші.

Оскільки компонент як елемент моделі може мати різну фізичну реалізацію, то іноді його зображують у формі спеціального графічного символу (стереотипу), що ілюструє конкретні особливості реалізації. Ці додаткові позначення не специфіковані в нотації мови UML. Проте вони спрощують розуміння діаграми компонентів, істотно підвищуючи її наочність (рис. 11.12).

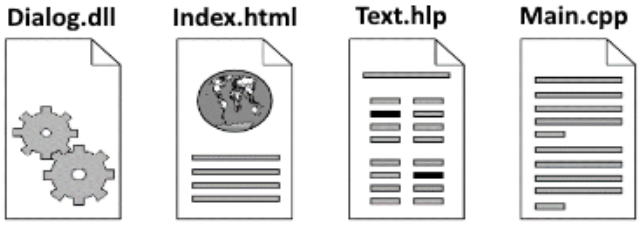


Рисунок 11.12 – Варіанти графічного зображення компонентів

Ці елементи іноді називають ***артефактами***, підкреслюючи при цьому їх закінчений інформаційний зміст, залежний від конкретної технології реалізації відповідних компонентів.

Інтерфейси. Інтерфейс – дуже важлива частина поняття «компонент». ***Інтерфейс*** – список операцій, які визначають послуги компонента (чи класу). Образно кажучи, інтерфейс – це роз'єм, який виступає із скриньки компонента. За допомогою інтерфейсних роз'ємів компоненти стикуються один з одним, об'єднуючись в систему.

Ще одна аналогія. Інтерфейс подібний до абстрактного класу, у якого відсутні атрибути і працюючі операції, а є тільки абстрактні операції (що не мають тіл). Усі операції інтерфейсу відкриті і видимі клієнтові (інакше вони втратили б всякий сенс). Отже, операції інтерфейсу тільки іменують послуги, не більше того. Інтерфейси можуть бути двох видів: забезпечені і потрібні.

Забезпечений інтерфейс описує послуги, виконання яких компонент пропонує (іншим компонентам). Кожній операції такого інтерфейсу повинен відповідати елемент реалізації всередині компонента. Іншими словами, забезпечений інтерфейс іменує набір послуг, що надаються компонентом (чи класом).

Необхідний інтерфейс описує послуги, які поставляються іншими компонентами. Необхідний інтерфейс декларує, що цьому компоненту (чи класу) для роботи вимагається прибігати до послуг іншого елементу, що надає цей інтерфейс.

У прямокутнику компонента можна показати секцію зі списком забезпечених інтерфейсів (розпочинається із стереотипу «provided») і списком необхідних інтерфейсів (розпочинається із стереотипу «required», рис. 11.13, а).

Найчастіше інтерфейси зображаються в формі піктограм, з'єднаних лініями з межею прямокутника компонента. Забезпечений інтерфейс малюється як коло, а необхідний інтерфейс – як півколо (рис. 11.13, б).

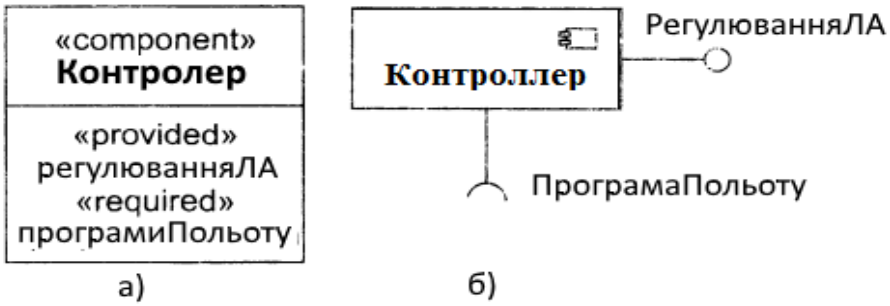


Рисунок 11.13 – Опис інтерфейсів в прямокутнику компонента

Розгорнутий спосіб представлення інтерфейсу ілюструє рис. 11.14. Тут інтерфейс зображається у вигляді прямокутника із стереотипом «interface». У прямокутнику інтерфейсу показуються його операції. Компонент, який реалізує забезпечений інтерфейс, підключається до нього відношенням реалізації. Компонент, який дістає доступ до послуг іншого компонента через необхідний інтерфейс, підключається до інтерфейсу відношенням залежності із стереотипом «use».

Рисунок 11.14 – Розгорнута форма представлення інтерфейсу

У одного компонента може бути декілька потрібних і декілька забезпечених інтерфейсів. Той факт, що між двома компонентами завжди знаходиться інтерфейс, усуває їх пряму залежність. Компонент, що використовує забезпечений інтерфейс, функціонуватиме правильно незалежно від того, який компонент реалізує цей інтерфейс.

Приклад діаграми компонентів. Діаграма компонентів показує компоненти в системі, тобто програмні підсистеми, з яких створюється програмна архітектура, а також залежності між компонентами. На рис. 11.15, а наведена діаграма компонентів для системи продажу квитків у філармонію [38].

Як бачимо, можна купити один квиток на концерт, зробити групову купівлю, придбати абонемент на увесь сезон. Купівлю можна виконати в інтернет-касі філармонії або в її офісі. Правда, групова купівля здійснюється тільки в офісі. Оплата робиться за допомогою кредитної карти. Пунктирними лініями тут показані стрілки залежності необхідних інтерфейсів від забезпечених інтерфейсів.

Другий вид діаграми у формі спеціальних графічних символів (стереотипів) приведений на рис. 11.15, б.

Діаграми компонентів слід застосовувати, коли система розділяється на компоненти і потрібно показати їх взаємовідносини за допомогою інтерфейсів.

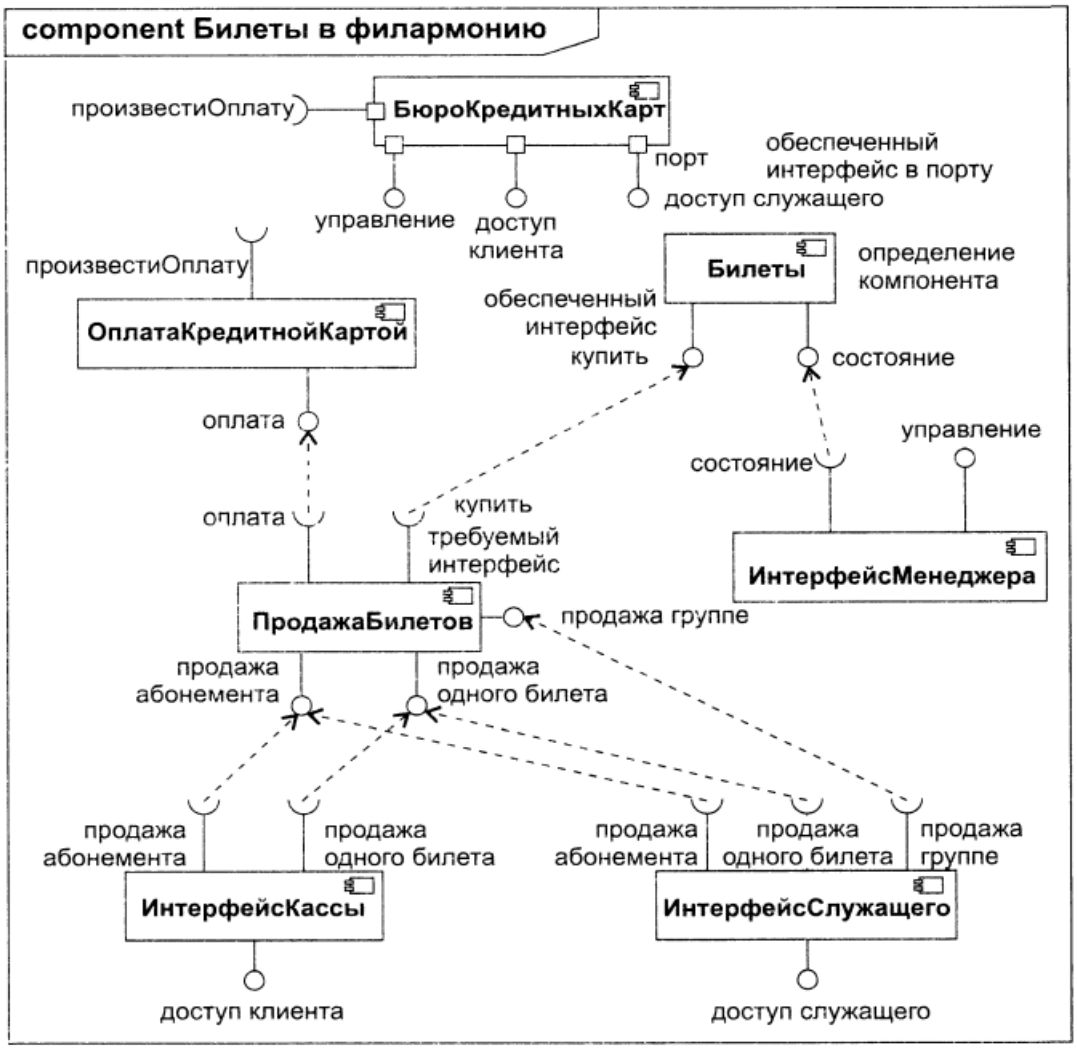


Рисунок 11.15, а – Діаграма компонентів

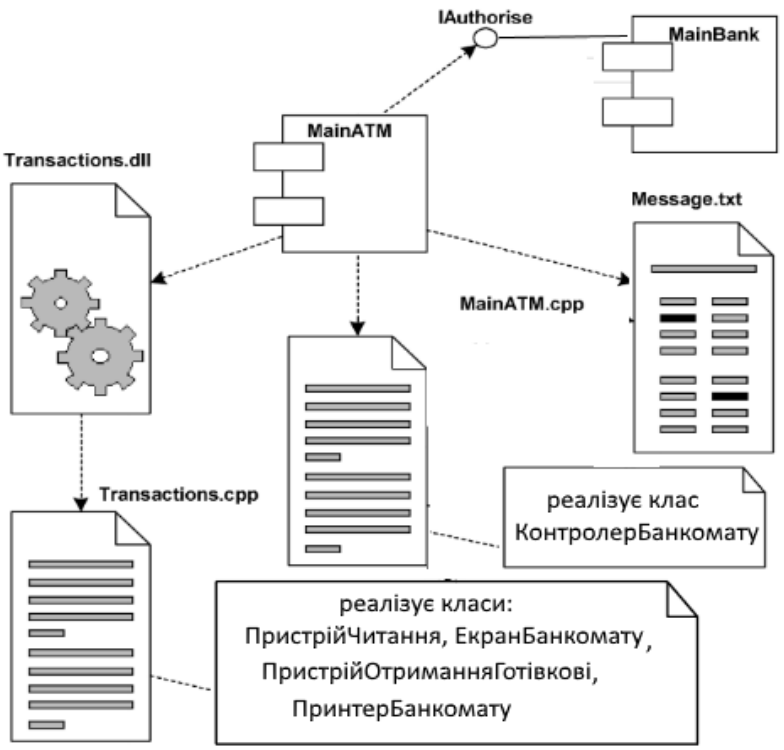


Рисунок 11.15, б – Діаграма компоентів системи управління банкоматом

11.3. Детальне проектування

Детальне проектування – це другий ступінь проектування, який іде за створенням архітектури. У ході цієї діяльності орієнтуються на максимальну підготовку до кодування програмної системи. Програмісти повинні отримати детальні проектні рішення, які забезпечать їх повною інформацією для створення програмного коду.

Нагадаємо взаємозв'язки між елементами всього попереднього ланцюжка розробки: **формування вимог – аналіз вимог – архітектурне проектування**.

При формуванні вимог створюються елементи Use Case, що документують побажання замовника, ці побажання деталізуються і формалізуються на етапі аналізу, перетворюючись на діаграми послідовності. На етапі аналізу вже доводиться спиратися на архітектурні риси майбутньої системи. Можна сказати, що паралельно з аналізом починається архітектурне проектування.

Архітектура і детальні вимоги живлять фазу деталізації проектування. Архітектурний скелет обростає деталями – класами, здатними реалізувати сценарії, описані діаграмами послідовності. Завершується детальне проектування у момент отримання повного плану для етапу програмування.

Як відомо класи можуть застосовуватися на стадії аналізу для складання глосарію системи або аналізу предметної області. Але надалі клас аналізу має бути уточнений в один або більше проектних класів, оскільки клас аналізу описаний на дуже високому рівні абстракції. Тут немає повного набору атрибутів, а набір операцій – це фактично тільки ескіз, що відбиває ключові сервіси, пропоновані класом.

При переході до проектування усі операції і атрибути класу мають бути повністю описані, тому нерідко він стає занадто великим. Якщо це відбувається, необхідно розбити його два або більше менших класів. Пам'ятайте, що завжди потрібно прагнути проектувати невеликі класи, що є самодостатніми, зв'язними елементами, які добре справляються з однією-двома функціями. За всяку ціну необхідно уникати великих класів, які роблять усе.

Вибраний метод реалізації визначає необхідну міру повноти описів проектних класів. Якщо планується передати модель проектного класу програмістам в якості керівництва для написання коду, проектні класи мають бути повними лише настільки, щоб забезпечити можливість ефективного виконання завдання.

Проте, якщо передбачається використати проектні класи для генерації коду за допомогою відповідним чином оснащеного інструментального засобу моделювання, їх описи мають бути повними в усіх відношеннях. Генератор коду, на відміну від програміста, не може заповнювати пропуски. Для цього треба зробити таке:

- закінчити набір атрибутів і повністю описати їх, включаючи ім'я, тип, видимість і (необов'язково) вживане за умовчанням значення;
- закінчити набір операцій і повністю описати їх, включаючи ім'я, список параметрів і повертаний тип.

Отже, **основним будівельним блоком детального проектування є класи.** Створення структури класів в мові UML підтримується діаграмою класів, призначеною для *статичного моделювання об'єктів*. Перейдемо до її розгляду.

11.3.1. Діаграми класів

Центральне місце в ООАП займає розробка логічної моделі системи у вигляді діаграми класів. Нотація UML надає широкі можливості для відображення додаткової інформації (абстрактні операції і класи, стереотипи, загальні і приватні методи, деталізовані інтерфейси, класи, що параметризуються). При цьому можливе використання графічних зображень для асоціацій і їх специфічних властивостей, таких як відношення агрегації, коли складовими частинами класу можуть виступати інші класи.

Діаграма класів описує типи об'єктів системи і різного роду статичні відношення, які існують між ними. На діаграмах класів відображаються також властивості класів, операції класів і обмеження, які накладаються на зв'язки між об'єктами. У UML термін **функціональність** застосовується в якості основного терміну, що описує і властивості, і операції класу.

Діаграми класів вважають основним засобом для представлення структури систем в термінах базових будівельних блоків і відношень між ними. До інших структурних діаграм належатья діаграми компонентів і розгортання.

Моделі класів визначають структури, що відбивають внутрішній стан системи. Вони ідентифікують класи і їх атрибути, включаючи відношення. Крім того, вони визначають операції, необхідні для реалізації вимог динамічної поведінки системи, зафіксованих в прецедентах використання. Класи, що реалізовані конкретною мовою програмування, відбивають статичну структуру і динамічну поведінку додатка.

Результатом моделювання класів є діаграма класів і пов'язана з нею текстова документація. У цьому розділі моделювання класів обговорюється після моделювання прецедентів використання, але на практиці ці види діяльності, як правило, виконуються паралельно. Ці дві моделі доповнюють одна одну, обмінюючись додатковою інформацією. Прецеденти використання полегшують виявлення класів, а моделі класів, навпаки, призводять до ідентифікації і перегляду прецедентів використання.

Як уже згадувалося, одну і ту ж діаграму UML можна використати в декількох ракурсах залежно від ступеня її деталізації. У концептуальному ракурсі діаграми класів можна застосовувати для візуалізації моделі предметної області на етапі аналізу. Надалі виявиться корисним застосовувати її для однозначного опису ситуації, коли діаграми класів використовуються в ракурсі проектування або на етапі реалізації (програмування).

На рис. 11.16 зображена типова діаграма класів, зрозуміла кожному, хто мав справу з обробкою замовлень клієнтів. Прямокутники на діаграмі представляють класи і розділені на три частини: ім'я класу (жирний шрифт),

його атрибути і його операції. На рис. 11.16 також показані два види зв'язків між класами: асоціації і узагальнення.

Вершини діаграм класів навантажені класами, а дуги (ребра) – відношеннями між ними. Діаграми використовуються:

- в ході аналізу – для вказівки ролей і обов'язків сутностей, які забезпечують поведінку системи;
- в ході проектування – для фіксації структури класів, які деталізують системну архітектуру.

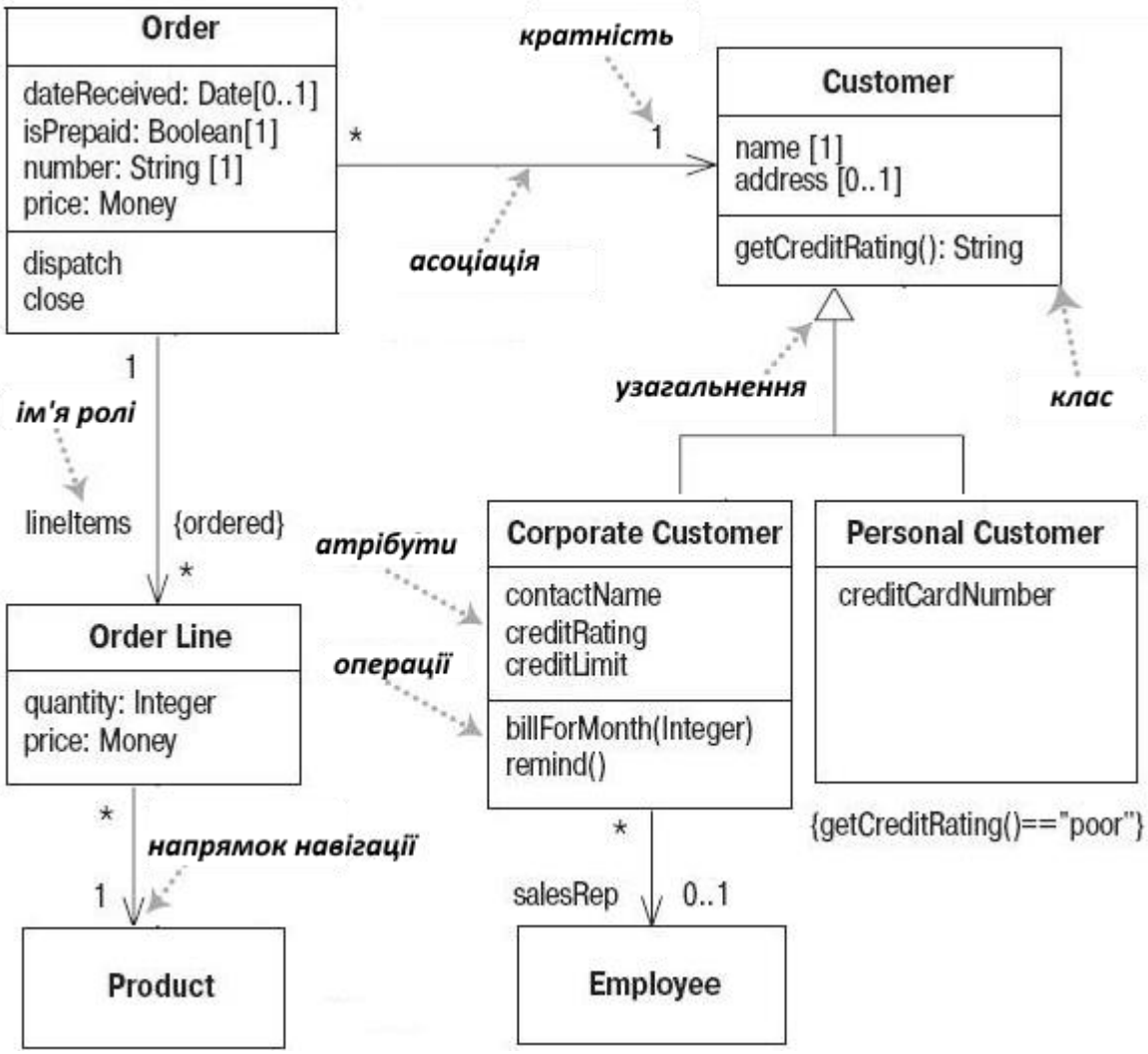


Рисунок 11.16 – Проста діаграма класів

Вершини в діаграмах класів. Вершина в діаграмі класів – це клас. Позначення класу показане на рис. 11.17. Ім'я класу вказується завжди, атрибути і операції – вибірково. Передбачено завдання зони дії атрибуту (операції). Якщо атрибут (операція) підкреслюється – його зоною дії є клас, інакше – зоною дії є екземпляр (рис. 11.18).

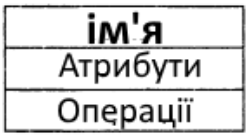


Рисунок 11.17– Позначення класу

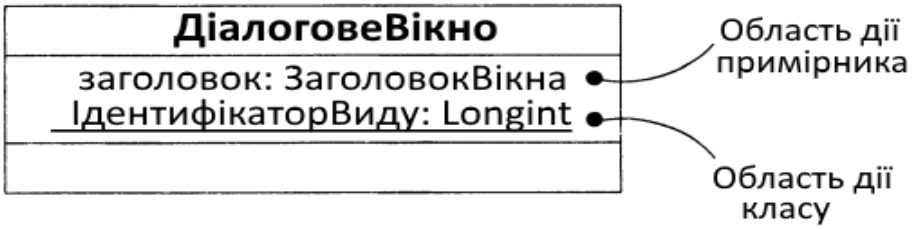


Рисунок 11. 18 – Атрибути рівнів класу і екземпляра

Що це означає? Якщо зоною дії атрибуту є клас, то усі його екземпляри (об'єкти) використовують загальне значення цього атрибуту, інакше – у кожного екземпляра своє значення атрибуту.

Атрибути. Атрибут описує властивість у вигляді рядка тексту усередині прямокутника класу. Загальний синтаксис представлення атрибуту має вигляд:
Видимість Ім'я: Тип [Множинність] = НачальнеЗначенияення {Властивості}

Розглянемо видимість і властивості атрибутів.

У мові UML визначені чотири рівні видимості.

public	Будь-який клієнт класу може використати атрибут (операцію), позначається символом +
package	Будь-який клієнт класу, оголошений в тому ж пакеті, може використати атрибут (операцію), позначається символом -
protected	Будь-який спадкоємець класу може використати атрибут (операцію), позначається символом #
private	Атрибут (операція) може використовуватися тільки самим класом, позначається символом -

Якщо видимість не вказана, вважається, що атрибут оголошений з публічною видимістю.

Приклади оголошення атрибутів.

початок	Тільки ім'я
+ початок	Видимість і ім'я
початок: Координати	Ім'я і тип
имяФамилня; String [0.1]	Ім'я, тип, множинність
левыйУгол: Координати=(0, 10)	Ім'я, тип, початкове значення
сума: Integer [3]{readOnly. ordered}	Ім'я, тип, множинність і властивості

Операції. Загальний синтаксис представлення операції має вигляд:
Видимість Ім'я (Список Параметрів): ВозвращаемыйТип {Властивості}

Приклади оголошення операцій.

записати	Тільки ім’я
+ записати	Видимість і ім’я
зареєстрований(і: Ім’я. ф: Прізвище)	Ім’я і параметри
балансСчета ( ): Integer	Ім’я і повертаний тип
нагрівати (){guarded}	Ім’я і властивість

У сигнатурі операції можна вказати нуль або більше параметрів, форма представлення параметра має такий синтаксис:

Напрям Ім’я: Тип = ЗначениеПоУмолчанию

Елемент **Напрям** може приймати одне з наступних значень.

in	Вхідний параметр, не може модифікуватися
ont	Вихідний параметр, може модифікуватися для передачі інформації в об’єкт, що викликає
inout	Вхідний параметр, може модифікуватися

Множинність. Іноді буває необхідно обмежити кількість екземплярів класу:

- задати нуль екземплярів (в цьому випадку клас перетворюється на утиліту, яка пропонує свої атрибути і операції);
- задати один екземпляр (клас-singleton);
- задати конкретну кількість екземплярів;
- не обмежувати кількість екземплярів (за замовчуванням).

Кількість екземплярів класу називається його ***множинністю.*** Вираження множинності записується в верхньому куті значка класу. Наприклад, як показано на рис. 11.19, **КонтролерКутів –** це клас-singleton, а для класу **ДатчикКута** дозволені три екземпляри.

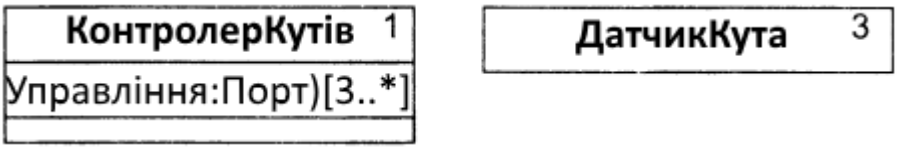


Рисунок 11.19 – Множинність

Множинність застосовна не лише до класів, але і до атрибутів. Множинність атрибуту задається виразом в квадратних дужках, записаним після його типу. Наприклад, на рисунку задані три і більше екземпляри атрибуту управління (у екземплярі класу **КонтролерКутів**).

Відношення в діаграмах класів. Відношення, використовувані в діаграмах класів, показані на рис. 11.20.

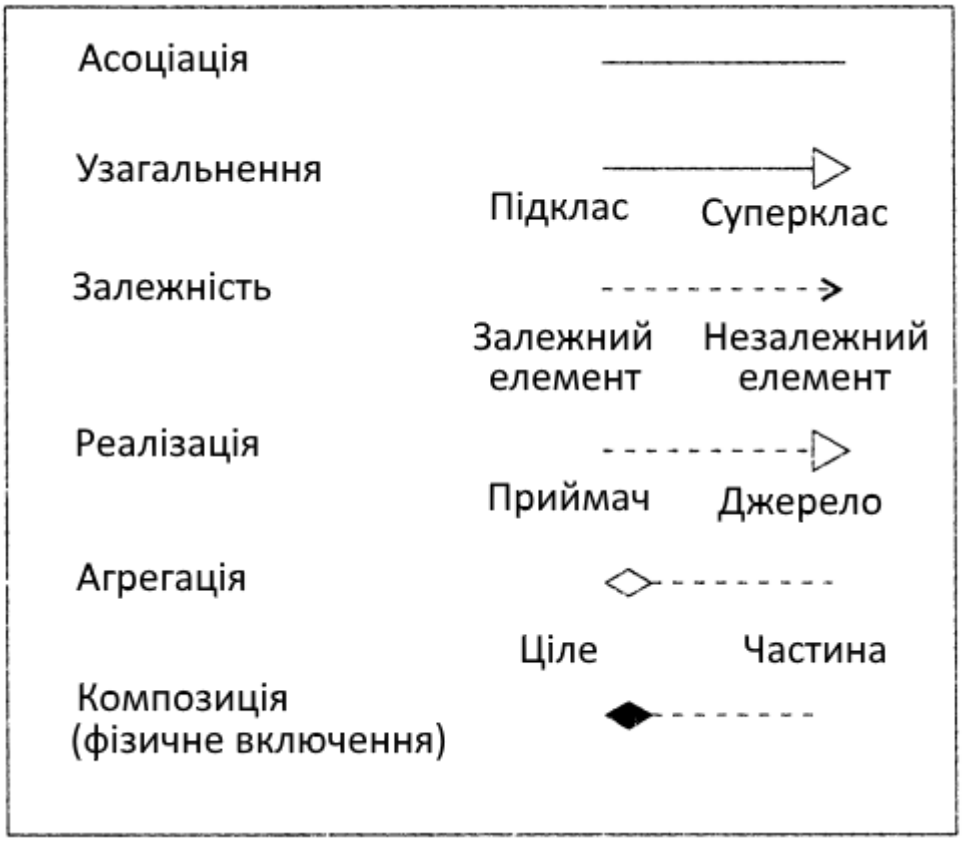


Рисунок 11.20 – Відношення в діаграмах класів

Асоціація. ***Асоціації*** відображають структурні відношення між екземплярами класів, тобто з'єднання між об'єктами. Кожна асоціація може мати мітку – ім'я, яке описує природу відношення.

Як показано на рис. 11.21, імені можна надати напрям – досить додати трикутник напрямку, який вказує напрямок, заданий для читання імені.

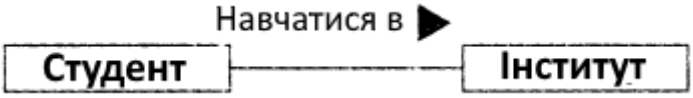
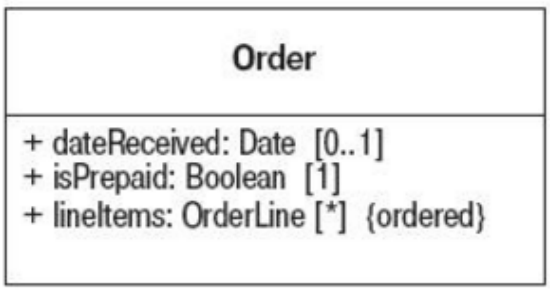


Рисунок 11.21 – Імена асоціацій

Ім'я властивості (разом з кратністю) розташовується на цільовому кінці асоціації. Цільовий кінець асоціації вказує на клас, який є типом властивості (рис. 11.22). Зокрема, асоціація може показувати кратність на обох кінцях лінії.

Коли клас бере участь в асоціації, він грає в цьому відношенні певну роль. Як показано на рис. 11.23, *роль асоціації* визначає, яким видається клас на одному полюсі асоціації для класу на протилежному полюсі асоціації. Один і той же клас в різних асоціаціях може грати різні ролі.



Подання властивостей замовлення у вигляді атрибутів

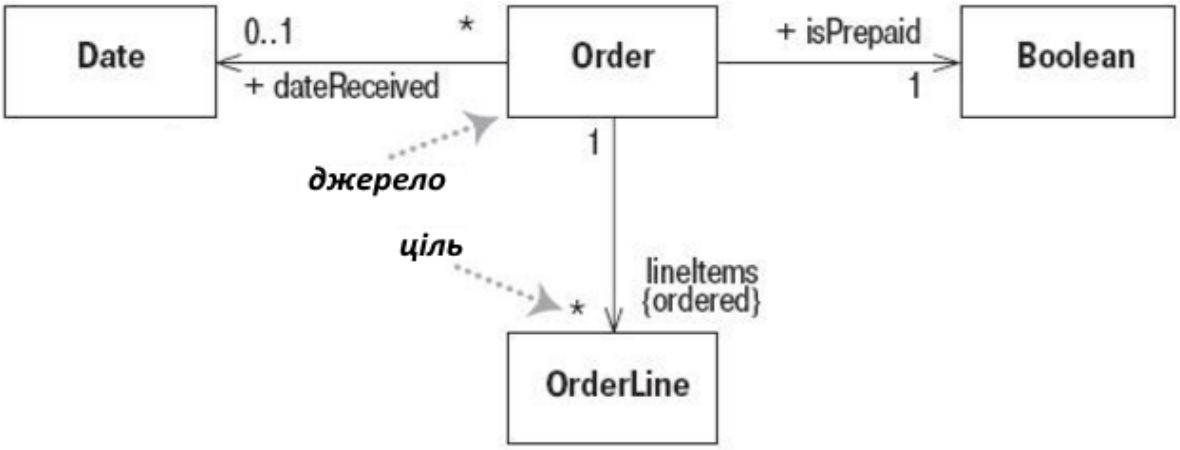


Рисунок 11.22 – Подання властивостей замовлення у вигляді асоціації

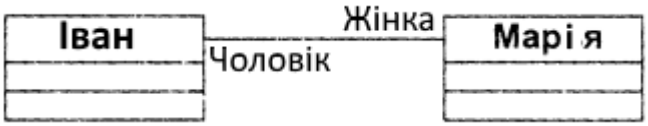


Рисунок 11.23– Ролі асоціації

Потужність. Часто важливо знати, як багато об’єктів може з’єднуватися через екземпляр асоціації. Ця кількість називається ***потужністю (кратністю***) ролі в асоціації, записується у вигляді виразу, що задає діапазон величин або одну величину (рис. 11.24). Запис потужності на одному полюсі асоціації визначає кількість об’єктів, що сполучаються з кожним об’єктом на протилежному полюсі асоціації. Наприклад, можна задати такі варіанти потужності:

- 5 – точно п’ять;
- * – необмежена кількість;
- 0..* – нуль або більше;
- 1 ..* – один або більше;
- 3..7 – певний діапазон:
- 1..3,7 – певний діапазон або число.

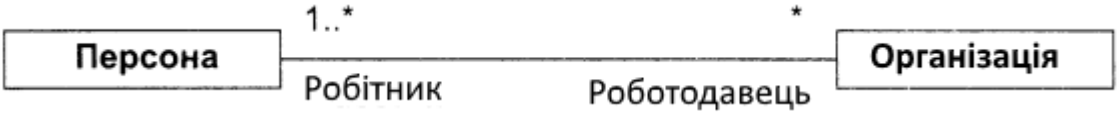


Рисунок 11.24 – Потужність

Досить часто виникає така проблема: як для об'єкту на одному полюсі асоціації виділити набір об'єктів на протилежному полюсі? Наприклад, розглянемо взаємодію між банком і клієнтом – вкладником. Як показано на рис. 11.25, ми встановлюємо асоціацію між класом **Банк** і класом **Клієнт.** У контексті **Банку** ми маємо **номерСчета,** який дозволяє ідентифікувати конкретного **Клієнта.** У цьому сенсі **НомерСчета** є атрибутом асоціації. Він не є характеристикою **Клієнта,** оскільки **Клієнтові** не обов'язково знати службові параметри його рахунку. Тепер для цього екземпляра **Банку** і цього значення **номераСчета** можна виявити нуль або один екземпляр **Клієнта**. У UML для вирішення цієї проблеми вводиться кваліфікатор – атрибут асоціації, чиї значення виділяють набір об'єктів, пов'язаних з об'єктом через асоціацію. Кваліфікатор зображається маленьким прямокутником, приєднаним до кінця асоціації. У прямокутник вписується атрибут – атрибут асоціації.

Рисунок 11.25 – Кваліфікація

Крім того, ролі в асоціаціях можуть мати позначки видимості. Наприклад, на рис. 11.26 показані асоціації між **Начальником** і **Жінкою,** а також між **Жінкою** і **Загадкою.** Для цього екземпляру **Начальника** можна визначити відповідні екземпляри **Жінки.** З іншого боку, **Загадка** приватна для **Жінки,** тому вона не доступна ззовні. Як показано па рисунку, з об'єкту **Начальника** можна переміщатися до екземплярів **Жінки** (і навпаки), але не можна бачити екземпляри Загадки для об'єктів **Жінки**.

Рисунок 11.26 – Видимість в асоціації

У мові UML асоціації можуть мати атрибути. Як показано на рис. 11.27, такі можливості відображаються за допомогою класів-асоціацій. Ці класи

приєднуються до лінії асоціації пунктирною лінією і розглядаються як класи з атрибутами асоціацій, або як асоціації з атрибутами класів.

Атрибути класу-асоціації характеризують не один, а пару об'єктів, в даному випадку – пару екземплярів **Професора** і **Університету**.

Рисунок 11.27 – Клас-асоціація

Відношення агрегації і композиції в мові UML вважаються різновидами асоціації, що вживаються для відображення структурних відношень між «цілим» (агрегатом) і його «частинами». *Агрегація* показує відношення за посиланням (у агрегат включені тільки покажчики на частини), *композиція* – відношення фізичного включення (у агрегат включені самі частини).

Узагальнення. ***Узагальнення*** (спадкоємство) – відношення між загальним предметом (суперкласом) і спеціалізованим різновидом цього предмета (підкласом). Підклас може мати одного батька (один суперклас) або декілька батьків (декілька суперкласів). У другому випадку говорять про множинне спадкоємство. Як показано на рис. 11.28, підклас ***Літаюча*** **шафа** є спадкоємцем суперкласів **Літаючий предмет** і **Сховище речей.** Цьому підкласу дістаються в спадок усі атрибути і операції двох класів-батьків.

Рисунок 11.28 – Множинне спадкоємство

Складніші проблеми виникають при спадкоємстві від двох класів, що мають загального батька. Говорять, що в результаті утворюється ромбовидна гратка спадкоємства (рис. 11.29). Вважаємо, що в підкласах **Офіціант** і **Співак** операція **работать()** суперкласу **Працівник** перевизначена відповідно до обов'язку підкласу (робота офіціанта полягає в обслуговуванні їжею, а співака – в співі). Виникає питання: яку версію операції **працювати()** успадкує **Співаючий_офыцыант**?

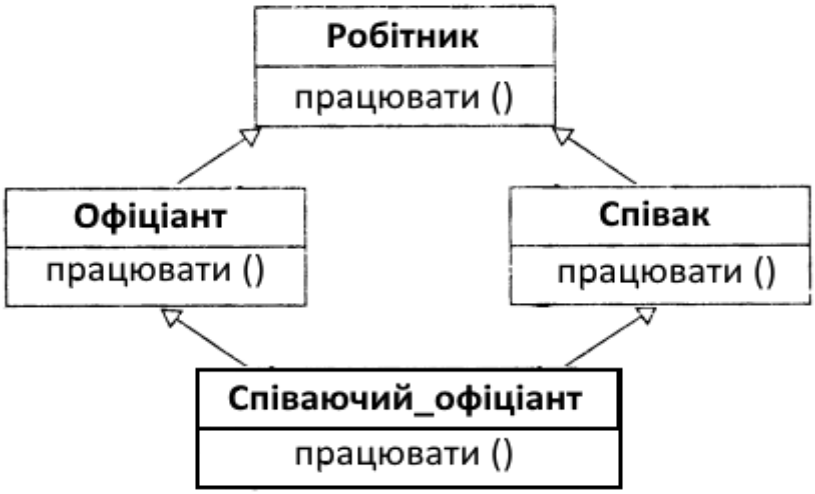


Рисунок 11.29 – Ромбовидні гратки спадкоємства

Залежність є відношенням використання між клієнтом (залежним елементом) і постачальником (незалежним елементом).

Наприклад, на рис. 11.30 показана залежність класу **Замовлення** від класу **Книга,** оскільки **Книга** використовується в операціях **проверкаДоступности(), додати()** і **видалити()** класу **Замовлення**.

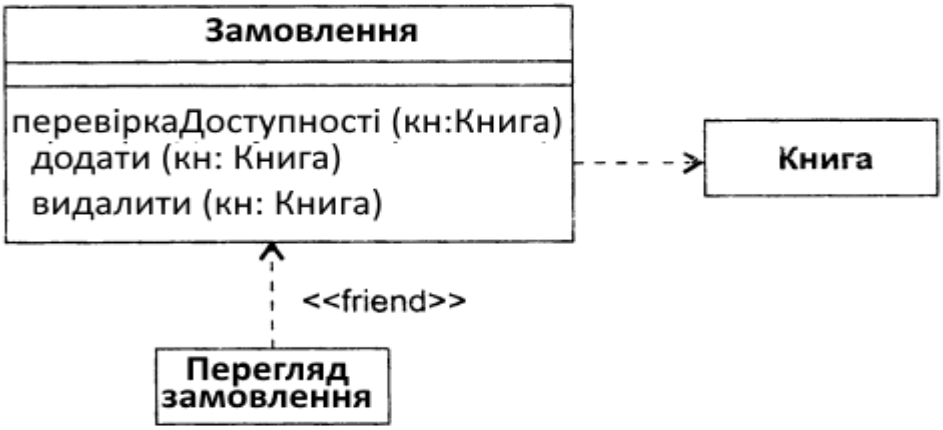


Рисунок 11.30 – Відношення залежності

На цьому рисунку зображена ще одна залежність, яка показує, що клас **Перегляд Замовлення** використовує клас **Замовлення.** Причому **Замовлення** нічого не знає про **Перегляд Замовлення**. Ця залежність помічена стереотипом «friend», який розширює просту залежність, визначену в мові.

Реалізація – відношення між двома елементами моделі, в якому один елемент ***(****клієнт)* реалізує поведінку, задану іншим *(постачальником*). Реалізація – відношення ціле-частина. Графічно реалізація представляється так само, як і спадкоємство, але з пунктирною лінією.

Агрегація – проста асоціація між двома класами, яка відбиває структурне відношення між рівноправними сутностями. Але іноді доводиться моделювати відношення типу «частина/ціле», в якому один з класів має вищий ранг (ціле) і складається з декількох менших за рангом (частин). Агрегація є частковим випадком асоціації і зображається у вигляді простої асоціації з незафарбованим ромбом з боку «цілого».

Композиція – строгіший варіант агрегації. Відома також як агрегація за значенням. Композиція має жорстку залежність часу існування екземплярів класу контейнера і екземплярів класів, що містяться. Якщо контейнер буде знищений, то увесь його вміст буде також знищений. Графічно представляється як і агрегація, але із зафарбованим ромбом.

11.3.2. Приклади діаграм класів

В якості першого прикладу на рис. 11.31 показана діаграма класів системи управління польотом літального апарату.

Тут представлений клас **ПрограммаПолета,** який має атрибут **траекторияПолета,** операцію-модифікатор **выполнятьПрограмму()** і операцію-селектор **прогнозОкончУправления().** Є асоціація між цим класом і класом **Контроллер СУ** – екземпляри програми задають параметри рухи, які повинні забезпечувати екземпляри контроллера.

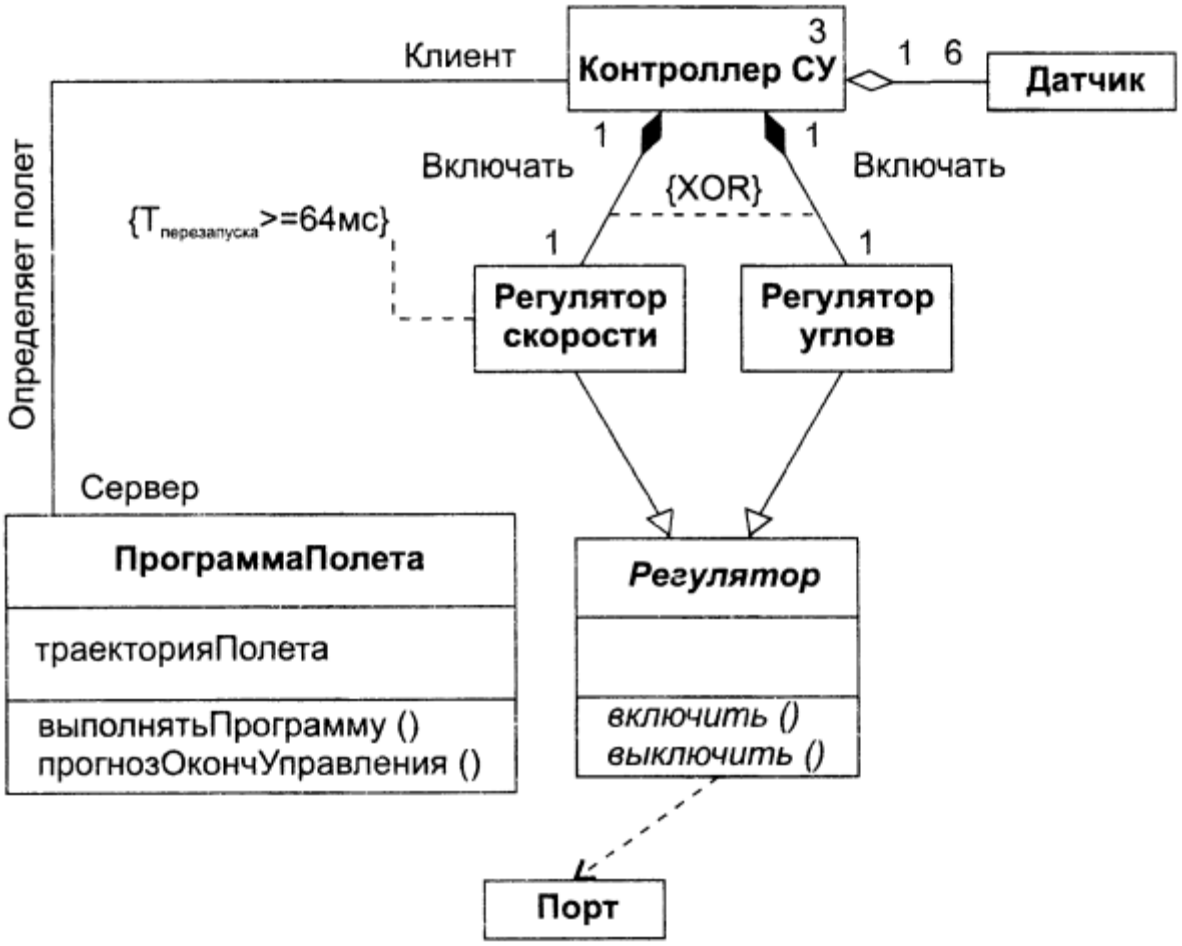


Рисунок 11.31 – Діаграма класів системи управління польотом

Клас **Контроллер СУ –** агрегат, чиї екземпляри включають по одному екземпляру класів **Регулятор швидкості** і **Регулятор кутів,** а також по шість екземплярів класу Датчик. Екземпляри Регулятора швидкості і Регулятора кутів включені в агрегат фізично (за допомогою відношення *композиція),* а екземпляри **Датчика –** за посиланням, тобто екземпляр **Контроллера СУ** включає лише покажчики на об’єкти-датчики. **Регулятор** швидкості і

Регулятор кутів – це підкласи абстрактного суперкласу ***Регулятор,*** який передає їм в спадок абстрактні операції включити() і вимкнути(). У свою чергу, клас ***Регулятор*** використовує конкретний клас **Порт**.

Як бачимо, асоціація має ім'я (**Визначає політ),** ролі учасників асоціації явно вказані **(Сервер, Клієнт).** Відношення композиції також мають імена **(Включати),** причому на ці відношення накладено обмеження – контроллер не може включати **Регулятор швидкості** і **Регулятор кутів** одночасно.

Для класу **Контроллер СУ** задано обмеження на множинність – допускається не більше трьох екземплярів цього класу. **Клас Регулятор** швидкості має обмеження іншого типу – повторне включення його екземпляра дозволяється не раніше, ніж через 64 мс.

В якості другого прикладу на рис. 11.32 наведена діаграма класів для інформаційної системи театру. Цю систему утворює 6 класів.

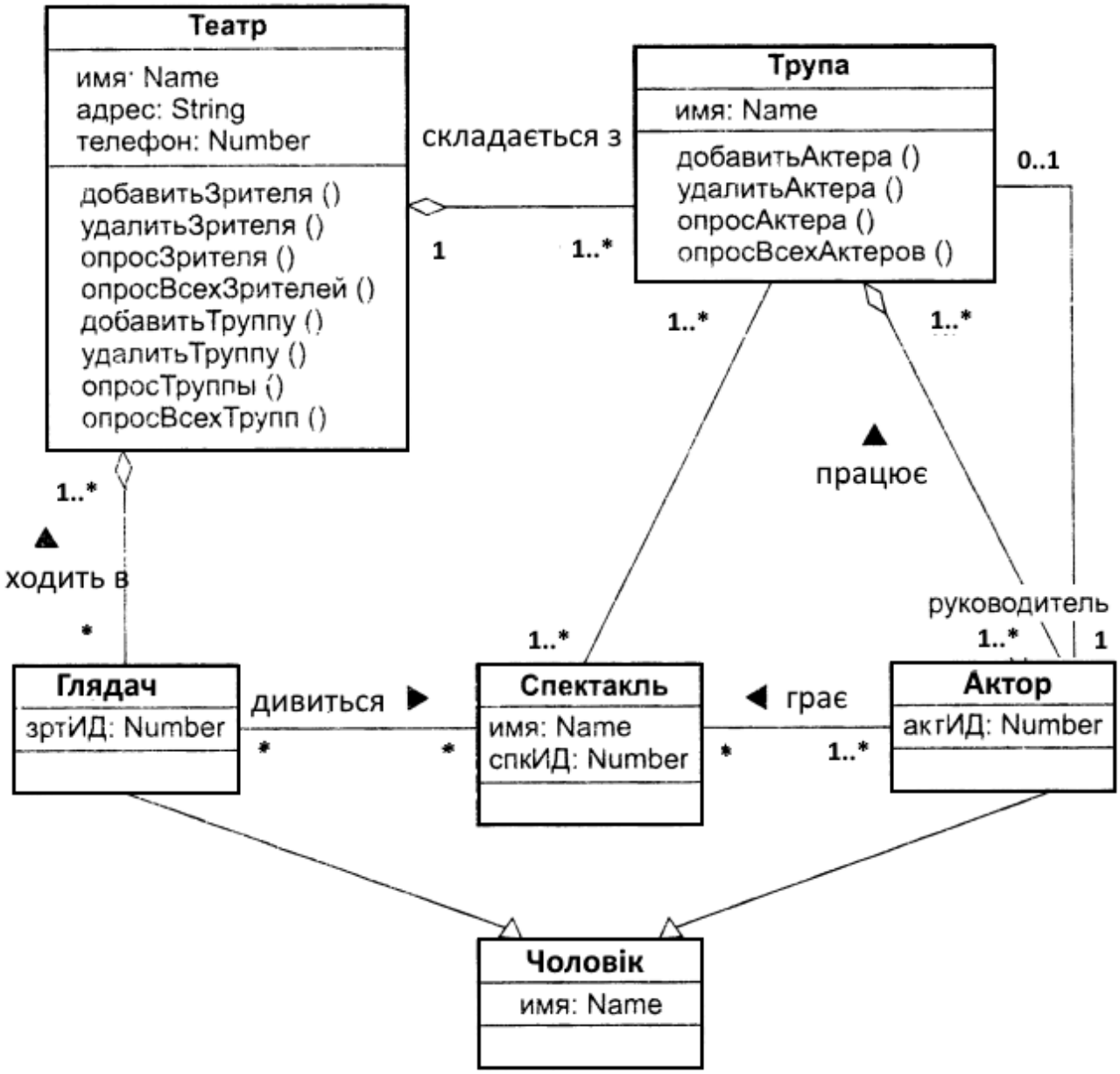


Рисунок 11.32 – Діаграма класів інформаційної системи театру

Класи-агрегати **Театр** і **Трупа** мають операції додавання і видалення своїх частин, які включаються в агрегати по посиланню. **Частинами Театру** є **Глядачі** і **Трупи,** а частинами **Трупи – Актори.** Відношення агрегації між класом **Театр** н класами **Трупа** і **Глядач** злегка відмінні. Театр може складатися з однієї або декількох труп, але кожна трупа знаходиться в одному і тільки одному театрі. З іншого боку, в театр може ходити будь-яка кількість глядачів (включаючи нульову кількість), причому глядач може відвідувати один або декілька театрів.

Між класами **Трупа** і **Актор** існує два відношення – агрегація і асоціація. Агрегація показує, що кожен актор працює в одній або декількох трупах, а в кожній трупі має бути хоч би один актор. Асоціація відображає, що кожною трупою управляє тільки один актор – художній керівник, а деякі актори не є керівниками. Асоціація між класами **Спектакль** і **Актор** фіксує, що в спектаклі має бути зайнятий хоч би один актор, втім, актор може грати у будь-якій кількості спектаклів (чи взагалі може нічого не грати).

Між класами **Спектакль** і **Глядач** теж визначена асоціація. Вона пояснює, що глядач може дивитися будь-яке число спектаклів, а на кожному спектаклі може бути будь-яке число глядачів.

І нарешті, на діаграмі відображено два відношення спадкоємства, що стверджує, що і в глядачах, і в акторах є людський початок.

11.3.3. Створення початкової діаграми класів

Робота із створення початкової діаграми класів вимагає вивчення змісту усіх ***діаграм послідовності***, що є результатом етапу аналізу. Проводиться вона в три етапи.

На першому етані виявляються і іменуються класи. Для цього видима кожна діаграма послідовності. Будь-яка роль в цій діаграмі повинна належати конкретному класу, для якого потрібно придумати ім’я. Наприклад, резонно припустити, що ролі **контроллер** повинен відповідати клас **Controller,** тому клас **Controller** слід ввести в діаграму. Звичайно, якщо в іншій діаграмі послідовності знову з’явиться подібна роль, то додатковий клас не утворюється.

На другому етані виявляються операції класів. На діаграмі послідовності така операція відповідає стрілці (і імені) повідомлення, що вказує па лінію життя учасника взаємодії. Наприклад, якщо до лінії життя ролі **контроллер** підходить стрільця повідомлення регулювати, то в клас **Controller** треба ввести операції **регулювати()**.

На третьому етапі визначаються відношення асоціації між класами – вони забезпечують пересилки повідомлень між відповідними екземплярами. Ці відношення просто відтворюють стрілки передачі повідомлень на діаграмах послідовності.

Звичайно, початкова діаграма класів далека від досконалості, але вона має безперечну гідність: реалізує набір детальних вимог, заданих діаграмами послідовності.

Подальші кроки проектувальника спрямовані на визначення атрибутів класів і оптимізацію усієї діаграми класів. Наприклад, в класах шукаються загальні риси (атрибути і операції). Якщо вони знайдуться, то їх виділяють в додатковий суперклас, який зв’язується із спорідненими класами відношеннями спадкоємства. Досліджується також можливість підлеглості класів, яка призводить до появи відношень агрегації.

На закінчення хотілося б ще раз нагадати, що діаграми класів складають фундамент UML, і тому їх постійне застосування є умовою досягнення успіху.

Основна трудність, пов’язана з діаграмами класів, полягає в тому, що вони настільки великі, що їх застосування може виявитися непомірно складним. Наведемо декілька корисних порад.

1. Не намагайтеся задіяти відразу усі доступні поняття. Розпочніть з найпростіших, описаних в цьому розділі: класів, асоціацій, атрибутів, узагальнень і обмежень. Звертайтеся до додаткових понять тільки якщо вони дійсно потрібні.
2. Не потрібно будувати моделі для усього на світі, замість цього слід сконцентруватися на ключових аспектах. Краще створити мало діаграм, які постійно застосовуються в роботі і відбивають усі внесені зміни, чим мати справу з великою кількістю забутих і застарілих моделей.
3. Найбільша небезпека, пов’язана з діаграмами класів, полягає в тому, що ви можете зосередитися виключно на структурі і забути про поведінку. Тому, малюючи діаграми класів для того, щоб розібратися в програмному забезпеченні, використайте які-небудь форми аналізу поведінки.

Нагадаємо, що варіанти використання застосовувалися для представлення вимог на стадії аналізу, і тому їх можна притягати для визначення ключових успадковних класів додатка. Тепер також можливо використати діаграми послідовності, побудовані за варіантами використання, і надавати відповідні класи з необхідними для виконання послідовностей методами.

Оскільки усі методи, необхідні для реалізації цього варіанту використання, тепер відомі, ми можемо представити їх на об’єктній моделі. Продовжуючи цей процес, ми отримаємо детальні модель класів і модель варіантів використання (у вигляді діаграм послідовності). Можна також отримати детальну модель переходів станів (якщо вона застосовна).

11.4. Розгортання програмної системи на апаратних засобах

Фізичне представлення програмної системи не може бути повним, якщо відсутня інформація про те, на якій платформі і на яких обчислювальних засобах вона реалізована. Якщо створюється проста програма, яка може виконуватися локально на комп'ютері користувача, не використовуючи ніяких розподілених пристроїв і мережевих ресурсів, то необхідності в розробці додаткових діаграм немає. Проте при створенні корпоративних або розподілених додатків необхідно візуалізувати мережеву інфраструктуру програмної системи.

Технології доступу і маніпулювання даними у рамках загальної схеми «клієнт-сервер» також вимагають розміщення великих баз даних в різних сегментах корпоративної мережі, їх резервного копіювання, архівації для забезпечення необхідної продуктивності системи в цілому. З метою специфікації програмних і технологічних особливостей реалізації розподіленої архітектури потрібне візуальне представлення цих аспектів.

Першою з діаграм фізичного представлення є *діаграма компонентів*. Друга форма фізичного представлення програмної системи по апаратних вузлах комп'ютерної системи – це ***діаграма розгортання***. У російській (українській) літературі їх називають також ***діаграмами розміщення***. Діаграма розгортання дозволяє зображувати «фізичну» структуру програмної системи. На етапі проектування діаграма розгортання використовується для представлення фізичної сукупності вузлів як основи для реалізації системи. Діаграми розгортання складаються з трьох основних елементів: *артефактів*, *вузлів* і *відношень* між ними.

11.4.1. Артефакти і вузли

Артефакт – це фізичний елемент, частина програмної системи. Як правило, він є виконуваною програмою, але може бути також текстовим файлом, документом, сценарієм або іншим елементом системи, пов'язаним з програмним кодом. Артефакт може бути маніфестацією (реалізацією) одного або декількох компонентів. Між артефактами можливі відношення залежності або композиції.

Артефакт зображається у вигляді прямокутника, що містить його ім'я, мітку із стереотипом «artifact» (рис. 11.33). Можливо також присутність піктограми у вигляді листка паперу (у правому верхньому кутку). На рисунку також показано, що за допомогою артефакту **ІнтерфейсКористувача.с** реалізований компонент **ІнтерфейсКаси.** Для цього використано відношення маніфестації (залежність із стереотипом <<manifest>>).

Рисунок 11.33 – Приклад артефакту і компонента, що реалізовується ним

Світ артефактів досить широкий і різноманітний. У мові UML для позначення нових різновидів артефактів застосовують механізм стереотипів. Стандартні стереотипи UML для артефактів представлені в таблиці. 11.

Таблиця 11.1 – Різновиди артефактів

Стереотип	Опис
<<exécutable>>	Програмний файл, який може виконатися у комп'ютерній системі (має розширення .exe)
<<library>>	Статична або динамічна об'єктна бібліотека (має розширення .dll)
<<file>>	Фізичний файл в контексті розробленої системи
<<source>>	Початковий файл, який можна відкомпілювати у виконуваний файл
<<script>>	Скриптовой файл, який може інтерпретуватися комп'ютерною системою
<<document>>	Узагальнений файл, який не є початковим або виконуваним файлом

Вузол – фізичний елемент, який існує в період роботи програмної системи і є комп'ютерним ресурсом, що має пам'ять, а можливо, і здатність обробки. *Розгортання* – це розміщення артефакту або набору артефактів на вузлі для виконання. Графічно вузол на *діаграмі розгортання* зображається у формі тривимірного куба. Вузол має ім'я, яке вказується усередині цього графічного символу. Самі вузли можуть представлятися як на рівні типу (рис. 11.34 а), так і на рівні екземпляра (рис. 11,34 б).Відеокамера

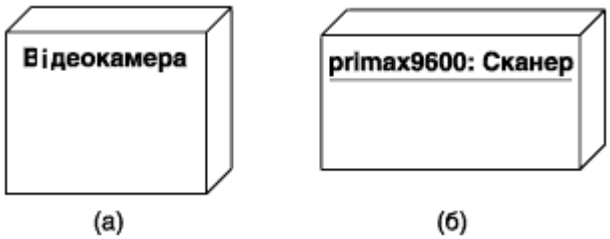


Рисунок 11.34 – Графічне позначення вузла

У першому випадку ім'я вузла записується у формі: <Ім'я типу *вузла* > без підкреслення і розпочинається із заголовної букви. У другому – ім'я вузла-екземпляра записується у виді: <ім'я вузла ':' Ім'я типу вузла>, а увесь запис підкреслюється. Ім'я типу вузла вказує на різновид вузлів, присутніх в моделі системи. Так, на представленому рис. 11.34, а) вузол з ім'ям Відеокамера відноситься до загального типу і ніяк не конкретизується. Другий *вузол* (рис. 11.34, б) є вузлом-экземпляром конкретної моделі сканера.

Зображення вузлів можуть розширюватися, щоб включити додаткову інформацію про специфікацію *вузла*. Якщо додаткова інформація відноситься до імені *вузла*, то вона записується під цим ім'ям у формі поміченого значення (рис. 11.35).

Рисунок 11.35 – Графічне зображення вузла-екземпляра з додатковою інформацією

Вузол є тим місцем, де фізично розміщуються артефакти, він грає роль квартири для артефактів (рис. 11.36, а). Як бачимо, розгортання зображається графічно шляхом вкладення символів артефактів в символ вузла.

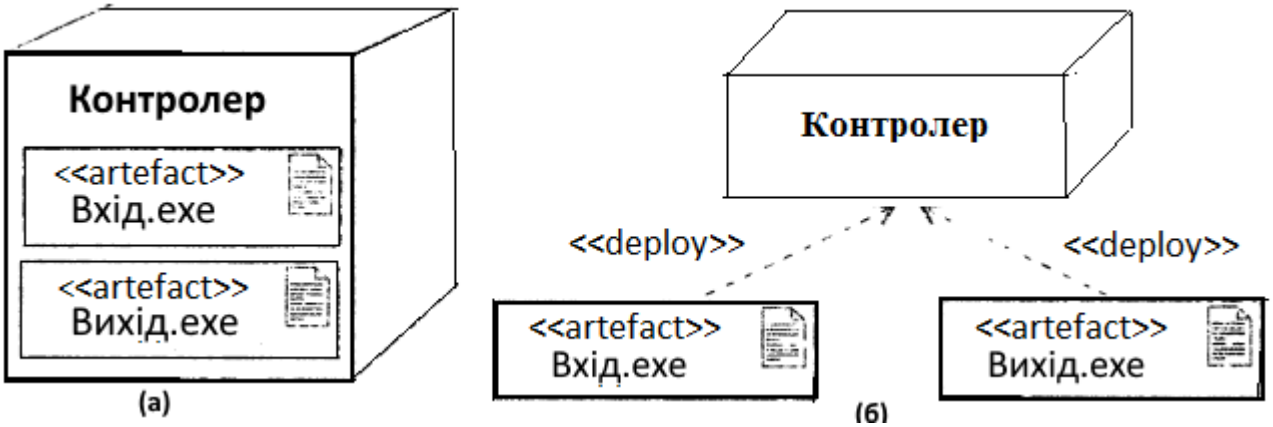


Рисунок 11.36 – Розміщення артефактів у вузлі

Якщо артефактів у вузлі досить багато, можна вибрати альтернативний спосіб: намалювати пунктирні стрілки, що йдуть від символів артефактів до символу розміщуючого вузла. Стрілки позначаються стереотипом «deploy» (розгортання, рис. 11.36, б).

Другий спосіб дозволяє показувати на діаграмі розгортання вузли з вкладеними зображеннями компонентів (рис. 11.37, б). Важливо пам'ятати, що такими вкладеними компонентами можуть виступати тільки виконувані компоненти і динамічні бібліотеки.

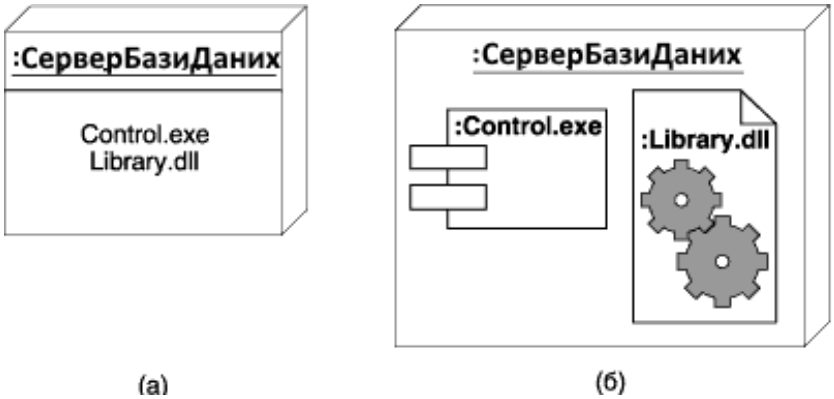


Рисунок 11.37 – Варіанти графічного зображення вузлів-екземплярів з розміщуваними на них компонентами

11.4.2. Побудова діаграм розгортання

Розробка *діаграми розгортання* розпочинається з ідентифікації усіх апаратних, механічних і інших типів пристроїв, які потрібні для виконання системою усіх функцій. Спочатку специфікуються обчислювальні *вузли* системи, що мають процесор і пам'ять. При цьому використовуються наявні в мові UML стереотипи, а, у разі відсутності останніх, розробники можуть визначити нові стереотипи. Окремі вимоги до складу апаратних засобів можуть бути задані у формі обмежень і помічених значень.

Складні програмні системи можуть реалізовуватися в мережевому варіанті на різних обчислювальних платформах і технологіях доступу до розподілених баз даних. Наявність локальної корпоративної мережі вимагає розв'язання цілого комплексу додаткових завдань з раціонального розміщення компонентів по вузлах цієї мережі, що визначає загальну продуктивність програмної системи.

Інтеграція програмної системи з мережею Інтернет визначає необхідність вирішення додаткових питань при проектуванні системи, таких як забезпечення безпеки, криптозахищеності і стійкості доступу до інформації для корпоративних клієнтів. Ці аспекти неабиякою мірою залежать від реалізації проекту у формі фізично існуючих вузлів системи, таких як сервери, робочі станції, брандмауери, канали зв'язку і сховища даних.

Технології доступу і маніпулювання даними у рамках загальної схеми «клієнт-сервер» також вимагають розміщення великих баз даних в різних сегментах корпоративної мережі, їх резервного копіювання, архівації, кешування для забезпечення необхідної продуктивності системи в цілому. Ці аспекти також вимагають візуального представлення з метою специфікації програмних і технологічних особливостей реалізації розподіленої архітектури.

Діаграми розгортання відображають відповідність конкретних програмних артефактів (наприклад, виконуваних файлів) обчислювальним вузлам (що виконують обробку). Вони показують розміщення програмних елементів у фізичній архітектурі системи і взаємодію (зазвичай мережеву) між фізичними елементами. Діаграма розгортання дозволяє краще зрозуміти фізичну архітектуру (чи архітектуру розгортання).

Графічно діаграма розгортання (рис. 11.38) – це граф з вузлів (чи екземплярів вузлів), сполучених асоціаціями, які показують існуючі комунікації. ***Вузли*** (чи їх екземпляри) можуть містити артефакти (чи їх екземпляри), що живуть і/або працюють у вузлах. Вузол належить до одного з двох типів:

1. *Вузол пристрою* – це фізичний (наприклад, цифровий або електронний) обчислювальний ресурс з пам'яттю і процесорним елементом, на якому працює програмне забезпечення.
2. *Виконуючий вузол оточення* – це програмний обчислювальний ресурс, що працює у рамках іншого вузла (наприклад, комп'ютера) і забезпечує виконання інших виконуваних програмних елементів.

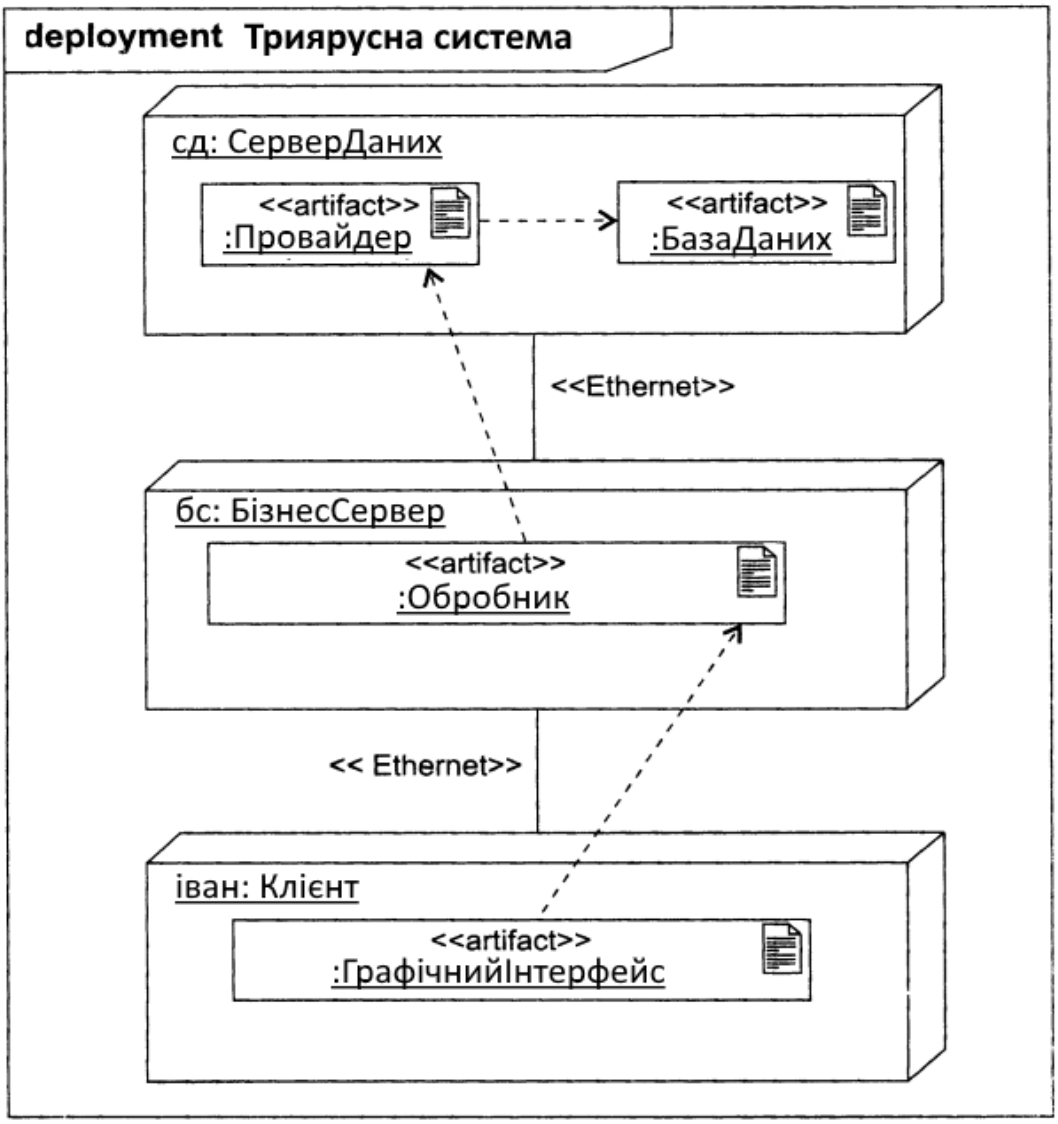


Рисунок 11.38 – Діаграма розгортання триярусної системи

Як показано на рис. 11.38, артефакти з'єднуються один з одним пунктирними стрілками залежностей. На цій діаграмі зображена типова триярусна система:

- ярус бази даних реалізований екземпляром **сд** вузла **СерверДаних**;
- ярус бізнес-логіки представлений екземпляром **бс** вузла **БізнесСервер**;
- ярус графічного інтерфейсу користувача утворений екземпляром **іван** вузла **Клієнт**.

У екземплярі сервера даних показано розміщення анонімного екземпляра артефакту **Провайдер** і анонімного екземпляра артефакту **БазаДаних.** Екземпляр вузла бізнес-сервера містить анонімний екземпляр артефакту **Обробник,** а екземпляр вузла клієнта – анонімний екземпляр артефакту **ГрафічнийІнтерфейс.** У діаграмі явно відображені три залежності між екземплярами артефактів. За допомогою стереотипів задані характеристики фізичних з'єднань між ярусами: усі вони визначені як Ethernet-з'єднанняя.

Розглянемо можливий варіант діаграми розгортання для програмної системи управління банкоматом зображений на рис. 11.39 [48].

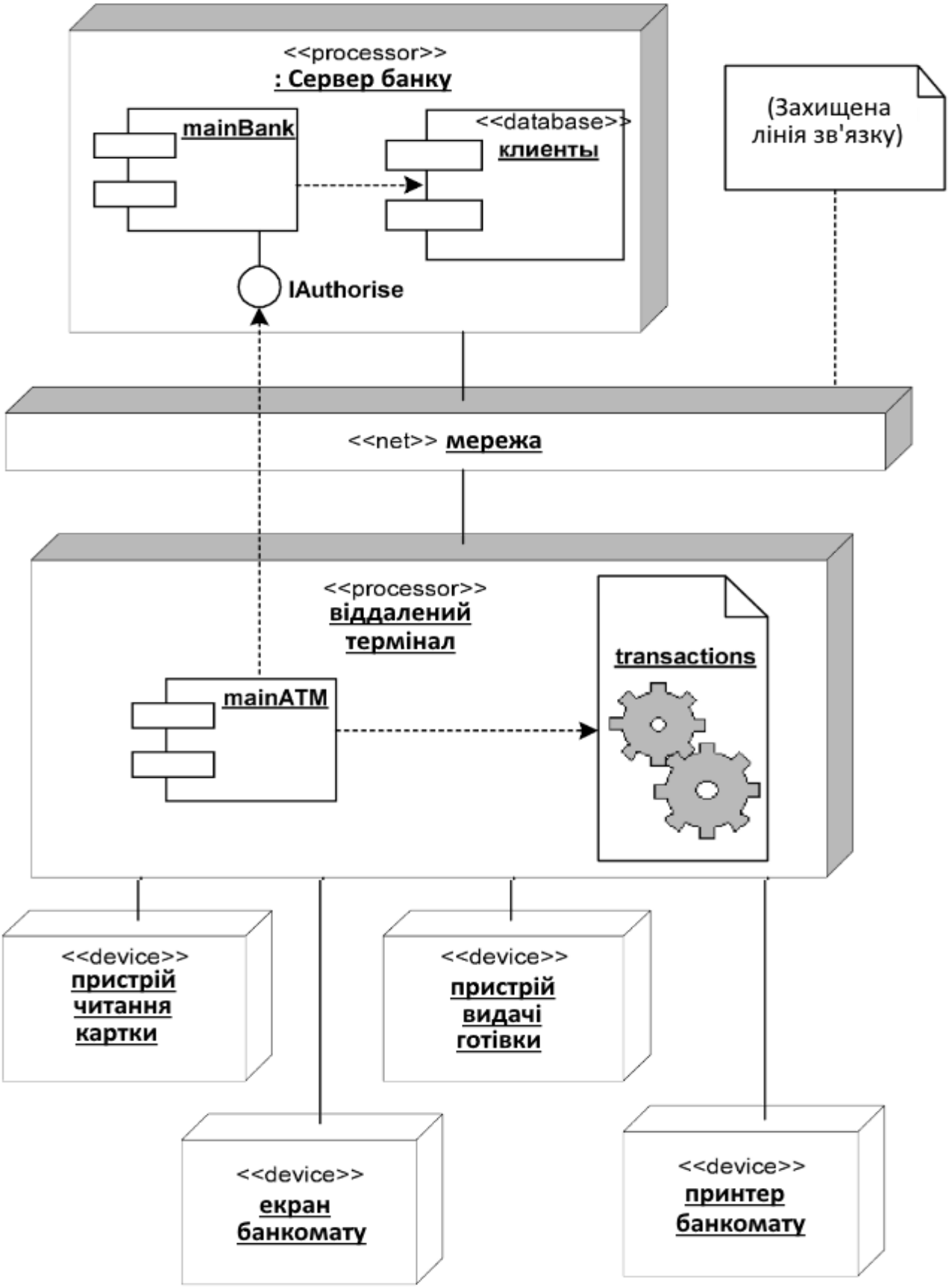


Рисунок 11.39 - Діаграма розгортання системи управління банкоматом

У рамках моделі, що розробляється, побудована діаграма розгортання, на якій зображені окремі вузли і фізичні канали комунікації між ними.

Ця діаграма містить 7 компонентів, 3 з яких зображені у формі ресурсоємних вузлів, а 4 – у формі пристроїв. При цьому на сервері банку і видаленому терміналі розміщені окремі компоненти моделі. Хоча графічне зображення цих вузлів робить зайвою вказівку текстових стереотипів, проте вони залишені для більшої наочності.

Як вже відзначалося раніше, діаграми розгортання можуть мати складнішу структуру, що включає вкладені компоненти, інтерфейси та інші апаратні пристрої. На даній діаграмі розгортання (рис. 11.39) явно вказана залежність компонента **mainATM** на видаленому терміналі від інтерфейсу **iAuthorise**, який реалізований компонентом **mainBank**, який, у свою чергу, размешен на анонімному узлі-екземплярі **Сервер банку**. Компонент **mainBank** залежить від компонента бази даних з ім'ям клієнта, який розгорнутий на цьому ж узлі.

Слід ще раз відмітити, що діаграма розгортання будується тільки у тому випадку, якщо не лише передбачається програмна реалізація моделі, що розробляється, але і росподілення фізичного розміщення її компонентів. Інакше цей тип канонічних діаграм може бути відсутнім в моделі, що характерно як для проектів документування і реинжиниринга бізнес-процесів, так і програм, що виконуються на одному комп'ютері.

Контрольні питання до розділу 11

1. Дайте визначення пакету.
2. У чому полягає різниця в зображенні порожнього і заповненого пакету.
3. Які рівні видимості в пакеті ви знаєте?
4. Поясніть поняття компонента і його інтерфейсу.
5. Які різновиди інтерфейсу ви знаєте?
6. Що є частиною пакету? Як вона іменується?
7. Які секції входять в графічне позначення класу?
8. Які властивості атрибутів вам відомі?
9. Який вигляд має форма представлення параметра операції?
10. Які властивості операцій вам відомі?
11. Які вершини і ребра утворюють діаграму розгортання?
12. Чим відрізняється екземпляр вузла від типу вузла?

РОЗДІЛ 12. МОДЕЛЮВАННЯ ПОВЕДІНКИ СИСТЕМИ

12.1. Модель поведінки

При створенні програмної системи недостатньо відповісти на питання **що робить система?** і **з чого вона складається?** – вимагається відповісти на питання **як працює система?** Відповідь на це питання дає *модель поведінки.* Поведінка реальної програми цілком і повністю визначається її кодом – як програма складена, так вона і виконується. Але програма – це просто запис алгоритму. Таким чином, ***модель поведінки*** **(behavior model) – це опис алгоритму роботи системи**.

Діаграма діяльності корисна для опису алгоритму дій, але вона не дає уявлення про поведінку певного об'єкту у рамках окремого варіанту використання або системи в цілому, що необхідно при об'єктно-орієнтованому програмуванні.

При проектуванні складної системи прийнято ділити її на частини, кожну з яких потім розглядати окремо. Таким чином, при об'єктній декомпозиції система розбивається на об'єкти або компоненти, які взаємодіють один з одним, обмінюючись повідомленнями. Повідомлення описують або представляють собою деякі події. Отримання об'єктом повідомлення активізує його і спонукає виконувати вказані його програмним кодом дії.

При цьому підході система стає подієво керованою, тому розробникам важливо знати, як повинен реагувати той або інший об'єкт на певні події. Ініціаторами подій можуть бути як об'єкти самої системи, так і її зовнішнє оточення.

Можна помітити, що додатки різних типів поводяться (з точки зору користувача) абсолютно по різному, що не може не позначатися на моделюванні поведінки.

Для користувача поведінка додатка проявляється передусім в інтерфейсі. В принципі можливі два архітектурні рішення інтерфейсу з користувачем:

- ініціатива належить програмі, тобто програма, будучи запущеною, виконує свою функцію, дотримуючись закладеного в неї алгоритму рішення задачі, запитуючи в міру необхідності інформацію у користувача (так званий ***активний інтерфейс***);
- ініціатива належить користувачеві, тобто програма, будучи запущеною, знаходиться в стані очікування команд користувача, які користувач подає, дотримуючись наявного у нього в голові алгоритму рішення задачі, а програма виконує команди (так званий ***пасивний інтерфейс***), що подаються.

Нині поширеним, особливо в наймасовіших застосуваннях для бізнесу, є друге рішення (тобто пасивний інтерфейс). Це обумовлено цілим рядом причин, у тому числі:

- збільшенням компактності багатофункціонального застосування при використанні пасивного інтерфейсу, тобто програма повинна

забезпечити здійснимість тільки порівняно невеликого числа порівняно простих команд, а підбір послідовності виконання команд, потрібних для розв'язання завдання, покладається на користувача;
- наявністю в сучасних системах програмування і операційних системах для цієї архітектури розвиненого механізму підтримки, що називається подієвим управлінням;
- сталою за останні декілька років звичкою масового користувача до пасивного інтерфейсу.

Існує множина способів опису алгоритмів, кожен з яких має свої переваги і недоліки, і призначений для застосування в різних ситуаціях. Наприклад, при описі алгоритмів, які призначені для виконання комп'ютером, використовуються мови програмування, але для опису алгоритмів, що виконуються людиною, мови програмування незручні і застосовуються інші способи.

Засоби моделювання поведінки в UML, зважаючи на різноманітність сфер застосування мови, повинні задовольняти набору різних і частково суперечливих вимог. Перерахуємо деякі з них.

1. Модель має бути досить детальною для того, щоб послужити основою для складання комп'ютерної програми.
2. Модель має бути компактною і осяжною, щоб служити засобом спілкування між людьми в процесі розробки системи і для обміну ідеями.
3. Модель не повинна залежати від особливостей реалізації конкретних комп'ютерів, засобів програмування і технологій.
4. Засоби моделювання поведінки в UML мають бути знайомими і звичними для більшості користувачів мови.

У UML передбачені декілька різних засобів для опису поведінки. Вибирання того або іншого засобу диктується типом поведінки, яку треба описати.

Так, у разі використання універсальної мови програмування ми не можемо автоматично переконатися в правильності програми: нам залишається тільки наполегливо її тестувати. Для опису життєвого циклу конкретного об'єкту, поведінка якого залежить від історії цього об'єкту, або ж вимагає обробки асинхронних стимулів, використовується кінцевий автомат у формі **діаграми станів**. При цьому стани кінцевого автомата відповідають станам об'єкту, тобто різним наборам значень атрибутів, а переходи відповідають виконанню операцій.

Кінцевий автомат орієнтований на представлення поведінки окремого об'єкту. Як і діаграми видів діяльності, діаграми кінцевих автоматів (станів) UML відбивають динамічну модель системи. За їх допомогою ілюструються події і стани об'єктів – транзакцій, прецедентів, людей тощо.

Стан об'єкту визначається поточними значеннями його атрибутів (як елементарних атрибутів, так і атрибутів, що означають інші класи).

Кінцевий автомат описує поведінку в термінах послідовності станів, через які проходить об'єкт впродовж свого життя. Ця послідовність розглядається як

відповідь на події і включає реакції на ці події. Автомат задає поведінку системи як цілісної, єдиної сутності, він моделює життєвий цикл єдиного об'єкту. В силу цього автоматний підхід зручно застосовувати для формалізації динаміки окремого, важкого для розуміння блоку системи.

Для опису потоку управління, тобто послідовності виконуваних елементарних кроків при виконанні окремої операції або реалізації складного варіанту використання, зручно використовувати **діаграми діяльності**.

Взаємодія декількох програмних об'єктів між собою описується **діаграмами взаємодії** в одній з двох еквівалентних форм: **діаграми кооперації** і **діаграми послідовності**. Для об'єктно-орієнтованої програми поведінка передусім визначається взаємодією об'єктів, тому діаграми цього типу мають також важливе значення при моделюванні поведінки в UML.

12.2. Діаграма кінцевого автомата

Кінцеві автомати в UML реалізовані досить своєрідно. З одного боку, в основу покладено класичне представлення автомата у формі графа станів-переходів. З іншого боку, до класичної форми додано велике число різних розширень і допоміжних позначень, які, строго кажучи, не обов'язкові, – без них в принципі можна було б обійтися – але дуже зручні і наочні при складанні діаграм: діаграми станів UML наочніші і виразні в порівнянні з класичними представленнями автоматів, але їх застосування вимагає більшої підготовленості користувача і пред'являє вищі вимоги до «кмітливості» і «уважності» інструментів моделювання.

Діаграма кінцевого автомата (діаграма станів) відображає кінцевий автомат, виділяючи потік управління, який іде від стану до стану. Кінцевий автомат – поведінка, яка визначає послідовність станів в ході існування об'єкту. Точніше, діаграма визначає для кожного стану об'єкту – дію, що виконується об'єктом при отриманні ним сигналу про подію. Один і той же об'єкт може виконувати різні дії у відповідь на одну і ту ж подію залежно від стану об'єкту.

На діаграмах станів застосовується всього один тип сутностей – *стани,* і всього один тип відношень – *переходи.* Сукупність станів і переходів між ними утворює *машину станів*.

У узагальненому виді діаграма кінцевого автомата показує:

- набір станів системи;
- події, які викликають перехід з одного стану в інший;
- дії, які відбуваються в результаті зміни стану.

У мові UML ***станом*** називають період в житті об'єкта, упродовж якого він задовольняє якійсь умові, виконує певну діяльність або чекає деякої події. Як представлено на рис. 12.1, стан зображається як прямокутник із закругленими кутами, що зазвичай включає його ім'я і підстани.

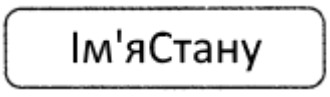


Рисунок 12.1 – Позначення стану

Переходи між станами відображаються поміченими стрілками (рис. 12.2).

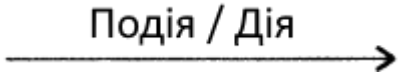


Рисунок 12.2 – Переходи між станами

Інакше кажучи, події викликають переходи, а дії є реакціями на події і переходи.

Подію, що запускає перехід, називають перемикальною подією. У мові UML визначено, що реакція на події проявляється у вигляді ефектів. Коли відбувається подія, залежно від поточного стану об’єкту спостерігається деякий ефект. Ефект задає поведінку, що реалізовується усередині автомата. Кінець кінцем, ефекти проявляють себе у виконанні дій, що змінюють стан об’єкту або повертають яке-небудь значення.

Приклади подій.

баланс < 0	Зміна в змозі
перешкоди	Сигнал (об’єкт з ім’ям)
зменшити(Тиск)	Виклик дії
after (5 seconds)	Закінчення періоду часу
when (time = 16:30)	Настання абсолютного моменту часу

Приклади дій.

касир. ПрипинитиВиплати()	Виклик однієї операції
Flt=new(Фільтр);flt. убратьПерешкоди ()	Виклик двох операцій
send нік.привіт	Посилка сигналу в об’єкт нік

Для відображення посилки сигналу використовують спеціальне позначення – перед ім’ям сигналу вказують службове слово send.

Під псевдостанами на діаграмі станів розуміються, знайомі вже нам початковий і кінцевий стан. Початковий стан зазвичай не містить ніяких внутрішніх дій і визначає точку, в якій знаходиться об’єкт за замовчуванням в початковий момент часу. Кінцевий стан також не містить ніяких внутрішніх дій і служить для вказівки на діаграмі області, в якій завершується процес зміни станів в контексті кінцевого автомата.

Для відображення переходу в початковий і кінцевий стан прийняті позначення, показані на рис. 12.3. Помітимо, що по суті початковий стан є псевдостаном, що відмічає точку старту.

Рисунок 12.3 – Перехід в початковий (старт) і кінцевий стан

Псевдостан – це тимчасовий стан, що формує внутрішню структуру переходу. При активізації псевдостану кінцевий автомат ще не закінчив виконання безперервного кроку переходу, тому не може обробляти події. Псевдостани використовуються для зв'язування фрагментів (сегментів) переходу. Перехід до одного з псевдостанів має на увазі, що подальший перехід до наступного стану буде зроблений автоматично, без допомоги події.

Кінцевий стан не є псевдостаном. Цей особливий стан, який залишається активним після завершення кінцевим автоматом безперервного кроку переходу. Як приклад на рис. 12.4 показана діаграма кінцевого автомата для системи охоронної сигналізації.

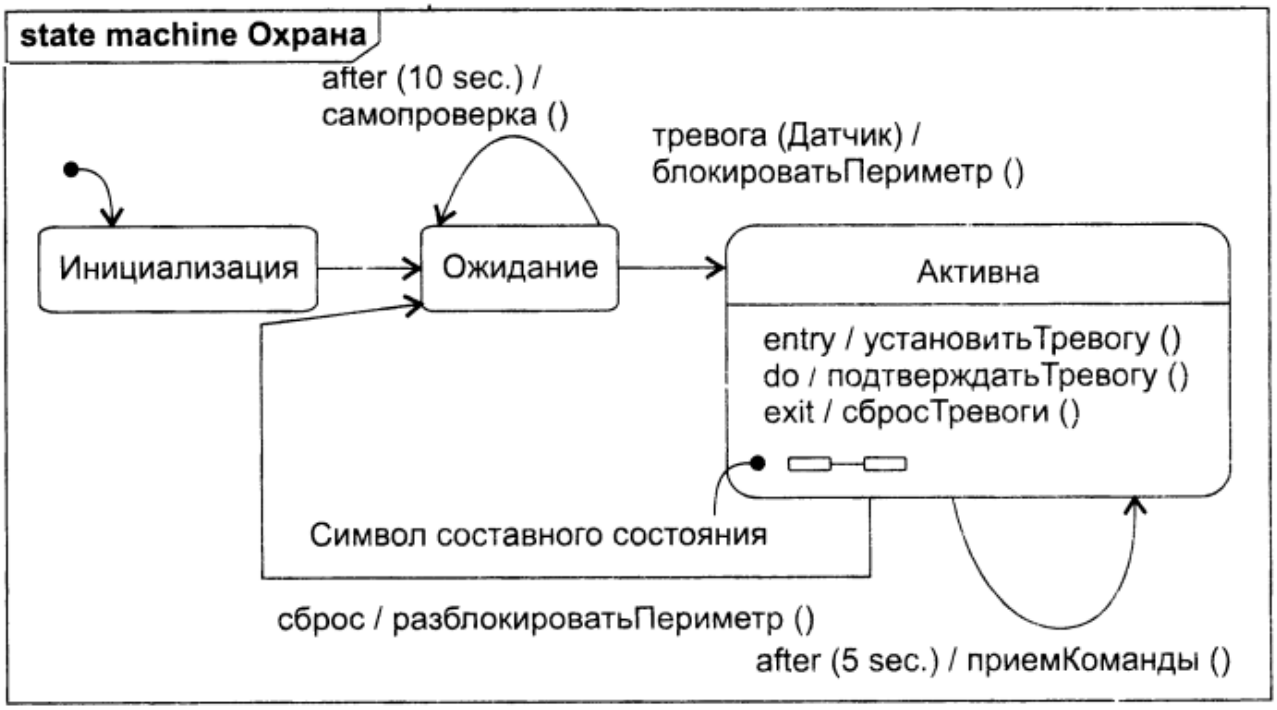


Рисунок 12.4 – Діаграма кінцевого автомата системи охоронної сигналізації

З рисунка видно, що система починає своє життя в стані **Ініціалізація,** потім переходить в стан **Очікування.** У цьому стані через кожні 10 секунд (по події **after(10 sec.)**) виконується самоперевірка системи (операція **самоперевірка()).** При настанні події **тривога(Датчик)** реалізуються дії, пов'язані з блокуванням периметра об'єкту, що охороняється, – виконується операція **блокироватьПериметр(),** і здійснюється перехід в стан **Активна.** У активному стані через кожні 5 секунд по події **after(5 sec.)** запускається операція **приемКоманды().** Якщо команда отримана (настала подія **скидання**), система повертається в стан **Очікування.** У процесі повернення розблоковується периметр об'єкта (операція **разблокироватьПериметр()),** що охороняється.

12.3. Дії в станах

Для вказівки дій, що виконуються при вході в стан і при виході із стану, використовуються мітки **entry** і **exit** відповідно.

Наприклад, як показано на рис. 12.5, при вході в стан **Активна** виконується операція **установитьТревогу()** з класу **Контроллер,** а при виході із стану – операція **сбросТревоги().**

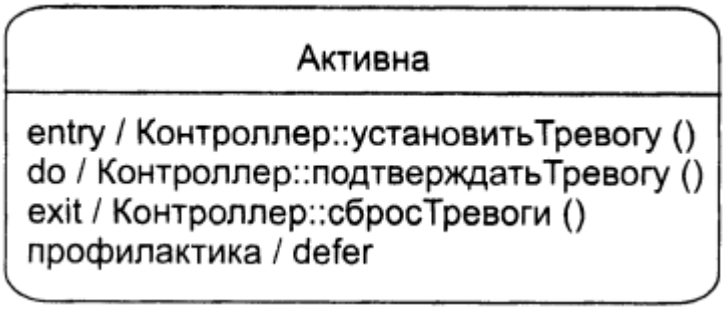


Рисунок 12.5 – Вхідні і вихідні дії і діяльність у стані **Активна**

Дія, яка повинна виконуватися, коли система знаходиться в цьому стані, вказується після мітки **do.** Вважається, що така дія починається при вході в стан і закінчується при виході з нього. Наприклад, у стані **Активна** це дія **подтверждатьТревогу().**

Можливі події, здійснення яких в цьому стані має бути відкладене. Такі події залишаються відкладеними до тих пір, поки система не перейде в інший стан, де вони вже не будуть відкладеними. Зарезервоване ім'я дії **defer** вказує на те, що подія відкладена в цьому стані і його підстанах:

имя-события / defer

У стані **Активна** (див. рис. 12.5) відкладеною є подія **профілактика**.

12.4. Умовні переходи

Між станами можливі різні типи переходів. Зазвичай перехід ініціюється перемикальною подією. Допускаються переходи і без подій – переходи після завершення (вони запускаються після закінчення діяльності в поточному стані). Нарешті, дозволені умовні або такі, що охороняються, переходи.

Правила позначки стрілок умовних переходів ілюструє рис. 12.6.

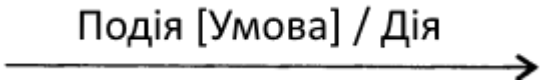


Рисунок 12.6 – Позначення умовного переходу

Порядок виконання умовного переходу:

1) відбувається подія (чи завершується діяльність у поточному стані);
2) обчислюється сторожова умова **УмоваПерехід**:
3) при **УмоваПерехід=true** запускається перехід і активізується дія, інакше перехід не виконується.

Приклад умовного переходу між станами **Ініціалізація** і **Очікування** наведений на рис. 12.7. Він відбувається по події **питаниеПодано**, але тільки у тому випадку, якщо досягнутий бойовий режим лазера.

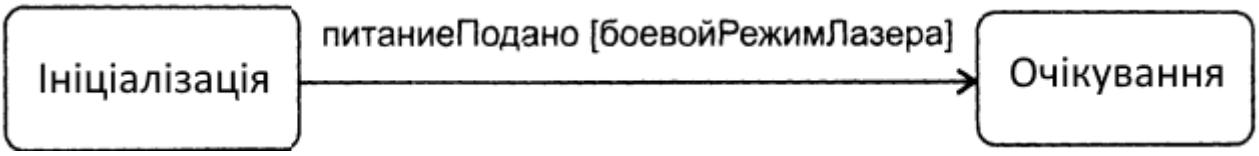


Рисунок 12.7 – Умовний перехід між станами

12.5. Композитні (складені) стани

Однією з найважливіших характеристик кінцевих автоматів в UML є підстан. Підстан дозволяє значно спростити моделювання складної поведінки. ***Підстан*** – цей стан, вкладений в інший стан.

Композитний (складений) стан – це стан, в який вкладена машина станів. Якщо вкладена тільки одна машина, то стан називається ***послідовним,*** якщо декілька **– *паралельним (ортогональним)***.

На рис. 12.8 показаний композитний стан, що містить в собі два підстани.

Рисунок 12.8 – Позначення підстанів

На рис. 12.9 наведена внутрішня структура композитного стану **Активна**.

Семантика вкладеності така: якщо система знаходиться в змозі **Активна**, то вона має бути точно в одному з підстанів: Перевірка, Дзвінок, Чекати. У свою чергу, в підстан можуть вкладатися інші підстани. Міра вкладеності підстанів не обмежується. Ця семантика відповідає випадку неортогональних (послідовних) підстанів.

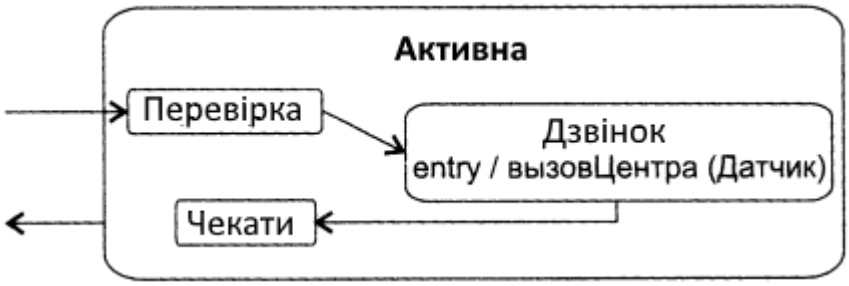


Рисунок 12.9 – Переходи в стані Активна

Іноді при поверненні в композитний стан виникає необхідність потрапити в той його підстан, який минулого разу був останнім. Такий підстан називають історичним. Інформація про історичний стан запам’ятовується. Як показано на рис. 12.10, подібна семантика переходів відображається значком історії – буквою **H** усередині кола.

При першому відвідуванні стану **Активна** автомат не має історії, тому відбувається простий перехід в підстан **Перевірка.** Припустимо, що в підстані

Дзвінок сталася подія **запит.** Засоби управління примушують автомат покинути підстан **Дзвінок** (і стан **Активна**) і повернутися в стан **Команди.** Коли робота в стані **Команди** завершується, виконується повернення в історичний підстан стану **Активна.** Оскільки тепер автомат запам'ятав історію, він переходить прямо в підстан **Дзвінок** (минувши підстан **Перевірка**).

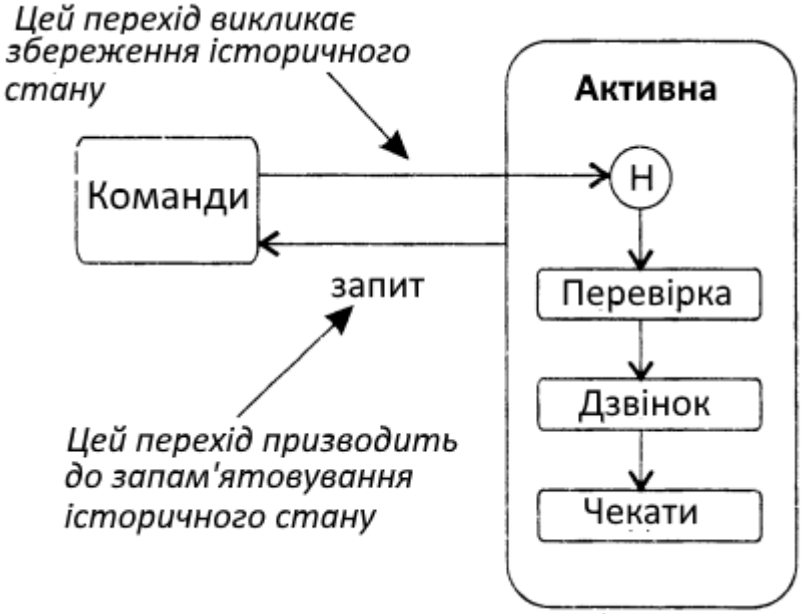


Рисунок 12.10 – Історичний стан

Як показано па рис. 12.11, для позначення композитного стану, що має усередині себе приховані (не показані на діаграмі) підстани, використовується символ «окуляри».

Рисунок 12.11 – Символ стану з прихованими підстанами

Можливі також **ортогональні** (паралельні) підстани – вони виконуються паралельно усередині композитного стану. В цьому випадку композитний стан розділяється на дві або більше областей. Його області називаються ортогональними областями. Якщо такий стан активний, то в кожній його ортогональній області активний рівно один з підстанів. Міра розпаралелювання незалежних дій дорівнює кількості ортогональних областей. Графічно ортогональні області відділяються одна від одної пунктирними лініями. Секція імені відділяється від областей суцільною лінією. Альтернативно можна вказати ім'я композитного стану на невеликому прямокутнику, приєднаному до символу цього стану. В цьому випадку секція з ім'ям не виглядає як ще одна область.

На рис. 12.12 зображений кінцевий автомат з композитним станом **Розробка,** у якого є три ортогональні області. Кожна з них, у свою чергу, ділиться на підстани. Окрім цього, є прості стани **Завершений** і **Зупинений**. Автомат моделює процес виконання програмного проекту.

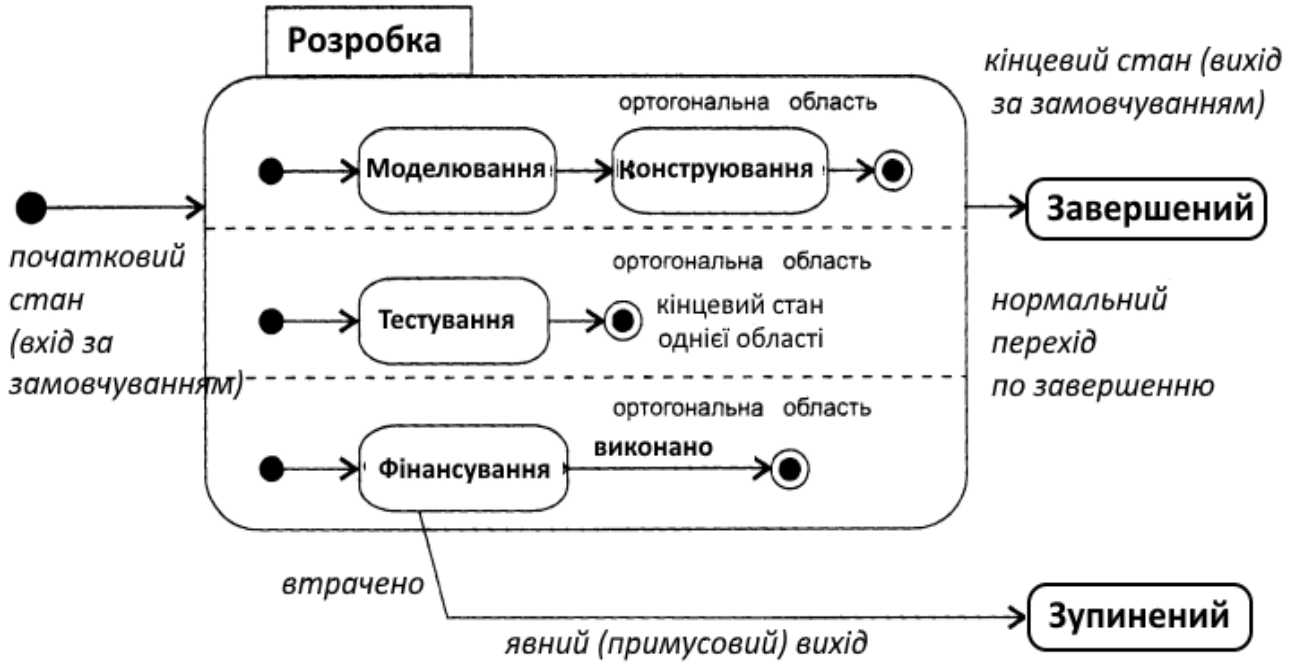


Рисунок 12.12 – Кінцевий автомат з паралельним композитним станом

При переході в композитний стан **Розробка** стають активними початкові підстани кожної області. Проходження по кожній з цих трьох ортогональних областей здійснюється паралельно. Якщо одна ортогональна область переходить в кінцевий стан раніше інших, управління в ній чекає, поки інші області не досягнуть свого кінцевого стану. Коли усі три підобласті досягають своїх кінцевих станів, управління з них зливається назад в загальний потік, і для зовнішнього композитного стану **Розробка** запускається перехід після завершення, після чого активним стає стан **Завершений**. У випадку якщо в той період, коли активний стан **Розробка,** відбувається подія втрачено (втрачено фінансування), усі три паралельні області припиняються, і активним стає стан **Зупинений**.

Коли є перехід в композитний стан з ортогональними областями, управління завжди розділяється на стільки ж паралельних потоків, скільки існує таких областей. Аналогічно, за наявності переходу з композитного стану з ортогональними областями паралельні потоки управління зливаються воєдино. Це відбувається завжди. Якщо усі ортогональні області досягають свого кінцевого стану або ж з’явився явний перехід з охоплюючого композитного стану, управління зливається в один потік.

В принципі завжди можна обійтися без використання складених станів. Наприклад, розглянемо всім відомий прилад – світлофор. На рис. 12.14 приведена еквівалентна машина станів світлофора (тобто, що описує ту ж саму поведінку, що і машина із складеними станами, рис. 12.13), що не містить складених станів. Порівняння діаграм на рис. 12.13 і рис. 12.14 є достатнім поясненням того, навіщо в UML введені складені стани.

Світлофор може знаходитися в двох основних станах:

1. Off – взагалі не працює – вимкнений або зламався.
2. On – працює.

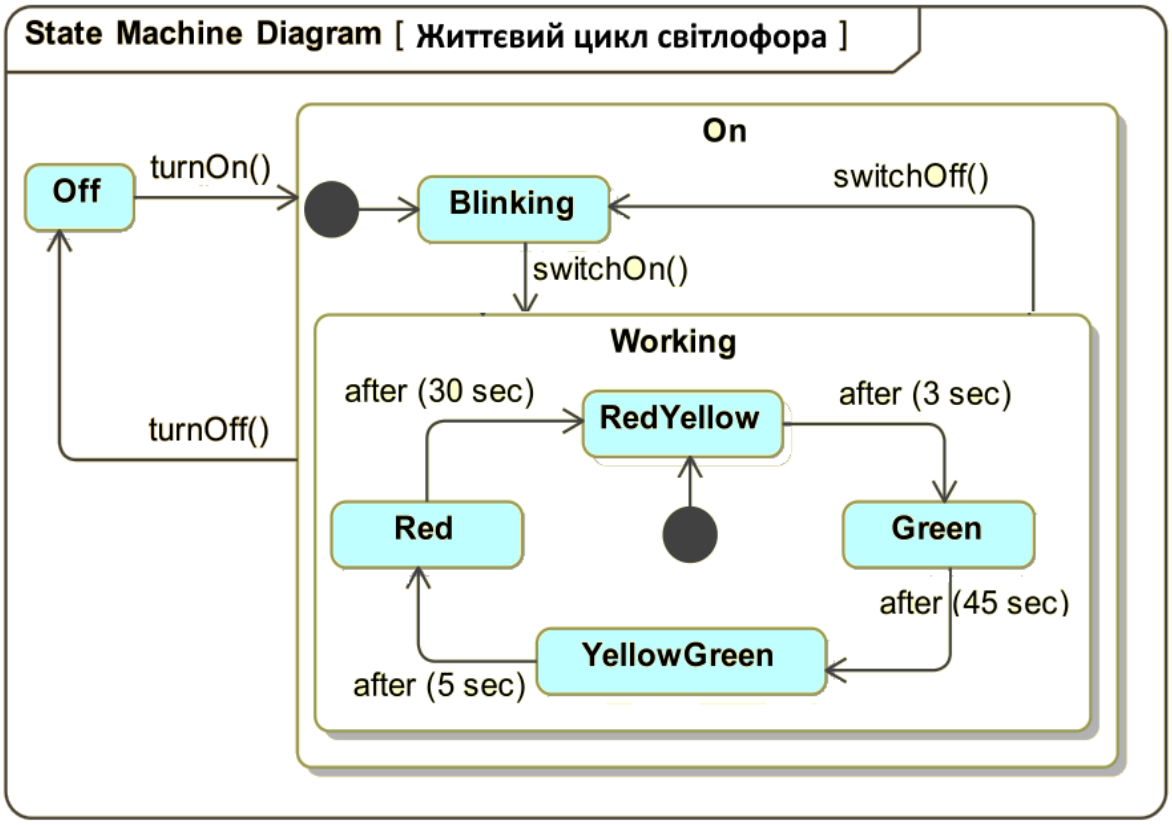


Рисунок 12.13 – Складені стани

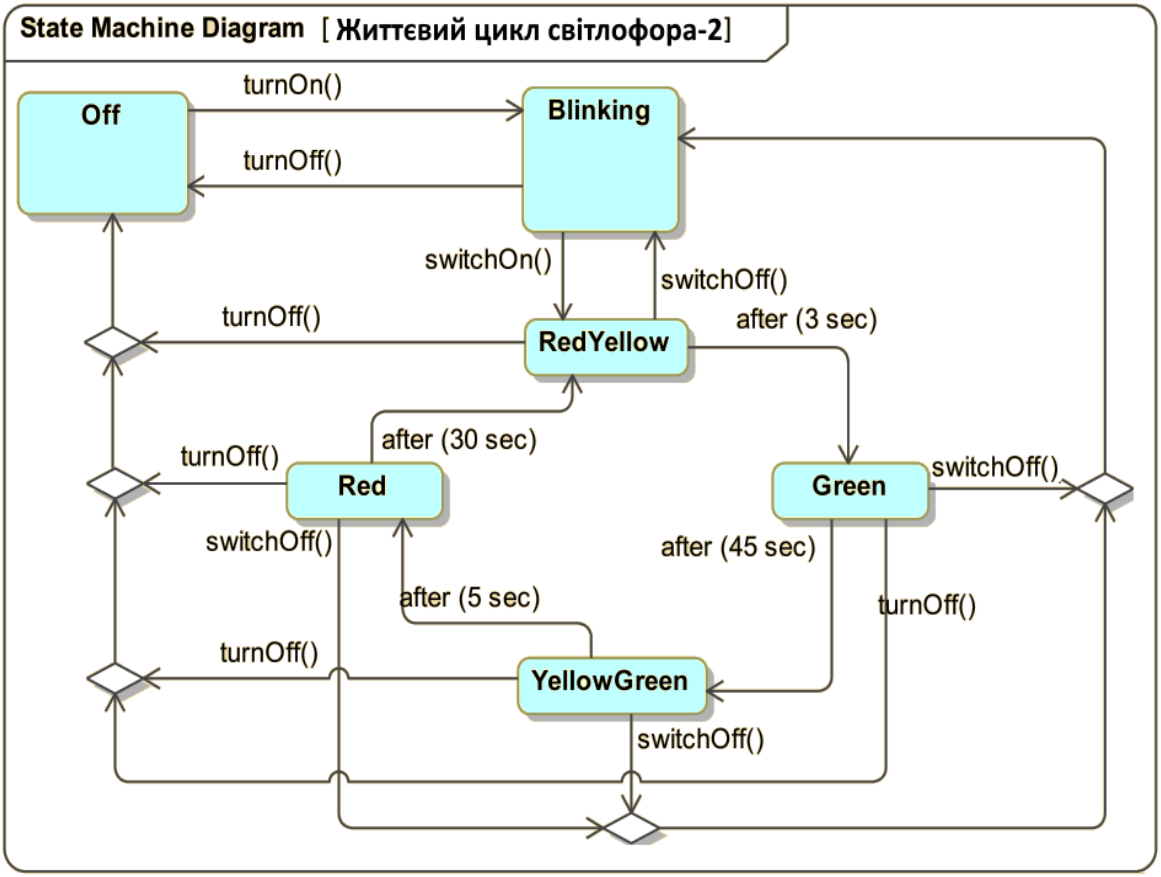


Рисунок 12.14 – Машина станів, що не містить складених станів

Але працювати світлофор може по-різному:

– Blinking – миготливий жовтий, дорожній рух не регулюється;
– Working – працює по-справжньому і регулює рух.

У останньому випадку у світлофора є 4 видимі стани, що є приписуючими сигналами для учасників дорожнього руху:

1. Green – зелене світло, рух дозволений;
2. YellowGreen – стан переходу з режиму дозволу в режим заборони руху (це справжній стан, світлофор знаходиться в ньому помітний час);
3. Red – червоне світло, рух заборонений;
4. RedYellow – стан переходу з режиму заборони в режим дозволу руху (цей стан відмінний від YellowGreen, світлофор подає дещо інші світлові сигнали, і учасники руху зобов'язані по-іншому на них реагувати).

12.6. Псевдостани управління

У діаграмах кінцевих автоматів для управління потоком переходів застосовуються фактично ті ж вузли управління, що і в діаграмах діяльності:

– *вибір* (choice – ромб з однією, що входить, і декількома вихідними стрілками);
– *розділення* (fork – жирна горизонтальна лінія з однією, що входить, і декількома вихідними стрілками);
– *злиття* (join – жирна горизонтальна лінія з декількома стрілками, що входять, і однією вихідною стрілкою).

У кінцевому автоматі ці вузли утворюють псевдостани. Псевдостан «вибір» дозволяє відобразити розгалуження переходів, стрілки, що виходять з нього, позначаються сторожовими умовами галуження. Залежно від значення умови псевдостан забезпечує вибір одного з багатьох переходів.

Варіанти ухвалення рішень на діаграмі станів можуть бути показані також як і на діаграмі діяльності за допомогою вузла вибору. При цьому перехід в стан вибору має бути тригерним і містити ім'я події. Переходи з псевдостану вибору в цільові стани повинні містити сторожові умови. Перехід, який повинен спрацьовувати, якщо жодна з умов не набуде значення «істина» повинен містити мітку «else» (рис. 12.15).

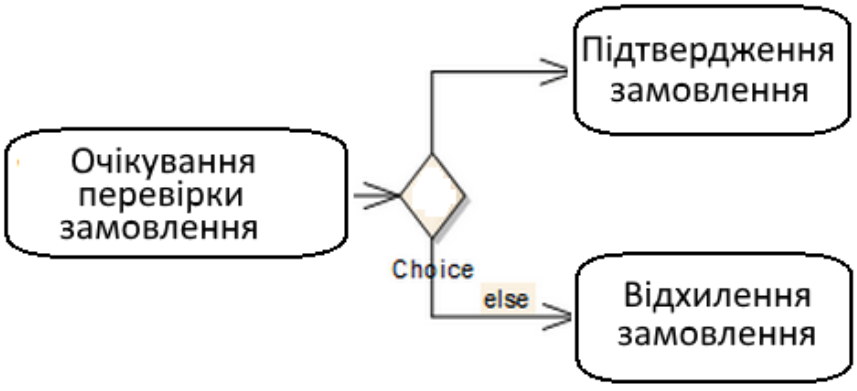


Рисунок 12.15 – Варіант ухвалення рішень

Псевдостани «розділення» і «злиття» дозволяють показати паралельні потоки підстанів, відмічаючи точки їх синхронізації при запуску (момент розділення) і при завершенні (момент злиття). На рис. 12.16 представлений варіант попереднього прикладу **Розробка** з явними переходами розділення і злиття.

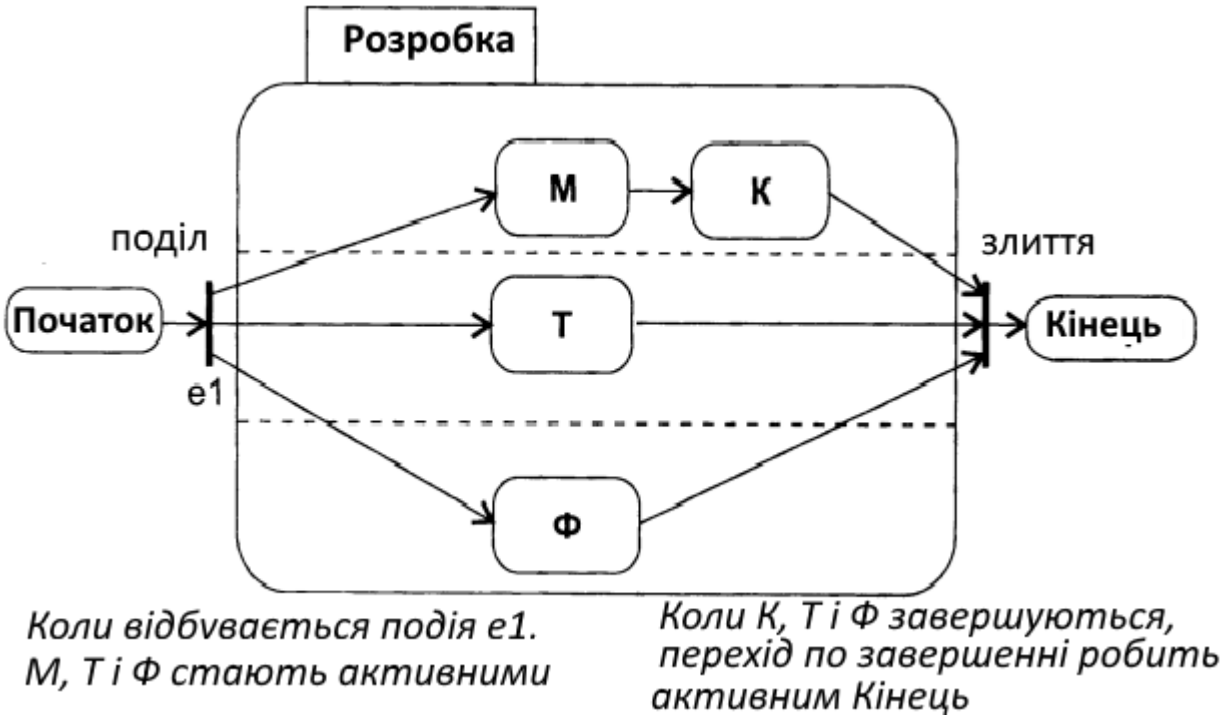


Рисунок 12.16 – Розділення і злиття

Розділення забезпечує перехід від одного стану **Початок** до трьох паралельних станів **М, Т** і **Ф** відразу. Графічно це виражає товста чорна лінія з однією стрілкою, що входить, і трьома вихідними, кожна з яких вказує на ортогональний стан.

У свою чергу, злиття реалізує перехід від трьох паралельних станів **К, Т** і **Ф** до єдиного стану **Завершений**. Тут три стрілки, що входять, і одна витікаюча. Злиття відбувається, якщо завершені усі три ортогональні (паралельних) стани.

Часто буває зручно визначати іменовані точки входу і виходу автомата, ялі сполучені з його внутрішніми станами. На рис. 12.17 приведений кінцевий автомат **Авторизація**.

Автомат виконує ідентифікацію клієнта. Він має стандартний вхід і стандартний вихід. У стандартному режимі дані користувача прочитуються з кредитної карти. Можливе і незвичайне, ручне введення даних. У цьому режимі користувач вводить свої дані вручну. Для ручного режиму заданий свій вхід – іменована точка **ручний набір** (позначається на межі автомата невеликим колом), яка безпосередньо з'єднується з композитним станом **Ручне введення.** При успішній авторизації автомат припиняє роботу у своєму кінцевому стані, на стандартному виході. Інакше він переходить в стан, сполучений з іменованою точкою виходу **відмова**. Точка виходу зображається на межі автомата невеликим колом з хрестиком.

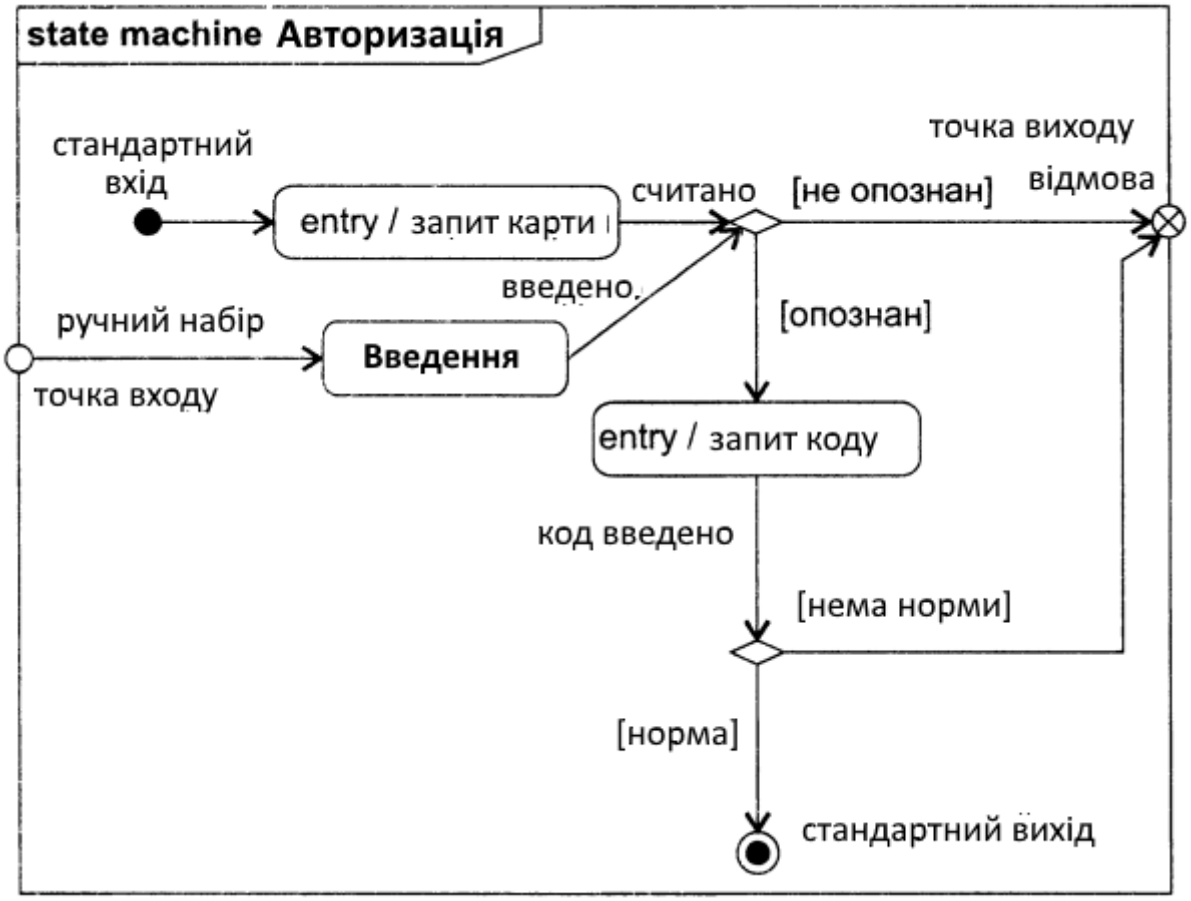


Рисунок 12.17 – Кінцевий автомат з іменованою точкою входу і виходу

Як бачимо, діаграма станів має схожу семантику з діаграмою діяльності, тільки діяльність тут замінена станом, переходи символізують дії. Таким чином, якщо для діаграми діяльності відмінність між поняттями «Діяльність» і «Дія» полягає в можливості подальшої декомпозиції, то на діаграмі станів діяльність символізує стан, в якому об’єкт знаходиться тривалу кількість часу, тоді як дія моментальна.

На закінчення наведемо діаграми станів UML в найзагальнішому вигляді (Діаграма автомата – рис. 12.18, Діаграма вкладеного автомата – рис. 12.19).

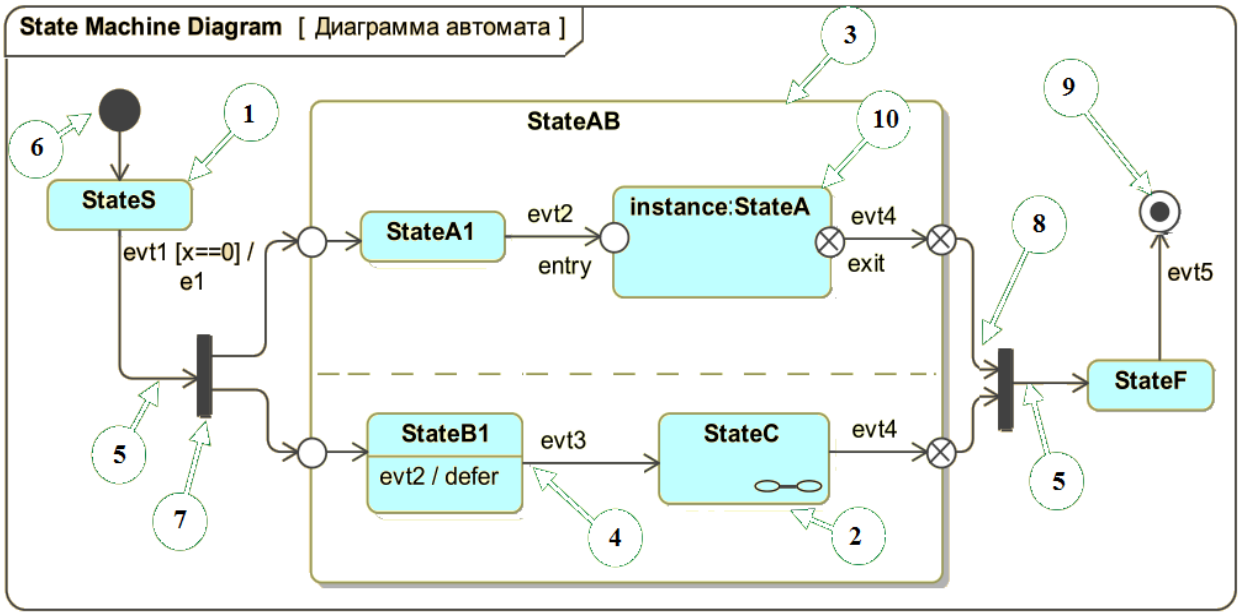


Рисунок 12.18 – Діаграма автомата

На рис. 12.18 відмічені такі сутності і відношення:
1. Простий стан. 2.Складений стан. 3. Складений ортогональний стан.
4. Простий перехід. 5. Складений перехід. 6. Початковий стан. 7. Розвилка.
8. Злиття. 9. Завершальний стан. 10. Посилальний стан.

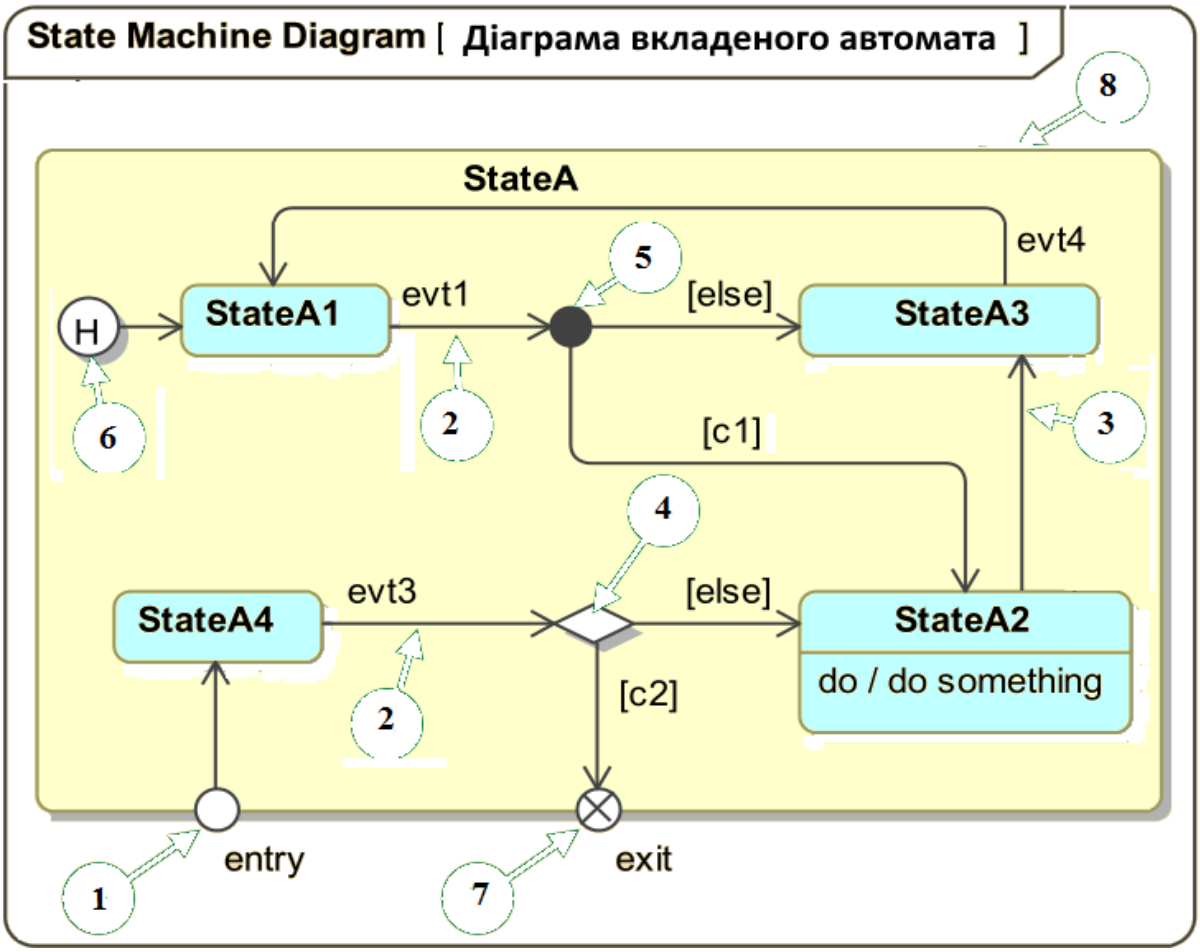


Рисунок 12.19 – Діаграма вкладеного автомата

На рис. 12.19 відмічені наступні сутності і відношення:
1. Точка входу. 2. Сегментований перехід. 3. Перехід після закінчення.
4. Стан вибору. 5. Перехідний стан. 6. Історичний стан.
7. Точка виходу. 8. Складений послідовний стан.

12.7. Застосування діаграм кінцевих автоматів

Зазвичай діаграми кінцевих автоматів застосовуються для моделювання поведінки подієво-керованих об'єктів. Застосовуйте діаграми станів тільки для тих класів, які проявляють цікаву поведінку, коли побудова діаграми станів допомагає зрозуміти, як усе відбувається. Автомати станів можна також використати при моделюванні поведінки графічного інтерфейсу, як реакції на дії користувача.

Якщо система чекає настання яких-небудь подій і виконує певні дії у відповідь, кінцевий автомат може бути легко використаний для специфікації необхідної поведінки в певному варіанті використання.

Подієво-керованим назвемо об'єкт, поведінка якого визначається його реакцією на зовнішні події. Говорять, що цей об'єкт управляється подіями. Його робота укладається в такий сценарій:

1. Очікування настання зовнішньої події.
2. При появі події об'єкт реагує на неї. Реакція залежить від його стану, визначуваного минулими подіями.
3. Повернення в стан очікування події, тобто перехід до пункту 1.

Сценарій застосовується упродовж усього життєвого циклу об'єкту.

Очевидно, що цьому визначенню відповідають екземпляри класів, елементи Use Case, та і програмні системи в цілому. Головне, щоб об'єкт моделювання управлявся подіями і розглядався як єдине ціле.

При моделюванні поведінки подієво-керованого об'єкту завжди визначають такі моменти:

- формують стани об'єкту (встановлюють початковий і кінцевий стани, інші стани упорядковують за часом так, щоб вони покрили увесь життєвий цикл об'єкту);
- встановлюють перелік подій, що викликають зміну станів;
- обмірковують дії, що виконуються при зміні станів;
- аналізують можливості спрощення автомата (за рахунок композитних станів і підстанів, застосування механізмів вибору, розділення, злиття, перехідних і історичних станів).

При використанні діаграми станів важливо ндотримуватись таких правил:

1. Діаграма станів повинна створюватися тільки для об'єктів, що мають реактивну поведінку. Не слід робити діаграму автоматів для усіх класів або об'єктів, досить вибрати тільки основні класи або об'єкти, що мають складну поведінку.

2. Діаграма станів має бути зосереджена на описі тільки одного аспекту поведінки об’єкту. Слід створювати діаграму автомата, що моделює поведінку тільки одного об’єкту. Якщо необхідно показати поведінку декількох, взаємозв’язаних об’єктів, допустимо створювати для них діаграму станів у рамках певного варіанту використання (діаграма станів для варіанту використання).
3. На діаграмі станів доцільно використати тільки ті елементи, які істотні для розуміння описуваного аспекту.

12.8. Діаграми діяльності

Діаграми діяльності також є засобом опису поведінки в UML, причому їх місце в мові допускає деякі різночитання. З одного боку, семантика діаграми діяльності визначена через семантику машини станів і в цьому сенсі діаграми діяльності вторинні по відношенню до діаграм станів.

З іншого боку, діаграми діяльності UML дуже близькі за своєю суттю до блок-схем, які є старим засобом опису поведінки.

З третього боку, діаграми діяльності в UML забезпечені додатковими синтаксичними засобами, що різко підвищують їх виразність і область застосування, навіть в порівнянні з машинами станів UML.

З четвертого боку, діаграми діяльності в найменшій мірі об’єктно-орієнтованні і містять найбільшу кількість свідомо допущених недомовленостей і довільних елементів (це зроблено для гнучкості і розширення сфери застосування).

Основних сутностей і відношень, вживаних на діаграмі діяльності, в деякому розумінні ще менше, ніж на діаграмі станів (хоча, здавалося б, менше вже нікуди – на діаграмі станів тільки стани і переходи).

Річ у тому, що основна сутність на діаграмі діяльності є частковим випадком простого стану (стан діяльності), а основне відношення – частковим випадком простого переходу (перехід після закінчення). У той же час всіляких додаткових позначень і варіантів нотації на діаграмі діяльності ще більше, ніж на діаграмі станів. Тому, щоб не загубитися в деталях, в наступних параграфах ми обговоримо зміст основних понять, а потім перейдемо до прикладів і основного нашого прикладу – інформаційної системи відділу кадрів (розділ 18).

12.8.1. Дія і діяльність

Дія в UML (версій 1.1-1.4) може бути одного з таких типів:

- надання значення;
- виклик операції;
- створення об’єкту;
- знищення об’єкту;
- повернення значення;
- посилка сигналу;
- останов.

Наведений список дуже схожий на список основних виконуваних операторів в звичайній мові програмування. Саме на цей «ефект пізнавання» і розраховували автори UML. Але UML не є мовою програмування, тому семантика дій до кінця в UML не визначається. Розрахунок робиться на те, що на абстрактному рівні всім усе зрозуміло. У таблиці. 12.1. наведені основні відомості про дії в UML.

Таблиця 12.1 – Дії в UML

Тип дії	Ключове слово
Присвоєння значення	**:=**
Виклик операції	**call**
Створення об’єкту	**create**
Знищення об’єкту	**destroy**
Повернення значення	**return**
Посилка сигналу	**send**
Останов	**terminate**
Дія, що не інтерпретується	будь-який текст

Ще один засіб, що відноситься до дій UML, називається повторювач. ***Повторювач*** – це вираз, що приписує виконати дію кілька разів (можливо, нуль разів). Синтаксично повторювач записується подібно до сторожової умови, в квадратних дужках, перед якими ставиться зірочка:

*** [повторювач] дія**

Діяльність (activity) в UML – цей опис поведінки у формі графа діяльності. Діяльність в UML моделює те ж, що і дія, тобто якусь змістовну активність під час роботи системи; у цьому сенсі діяльність подібна до дії, але діяльність протиставляється дії з усіх характеристичних ознак.

12.8.2. Граф діяльності

Призначення графа діяльності не міняється в UML від версії до версії, але спосіб визначення семантики, деталі нотації і назви елементів моделювання міняються. У діаграмах діяльності спостерігається «мирне співіснування» старого і нового.

Перерахуємо сутності і відношення, які застосовуються на діаграмі діяльності:

Вузли управління:

- початковий вузол (initial node) і завершальний вузол (final node);
- завершення потоку (flow final node);
- розгалуження управління (decision node);
- з’єднання управління (merge node);
- розвилка управління (fork node);
- злиття управління (join node);
- посилка сигналу (send signal action) і прийом сигналу (accept event action);
- прийом сигналу від таймера (accept time event action).

Вузли дій:

- дія (action);
- діяльність (activity);
- виклик діяльності (activity invocation).

Вузли даних:

- вхідний контакт (input pin) і вихідний контакт (output pin);
- параметр діяльності (activity parameter);
- центральний буфер (central buffer);
- сховище даних (data store).

Складені вузли:

- розбиття (partition);
- область розкладання (expansion region);
- область переривання (interraptible region).

На рис. 12.20 приведена узагальнена діаграма діяльності.

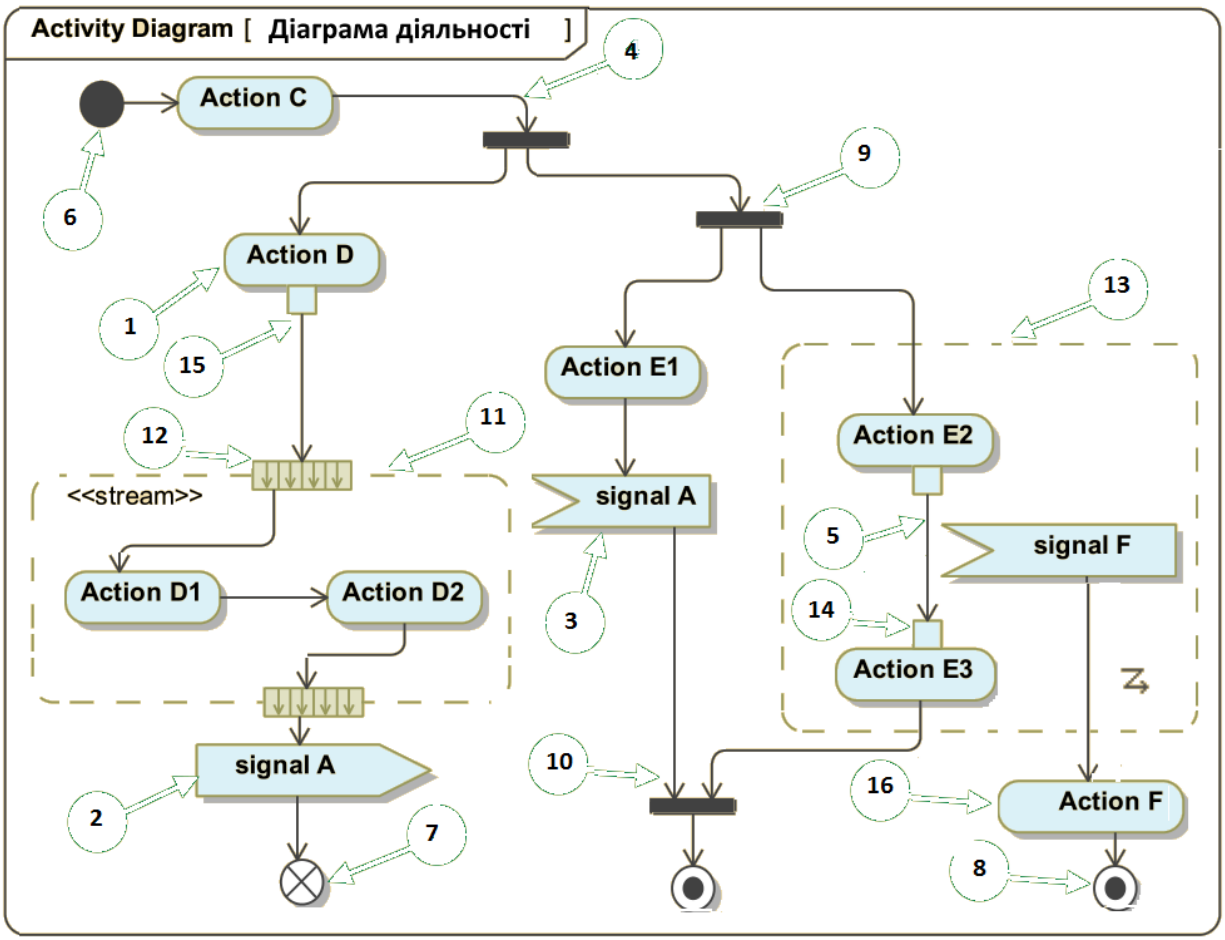


Рисунок 12.20 – Узагальнена діаграма діяльності

Сутності і відношення на діаграмі діяльності (рис. 12.20).

1. дія; 2. посилка сигналу; 3. прийом сигналу;
4. потік управління; 5. потік даних; 6. початковий вузол;
7. завершення потоку; 8. завершальний вузол; 9. розвилка управління;

10. злиття управління; 11. область розкладання; 12. вузол розкладання; 13; область переривання; 14. вхідний контакт; 15. вихідний контакт; 16. обробник переривання.

У рамках семантики машини станів автори UML зуміли описати семантику звичайних блок-схем, в яких немає ніяких подій, а є послідовна передача управління наступній діяльності після закінчення попередньої діяльності.

У розділі 18 ми обговоримо утримування інших понять діаграми діяльності на нашому основному прикладі – інформаційній системі відділу кадрів.

Підводячи підсумки опису поведінки діаграмами діяльності, обговоримо сфери застосування цих діаграм.

З одного боку, діаграми діяльності – загальний і потужний засіб опису алгоритмів. Причому не обов'язково програмно реалізованих алгоритмів: діаграми діяльності з успіхом можна застосовувати для моделювання поведінки людей, пристроїв і організацій при виконанні бізнес-процесів.

З іншого боку, діаграми діяльності, так само як і блок-схеми, можна застосовувати практично як засіб візуального структурного програмування.

Таким чином, діаграми діяльності застосовуються для опису поведінки на найвищому рівні абстракції, найвіддаленішому від програмної реалізації, і на найнижчому рівні, практично на рівні програмного коду.

12.9. Діаграми взаємодії

Діаграми взаємодії призначені для моделювання поведінки шляхом опису взаємодії об'єктів для виконання деякого завдання або досягнення певної мети. Взаємодія відбувається шляхом обміну повідомленнями.

Діаграми взаємодії застосовуються на різних рівнях моделювання: як для опису поведінки окремих операцій, так і цілих варіантів використання. Цей тип діаграм дозволяє описувати не лише взаємодію програмних об'єктів (екземплярів класів), але і взаємодію екземплярів інших класифікаторів: дійових осіб, варіантів використання, компонентів та ін.

Діаграми взаємодії зображаються в декількох різних графічних формах, з яких найважливішими є **діаграми послідовності** і **діаграми комунікації**.

Діаграми послідовності і діаграми комунікації семантично еквівалентні, хоча графічно виглядають зовсім по-різному. Семантично ці діаграми еквівалентні тому, що описують одне і те ж: послідовність передачі повідомлень між об'єктами в процесі їх взаємодії. А виглядають по-різному вони тому, що в діаграмі послідовності графічно підкреслюється впорядкованість в часі передаваних повідомлень, тоді як в діаграмі комунікації на передній план висувається структура зв'язків між об'єктами, по яких передаються повідомлення.

12.9.1. Повідомлення

Повідомлення (message) – це передача управління і даних від одного об'єкту (посилача) до іншого (одержувачеві).

Відправка повідомлення є **дією,** а отримання повідомлення **– подією.** У UML з передачею інформації і відправкою повідомлень пов'язані такі дії:

- виклик методу (call);
- створення об'єкту (create);
- знищення об'єкту (destroy);
- повернення значення (return);
- посилка сигналу (send).

Дія записується у вигляді тексту над (чи поряд з) стрілкою, що символізує повідомлення. Синтаксис виклику методу має відмінності в UML 1 і в UML 2. У UML 1 має місце такий синтаксис:

змінні := ІМ'Я (аргументи)

У UML 2 використовується дещо інший синтаксис:

атрибути – ІМ'Я (аргументи) : змінні

Атрибути зліва від знаку «=» це атрибути об'єкту, що відправляє повідомлення, призначені для зберігання повернутих значень. Змінні праворуч від знаку «:» – це локальні змінні взаємодії для набуття повернутих значень. Можна не використати ні атрибути, ні змінні, можна використати або те, або інше, а можна використати і те, і інше.

Повідомлення має відправника і одержувача, але якщо одержувачів декілька, то таке повідомлення називається ***широкомовним*** (broadcast).

Повідомлення може бути ***умовним***. При виконанні певної сторожової умови повідомлення відправляється, інакше нічого не відбувається. Сторожова умова записується, як завжди, в квадратних дужках.

Оскільки отримання повідомлення є подією, то одержувач повідомлення разом з інформацією отримує і управління, для того щоб мати можливість виконати дії, що ініціюються отриманим повідомленням. У UML розрізняється декілька типів передачі управління за допомогою повідомлення. Щоб відрізнити тип передачі повідомлення, в UML застосовується спеціальна графічна нотація, а саме розрізняються види стрілок, якими позначаються повідомлення. Хоча на діаграмах комунікації і послідовності повідомлення позначаються по-різному, принципи зображення однакові і перераховані в таблиці. 12.2.

Таблиця 12.2 – Типи передачі повідомлень

Вид стрілки	Тип передачі повідомлення
	Вкладений (синхронний) потік управління. Відправник може відправити наступне повідомлення тільки після того, як завершиться виконання усіх дій, ініційованих цим повідомленням.
 Тільки UML 1	Простий потік управління. Цей тип передачі має на увазі, що управління передається від відправника повідомлення одержувачу (можливо, безповоротно).

 у UML 1  У UML 2	Асинхронний потік управління. Повідомлення асихронно передається від відправника одержувачеві, при цьому у відправника зберігається свій потік управління, не залежний від потоку управління одержувача.
	Повернення управління. Цей тип передачі має на увазі повернення управління після виконання усіх дій, ініційованих передачею повідомлення з вкладеним потоком управління.

12.9.2. Діаграми послідовності

Діаграма послідовності призначена для моделювання поведінки у формі опису протоколу сеансу обміну повідомленнями між взаємодіючими екземплярами класифікаторів під час виконання одного з можливих сценаріїв.

Вісь часу і час життя об'єктів. На діаграмі послідовності вважається виділеним один напрям, що відповідає плину часу. За умовчанням вважається, що час тече зверху вниз, але це не обов'язково, наприклад, можна вважати, що час тече зліва направо, обумовивши це спеціальним коментарем. У наших прикладах використовується виключно нотація за умовчанням: час завжди тече зверху вниз. Саму вісь часу не відображають.

Повідомлення зображаються прямими стрілками різного виду (таблиця. 12.2). Якщо передача повідомлення вважається миттєвою (тобто, час передачі дуже малий), то стрілка горизонтальна (тобто, перпендикулярна осі часу). Якщо ж треба відобразити затриману доставку повідомлення, то стрілку трохи нахиляють, так щоб кінець стрілки був нижчий початку.

Лінія життя і стрілки повідомлень. Паралельно осі часу від усіх об'єктів, що беруть участь у взаємодії, відходить пряма пунктирна лінія, яка називається лінією життя (lifeline). Лінія життя представляє об'єкт у взаємодії: якщо стрілка відходить від лінії життя об'єкту, то це означає, що цей об'єкт відправляє повідомлення, а якщо стрілка повідомлення входить в лінію життя, то це означає, що цей об'єкт отримує повідомлення. Якщо ж стрілка перетинає лінію життя об'єкту, то це нічого не означає – повідомлення пролетіло мимо. Якщо в процесі взаємодії об'єкт закінчує своє існування, то лінія життя обривається і в цьому місці ставиться косий хрест. Над стрілкою повідомлення вказується текстова частина.

Якщо треба в явному виді вказати обмеження за часом, наприклад, вказати, що час затримки доставки повідомлення має бути обмежений зверху, то на діаграму в потрібному місці (має значення тільки положення по вертикалі) поряд з початком або кінцем стрілки повідомлення поміщають довільні ідентифікатори, які називаються мітками часу (time observation), і додають часові обмеження, задаючі потрібні умови на значення міток часу.

Мітка часу – це іменована точка на лінії життя. Перед міткою часу ставиться символ «@».

Повернення управління. Для наочності на діаграмі послідовності можна показати в явному виді повернення управління **(<- –** і, можливо, повернене

значення), хоча це не обов’язково: повернення управління мається на увазі при використанні повідомлення типу «виклик методу».

Сутності і відношення на діаграмі послідовності. На рис. 12.21 наведена узагальнена діаграма послідовності. Детальніша інформація про діаграму послідовності наведена в параграфі 18.3.2 при розгляді прикладу інформаційної системи відділу кадрів.

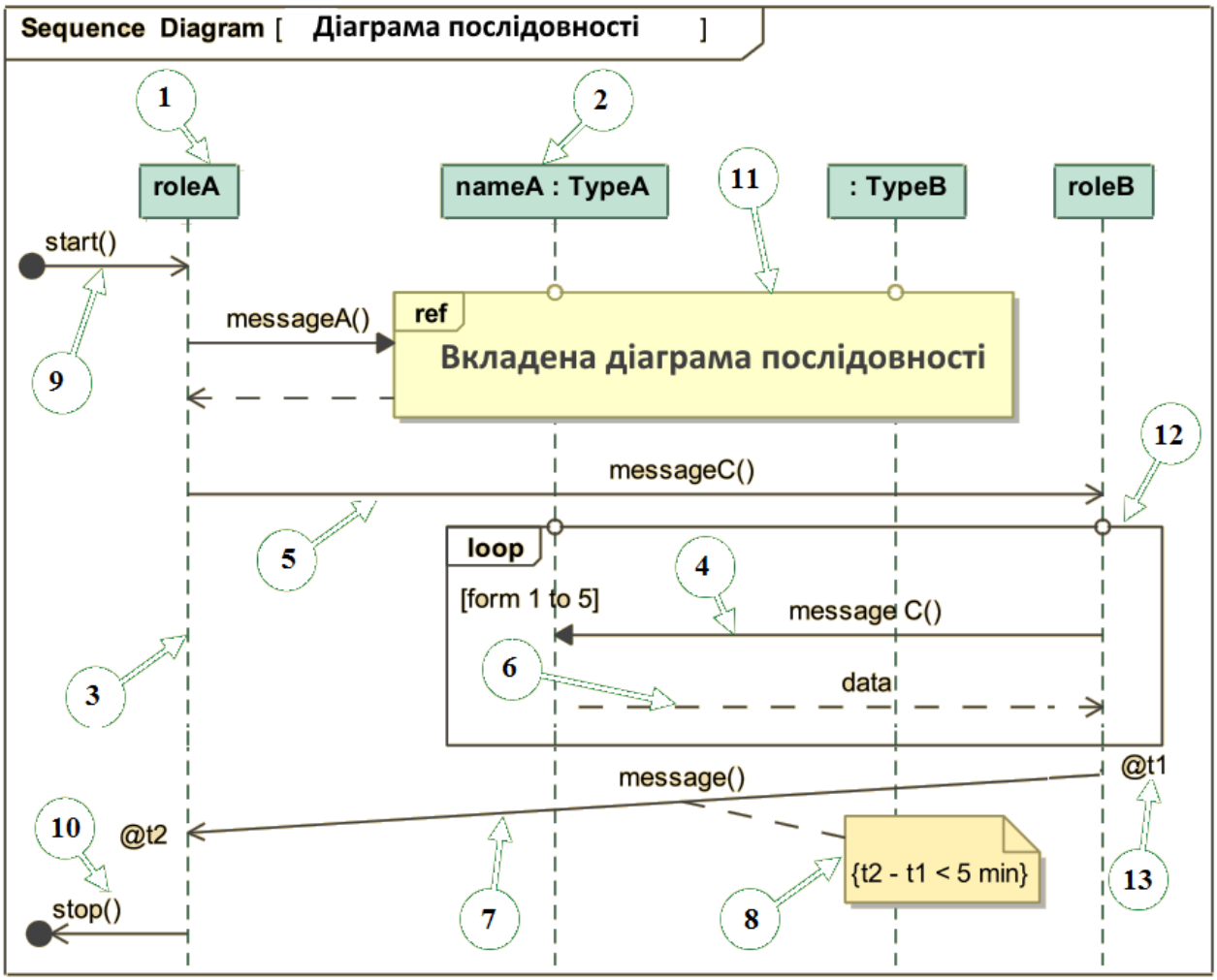


Рисунок 12.21 – Узагальнена діаграма послідовності

Сутності і відношення на діаграмі послідовності (рис. 12.21).

1. роль;
2. об’єкт;
3. лінія життя;
4. синхронний виклик;
5. асинхронний виклик;
6. повернення результату;
7. затримана доставка повідомлення;
8. обмеження тривалості;
9. знайдене повідомлення;
10. втрачене повідомлення;
11. посилання на взаємодію;
12. складений крок взаємодії;
13. мітка часу.

12.9.3. Діаграми комунікації

Діаграми комунікації – це особливий вид діаграм взаємодії, акцентованих на обміні даними між різними учасниками взаємодії. Замість того щоб малювати кожного учасника у вигляді лінії життя і показувати послідовність повідомлень, розташовуючи їх по вертикалі, як це робиться в діаграмах послідовності, комунікаційні діаграми допускають вільне розміщення учасників, дозволяючи малювати зв'язки, що показують відносини учасників, і використовувати нумерацію для подання послідовності повідомлень.

Графічних позначень на діаграмі комунікації значно менше, проте цей тип діаграм графічно дуже лаконічний і, проте, надзвичайно виразний.

Нагадаємо, що спочатку ці діаграми називалися діаграмами кооперації і були перейменовані в діаграми комунікації в UML 2.0.

Діаграма комунікації (так само як і діаграма послідовності) описує поведінку як *взаємодію*, тобто як протокол обміну повідомлень між об’єктами. Один і той же об’єкт може брати участь в різних взаємодіях, граючи в них різні ролі. Таким чином, взаємодія завжди відбувається в певному контексті, який визначається множиною об’єктів, що беруть участь у взаємодії, і зв’язків.

Номер повідомлення визначається відповідно до положення повідомлення в послідовності повідомлень цієї взаємодії. Якщо у взаємодії використовуються тільки прості або асинхронні повідомлення, то повідомлення просто нумеруються, зазвичай підряд: 1, 2, 3 і т. д.

Повідомлення з меншими номерами передують сполученням з великими номерами. Якщо ж використовуються вкладені потоки управління, тобто повідомлення типу «виклику операції», то повідомлення нумеруються складнішим чином. Припустимо, що повідомлення виклику деякої операції має номер *x. Тоді повідомлення, що відправляються при виконанні цієї операції, матимуть номери x.1, x.2 x.3* і т. д.

Розглянемо приклад з інформаційної системи відділу кадрів (Створення підрозділу, рис. 12.22).

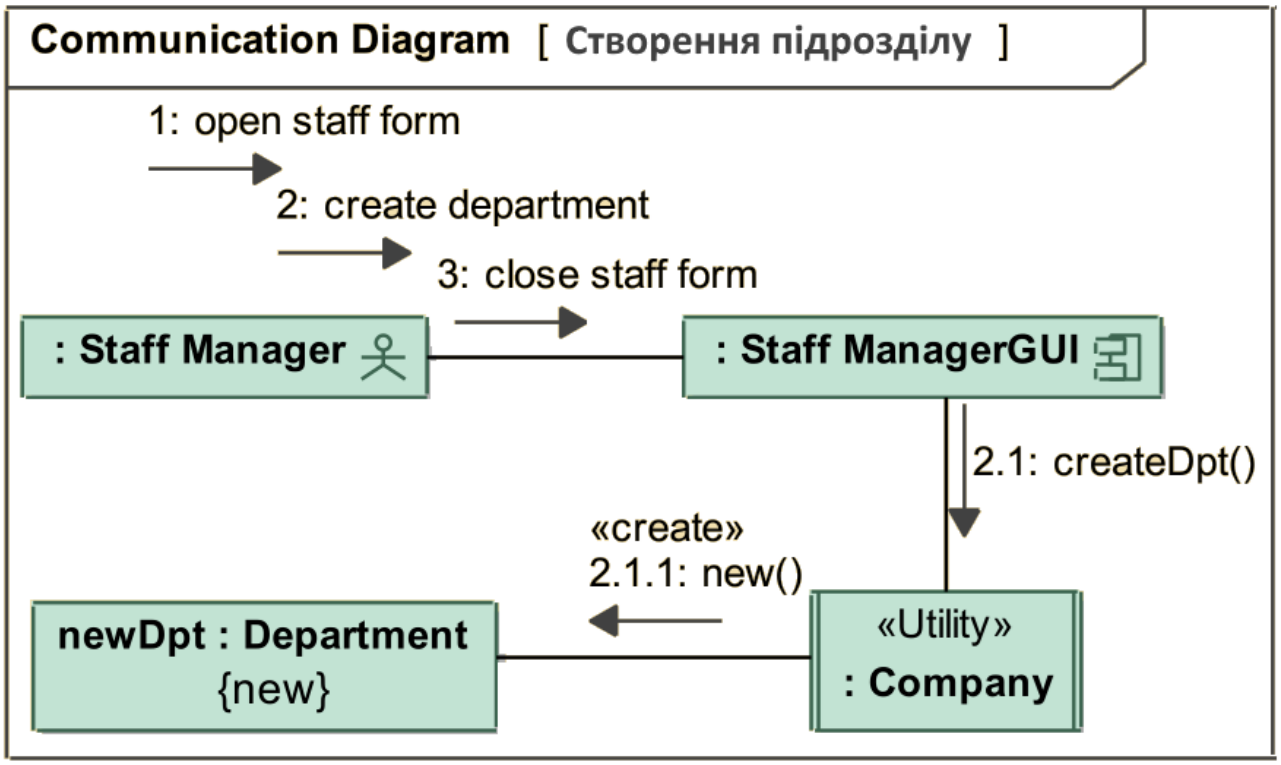


Рисунок 12.22 – Діаграма комунікації

Контрольні питання до розділу 12

1. Охарактеризуйте вершини і дуги діаграми кінцевого автомата. У чому полягає призначення цієї діаграми?
2. Як відображаються дії в станах діаграми кінцевого автомата?
3. Як показуються умовні переходи між станами?
4. Які дії Ви знаєте в діаграмі діяльності?
5. Призначення діаграми послідовності для моделювання поведінки.
6. Як раніше (UML 1) називалися діаграми комунікації?

РОЗДІЛ 13. ОСОБЛИВОСТІ РОЗРОБКИ БАЗ ДАНИХ З ЕЛЕМЕНТАМИ UML

Невід'ємною частиною численної категорії програмних систем є бази даних. У цьому розділі обговорюються: термінологія баз даних, розширення мови UML, що забезпечують моделювання баз даних, а також особливості розробки реляційних баз даних в об'єктно-орієнтованому ПЗ [31; 32; 39].

13.1. Основні поняття баз даних: моделі даних

Нині застосовують такі моделі даних: ієрархічну, мережеву, реляційну, об'єктно-орієнтовану, об'єктно-реляційну.

У **ієрархічній моделі** структура даних представляється у вигляді дерева. Це означає, що кожен запис у БД може мати скільки завгодно нащадків, але тільки одного батька. Така структура накладає жорсткі обмеження на організацію логічних зв'язків між даними.

У **мережевій моделі** допускаються довільні зв'язки між даними. Недоліками є висока складність структури БД, труднощі в її розумінні і обробці.

У **реляційній моделі** усі дані знаходяться в таблицях. Зв'язки між таблицями встановлюються за співпадаючими значеннями в стовпцях, що мають однакові імена (ключових стовпцях). Перевагами реляційної моделі є простота, гнучкість, зрозумілість, зручність реалізації.

У **об'єктно-орієнтованій моделі** структура БД представляється у вигляді графа, вузлами якого є об'єкти. Тут з'являється можливість прозорого відображення складних взаємозв'язків об'єктів, можливість «елегантного» застосування усієї потужності об'єктно-орієнтованих механізмів (інкапсуляції, спадкоємства, поліморфізму). Недоліками моделі є висока понятійна складність БД, незручність обробки даних, низька швидкість виконання запитів.

Об'єктно-реляційна модель пропонує гібридне рішення, засноване на застосуванні як реляційної, так і об'єктно-орієнтованої моделі. У цій моделі структура БД представляється набором таблиць, але в самих таблицях зберігаються об'єкти. При цьому зменшуються труднощі, які довелося б здолати на шляху перетворення чисто реляційної БД в чисту об'єктно-орієнтовану БД.

13.2. Розширення UML для моделювання баз даних

Нині стандарт мови UML не має засобів для моделювання баз даних. Найвідоміші підходи до вирішення цієї проблеми запропоновані корпорацією Rational і С. Амблером. Ми ж за основу розширення UML приймемо позначення С. Амблера як найпростіші і такі, що пропрацьовані для реляційної БД [37].

13.2.1. Типи моделей даних

Відомо, що при моделюванні БД послідовно застосовують три типи моделей:

1. ***Концептуальні моделі даних***. Ці моделі створюють на першому кроці моделювання і використовують для дослідження понять проблемної області з точки зору замовника. Основними елементами тут є бізнес-моделі.
2. ***Логічні моделі даних.*** Логічні моделі створюють на другому кроці моделювання. Вони фіксують вимоги до системи з точки зору розробника і описують логічну організацію системи з базою даних, що реалізовує ці вимоги (в термінах сутностей даних і відношень між сутностями). Основними елементами логічних моделей є діаграми класів.
3. ***Фізичні моделі даних.*** Фізичні моделі створюють на третьому кроці моделювання. За їх допомогою проектують внутрішню схему бази даних, зображуючи таблиці даних, атрибути (стовпці) таблиць і відношення між таблицями.

Тип моделі має бути позначений або за допомогою відповідного стереотипу (таблиця. 13.1), або вказаний як текстовий коментар в примітці UML. У разі фізичної моделі різновид механізму збереження даних слід позначити як один із стереотипів (таблиця. 13.2).

Таблиця 13.1 – Стереотипи для вказівки типу моделі

Стереотип	Тип моделі
<<Концептуальна модель даних>>	Концептуальна модель даних
<<Логічна модель даних>>	Логічна модель даних
<<Фізична модель даних>>	Фізична модель даних

Таблиця 13.2 – Стереотипи для різних механізмів збереження даних

Стереотип	Різновид механізму збереження даних
<<Ієрархічна БД>>	Ієрархічна база даних
<<Мережева БД>>	Мережева база даних
<<Реляційна БД>>	Реляційна база даних
<<Об’єктно-орієнтована БД>>	Об’єктно-орієнтована база даних
<<Об’єктно-реляційна БД>>	Об’єктно-реляційна база даних

13.2.2. Таблиці, сутності, представлення і відношення

Для позначення таблиць, сутностей і представлень використовуються прямокутники класів. Ці прямокутники мають відповідні стереотипи (таблиця. 13.3).

Таблиця 13.3 – Стереотипи для вершин в моделях даних

Стереотип	Тип моделі	Застосування
<<Таблиця>>	Фізична	Зазвичай у фізичній моделі цей стереотип мається на увазі за замовчуванням
<<Асоціативна таблиця>>	Фізична	Застосовний до асоціативних таблиць у фізичній моделі для реляційної БД
<<Представлення>>	Фізична	Застосовний при моделюванні представлення. Для вершини-представлення вказуються залежності від таблиць-джерел даних
<<Індекс>>	Фізична	Застосовний при моделюванні Індексу. Прийнято вказувати на залежність індексу від індексованого стовпця таблиці
<<Сутність>>	Логічна Концептуальна	Додаткове позначення, яке за замовчуванням є типом моделі
<<Процедури, що зберігаються>>	Фізична	Застосовний до класу, який містить тільки сигнатури операцій для тих процедур БД, що зберігаються

Наведемо приклад логічної моделі даних (рис. 13.1) і приклад фізичної моделі даних (рис. 13.2). Прямокутники класів в концептуальних і логічних моделях даних означають визначення сутностей, для яких стереотип є доповненням. У фізичній же моделі даних для реляційної БД передбачається, що будь-який прямокутник класу без стереотипу – це таблиця.

Відмітимо, що у фізичній моделі (рис. 13.2) є три таблиці (**Викладач, Зарплата, Посада),** одне представлення **Ввикладач** і один індекс **ІндексВикладач.** Представлення залежить від двох звичайних таблиць. Індекс залежить від таблиці Викладач, в ній вказано, що індексується стовпець таблиці **Номер Викладача**.

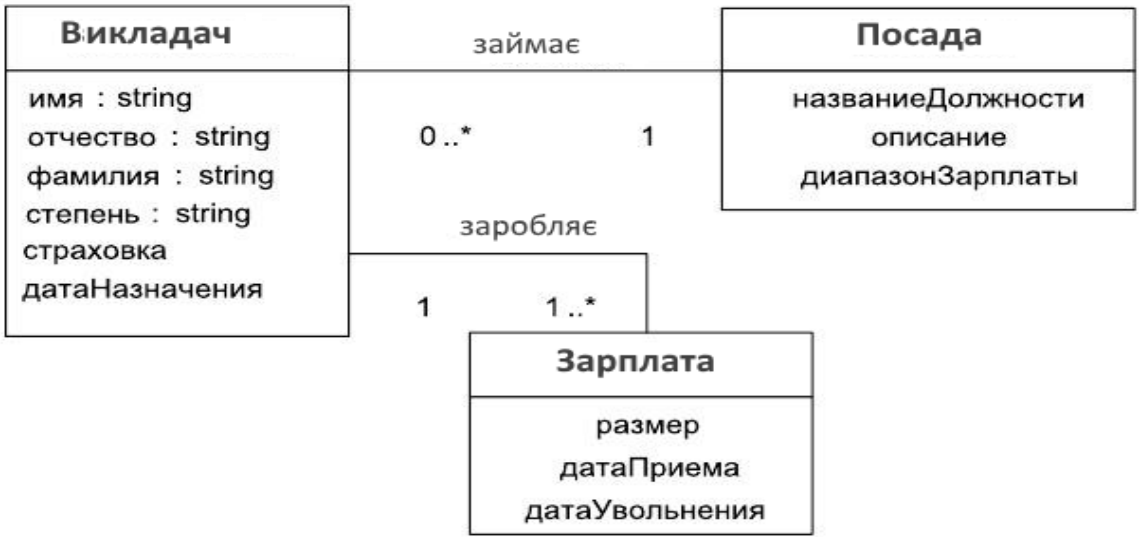


Рисунок 13.1 – Логічна модель даних

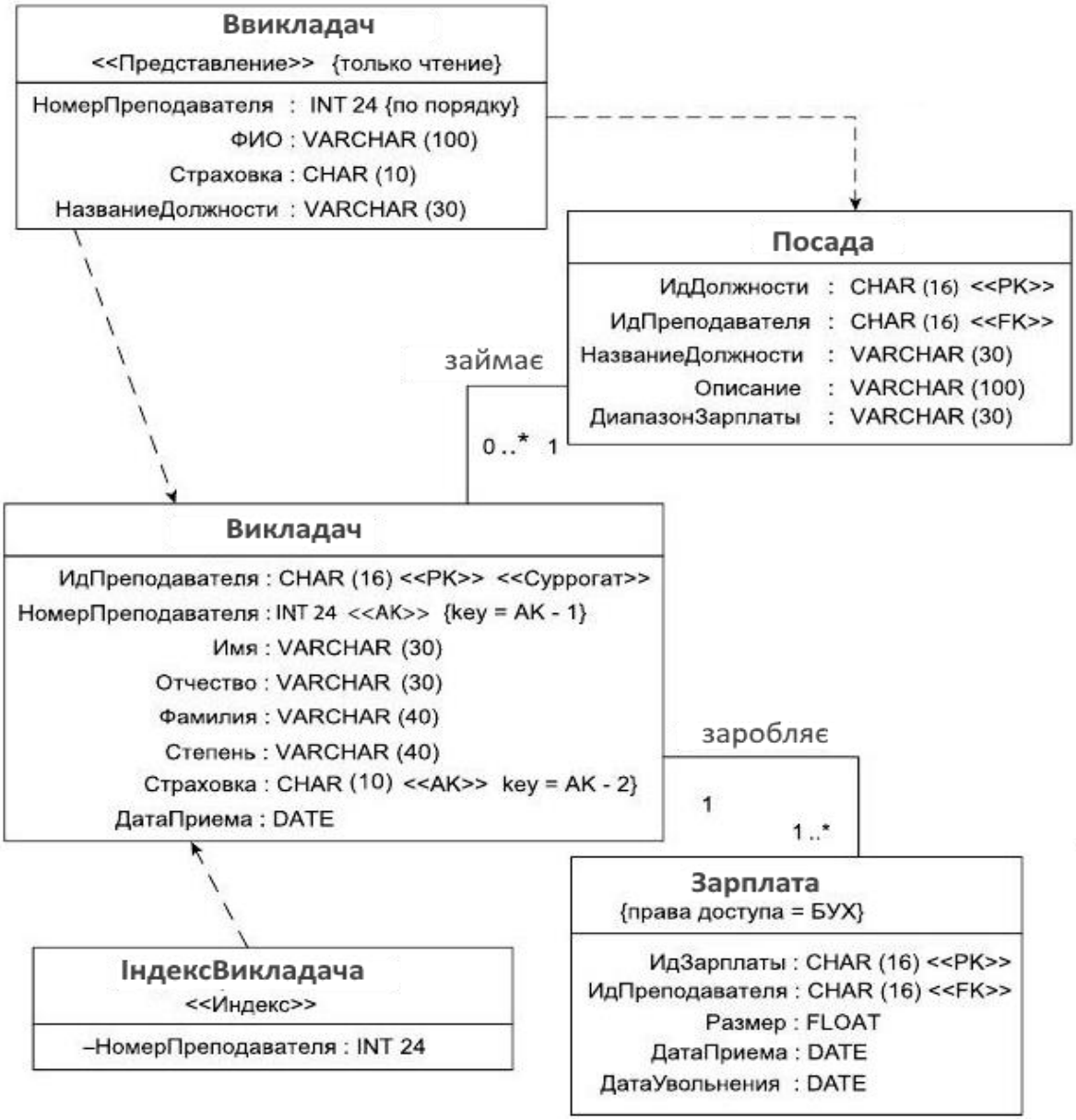


Рисунок 13.2 – Фізична модель даних

Атрибути даних в концептуальних і логічних моделях даних, так само як і стовпці (атрибути) у фізичних моделях даних, моделюються з використанням стандартного позначення атрибуту даних UML. Мається на увазі, що атрибут сутності з логічної моделі автоматично перетвориться у відповідний атрибут (стовпець) таблиці фізичної моделі. Стовпці таблиці розділяють на ключові або неключові. У свою чергу, ключовий стовпець може бути первинним ключем, зовнішнім ключем або ж комбінацією первинного і зовнішнього ключа. Обмеження на значення стовпця вказуються за допомогою звичайних обмежень UML. Мається на увазі, що видимість стовпців завжди задається як публічна.

Відношення між вершинами моделей даних зображаються за допомогою стандартних позначень UML, використовуються різновиди відношень, застосовані до діаграм класів.

13.2.3. Ключі, обмеження, тригери і процедури, що зберігаються

Нагадаємо основні різновиди ключів:

1. Первинний ключ – це стовпець (чи декілька стовпців), що унікально ідентифікує рядок (запис) таблиці.
2. Зовнішній ключ – це стовпець таблиці, що є первинним ключем іншої таблиці і забезпечує зв'язок з цією таблицею. Зовнішній ключ може одночасно бути і первинним ключем власної таблиці.

Тригери і процедури, що зберігаються, – це іменовані блоки коду SQL, які заздалегідь відкомпілювалися і зберігаються на сервері для того, щоб швидко здійснювати обробку запитів, валідацію даних і інші часто виконувані функції.

Процедурою, що зберігається, називається іменований набір команд SQL, що заздалегідь відкомпілювалися, який може викликатися з клієнтського застосування або з іншої процедури, що зберігається.

Тригером називається процедура, яка виконується автоматично як реакція на подію.

Наведемо комплексний приклад, що ілюструє у фізичній моделі ключі, обмеження, тригери і процедури, що зберігаються (рис. 13.3).

Обговоримо цей приклад. По-перше, тут описані такі ключі:

- **ІдЗамовлення –** перший елемент первинного ключа таблиці **ЕлементЗамовлення** і первинний ключ таблиці **Замовлення.** Крім того, цей стовпець є зовнішнім ключем для зв'язку таблиці **ЕлементЗамовлення** з таблицею **Замовлення**;
- стовпець **НаборЕлементівЗамовлення –** другий елемент первинного ключа таблиці **ЕлементЗамовлення**;
- стовпець **ІдЕлементаЗаказу** – другий альтернативний ключ таблиці **ЕлементЗаказу**;
- стовпець **ІдЕлемента –** перший елемент першого альтернативного ключа таблиці **ЕлементЗамовлення**;

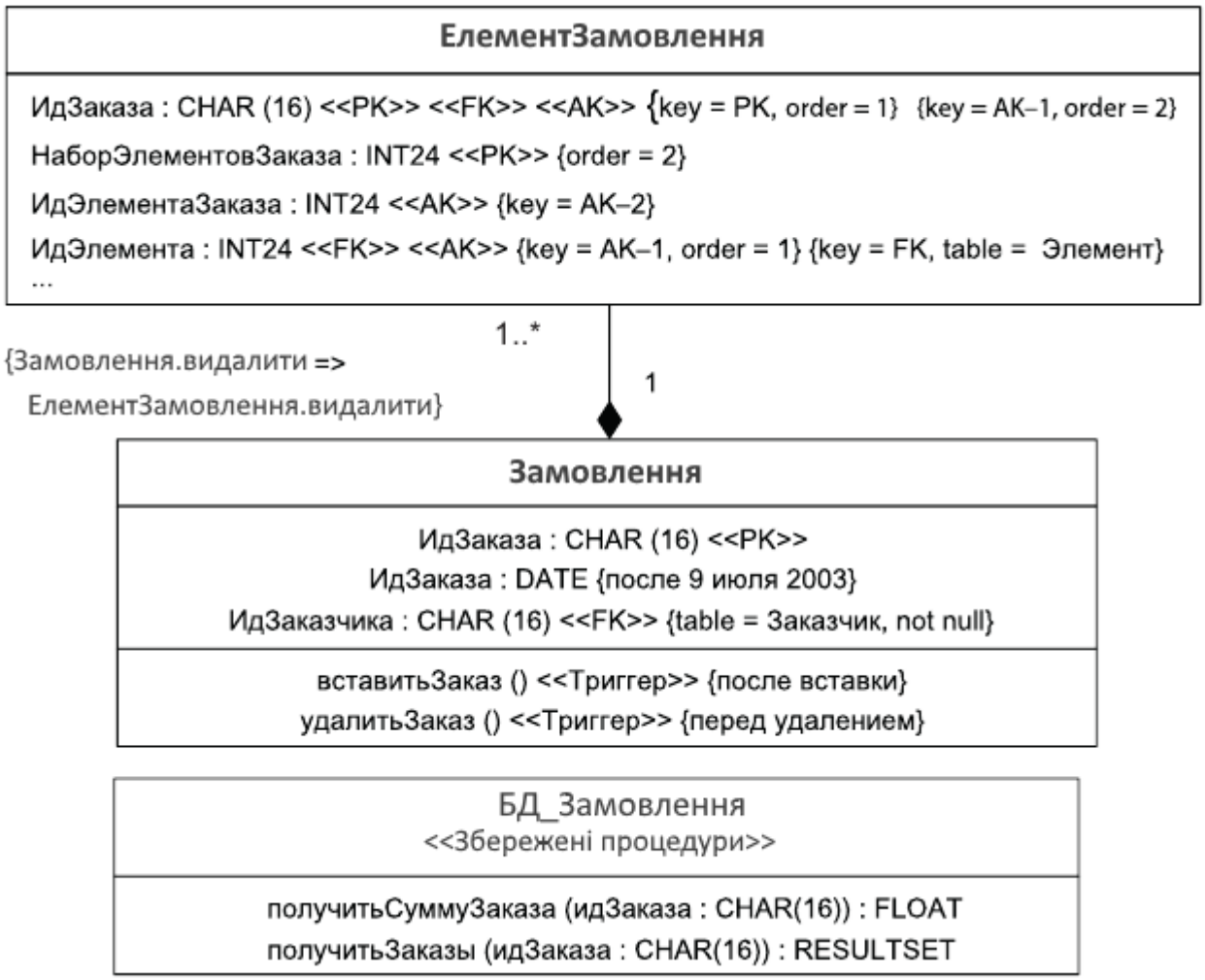


Рисунок 13.3 – Ключі, обмеження, тригери і процедури, що зберігаються, у фізичній моделі даних

- стовпець **ІдЭлемента –** є також зовнішнім ключем до таблиці **Елемент**;
- стовпець **ИдЗамовник а –** зовнішній ключ до таблиці **Замовник**.

По-друге, тут описані такі обмеження:

- для стовпця **ДатаЗаказу** визначено обмеження домена, яке вказує, що це повинно бути пізніше а 9-го липня 2003 року;
- для стовпця **ІдЗамовника** визначено обмеження стовпця, він не має бути нульовим;
- для БД задано обмеження посилальної цілісності між двома таблицями **Замовлення і ЕлементЗамовлення**, воно підписує стрілку композиції. Видно, що при видаленні замовлення елементи замовлення також мають бути видалені.

По-третє, тут показані тригери: використовується позначення для операції із стереотипом **<<Тригер>>.** Умови запуску процедур-тригерів відмічені обмеженнями **{після вставки}** і **{перед видаленням}.**

По-четверте, на рис. 13.3 показані процедури, що зберігаються: використаний окремий клас із стереотипом **<<Хранимые процедури**. Цей клас перераховує сигнатури операцій процедур, що зберігаються, із застосуванням

позначень стандарту UML. Інший підхід полягає в застосуванні стереотипу **<<Хранимая процедура>>** до кожної сигнатури окремої операції. Відмітимо, ім'я цього класу повинне співпадати з ім'ям бази даних або з ім'ям пакету у БД.

На закінчення відмітимо, що в раніше наведеній фізичній моделі (див. рис. 13.2) теж були показані ключі і обмеження. Наприклад, було відмічено, що стовпець **ІдВикладача** є первинним сурогатним ключем. На таблицю **Зарплата** накладено обмеження доступу: тільки співробітникам з відділу **БУХ** дозволяють звертатися до її вмісту. Для представлення **Ввикладач** теж введені обмеження: доступ дозволений тільки для читання, а записи впорядковані за стовпцем **НомерВикладача**.

13.3. Відображення атрибутів об'єктів і класів в реляційну БД

Між технологіями для розробки об'єктно-орієнтованих програмних систем з використанням БД, тобто між об'єктною і реляційною технологіями, існує повна невідповідність. Для подолання цієї невідповідності необхідно розібратися в суті відображення об'єктів в реляційні бази даних і особливостях реалізації цих відображень. Термін «відображення» застосовуватимемо для позначення того, як об'єкти (класи) і їх відношення відображаються в таблиці, що зберігаються, і відношення між ними у базі даних.

Розпочнемо з атрибутів (елементів даних) класу. Атрибут може відобразитися в нуль або декілька стовпців в реляційній базі даних. При чому тут нуль? Річ у тому, що не усі атрибути потрібно запам'ятовувати у БД. Очевидно, що немає потреби зберігати нестійкі атрибути, тобто ті, які використовуються для тимчасових обчислень.

Наприклад, об'єкт класу **Замовлення** може мати атрибут **средняяЦенаПокупки,** яке потрібне в додатку, але не зберігається у базі даних, тому що розраховується при необхідності. Крім того, деякі атрибути об'єктів самі є об'єктами, наприклад об'єкт класу **Замовник** має в якості атрибуту об'єкт класу **Адреса.** Цей факт відображається відношенням між двома класами, яке треба відобразити; мало того, атрибути самого класу **Адреса** теж мають бути відображені. Важливо відмітити, що визначення «атрибут відображається в нуль або декілька стовпців» за своєю суттю рекурсивне.

Найпростіше відображення – це відображення окремого атрибуту в окремий стовпець. Воно стає гранично простим, коли їх базові типи співпадають. Наприклад, атрибут є числом, а стовпець – числом з плаваючою точкою, або тип атрибуту – рядок і тип стовпця – рядок.

Зручно вважати, що класи прямо відображають в таблиці, в стилі «один-в-один», але це далеко не завжди можливо. За винятком дуже простих баз даних, не вдається отримувати класи в таблицях виду «один-до-одного»: втручається механізм спадкоємства і необхідність його відображення. Проте в якості початкового кроку переважне відображення одного класу в одну таблицю.

Розглянемо приклад простого відображення логічної моделі, представленої діаграмою класів, у фізичну модель даних (рис. 13.4). Приклад ілюструє відображення атрибутів класів в стовпці таблиць БД. Наприклад, атрибут **датаВыполнения** класу **Замовлення** відображається в стовпець **ДатаВыполнения** таблиці **Замовлення,** а атрибут **заказанноеКоличество** класу **ЭлементЗамовлення** відображається в стовпець **ЗаказанноеКоличество** таблиці **ЭлементЗамовлення**.

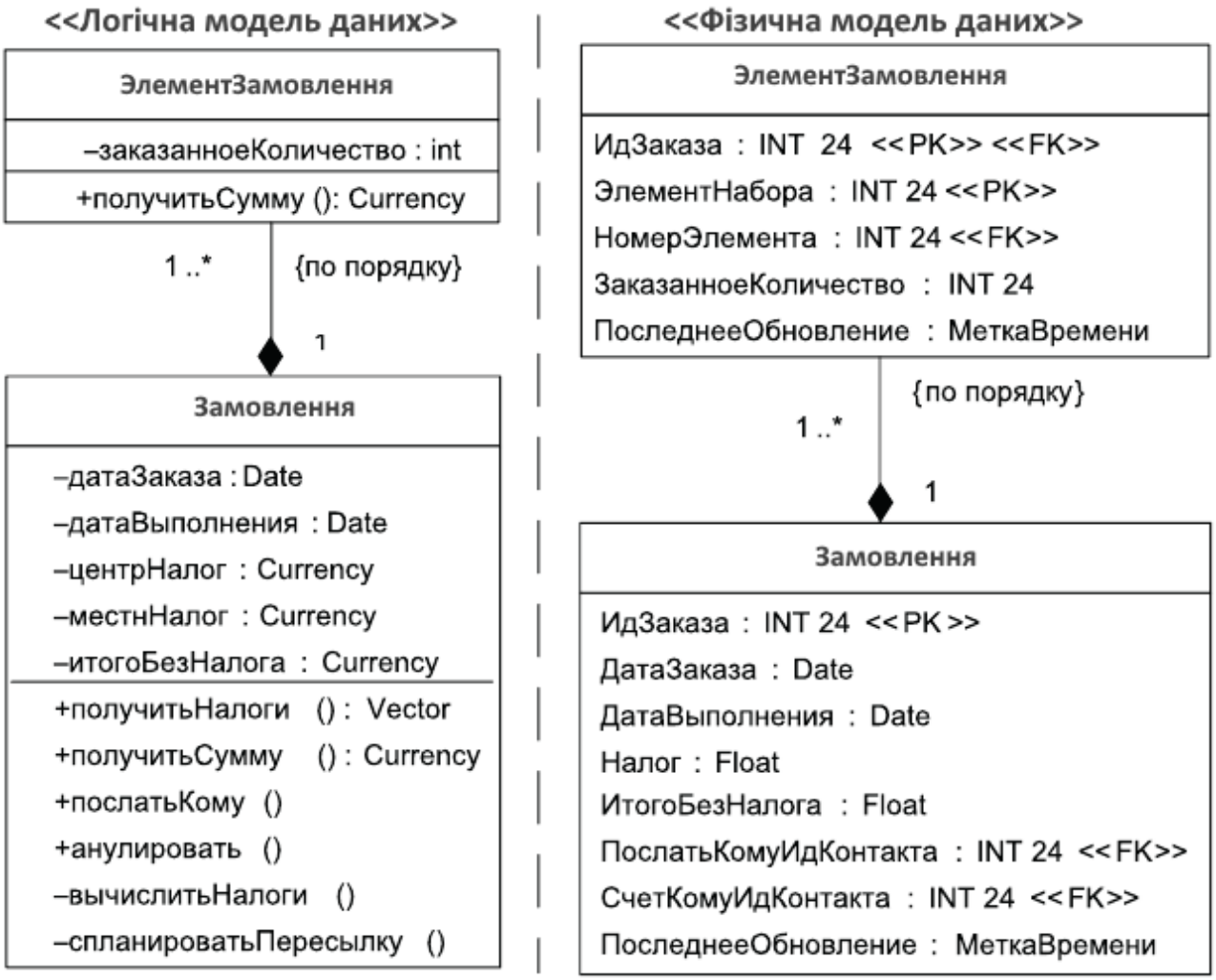


Рисунок 13.4 – Приклад простого відображення атрибутів класів

Та все ж в структурі класів і таблиць є відмінності:

1. У класі є два атрибути для фіксації податку (**центрНалог, местнНалог),** а в таблиці тільки один стовпець **Податок.** Очевидно, що при збереженні об’єкту значення двох атрибутів складаються і заносяться в стовпець **Податок** таблиці.
2. На відміну від класів в таблицях вказані ключі. Рядки в таблицях однозначно визначені первинними ключами, а відношення між рядками встановлюються за допомогою зовнішніх ключів. З іншого боку, відношення між класами реалізуються через посилання на класи, а не через зовнішні ключі. Як наслідок, для повного збереження в замовленні об’єктів і їх відношень об’єкти повинні знати про значення ключів,

використовуваних у базі даних, щоб ідентифікувати їх. Цю додаткову інформацію називають *тіньовою (прихованою) інформацією*.

3. У кожній моделі використовуються різні типи даних. Атрибут **итогоБезНалога** класу **Замовлення** має тип **Currency,** тоді як стовпець **ИтогоБезНалога** таблиці **Замовлення** має тип **Float.** При реалізації цього відображення доведеться забезпечити двостороннє приведення значень (без втрати інформації).

13.4. Відображення дерев спадкоємства в реляційну БД

Суть такого відображення полягає в застосуванні спеціальних прийомів збереження успадкованих атрибутів об'єктів (при їх розміщенні в реляційній БД). Розглянемо відомі методики для відображення спадкоємства.

Відображати будемо просту ієрархію класів, що включає один абстрактний клас ***Особа,*** *у якого є два конкретні клас-спадкоємці* **Студент** і **Службовець** (рис. 13.5). Нагадаємо, що ім'я абстрактного класу записується курсивом. Крім того, приймемо, що у класу **Службовець** також є спадкоємець: клас **Викладач**. Для спрощення можна вважати, що кожен клас має в розпорядженні тільки один власний атрибут.

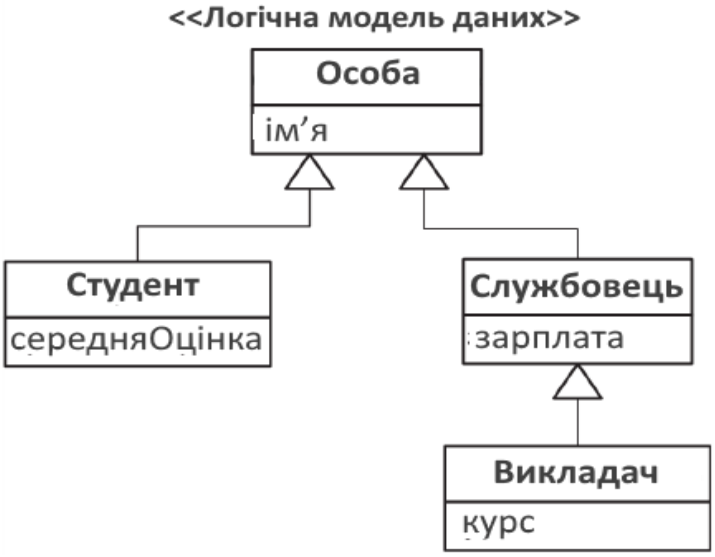


Рисунок 13.5 – Отображаемое дерево успадкування класів

13.4.1. Відображення дерева спадкоємства в єдину таблицю

За цією методикою атрибути усіх класів зберігаються в одній таблиці. Наприклад, результат відображення класів ***Особа,*** **Студент** і **Службовець** може бути представлений у вигляді таблиці **Особа** (рис. 13.6). Відмітимо, що в якості імені таблиці слід використати ім'я кореневого класу дерева спадкоємства.

На рисунку видно, що в таблицю додані два службові стовпці – **ИдОсоби** і **ТипОсоби.** Перший стовпець розглядається як первинний ключ таблиці (сурогатний ключ), він помічений стереотипом **<<PK>>**, а другий стовпець містить код, що вказує, чи являється особа студентом, службовцем або, можливо, і тим і іншим.

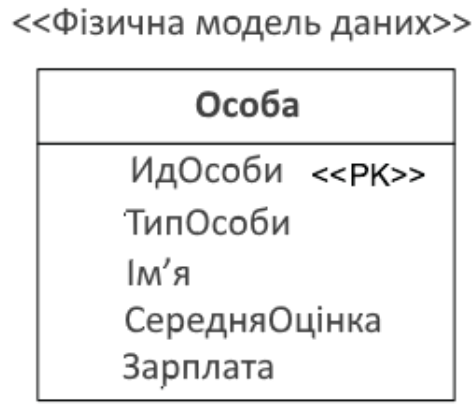


Рисунок 13.6 – Відображення дерева спадкоємства в єдину таблицю

Стовпець **ТипОсоби** зобов’язаний визначити тип об’єкту, що зберігається в певному рядку таблиці. Наприклад, значення «*служ»* означало б, що особа є службовцем, «*студ*» означало б студента, а «*обоє*» означало б обидва типи. Хоча цей підхід простий, він може відмовити у міру росту числа типів і їх комбінацій. Наприклад, при необхідності обліку викладача доведеться додати значення *«виклад»*. Тепер значення *«обоє»*, представляючи тільки два типи, стає безглуздістю.

Крім того, може знадобитися додаткова комбінація із залученням викладача, наприклад, хтось може бути і викладачем, і студентом, так що знадобиться код для цієї комбінації. Більше універсальний підхід, в якому стовпець типу особи замінюється стовпцями для булевих змінних **цеСтудент, цеСлужбовець** і **цеВикладач** (рис. 13.7).

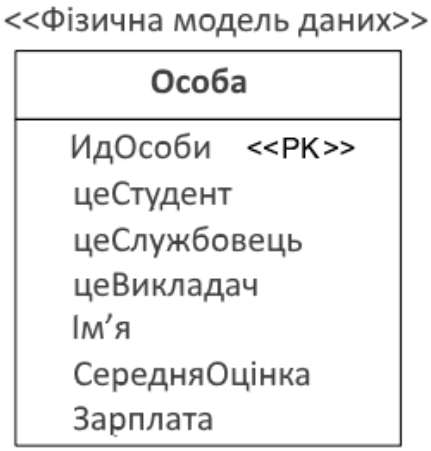


Рисунок 13.7 – Відображення дерева спадкоємства в таблицю з булевими змінними

Сфера застосування методики відображення: доцільно використати для простих і (чи) плоских дерев спадкоємства, в якому відсутнє або мале перекриття між типами дерева.

13.4.2. Відображення кожного конкретного класу в окрему таблицю

За цією методикою таблиця створюється для кожного конкретного класу. Кожна таблиця включає і власні атрибути, реалізовані класом, і його

успадковані атрибути. Відповідна фізична модель даних для нашого дерева спадкоємства включає три таблиці (рис. 13.8).

Рисунок 13.8 – Відображення конкретних класів в таблиці

Тут кожному з конкретних класів **Студент, Службовець** і **Викладач** відповідає таблиця, але немає таблиці для абстрактного класу ***Особа.*** Причина зрозуміла, адже на основі абстрактного класу об'єкти не створюються. Кожній таблиці призначений її власний первинний ключ: **ИдСтудента, ИдСлужбовця** і **ИдВикладача** відповідно.

13.4.3. Відображення кожного класу в окрему таблицю

За цією методикою таблиця створюється для кожного класу. Приклад фізичної моделі даних для нашого дерева спадкоємства включає чотири таблиці (рис. 13.9). Звернемо увагу на зв'язки між таблицями.

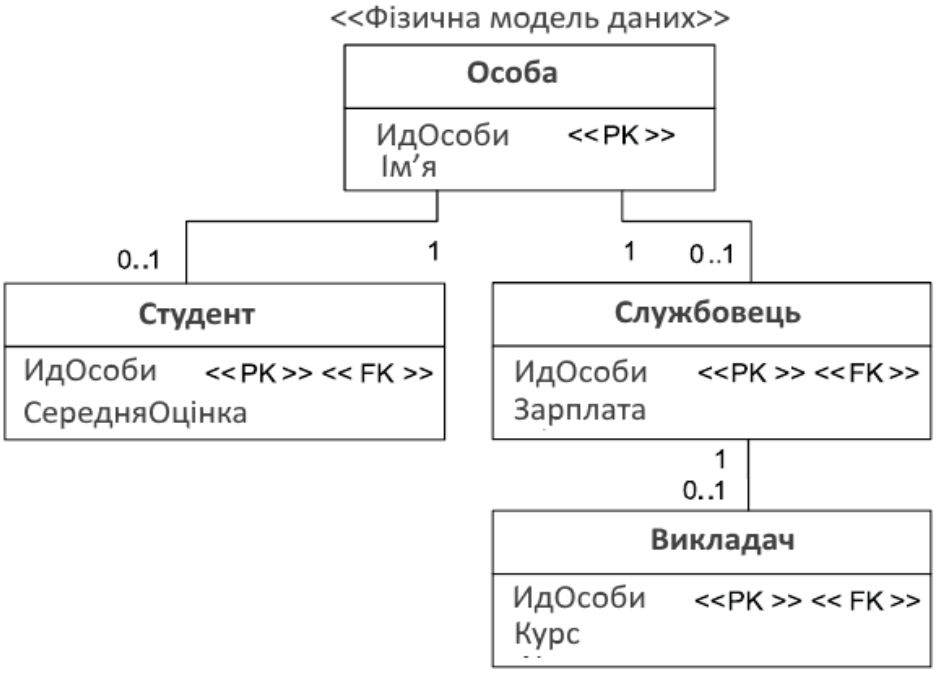


Рисунок 13.9 – Відображення кожного класу в окрему таблицю

Дані для класу **Студент** тепер зберігаються в двох таблицях **(Особа** і **Студент).** Для отримання цих даних потрібно звернутися до двох таблиць, тобто виконати два окремі читання, по одному на кожну таблицю. Відповідно дані класу **Викладач** зберігаються в трьох таблицях **(Особа, Службовець** і **Викладач**).

Обговоримо застосування ключів. Відмітимо, що **ИдЛичности** використовується як первинний ключ в усіх таблицях. Для таблиць **Студент, Службовець** і **Викладач** стовпець **ИдЛичности** є і первинним ключем, і зовнішнім ключем. У разі **Студента ИдЛичности,** його первинний ключ, як зовнішній ключ він використовується для підтримки відношення з таблицею **Особа.** Це позначено застосуванням двох стереотипів, **<<PK>>** і **<<FK>>.**

Контрольні питання до розділу 13

1. Охарактеризуйте відомі моделі баз даних. Сформулюйте їх переваги і недоліки.
2. Яку роль грають первинні ключі таблиці в реляційній базі даних? У чому різниця між природними і сурогатними ключами?
3. Яку роль в реляційних базах даних грають процедури, що зберігаються, і тригери? Яка між ними різниця?
4. Як позначаються в мові UML ключі, обмеження, тригери і процедури, що зберігаються? Назвіть переваги і недоліки цих позначень.

РОЗДІЛ 14. ПРОЕКТУВАННЯ ІНТЕРФЕЙСУ КОРИСТУВАЧА

14.1. Основні правила створення інтерфейсу

Фахівці зазвичай формулюють деякий набір принципів і правил, що дозволяють як оцінювати зручність інтерфейсу, так і пропонувати рішення, що підвищують його зручність. Наведемо ці правила [38].

Правило доступності. Система має бути настільки зрозумілою, щоб користувач, що ніколи раніше не бачив її, але добре розбирається в предметній області, міг без жодного навчання почати її використати. Це правило служить деяким ідеалом, до якого потрібно прагнути, оскільки на практиці досягти такої міри доступності майже ніколи не вдається. Проте, фахівці продовжують робити в цьому напрямі усе можливе.

Правило ефективності. Система не повинна перешкоджати ефективній роботі досвідчених користувачів, що працюють з нею довгий час. Очевидним прикладом порушення цього правила є націленість системи тільки на новачків, використання засобів, які добре підходять для недосвідченого користувача, обмежуючи його в можливості зробити щось не так, але неефективні для експерта, який і так знає, що і де йому треба зробити.

Правило безперервного розвитку. Система повинна сприяти безперервному росту знань, умінь і навичок користувача і пристосовуватися до його досвіду, що міняється. Погані результати приносить надання тільки базових можливостей або залишення початкуючого користувача наодинці із складним інтерфейсом, яким упевнено користуються експерти. Порушення безперервності при переході від одного набору можливостей до іншого також приносить незручності, оскільки користувач вимушений розбиратися з доданими можливостями в новому контексті.

Більшість користувачів можна розділити на три групи: новачки, досвідчені і середні, які вже знають більше, ніж новачки, і не роблять стільки помилок, але ще не придбали автоматизму при виконанні більшості операцій. Новачкам потрібна допомога в освоєнні нової для них системи і контроль за їх діями, досвідченим користувачам. Про середніх же користувачів часто забувають, хоча переважна більшість користувачів програмного забезпечення належать саме до цієї категорії. Їм потрібні досить висока ефективність і гнучкість разом з можливістю швидко отримувати адекватну допомогу з різноманітних питань, що виникають час від часу.

Правило дотримання контексту. Система має бути погоджена з контекстом, в якому їй належить працювати. Це правило вимагає від системи бути працездатною не «взагалі», а саме в тому оточенні, в якому нею користуватимуться. У контекст можуть входити специфіка і об’єми вхідних і вихідних даних, тип і цілі організацій, в яких система повинна працювати, рівень користувачів, зашумленість приміщень тощо.

14.2. Принципи розробки інтерфейсу користувача

Представлені вище правила визначають загальні вимоги, яким повинен задовольняти зручний інтерфейс. Наступні принципи дозволяють знаходити рішення, що підвищують зручність призначеного для користувача інтерфейсу [38].

Принцип структуризації. Призначений для користувача інтерфейс має бути доцільно структурований. Близькі за змістом, споріднені його частини мають бути пов'язані видимим чином, а незалежні – розділені; схожі елементи повинні виглядати схоже, а несхожі – розрізнятися.

Принцип простоти. Найпоширеніші операції повинні виконуватися максимально просто. При цьому мають бути видимі посилання на складніші процедури.

Принцип видимості. Усі функції і дані, що необхідні для розв'язання певного завдання, мають бути видні, коли користувач намагається її вирішити.

Принцип зворотного зв'язку. Користувач повинен отримувати повідомлення про дії системи і про важливі події всередині неї. Повідомлення мають бути інформативними, короткими, однозначними і написаними мовою, зрозумілою користувачеві.

Принцип толерантності. Інтерфейс має бути гнучким і терпимим до помилок користувача. Збиток від помилок повинен знижуватися за рахунок можливості відміни і повтору дій і за рахунок розумної інтерпретації будь-яких розумних дій користувача і введених їм даних. По можливості слід уникати взаємодії (модальних діалогів) заснованої на обмеженні свободи користувача.

Принцип повторного використання. Слід намагатися багаторазово використати внутрішні і зовнішні компоненти, забезпечуючи тим самим уніфікованість інтерфейсу і схожість між його схожими елементами.

Розглянемо деякі питання розробки ергономічного інтерфейсу користувача.

14.3. Взаємодія між користувачем і комп'ютером

Людиномашинний інтерфейс забезпечує зв'язок між користувачем і комп'ютером, він дозволяє досягати поставлених цілей, успішно знаходити розв'язання поставленої задачі. Взаємодія – обмін діями і реакціями на ці дії між комп'ютером і користувачем. Існує ряд стилів взаємодій, які підрозділяються на два види [38].

1. Використання інтерфейсу мови команд – введення команд текстовими засобами.
2. Безпосереднє маніпулювання.

Таким чином, є ряд способів, якими користувач міг би зв'язуватися з комп'ютером:

- мови команд – користувач управляє системою, вводячи відповідні команди в текстовому режимі;

– питання і відповідь – діалог, де комп’ютер ставить питання, а користувач відповідає йому (чи навпаки);
– форми – користувач заповнює форми або поля діалогу, вводячи дані у відповідні поля;
– меню – користувач забезпечений рядом опцій і управляє системою, вибираючи необхідні пункти;
– пряме маніпулювання – користувач управляє об’єктами на екрані за допомогою пристрою маніпулювання типу миші.

У програмній системі, що розробляється, також застосований комплексний підхід до створення інтерфейсу. Тут використовується пряме маніпулювання, меню, форми і діалоги.

Мета створення ергономічного інтерфейсу полягає в тому, щоб відобразити інформацію настільки ефективно, наскільки це можливо для людського сприйняття, і структурувати відображення на дисплеї так, щоб притягнути увагу до найбільш важливих одиниць інформації. Основна ж мета в тому, щоб мінімізувати загальну інформацію на екрані і представити тільки те, що є необхідним для користувача.

14.4. Розміщення інформації на екрані

Кількість інформації, що відображається на екрані, називається *екранною щільністю*. Дослідження показали, що чим менше екранна щільність, тим інформація, що відображається, доступніша і зрозуміліша для користувача, і навпаки, якщо екранна щільність велика, це може викликати труднощі в засвоєнні інформації і її ясному розумінні. Проте досвідчені користувачі можуть віддавати перевагу інтерфейсам з великою екранною щільністю. Інформація на екрані може бути згрупована і впорядкована в значимі частини. Це може бути досягнуто використанням кадрів (фреймів), методів типу колірного кодування, рамок, негативного зображення або інших методів для привертання уваги.

Виділення елементів інтерфейсу яскравістю. Для привертання уваги до яких-небудь елементів інтерфейсу можна скористатися виділенням цих елементів більшою яскравістю на тлі інших, темніших. Проте важливо не перестаратися з цим методом, оскільки велика кількість яскравих елементів здатна викликати дискомфорт у користувача і можна отримати зворотний ефект – перевантаження інтерфейсу. Застосовувати цей метод слід тільки при необхідності. Існує декілька способів виділення яскравістю:
– рух (миготіння або зміна позиції) – ефективний метод;
– яскравість – не дуже ефективний метод, оскільки люди здатні розрізнити лише декілька рівнів яскравості;
– колір – використання може бути надзвичайно ефективним;
– форма (символ, шрифт, форма символу) використовується для того, щоб диференціювати різні категорії даних;
– розмір (тексту, символів) – зазвичай застосовують збільшення виділеного об’єкту в 1,5 рази;

– відтінення (різна текстура об’єктів) – ефективний метод для привертання уваги до якої-небудь частини екрану;
– оточення (підкреслення, рамки, інвертоване зображення) – дуже ефективний спосіб.

Використання кольору при проектуванні ергономічного інтерфейсу. Колір може поліпшити інтерфейс, але для багатьох систем використання кольору практично не впливає на ефективність роботи користувача. Основне призначення кольору – створення інтерфейсів, цікавіших для користувачів; проте є випадки, коли колір може допомогти проектувальникові інтерфейсу. Особливо це ефективно при групуванні інформації, виділенні відмінностей між інформацією або простими повідомленнями (помилки, стани тощо).

Колір – потужний візуальний інструмент, і застосовувати його потрібно дуже обережно, щоб не викликати у користувача дискомфорт від помилкових колірних комбінацій. Перерахуємо деякі принципи використання кольору, якими треба керуватися при проектуванні ергономічного інтерфейсу:

– необхідно обмежити число кольорів до 4 на екрані і до 7 – для послідовності екранів;
– для неактивних елементів краще використати бліді кольори;
– при відображенні стану, як правило, червоний означає небезпеку (стоп), зелений – продовження роботи, жовтий – колір попередження;
– для привертання уваги найефективніші білий, жовтий і червоний кольори;
– при необхідності впорядкування даних (чи розділення даних) можна використати спектр 7 кольорів (веселка);
– для угрупування даних, об’єднання і подібності треба використати сусідні кольори спектру: помаранчево-жовті, синьо- фіолетові.

Важливо відмітити, що близько 9% людей не розрізняють кольорів (зазвичай, червоно-зелені поєднання); проте ці люди можуть відрізняти чорно-білі відтінки, тому проектувальники автоматизованих систем повинні перевіряти, чи не порушує використання різних кольорів в інтерфейсах сприйняття користувачів цієї категорії [38].

Несуперечність і стандартизація. Дані на екрані слід розташовувати так, щоб користувач знав, де знайти і де чекати виведення необхідної інформації:

– інформація, на яку слід негайно звернути увагу, повинна завжди відображатися на видному місці, щоб захопити увагу користувача (наприклад, застережливі повідомлення і повідомлення про помилки);
– не дуже часто потрібна інформація (наприклад, довідка) не повинна відображатися, але має бути доступна за потреби. Наприклад, іконка «довідки» або відповідна опція меню має бути доступна на кожному екрані.

Тексти і діалоги. Наведемо деякі принципи, якими необхідно керуватися при створенні текстових діалогів і відображень:

- текст в нижньому регістрі читається приблизно на 13% швидше, ніж текст, надрукований повністю у верхньому регістрі;
- символи верхнього регістру найефективніші для передачі інформації, яка повинна привернути увагу. Не використовуйте верхній регістр, якщо ви не хочете виділяти яку-небудь інформацію;
- вирівняний по правому краю текст важче читати, чим рівномірно розподілений текст з невирівняним правим полем;
- оптимальний інтервал між рядками рівний або трохи більше, чим висота символів.

Меню. Необхідний елемент автоматизованої системи – меню, що дозволяє користувачеві виконувати завдання усередині додатка і управляти процесом рішення. Меню – набір опцій, що відображаються на екрані, де користувачі можуть вибирати і виконувати дії, тим самим роблячи зміни в стані інтерфейсу. Перевага меню в тому, що користувачі не повинні пам'ятати назву елементу або дії, яку вони хочуть виконати, вони повинні тільки розпізнати його серед пунктів меню.

Таким чином, меню може використовувати навіть недосвідчений користувач. Проте проект меню має бути ретельно продуманий – щоб меню було ефективним, назви його пунктів мають бути очевидними.

Меню може займати багато екранного місця, але вирішенням цієї проблеми може бути використання спливаючого або спадаючого меню, яке викликається клацанням по піктограмі, рядку меню або іншому об'єкту.

Основні принципи створення меню. В процесі проектування системи меню додатка необхідно прийняти найкращий спосіб відображення меню, щоб воно було зрозумілим і легким у використанні. Зазвичай команди меню впорядковані деяким ієрархічним способом. Основна проблема полягає в тому, щоб правильно розподілити різні пункти меню по різних рівнях і правильно їх згрупувати.

Принципи проектування меню:

- структура меню повинна відповідати структурі завдання, що розв'язується системою; організація меню повинна відобразити найефективнішу послідовність кроків, що ведуть до розв'язання поставленої задачі;
- пункти меню мають бути короткими, граматично правильними і відповідати своєму заголовку. Порядок пунктів меню вибирається відповідно до угоди, частоти і порядку використання, а також залежно від потреб завдання або користувача;
- вибір пунктів меню має бути забезпечений декількома способами – за допомогою клавіатури, за допомогою миші і через інші об'єкти призначеного для користувача інтерфейсу. Важливо зафіксувати поєднання клавіш, що легко запам'ятовуються, для швидшого доступу до пунктів меню, оскільки це дуже економить час.

Форми. Форми – основний елемент інтерфейсу. Призначення форм – зручне введення і перегляд даних, стану, повідомлень автоматизованої системи.

Основні принципи проектування форм:

- форма проектується для зручнішого, зрозумілого і швидкого рішення поставленої задачі. Якщо форма переноситься з паперової форми, то пересування по суміжних полях не повинне викликати утруднень у користувача;
- розміщення інформаційних одиниць на просторі форми повинне відповідати логіці її майбутнього використання: це залежить від необхідної послідовності доступу до інформаційних одиниць, частоти їх використання, а також від відносної важливості елементів;
- логічні групи елементів необхідно відділяти пропусками, рядками, колірними або іншими візуальними засобами;
- взаємозалежні або пов’язані елементи повинні відображатися в одній формі.

При розробці форм необхідно продумати і вказати, які кнопки в смузі системного меню мають бути доступні в тому або іншому вікні, чи повинне вікно допускати зміни користувачем його розміру, яким має бути заголовок вікна.

Без особливої необхідності не треба робити вікна зі змінюваними розмірами. При зміні розмірів, якщо не застосовані спеціальні прийоми, порушується компонування вікна, що негативно позначається на роботі користувача. Має сенс створити вікно зі змінюваними розмірами, якщо це дозволяє користувачеві змінювати корисну площу розташованих в нім компонентів відображення і редагувати інформацію: текст, зображення, списки тощо. Якщо проектується змінюване вікно, то необхідно вжити заходи, щоб компоненти у вікні при цьому теж змінювали свої розміри або місце розташування, рівномірно розподіляючись по площі вікна і не залишали порожніх місць.

При проектуванні форм необхідно прагнути до використання обмеженого набору кольорів і приділяти увагу їх правильному поєднанню. Для фону форми обираються нейтральні кольори (світло-сірі). Колір не повинен використовуватися як основний засіб передачі інформації, потрібно вибирати системні кольори, які користувач може перебудовувати на свій розсуд.

Елементи, що управляють, і функціонально пов’язані з ними компоненти екрану слід зорово об’єднувати в групи, заголовки яких коротко і чітко пояснюють їх призначення. Кожне вікно повинне мати деяку центральну тему, яка підпорядковується його композиції. Користувач повинен розуміти, для чого призначено це вікно і що в нім найважливіше. Неприпустимо перевантажувати вікно великим числом елементів управління введення і відображення інформації.

Також неприпустимо, щоб схожі за функціями органи управління в різних вікнах називалися по-різному або розміщувалися в різних місцях вікон. Важливо потурбуватися про те, як додаток впишеться в загальну організацію робочого простору системи і як воно взаємодіятиме з іншими застосуваннями.

Дизайн заголовків і полів. Для окремих полів заголовок має бути вирівняний по лівому краю; для полів списків заголовок має бути вищий і

лівіший по відношенню до основного поля; числові поля вирівнюються по правому полю.

Довгі колонкові поля або довгі стовпці інформаційних одиниць з поодинокими полями необхідно об'єднувати в групи по п'ять елементів, що розділяються порожнім рядком, – це допомагає користувачеві подумки обробляти інформацію по виділених групах.

У формах з великою кількістю інформації важливо використати назви розділів, які однозначно свідчать про характер інформації, що належить їм. Необхідно чітко розділити відображення заголовків і безпосередньо полів введення. Заголовки мають бути короткими, знайомими і змістовними.

Поля, що необов'язкові для заповнення або не мають особливої важливості, повинні відрізнятися візуально (кольором або іншими ефектами) від полів важливих і обов'язкових для заповнення.

Формати введення. Слід забезпечити введення значень за замовчуванням в усі поля, які це допускають і де така функція не дратуватиме користувача. Можна призначити клавіші або коди для введення значень, що часто повторюються. Вхідні дані мають бути значимими і загальноприйнятими. Не слід об'єднувати поля введення чисел і символів, оскільки числові і алфавітні клавіші знаходяться незручно один відносно одного на клавіатурі.

Необхідно виключити часте перемикання між верхнім і нижнім регістрами для прискорення введення даних. Не можна вимагати від користувача введення незначимих цифр (наприклад, замість 00000010 користувач повинен ввести тільки 10). Аналогічно, не можна вимагати від користувача ввести інформацію, яка була заздалегідь введена або яка може бути автоматично отримана з системи. Бажано використати значення за замовчуванням, щоб мінімізувати процес введення інформації.

Організація системи навігації і системи відображення станів. Навігація забезпечує користувачеві можливість переміщення між різними екранами, інформаційними одиницями і підпрограмами в автоматизованій системі. У повноцінній системі користувач завжди може отримати інформацію про стан системи, про процес виконання або активну підпрограму.

Загальні принципи проектування. Існує ряд навігаційних засобів і прийомів, які допомагають користувачеві орієнтуватися в системі. Вони включають використання заголовків сторінок для кожного екрану, номерів сторінок, рядків і стовпців, відображення поточного імені файлу вверху екрану.

Тип системи навігації залежить від прийнятого стилю інтерфейсу. Для інтерфейсів мови команд існує дуже мало способів забезпечення повноцінної навігації. У інтерфейсах з меню можна використати ієрархічно структуроване меню. Діалогові інтерфейси самі по собі захищають користувача від помилкових дій. Інформація стану зазвичай відображається внизу екрану і містить в собі дані про кількість записів, число оброблених одиниць, процес друку, черги друку і так далі

Проектування повідомлень. Повідомлення потрібні для спрямування дій користувача в потрібну сторону, підказок і попереджень при виконанні необхідних дій на шляху розв'язання задачі. Вони також включають

підтвердження дій з боку користувача і підтвердження з боку системи, що завдання виконані успішно або з якихось причин не виконані. Повідомлення можуть бути виведені у формі діалогу, екранних заставок і тому подібне. Повідомлення можуть запропонувати користувачеві:

- вибрати із запропонованих альтернатив опцію або набір опцій;
- ввести інформацію;
- вибрати опцію з набору опцій, які можуть змінюватися залежно від поточного контексту;
- підтвердити фрагмент введеної інформації перед продовженням введення.

Повідомлення можуть бути поміщені в модальні діалогові вікна, які змушують користувача відповісти на питання перш, ніж почне виконуватися будь-яка інша дія. Це може бути корисно, коли система змушена змусити користувача обдумати рішення перед продовженням роботи. Немодальні діалогові вікна дозволяють працювати з іншими елементами інтерфейсу, тоді як саме вікно може ігноруватися.

14.5. Запобігання, виявлення і виправлення помилок

Звичайна людина у нормальному стані здійснює багато помилок різного роду. Можна сказати, що людина, на відміну від комп'ютера, є адаптивною аналоговою системою і успішність її «функціонування» в набагато більшому ступені визначається не точністю виконання дій і формулювання думок, а здатністю швидко видати хороше наближення до потрібного результату і досить швидко виправитись, якщо це необхідно.

Помилки користувача можуть бути засновані на неправильному розумінні дії або порядку дій або бути випадковими, неумисними, наприклад, друкарська помилка при введенні тексту.

Помилки другого виду можуть бути розділені ще на шість підвидів:

- неточність у виборі опції (наприклад, користувач випадково натиснув кнопку «Вихід» і програма закрилася);
- втрата активності, коли користувач забуває необхідну послідовність дій для продовження роботи;
- помилка режиму або стану, коли користувач думає, що він знаходиться в одному стані, а фактично – в іншому; наприклад, режим вставки замість режиму друку поверх тексту в текстовому процесорі.

Користувач завжди робитиме помилки навіть у відмінній програмній системі, тому в системі, що розробляється, завжди має бути передбачений захист від помилок. Техніка такого захисту включає такі аспекти:

- примусові дії в системі, які запобігають або утрудняють появу помилок;
- забезпечення хороших і інформативних повідомлень про помилки;
- забезпечення нормальної діагностики системи, в процесі якої користувачеві пояснюється, в чому суть помилки, і вказуються шляхи її виправлення.

Розглянемо основні принципи обробки помилок у формах введення:

- забезпечення можливості посимвольного редагування введених записів для виправлення помилок введення (друкарських помилок);
- якщо помилка виявлена системою, бажано повернути курсор в поле з помилковими даними і яким-небудь чином виділити це поле;
- виводити значимі повідомлення про помилки, що використовують стиль мови користувача і відповідну термінологію;
- виводити повідомлення про помилки, які пояснюють і пропонують шляхи їх усунення.

Ефективність запобігання і подолання помилок користувачів тим вище, чим рідше користувачі помиляються при роботі з цим інтерфейсом і чим менше часу і зусиль вимагається для подолання наслідків вже зроблених помилок.

Контрольні питання до розділу 14

1. Перерахуйте загальні принципи і правила створення зручного інтерфейсу.
2. Як використовується колір при розробці ергономічного інтерфейсу?
3. Перерахуйте основні вимоги до компонування форм.
4. Як можна запобігти появі помилок при введенні?
5. Як правильно створити меню?

РОЗДІЛ 15. ВИБІР СТРАТЕГІЇ ТЕСТУВАННЯ І РОЗРОБКА ТЕСТІВ

15.1. Рівні тестування

Тестуванням називається виконання програми з метою виявлення помилок. Відладкою називається локалізація і виправлення помилок.

Модульне тестування є процесом перевірки окремих програмних процедур (модулів) і підпрограм, що входять до складу програм або підпрограмних систем. Модульне тестування робиться безпосереднім розробником і дозволяє перевіряти усі внутрішні структури і потоки даних в кожному модулі.

Інтеграційне тестування проводиться для перевірки спільної роботи окремих модулів і передує тестуванню усієї системи як єдиного цілого. В ході інтеграційного тестування перевіряються зв'язки між модулями, їх сумісність і функціональність. Воно здійснюється незалежним тестувальником і входить до складу етапу тестування. Елементи інтеграційного тестування:

- перевірка функціональності – перевірка відповідності окремих функцій, що виконуються сукупностями модулів, функціям, заданим в специфікаціях вимог;
- перевірка проміжних результатів – перевірка усіх проміжних результатів і файлів на наявність і коректність;
- перевірка інтеграції – перевірка коректності взаємної передачі модулями інформації.

Системне тестування призначене для перевірки програмної системи в цілому, її організації і функціонування на відповідність специфікаціям вимог замовника. Його проводить незалежний тестувальник після успішного завершення інтеграційного тестування.

Елементи системного тестування:

- граничне тестування – тестування в граничних умовах;
- прогоночне тестування – тестування усіх функціональних характеристик реальної роботи системи;
- цільове тестування – тестування на цільовій платформі;
- перевірка документації – перевірка призначеної для користувача документації на коректність;
- інші тести, що визначаються тестувальником.

Вихідне тестування – завершальний етап тестування, на якому перевіряється готовність ПП для постачання замовникові. Цей вид тестування проводить незалежний тестувальник.

Приймальне тестування проводиться організацією, що відповідає за інсталяцію, супровід програмної системи і навчання кінцевого користувача.

15.2. Технології тестування

Технологія тестування, яка застосовується на етапі розробки програмного забезпечення, називається тестуванням «скляного ящика» («білого ящика») в протилежність класичному поняттю «чорного ящика».

При тестуванні «чорного ящика» програма розглядається як об'єкт, внутрішня структура якого невідома. Тестувальник вводить дані і аналізує результат, але він не знає, як саме працює програма.

При тестуванні «скляного ящика» ситуація абсолютно інша. Тестувальник (в даному випадку сам програміст) розробляє тести, грунтуючись на знанні початкового коду, до якого він має повний доступ. В результаті він отримує певні переваги.

1. Спрямованість тестування. Програміст може тестувати програму частинами, розробляти спеціальні тестові підпрограми, які викликають тестований модуль і передають йому дані, що цікавлять програміста. Окремий модуль набагато легше протестувати саме як «скляний ящик».

2. Повне охоплення коду. Програміст завжди може визначити, які саме фрагменти коду працюють в кожному тесті. Він бачить, які ще гілки коду залишилися непротестованими, і може підібрати умови, в яких вони будуть протестовані. Нижче описано, як відстежувати міру охоплення програмного коду проведеними тестами.

3. Можливість управління потоком команд. Програміст завжди знає, яка функція повинна виконуватися в програмі наступною і яким має бути її поточний стан. Щоб з'ясувати, чи працює програма так, як він думає, програміст може включити в неї налагоджувальні команди, що відображають інформацію про хід її виконання, або скористатися для цього спеціальним програмним засобом, що називається відладчиком.

4. Можливість відстежування цілісність даних. Програмістові відомо, яка частина програми повинна змінювати кожен елемент даних. Відстежуючи стан даних (за допомогою того ж відладчика), він може виявити такі помилки, як зміна даних не тими модулями, їх невірна інтерпретація або невдала організація. Програміст може самостійно автоматизувати тестування.

Тестування «скляного ящика» – частина процесу програмування. Програмісти виконують цю роботу постійно, вони тестують кожен модуль після його написання, а потім ще раз після інтеграції його в систему. При виконанні модульного тестування можна використати технологію або структурного, або функціонального тестування, або одночасно і ту, і іншу.

15.3. Програмні помилки

Усі програмні помилки можна віднести до відповідних категорій. Розглянемо ті, що зустрічаються найчастіше.

1. Функціональні недоліки властиві програмі, якщо вона не робить того, що повинна, виконує одну зі своїх функцій погано або не повністю. Функції програми мають бути детально описані в її специфікації, і саме на основі затвердженої специфікації тестувальник будує свою роботу.

2. Недоліки призначеного для користувача інтерфейсу. Оцінити зручність і правильність роботи призначеного для користувача інтерфейсу можна тільки в процесі роботи з ним. Бажано, щоб в цій роботі брав участь сам користувач.

3. Недостатня продуктивність. При розробці програмного продукту дуже важливою його характеристикою може виявитися швидкість роботи, іноді цей критерій задається у вимогах замовника. Погано, якщо у користувача створюється враження, що програма працює повільно.

4. Некоректна обробка помилок. Процедури обробки помилок – дуже важлива частина програми. Правильно визначивши помилку, програма повинна видати про неї повідомлення.

5. Некоректна обробка граничних умов. Існує багато різних граничних ситуацій. Будь-який аспект роботи програми, до якого застосовані поняття «більше» або «менше», «раніше або пізніше», «коротше» або «довше», обов'язково має бути перевірений на межах діапазону.

6. Помилки обчислень. Сюди належать помилки, викликані неправильним вибором алгоритму обчислень, неправильними формулами. Найпоширенішими є помилки округлення.

7. Помилки управління потоком. За логікою роботи програми услід за першою дією повинна бути виконана друга. Якщо замість цього виконується третя або четверта дія, значить, в управлінні потоком припустилися помилки.

8. Перевантаження. Збої в роботі програми можуть відбуватися із-за нестачі пам'яті або відсутності інших необхідних системних ресурсів.

Контрольні питання до розділу 15

1. Дайте визначення технологіям тестування «білого ящика» і «чорного ящика».
2. Перерахуйте основні види тестування.
3. Що означає ситуація перегонів? Яким чином її можна протестувати?
4. Що є програмною помилкою? Які категорії програмних помилок ви знаєте?

ЧАСТИНА 5. ПРИКЛАДИ МОДЕЛЮВАННЯ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ ІС

У будь-якій людській діяльності щодня працюють з моделями. Справа в тому, що модель – спрощене представлення реальності. Моделі «з висоти пташиного польоту» наочно демонструють бажану структуру і поведінку системи (зазвичай візуально). Моделі дозволяють краще зрозуміти загальну картину – ескіз майбутнього рішення. При необхідності ескіз доповнюється деталями. Так формується остаточний спосіб вирішення проблеми. Зазвичай моделювання включає дві дії: аналіз і проектування. Модель аналізу покращує розуміння вимог до ПО, а модель проектування показує ескіз структури і поведінки ПО, що виконує ці вимоги.

У цій частині посібника на конкретних прикладах (інформаційна система відділу кадрів і система реєстрації для ВНЗ) будуть розглянуті усі стадії при проектуванні ПЗ з використанням мови UML (моделювання використання, моделювання структури і моделювання поведінки) [12; 22]. Розробка і використання моделей мови UML здійснюється у рамках загальної концепції об’єктно-орієнтованого аналізу і проектування.

РОЗДІЛ 16. МОДЕЛЮВАННЯ ВИКОРИСТАННЯ ІС ВІДДІЛУ КАДРІВ

Моделювання використання покликане відповісти на питання: «що корисного робить система у зовнішньому світі»?, і це питання – одне з перших, на які необхідно дати відповідь.

У трьох розділах цієї частини посібника розглядатимемо один наскрізний приклад моделювання порівняно нескладного додатка – **інформаційної системи відділу кадрів** (ВК). Вибір прикладу обумовлений наступними міркуваннями.

По-перше, предметна область до певної міри знайома усім. Таким чином, суть завдання заздалегідь ясна, і можна зосередити увагу на тонкощах застосування UML, а не на поясненні особливостей предметної області.

По-друге, інформаційна система відділу кадрів – це типове офісне застосування з найпоширенішого класу систем автоматизації діловодства. UML якнайкраще підходить для моделювання саме таких систем.

Потрібно відмітити, що метою розгляду цього конкретного прикладу є ілюстрація можливостей мови UML. Може здатися, що послідовність розгляду прикладів є протоколом реалізації процесу моделювання, в цілому це не так. Приклад розглядається в порядку викладу різних конструкцій мови UML. Розглядати процес побудови моделі безвідносно стадій (фаз) процесу розробки було б неправильно.

Отже, поставимо себе на місце розробника і припустимо, що в нашому розпорядженні є такий текст, що поступив від замовника.

ТЕХНІЧНЕ ЗАВДАННЯ
*Інформаційна система «Відділ кадрів» (скорочено **ІС ВК**) призначена для введення, зберігання і обробки інформації про співробітників і рух кадрів. Система повинна забезпечувати виконання таких основних функцій:* *1. Прийом, переведення і звільнення співробітників.* *2. Створення і ліквідація підрозділів.* *3. Створення вакансій і скорочення посад.*

Звичайно, «технічне завдання» з одного абзацу тексту і трьох нумерованих пунктів – це не більше ніж навчальний приклад. Проте навіть на цьому прикладі видно багато характерних «особливостей» подібних документів, які, занадто часто зустрічаються в реальному житті. З одного боку, щось написане, а з іншого боку не дуже зрозуміло, що робити далі.

Без всяких пояснень замовник використовує терміни своєї предметної області – розробник повинен їх знати і розуміти. Вимог до реалізації немає зовсім. Функції не впорядковані за пріоритетами: не ясно, що є критично важливим, а чим можна поступитися у разі потреби. Загалом, можна розкритикувати це «технічне завдання».

Проте, яким би неповним не було це технічне завдання, після отримання відповіді на питання «хто винен»?, встає питання «що робити»?.

1. Можна заявити, що це технічне завдання нікуди не годиться і відмовитися працювати із замовником, який його представив. Це просте і надійне рішення, але...
2. Можна зажадати уточнити технічне завдання, включивши в нього відповіді на усі питання розробника. Це в усіх відношеннях ідеальний варіант, тільки, на жаль, так не буває.
3. Можна, нарешті, спробувати щось зробити самим з наявним матеріалом.

Виберемо третій шлях, маючи на меті показати, як, **застосовуючи UML, можна поступово перетворити розпливчатий опис додатка на цілком чітку модель, придатну для реалізації.**

16.1. Значення моделювання використання

При першому знайомстві з діаграмами використання в UML у розробників програмного забезпечення, особливо у досвідчених розробників, часто виникає питання: навіщо це треба? При цьому такого питання відносно інших засобів UML не виникає, оскільки відповідь на нього у більшості випадків очевидна аналогічно. Дійсно, розглянемо декілька прикладів типових асоціацій, що виникають у розробників при першому знайомстві з UML.

Діаграми діяльності – це не що інше, як блок-схеми алгоритмів. Розробник програм, особливо зі стажем, прекрасно розуміє призначення і межі застосовності блок-схем. Питання «навіщо»? просто не виникає, виникають інші питання: чи варто використати блок-схеми в цьому проекті, яку з альтернативних систем позначень застосовувати і тому подібні.

Діаграми станів – це розширення концепції кінцевих автоматів, які є основним апаратом в цілому ряду областей програмування: транслятори, логічне управління і інші. Якщо в конкретній предметній області, наприклад, в області конструювання відповідальних систем управління, наказано застосовувати кінцеві автомати, то розробники застосовують їх, не запитуючи, «навіщо»?.

Діаграми класів, в яких показані атрибути і операції класів, а також взаємозв'язки (асоціації) між ними, дуже схожі на діаграми «сутність-зв'язок» (ERD – Entity – Relationship Diagram), добре відомі розробникам додатків баз даних.

Перелік асоціацій, що виникають у розробників, можна продовжувати, для більшості засобів UML неважко відшукати аналоги серед широко використовуваних практичних методів конструювання програмних систем. І це не дивно, адже саме ці методи були уніфіковані за допомогою UML. А ось для діаграм використання відомий аналог вказати важче. Спробуємо пояснити прагматику моделювання використання на конкретному прикладі.

16.1.1. Підходи до моделювання

Переваги моделювання використання UML спробуємо виявити, порівнюючи його з іншими підходами. Відомо багато різних підходів до моделювання і подальшого проектування, тобто рекомендацій, що робити після отримання технічного завдання. Розглянемо три з них і спробуємо вказати ті підводні камені, які моделювання використання дозволяє обійти.

Найзаслуженішими є, мабуть, такі **методи структурного проектування**.

1. ***Програмування зверху вниз*** – це узагальнююча назва для модульного програмування без оператора GoTo методом покрокового уточнення.

При програмуванні зверху вниз процес полягає в наступному. Початкове завдання розбивається на підзадачі до тих пір, поки кожна окрема підзадача не стане настільки простою, що її реалізація стає очевидною.

Парним до програмування «зверху вниз» є ***програмування «від низу до верху»***, при якому рівень мови програмування підвищується (наприклад, за допомогою визначення модулів) до тих пір, поки він не стане настільки високим і близьким до початкового завдання, що її реалізація стане очевидною.

2. ***Програмування вшир*** – це коли, починаючи з найпершого кроку, створюється і на усіх подальших кроках підтримується працездатна версія програми. В термінах програмування зверху вниз це означає проведення одного шляху в дереві послідовних уточнень до кінця (стовбур дерева), а потім поступове нарощування дерева вшир. У результаті такого підходу структура додатка виявляється відповідною структурі команди розробників, а не початковому завданню.

Якщо, наприклад, в розробці інформаційної системи відділу кадрів буде задіяно двох провідних розробників, то буде виділено дві основні підсистеми, а якщо три, то і підсистем виявиться три. При покрокової деталізації таке виділення підсистем майже неминуче.

Розглянемо другий підхід до моделювання. Всякому ясно, що інформаційна система відділу кадрів – цей **додаток баз даних**.

При проектуванні додатків баз даних перший крок добре відомий: після отримання вимог треба скласти схему бази даних, тобто визначити склад таблиць бази і полів в таблицях, призначити первинні ключі в таблицях і встановити зв'язки між таблицями за допомогою зовнішніх ключів.

Серед десятка знайомих інформаційних систем відділу кадрів приблизно в половині випадків (у тому числі в системах, що комерційно продаються) номер позиції в штатному розписі (особовий рахунок) зроблений ключем таблиці співробітників. На перший погляд таке рішення допустиме: номер позиції в штатному розписі унікальний, однозначно відповідає співробітникові і може бути використаний в якості ключа (на відміну від прізвища співробітника, яке цілком може не бути унікальним). Проте повний аналіз усього застосування показує, що це груба проектна помилка, що породжує проблеми на пізніших стадіях розробки і явну неефективність в роботі системи. Щоб зрозуміти, в чому помилка, треба розглянути операцію «переведення співробітника», яка призводить до зміни номера позиції в штатному розписі, тобто первинного ключа в записі співробітника.

3. Нарешті, розглянемо наймодніший **об'єктно-орієнтований підхід** до моделювання. Захисники цього підходу перший крок проектування описують приблизно так: треба виділити ***словник предметної області*** (тобто набір основних понять), зіставити цим поняттям класи проектованої системи, визначити їх атрибути і операції і далі усе піде як належить.

16.1.2. Переваги моделювання використання

Наша мова і мислення влаштовані так, що найпростішою, зрозумілішою і чіткішою формою викладу думок є так звані прості твердження.

Просте твердження має таку граматичну форму: ***належний – присудок – пряме доповнення.*** Чи, в логічних термінах, ***суб'єкт – предикат – об'єкт.*** Наприклад***:*** *начальник звільняє співробітника, директор створює відділ*. Звичайно, використання такої форми не гарантує від помилок, але завдяки простоті і наочності форми їх легше помітити.

Моделювання використання безпечніше і надійніше за альтернативні методи, які були розглянуті у розділі 10, тобто за інших рівних умов дозволяє вчинити менше грубих проектних помилок на ранніх стадіях проектування. У цьому полягає основна перевага цього методу.

16.2. Діаграми використання

Діаграми використання являються, безумовно, найстабільнішим елементом UML – вони не мінялися і, фактично, набули закінченої форми задовго до появи мови. Одночасно ці діаграми мають найпростішу нотацію: всього два основні типи сутностей (*дійові особи і варіанти використання) і три типи відношень (залежності, асоціації, узагальнення*).

16.2.1. Дійові особи

Розглянемо наш приклад з інформаційною системою відділу кадрів. Важко уявити собі організацію, в якій реорганізація внутрішньої структури і процес найму персоналу виконуються автоматично, без участі людини, тому у нашої системи, очевидно, будуть користувачі.

Спираючись на поради, які дані у розділі 10, до нашого прикладу ми в першому наближенні схильні виділити дві категорії користувачів:

1) менеджер персоналу, який працює з конкретними людьми;
2) менеджер штатного розпису, який працює з абстрактними посадами і підрозділами.

Бізнес-процес користувача першої категорії включає не лише роботу з додатком, але і бесіди з конкретними людьми.

Користувачі другої категорії, очевидно, повинні мати спеціальні права доступу, оскільки навряд чи допустимо, щоб хто завгодно міг створювати і знищувати підрозділи на підприємстві.

На наступному фрагменті діаграми використання починаємо формувати представлення використання інформаційної системи відділу кадрів (рис. 16.1). Менеджер персоналу має ім'я **Personnel Manager,** а менеджер штатного розпису **– Staff Manager**, відповідно до використовуваної дисципліни імен.

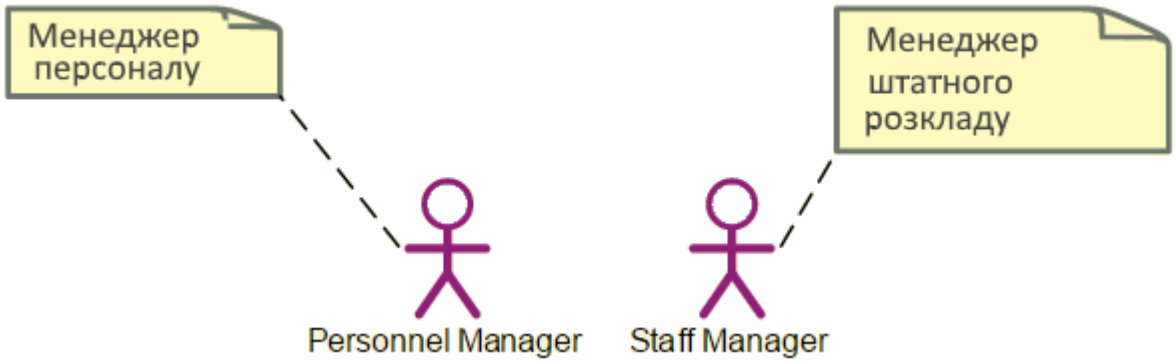


Рисунок 16.1 – Дійові особи ІС ВК

Для UML поки що немає досить сталої дисципліни імен, але деякий набір рекомендацій можна знайти в літературі. Ми, по можливості, будемо дотримуватись цих рекомендацій. Зокрема, в якості імен дійових осіб рекомендується використати ***іменник*** (можливо, з визначальним словом), а в якості імен варіантів використання **– *дієслово*** (можливо, з доповненням). Ці правила засновані на семантиці моделювання використання.

16.2.2. Варіанти використання

У нашому прикладі простий аналіз тексту технічного завдання (наведеного в параграфі 16.1.1) виявляє сім варіантів використання:

1) прийом співробітника;
2) переведення співробітника;
3) звільнення співробітника;
4) створення підрозділу;

5) ліквідація підрозділу;
6) створення вакансії;
7) скорочення посади.

Спираючись на знання предметної області, яке не відбите в технічному завданні (характерний випадок), помітимо, що термін «вакансія» є скороченням обороту «вакантна посада», тобто посада в деякому особливому стані. Саме ж слово «посада» багатозначне. Це може бути і позначення конкретного робочого місця – позиції в штатному розписі, і позначення сукупності таких позицій, що мають загальні ознаки: функціональні обов'язки, зарплата і т. п.

Наприклад, «в організації розрізняються посади: програміст, аналітик, керівник проекту» або «у відділі розробки передбачені такі посади: 9 програмістів, 3 аналітики і 2 керівники проектів». Кадрові працівники легко розрізняють ці випадки по контексту. Приймемо робочу гіпотезу про те, що автор технічного завдання використав слово «посада» в першому сенсі і отримаємо набір варіантів використання, представлений на рис. 16.2.

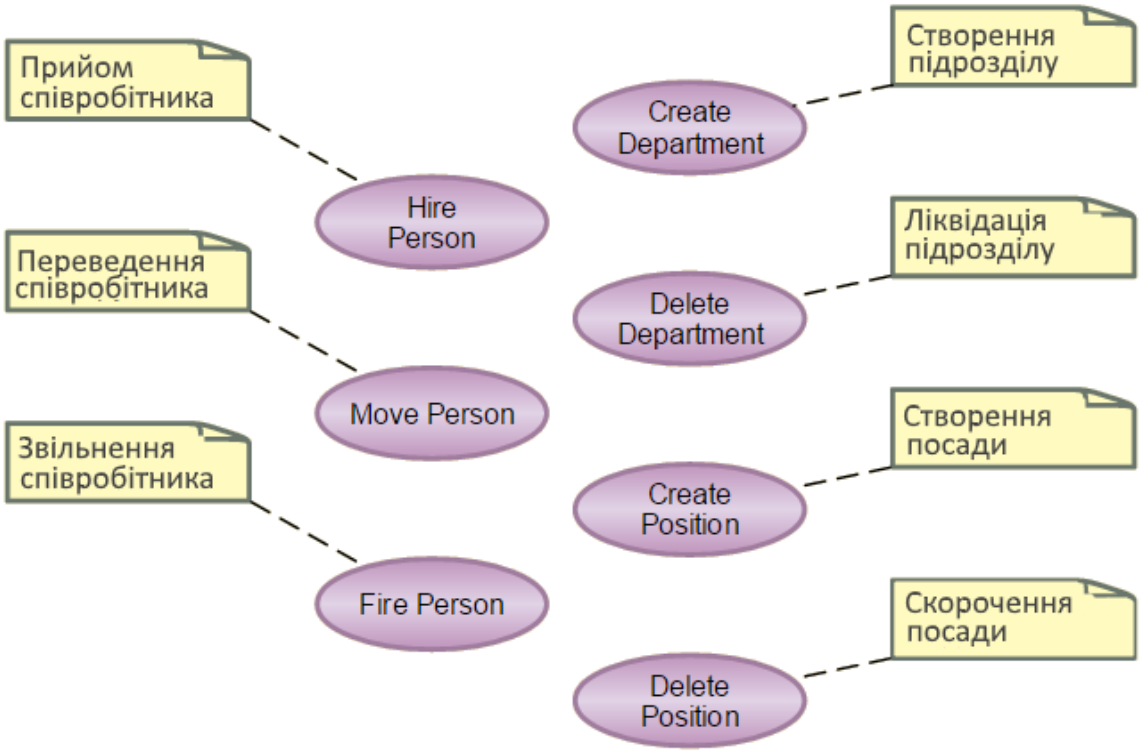


Рисунок 16.2 – Варіанти використання ІС ВК

16.2.3. Коментарі

У UML послідовно проводиться такий важливий принцип: уся інформація, яку користувач вносить в модель, має бути збережена інструментом у внутрішньому уявленні моделі і пред'явлена на першу вимогу, навіть якщо інструмент не уміє обробляти цю інформацію. Коментарі є найважливішим прикладом реалізації цього принципу.

Повертаючись до нашого прикладу, буде зовсім не зайвим указати, що інформацію про стан кадрів треба зберігати постійно, тобто вона не повинна зникати після завершення сеансу роботи з системою (рис. 16.3).

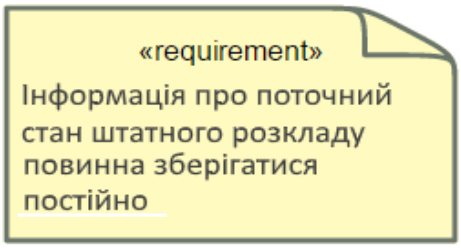


Рисунок 16.3 – Нефункціональна вимога до ІС ВК

16.2.4. Відношення на діаграмах використання

Як уже було відмічено, на діаграмах використання застосовуються такі основні типи відношень:

- асоціація між дійовою особою і варіантом використання;
- узагальнення між дійовими особами;
- узагальнення між варіантами використання;
- залежності між варіантами використання.

Асоціація між дійовою особою і варіантом використання показує, що дійова особа тим або іншим способом взаємодіє (надає початкові дані, отримує результат) з варіантом використання.

Стосовно нашого прикладу в першому наближенні можна позначити асоціації, представлені на діаграмі показаній на рис. 16.4.

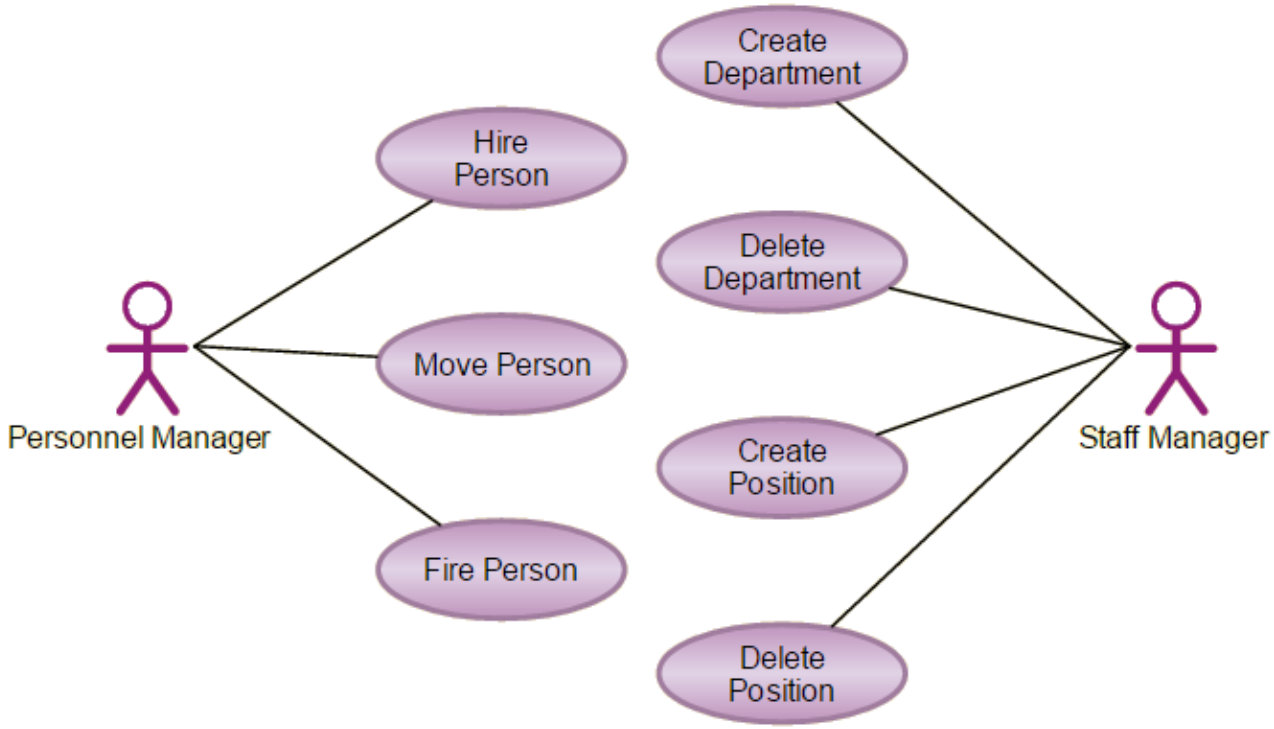


Рисунок 16.4 – Асоціації між дійовими особами і варіантами використання

Узагальнення між дійовими особами показує, що одна дійова особа наслідує усі властивості (зокрема, участь в асоціаціях) іншої дійової особи.

Таке узагальнення є дуже потужним засобом моделювання.

По-перше, за допомогою узагальнення між дійовими особами легко показати ієрархію категорій користувачів системи, зокрема, ієрархію прав доступу до виконуваних функцій і даних, що зберігаються.

ЗМІНИ В ТЕХНІЧНОМУ ЗАВДАННІ

Серед усіх користувачів інформаційної системи слід виділити особливу категорію користувачів (вище керівництво), якій дозволений доступ до будь-яких даних і операцій.

Цю зміну у вимогах можна відобразити в моделі системи так, як показано на рис. 16.5.

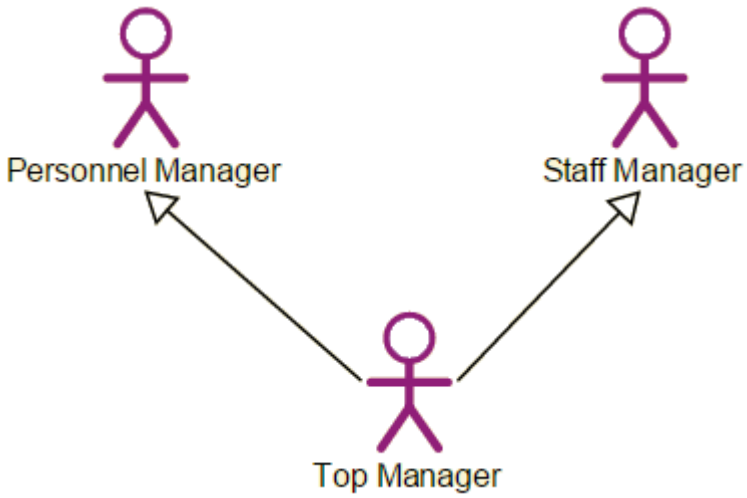


Рисунок 16.5 – Ієрархія категорій користувачів ІС ВК

По-друге, дійова особа, будучи класифікатором, може бути абстрактним класифікатором, тобто таким класифікатором, який не може мати безпосередніх екземплярів. Введення абстрактних дійових осіб дозволяє без втрати інформації скоротити кількість безпосередніх асоціацій в моделі, зробивши її лаконічнішою, а значить наочнішою і кориснішою.

ЗМІНИ В ТЕХНІЧНОМУ ЗАВДАННІ

Інформаційна система повинна надавати можливість переглядати дані без внесення в них яких-небудь змін.

Цю вимогу слід оформити у вигляді додаткового варіанту використання – **Browse.** Якщо ми проектуємо систему відділу кадрів для звичайної організації, а не для державної секретної служби, то розумно припустити, що переглядати дані можуть усі категорії користувачів. В цьому випадку можна встановити асоціації між новим варіантом використання і усіма дійовими особами, а можна поступити так, як показано рис. 16.6, тобто ввести узагальненого абстрактного користувача **User 1,** який буде пов’язаний асоціацією з варіантом використання

Browse 2. При цьому усі спеціалізації 3 і 4 узагальненого користувача автоматично будуть пов'язані асоціацією з варіантом використання **Browse**.

Узагальнення між варіантами використання показує, що один варіант використання є частковим випадком (підмножиною множини сценаріїв) іншого варіанту використання.

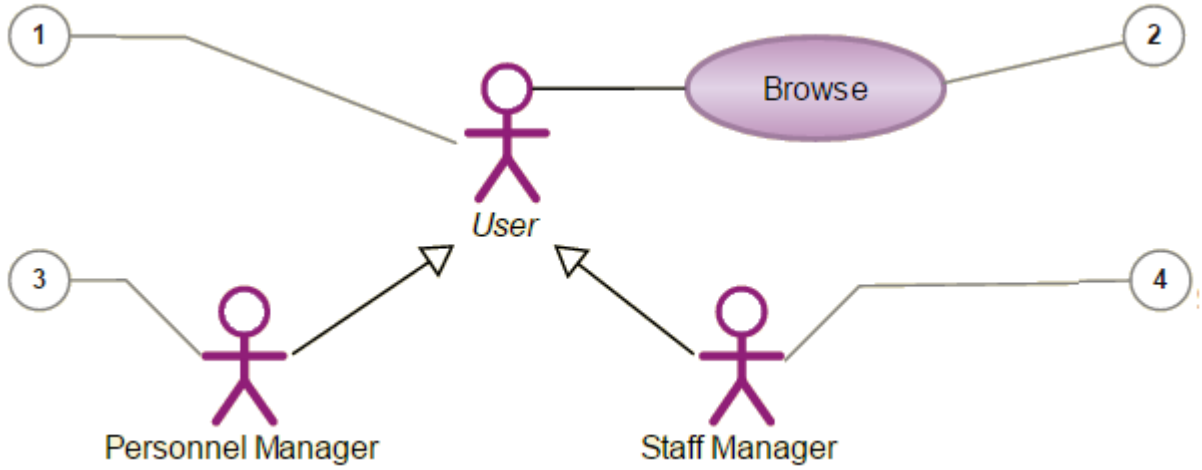


Рисунок 16.6 – Абстрактна дійова особа

Узагальнюючий варіант використання, будучи класифікатором, може бути абстрактним класифікатором. Наприклад, такий важливий для співробітника варіант використання, як звільнення, насправді є абстракцією: у кожному конкретному випадку має місце рівно один з можливих окремих випадків звільнення, які призводять до одного і тому ж результату з точки зору менеджера персоналу, але дуже різні з точки зору співробітника.

ЗМІНИ В ТЕХНІЧНОМУ ЗАВДАННІ

Система повинна підтримувати два способи звільнення співробітника: за ініціативою адміністрації і за власним бажанням.

Цю обставину можна відбити в моделі так, як показано на наступному фрагменті діаграми (рис. 16.7).

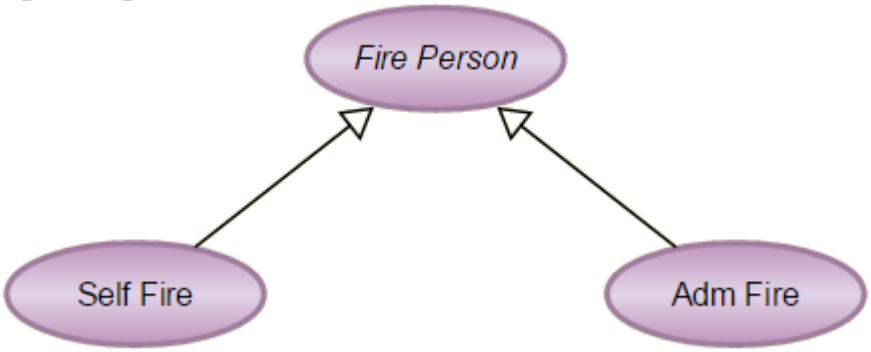


Рисунок 16.7 – Узагальнення варіантів використання

Узагальнений абстрактний (ім'я написане курсивом) варіант використання ***Fire Person*** має дві спеціалізації, які відповідають звільненню

працівника за власним бажанням **Self Fire** і за ініціативою адміністрації **Adm Fire**.

Залежність між варіантами використання показує, що один варіант використання залежить від іншого варіанту використання.

У UML є два стандартні стереотипи залежності між варіантами використання, які в деякому сенсі двоїсті одна одній:

1. **«include» –** показує, що в кожен сценарій залежного варіанту використання у визначеному місці вставляється в якості підпослідовності дій в сценарій незалежного варіанту використання.
2. **«extend» –** показує, що в деякий сценарій незалежного варіанту використання може бути у визначеному місці вставлений в якості підпослідовності дій сценарій залежного варіанту використання. Інший варіант інтерпретації: у визначеному місці сценарію незалежного варіанту використання викликається для виконання сценарій залежного варіанту використання. При цьому послідовність дій в сценарії, що викликається, визначається місцем, звідки він був викликаний, тобто з якого варіанту використання.

На перший погляд не дуже зрозуміло, чим відрізняється семантика цих залежностей – адже обидві вони відображають відношення включення для послідовностей дій.

ЗМІНИ В ТЕХНІЧНОМУ ЗАВДАННІ

При звільненні співробітника має бути здійснена виплата грошової компенсації за невикористану відпустку. У разі вимушеного скорочення можлива виплата вихідної допомоги.
Обліковий запис співробітника при звільненні має бути заблокований.

Відмітимо, що звільнення співробітника – один з найскладніших варіантів використання в реальних системах управління персоналом. Отже, нам відомо, що при звільненні співробітника слід з метою інформаційної безпеки заблокувати (чи видалити) обліковий запис користувача в локальній мережі організації. Причому ця дія має бути виконаною у будь-якому сценарії звільнення.

З іншого боку, як сказано в технічному завданні, при виконанні певних умов при звільненні іноді виплачується деяка грошова компенсація (за невикористану відпустку, вихідна допомога при скороченні і так далі).

Усе це приклади послідовностей дій (тобто варіантів використання), які цілком можуть бути затребувані як при звільненні, так і окрім нього. Відношення залежності між цими варіантами використання можуть бути показані на діаграмі використання, наприклад, так, як це зроблено на рис. 16.8.

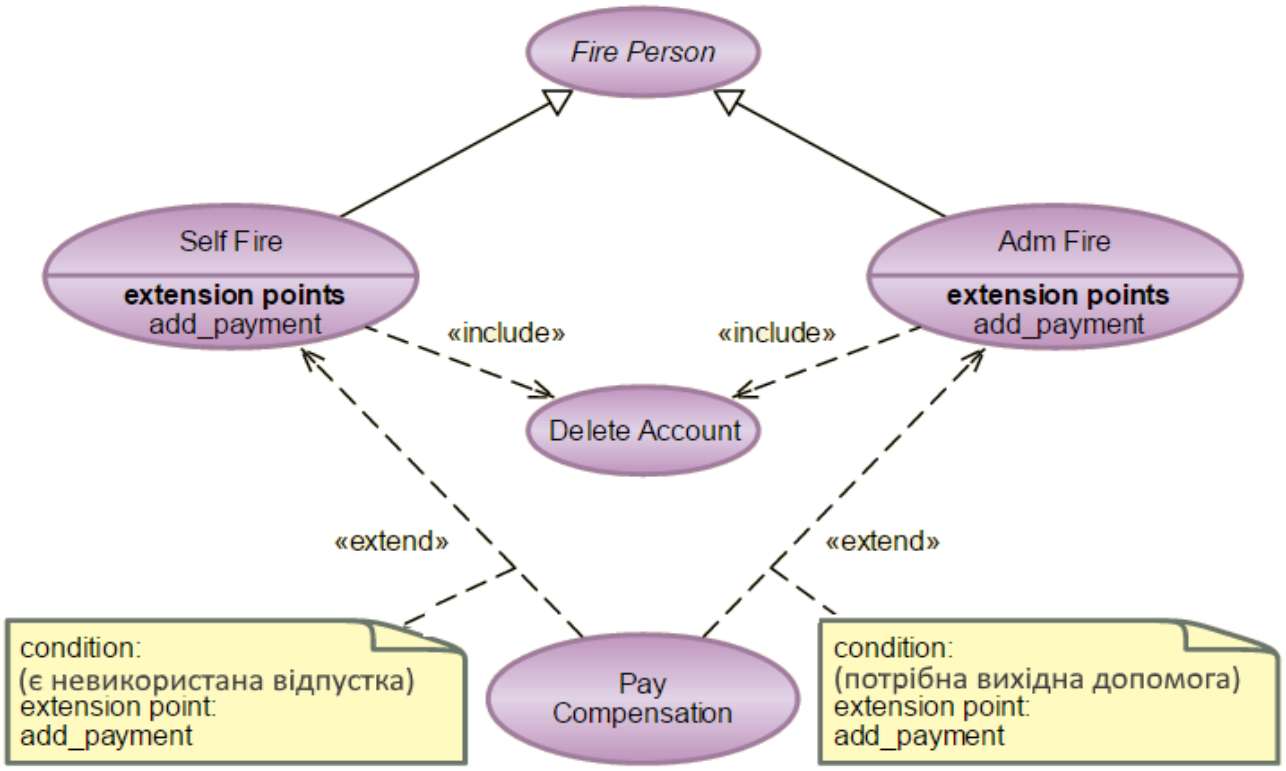


Рисунок 16.8 – Залежності між варіантами використання

Останній приклад можна відобразити ще компактніше (рис. 16.9), комбінуючи можливості відношень узагальнення і залежності, подібно до того, як скомбіновані можливості відношень узагальнення і асоціації на рис. 16.6 **Абстрактна дійова особа**.

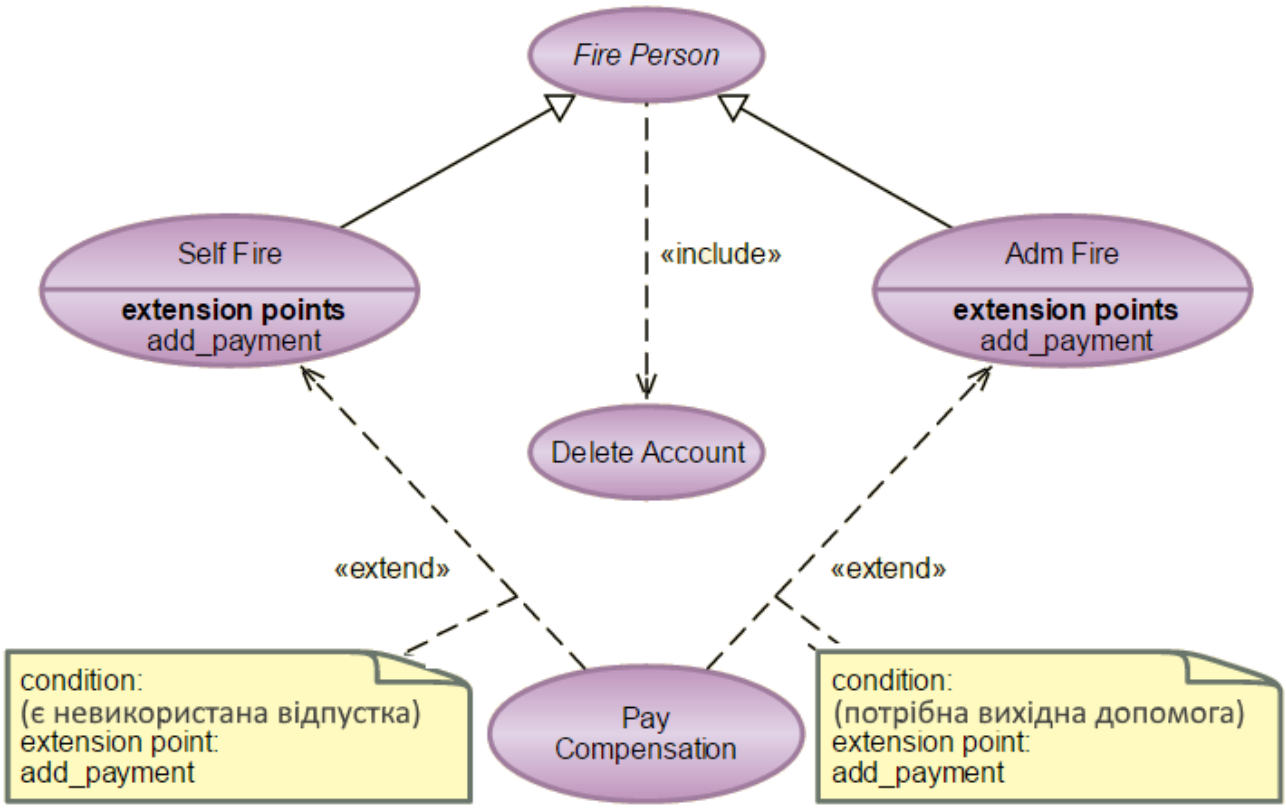


Рисунок 16.9 – Комбінація відношень узагальнення і залежності

16.2.5. Способи застосування моделей використання

Якщо подивитися на модель використання з найзагальнішої точки зору, то неважко помітити, що в моделі є присутніми:

- внутрішня модельована система (у нашому випадку ІС ВК), у формі набору варіантів використання, можливо пов'язаних залежностями і узагальненнями;
- зовнішнє оточення, у формі набору дійових осіб, можливо пов'язаних узагальненнями;
- зв'язок між модельованою системою і зовнішнім оточенням у формі асоціацій між дійовими особами і варіантами використання.

Зазвичай абсолютно ясно, що знаходиться усередині модельованої системи, а що зовні. Якщо це чого-небудь неясно, або ж вимагається збільшити наочність діаграм, то можна скористатися спеціальною конструкцією, яка називається «Межі системи».

Межі системи **–** це графічний коментар у формі прямокутної рамки, що вживається на діаграмах використання і відділяє внутрішню частину системи від її зовнішнього оточення. Внутрішня частина, що виділяється межами, має в UML конкретну назву – суб'єкт.

Суб'єкт **(subject) –** це класифікатор, який реалізує поведінку, що декларується варіантами використання.

Якщо межі системи використовуються на діаграмі, то можна вказати ім'я (і стереотип), які належатимуть до суб'єкта.

На наступній діаграмі (рис. 16.10) ми побудували приклад аналогічний представленому на рис. 16.4 **Асоціацій між дійовими особами і варіантами використання**, але використали інші можливості нотації UML.

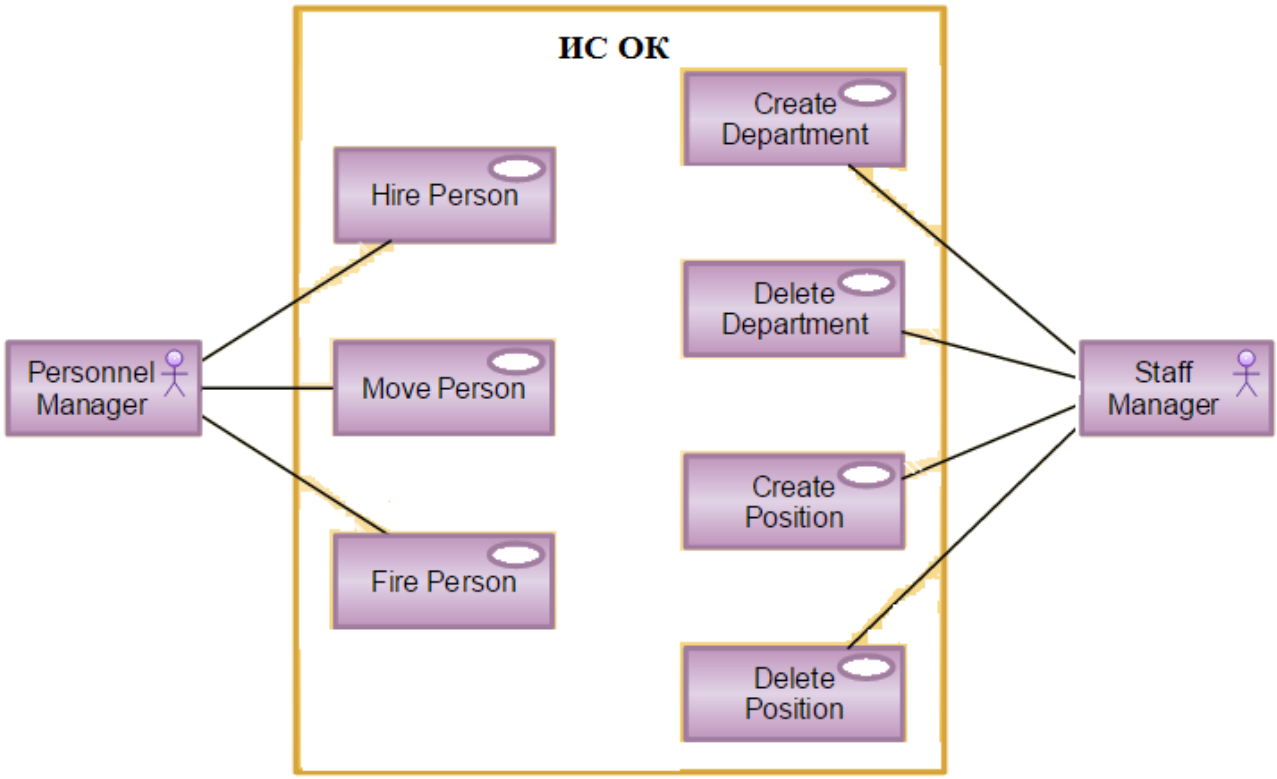


Рисунок 16.10 – Межі системи ІС ВК

У наведених прикладах дійовими особами є категорії користувачів. Це не випадково, у більшості випадків дійовими особами в моделях UML дійсно є категорії користувачів, але не завжди. У визначеннях дійової особи (параграф 16.2.1) і варіанту використання (параграф 16.2.2) ніщо не вказує на обмеження в застосуванні цих понять.

Вище ми відмітили, що **варіанти використання описують внутрішність системи**, а **дійові особи – її оточення**. Так от, системою може виступати будь-яка сутність, для якої можна визначити функціональні або нефункціональні вимоги. Це може бути і підсистема головної системи, окремий компонент і просто клас.

Якщо ми розглядаємо модель використання деякої підсистеми, то інші підсистеми (що взаємодіють з тією, що розглядається) будуть дійовими особами для даної підсистеми. Якщо ми розглядаємо модель використання деякого класу, то інші класи (що взаємодіють з тим, що розглядається) будуть дійовими особами для даного класу.

Розглянемо приклад, пов'язаний зі взаємодією нашої інформаційної системи відділу кадрів із зовнішнім програмним оточенням.

ЗМІНИ В ТЕХНІЧНОМУ ЗАВДАННІ

Система повинна підтримувати інтерфейс прикладного програмування (API) що дозволяє зовнішнім програмним засобам одержувати доступ до інформації, що зберігається в системі.

Було б самовпевненим вважати, що проектована інформаційна система відділу кадрів є першою і єдиною програмою, яка експлуатується на підприємстві. Швидше за все, таких програм сотні, і з десятком з них повинна взаємодіяти система відділу кадрів, що проектується. Заздалегідь усе передбачити дуже важко. Тому задовольнити нову вимогу найпростіше таким чином. Передбачити в інформаційній системі відділу кадрів інтерфейс для доступу до даних і кожного разу, коли потрібно буде надати конкретний API для нового клієнта, реалізовувати новий модуль, який адаптує наявний інтерфейс інформаційної системи до потрібного.

На діаграмі наведеній на рис. 16.11 дійова особа **ІС ВК** – це наша інформаційна система відділу кадрів, дійова особа **ERP** (Enterprise Resource Planning – планування ресурсів підприємства) – деяка інша інформаційна система, а **Adapter** – модуль з єдиним варіантом використання. Зверніть увагу, що на цій діаграмі усі сутності – програмні системи, ніяких користувачів тут немає.

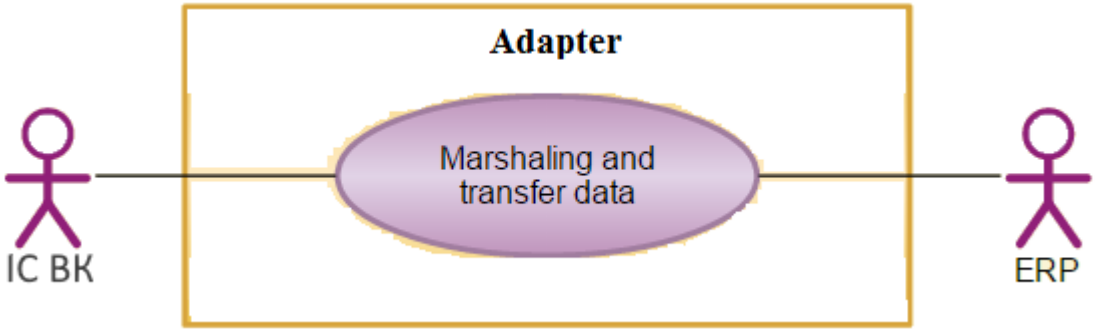


Рисунок 16.11 – Програмні системи в якості дійових осіб

16.3. Реалізація варіантів використання

Після того, як побудовано представлення використання (результат моделювання використання), тобто виділені дійові особи, варіанти використання і встановлені відношення між ними, встає природне питання: що далі?

Повторимо ще раз: варіант використання – це опис множини послідовностей подій або дій (сценаріїв), що доставляють значимий для дійової особи результат. Найчастіше використовуваний метод опису безлічі послідовностей дій полягає в указанні алгоритму, виконання якого доставляє послідовність дій з необхідної множини.

16.3.1. Реалізація діаграмами діяльності

Один із способів реалізації варіанту використання – описати алгоритм за допомогою діаграми діяльності. З одного боку, ***діаграма діяльності*** – це повноцінна діаграма UML, з іншого боку, діаграма діяльності незначно відрізняється від блок-схеми (а тим самим і від псевдокоду).

Наприклад, в інформаційній системі відділу кадрів прийом співробітника може бути організований так, як показано на діаграмі (рис. 16.12).

Чи наблизила нас поява цієї діаграми в моделі до завершення роботи над системою? З одного боку, замість однієї сутності, що підлягає реалізації (варіант використання **Hire Person**) з'явилося п'ять нових: три умови і дві діяльності, які у свою чергу тепер потребують реалізації.

З іншого боку, кожна з цих нових сутностей здається простішою і зрозумілішою, а значить швидше і надійніше за ту, що реалізовується. Крім того, цю діаграму можна показати замовникові, щоб перевірити, чи дійсно проектована нами логіка роботи системи відповідає тому бізнес-процесу, який існує в реальності.

Застосування діаграм діяльності для реалізації варіантів використання не дуже наближає до появи цільового артефакту – програмного коду, проте може привести до глибшого розуміння сутності завдання і навіть відкрити несподівані можливості поліпшення додатка, які було важко угледіти в первинній постановці завдання.

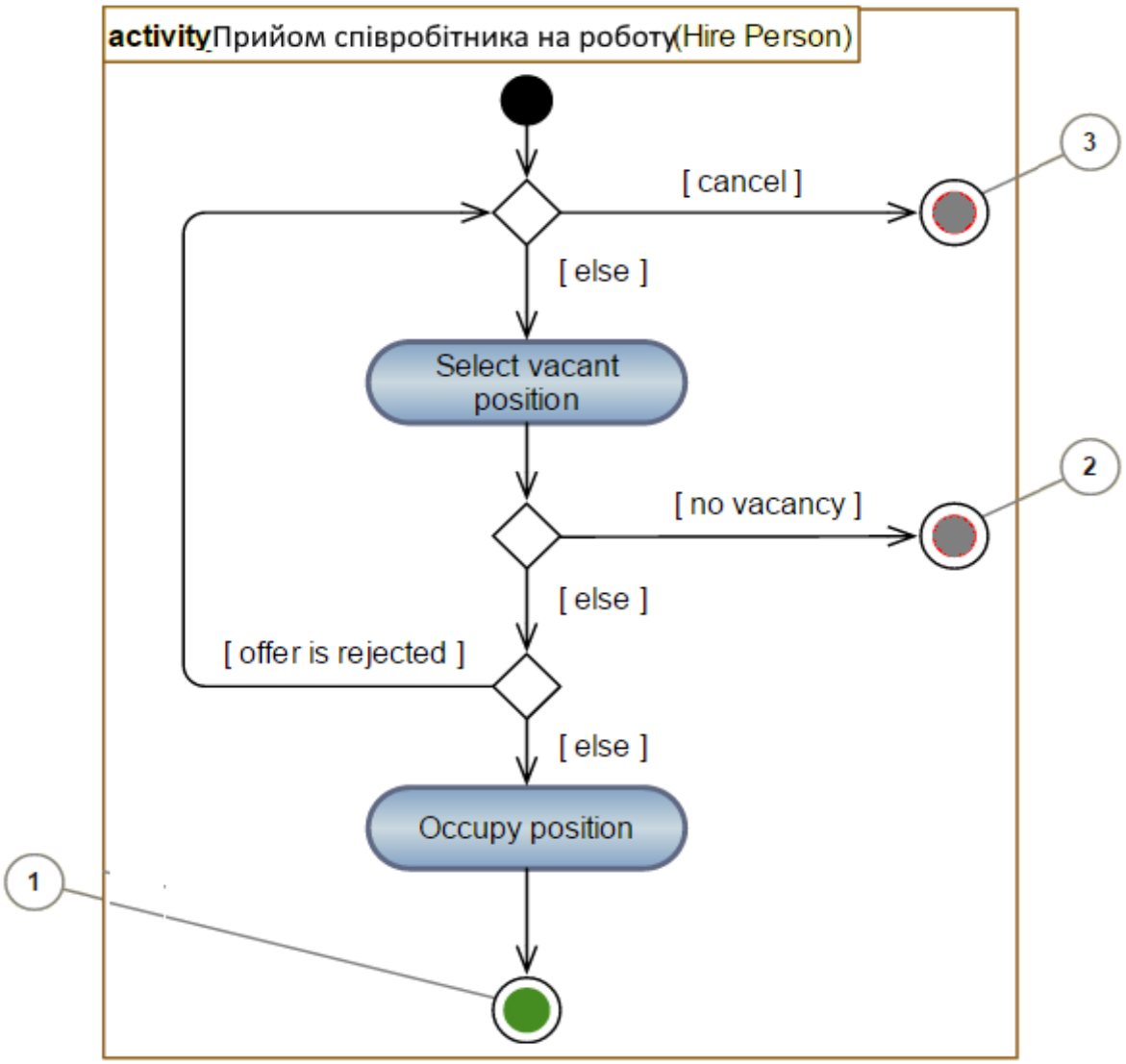


Рисунок 16.12 – Реалізація варіанту використання за допомогою діаграми діяльності

Наприклад, розглядаючи (чисто формально) схему процесу на рис. 16.12, ми бачимо, що процес може мати три результати.

Нормальне завершення 1, яке припускає обов'язкове виконання діяльності **Occupy position**. Резонно припустити, що при виконанні цієї діяльності у базу даних буде записана якась інформація, що, поза сумнівом, є значимим для користувача результатом.

Виняткова ситуація 2, що виникає у тому випадку, коли процес не може бути нормально завершений, тобто ми хотіли прийняти людину на роботу, і вона була згідна, а поточний стан штатного розкладу не дозволив цього зробити. Факт виникнення такої ситуації, хоча формально і не є значимим результатом для менеджера персоналу, насправді може бути дуже важливий для вищого керівництва або менеджера штатного розпису.

Завершення процесу без досягнення якого-небудь видимого результату 3. Наприклад, менеджер персоналу зробив кандидатові якісь пропозиції, але жодне з них кандидата не влаштувало, після чого вони розлучилися, неначе нічого і не було.

Цей простий аналіз наштовхує на такі міркування. Варіант використання повинен доставляти значимий результат, отже, якщо результату немає, то щось спроектовано не так, як треба.

Дійсно, майже усі практичні інформаційні системи відділу кадрів обов'язково накопичують статистичну інформацію про усі проведені кадрові операції. Така статистика абсолютно потрібна для так званого аналізу руху кадрів – важливої складової процесу управління організацією. Проте далеко не всі системи дозволяють враховувати і не проведені операції. Між тим, облік цієї інформації, наприклад, облік причин, з яких кандидати не приймають запропонованої роботи або, навпаки, причин, з яких організації відкидають кандидатів, може бути дуже корисний для дослідження ринку праці і формування кадрової політики організації.

Трохи подумавши над цією проблемою (чи порадившись із замовником), можна удосконалити нашу діаграму, наприклад, так, як показано на рис. 16.13.

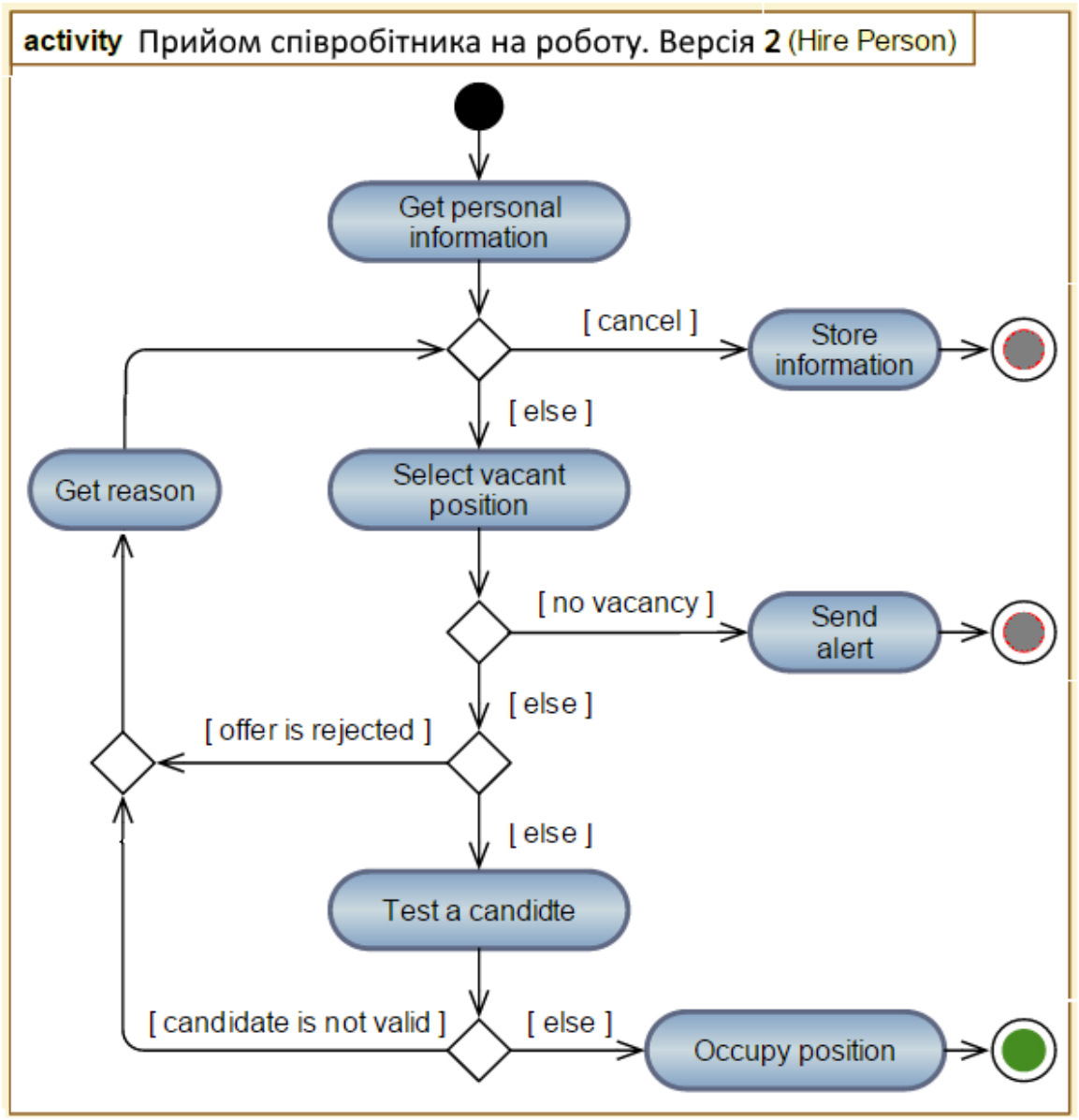


Рисунок 16.13 – Вдосконалена реалізація варіанту використання

Ось тепер (формально) усе добре: інформація не втрачається. Більше того, має сенс повернутися до представлення використання і подивитися, чи не треба включити в модель нові варіанти використання (можливо, як низькопріоритетні і такі, що підлягають реалізації в подальших версіях системи). Так реалізація варіантів використання може призводити до зміни і удосконалення самих варіантів використання. Моделювання має ітеративний характер, про що ми вже говорили раніше.

16.3.2. Реалізація діаграмами взаємодії

При складанні діаграм взаємодії для декількох варіантів використання можуть бути задіяні одні і ті ж класи, що грають різні ролі в різних кооперaціях. Однакова поведінка і структура використання, що проявляються в різних варіантах, виявляються реалізованими одним класом. Такий стиль моделювання повністю відповідає об'єктно-орієнтованому підходу і забезпечує концептуальну цілісність системи, що проектується.

Як уже говорилося (розділ 10) у діаграм взаємодії UML є істотне обмеження – діаграми взаємодії дозволяють описати протокол виконання алгоритму, але не сам алгоритм.

Якщо алгоритм виконання варіанту використання лінійний, то проблеми немає – лінійні алгоритми суть протоколи свого виконання. Для галужень і циклів діаграми взаємодії містять деякі засоби моделювання. Іноді цих засобів може виявитися достатньо.

Що ж робити, якщо все-таки ніяк не вдається побудувати вичерпну діаграму взаємодії для варіанту використання? У цьому випадку застосовують такі методи.

1. Реалізують варіант використання декількома діаграмами взаємодії. Кожна діаграма описує окремий сценарій, а всі разом вони дають достатнє уявлення про реалізацію варіанту використання.
2. Представляють використання так, щоб виключити варіанти використання, що важко реалізовуються. Наприклад, за допомогою узагальнення виділити варіанти використання, які описують одноріднішу множину сценаріїв і легше реалізуються діаграмами взаємодії.

Спробуємо проілюструвати сказане на прикладі реалізації діаграмами взаємодії варіанту використання **Hire Person** (прийом співробітника на роботу) інформаційної системи відділу кадрів.

Спочатку розглянемо типовий сценарій, коли прийом проходить без всяких ускладнень: є вакантне робоче місце і кандидат готовий його зайняти. Діаграма послідовності для такого сценарію приведена на рис. 16.14.

На наведеній діаграмі послідовність відправних повідомлень приблизно відповідає послідовності дій на діаграмі діяльності рис. 16.12. **Реалізація варіанту використання за допомогою діаграми діяльності** у тому випадку, коли потік управління проходить по діаграмі зверху вниз один раз.

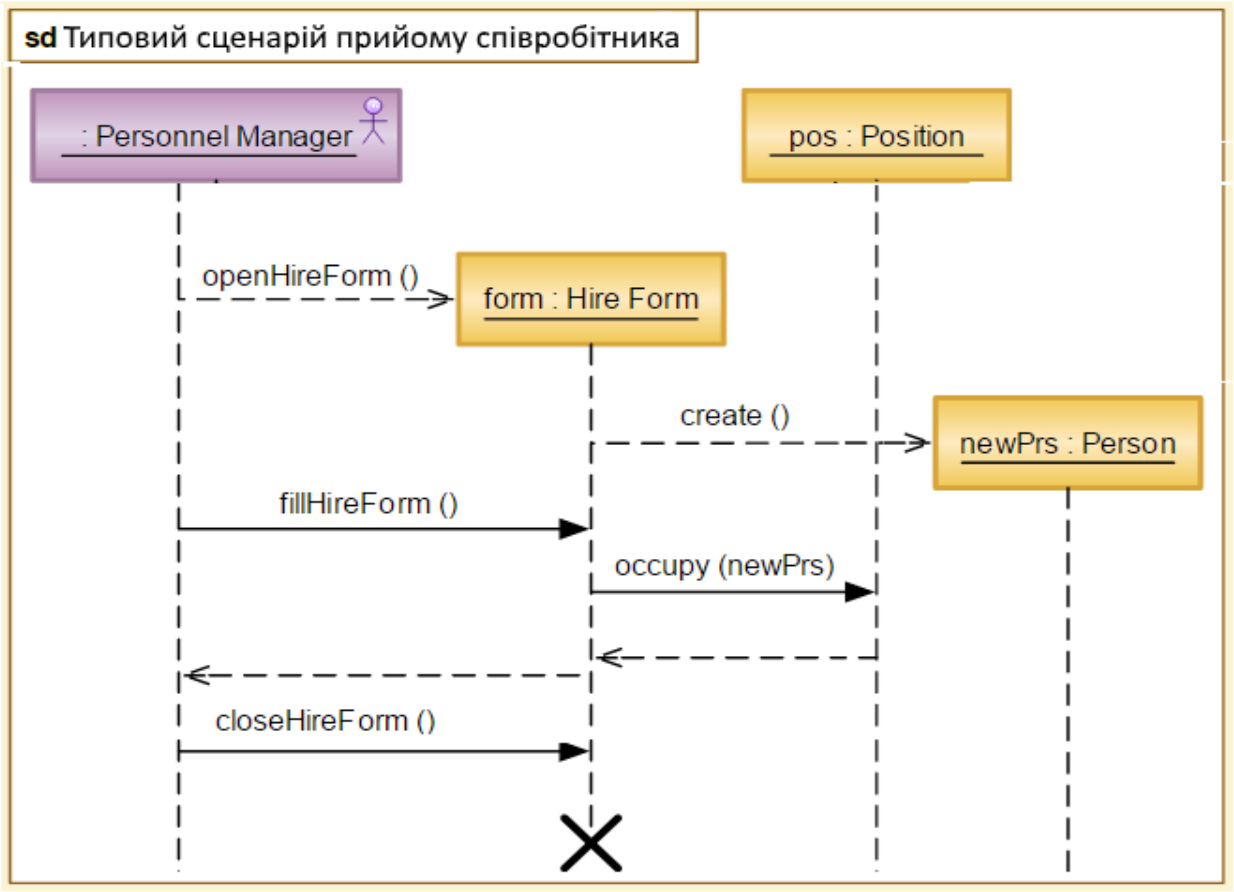


Рисунок 16.14 – Діаграма послідовності для типового сценарію

Таким чином, представлена діаграма послідовності до деякої міри визначає типовий сценарій варіанту використання **Hire Person**. Проте, окрім визначення послідовності виконуваних дій, ця діаграма містить і іншу інформацію, істотну для подальшого проектування.

Побудувавши таку діаграму взаємодії, ми постулювали існування в системі деяких класів (можливо, ще не всіх), екземпляри яких повинні взаємодіяти для забезпечення необхідної поведінки модельованого варіанту використання. Дійова особа **Personnel Manager** вже була визначена при моделюванні використання, тут же в нашій моделі з'явилися нові сутності:

- клас **Hire Form**, відповідальний за інтерфейс, необхідний для виконання варіанту використання прийом співробітника;
- клас **Person**, відповідальний за зберігання даних про конкретного співробітника;
- клас **Position**, відповідальний за зберігання даних і виконання операцій з конкретною посадою.

На цій стадії моделювання список операцій вказаних класів ще далеко неповний (а атрибути доки зовсім відсутні), проте сам факт появи класів в моделі є важливим рішенням, що істотно наближає нас до реалізації системи.

Більшість CASE-інструментів підтримують такий режим роботи. Складаючи деяку діаграму, (наприклад, в нашому випадку діаграму послідовності), можна визначити в моделі деякі сутності (наприклад, класи),

потрібні для моделювання в даний момент, не відбиваючи їх поки що ні на якій діаграмі.

Повертаючись до нашого прикладу, відмітимо, що попередня діаграма послідовності семантично не повна: вона не відбиває усі сценарії варіанту використання, які ми виявили (порівняйте).

Як уже було сказано, в цьому випадку можна скласти додаткові діаграми взаємодії, що реалізують альтернативні сценарії варіанту використання. Наприклад, на рис. 16.15 показаний сценарій прийому співробітника, що відповідає винятковій ситуації, коли немає вакантних посад. Цього разу ми описуємо сценарій у формі ***діаграми комунікації***.

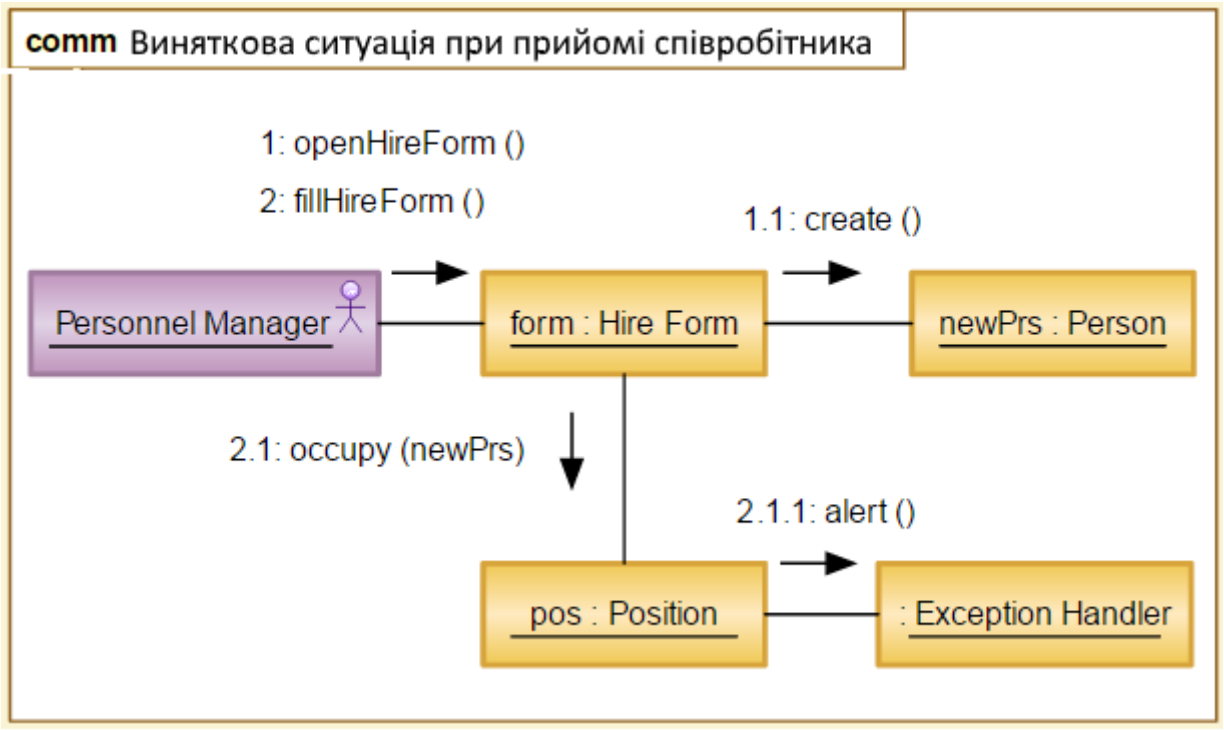


Рисунок 16.15 – Діаграма комунікації для виняткової ситуації

Побудова цієї діаграми виявила необхідність включення в модель (принаймні) ще одного класу – **Exceptions Handler**, який несе відповідальність за обробку виникаючих виняткових ситуацій. Звичайно, той спосіб вирішення проблеми, який ми пропонуємо на діаграмі далеко не єдиний, і, можливо, не найкращий з відомих вам. Справа не в конкретному способі обробки виключень – це питання технології програмування, а в тому, що побудова діаграми взаємодії виявила сам факт необхідності передбачити обробку виняткових ситуацій в системі. Якби ми не побудували діаграму комунікації для альтернативного сценарію, ми могли б випустити з уваги цю обставину і, тим самим, вчинити грубу проектну помилку.

З вище викладеного можна зробити такі висновки: реалізація варіантів використання діаграмами взаємодії є найтрудомісткішим і найскладнішим методом, але цей метод краще всього погоджений з об’єктно-орієнтованим підходом.

РОЗДІЛ 17. МОДЕЛЮВАННЯ СТРУКТУРИ ІС ВІДДІЛУ КАДРІВ

У цьому розділі ми розглянемо способи відповіді на питання **з чого складається наша ІС відділу кадрів**? Спочатку ми обговоримо загальні принципи моделювання структури для нашої ІС відділу кадрів, а потім – різноманітні конкретні засоби моделювання структури.

17.1. Сутності на діаграмі класів

Діаграма класів є основним засобом моделювання структури в UML, а клас, відповідно, основною структурною одиницею.

Класи. Опис класу може включати багато різних елементів, і щоб вони не плуталися, в мові передбачено групування елементів опису класу за ***секціями***. Стандартних секцій три:

- ***секція імені*** – разом з обов'язковим ім'ям може містити також стереотип, кратність і список іменованих значень;
- ***секція атрибутів*** – містить список описів атрибутів класу;
- ***секція операцій*** – містить список описів операцій класу.

Як і усі основні сутності UML, клас обов'язково має ім'я, а отже, секція імені не може бути опущена.

Разом із стандартними секціями, опис класу може містити і довільну кількість додаткових секцій. Якщо секцій більше за одну, то внутрішність прямокутника ділиться горизонтальними лініями на частини, що відповідають секціям (рис. 17.1).

Обов'язкове ім'я класу може бути виділене *курсивом і в цьому випадку цей* **клас є абстрактним**, тобто таким, що не має безпосередніх екземплярів. Якщо ім'я підкреслене, то це вже не ім'я класу, а ім'я об'єкту.

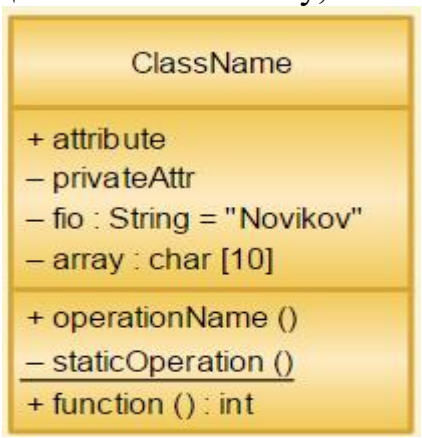


Рисунок 17.1 – Типова нотація класу

Розглянемо приклад секції імені класу для нашої інформаційної системи відділу кадрів. Якщо припустити, що ІС відділу кадрів, яка проектується, використовуватиметься на одному підприємстві, то хорошим рішенням буде визначення службового класу **Company** із стереотипом **«utility»** для зберігання глобальних атрибутів і операцій ІС відділу кадрів (рис 17.2).

Рисунок 17.2 – Секція імені служби ІС відділу кадрів

Атрибути. ***Атрибут*** – це іменоване місце, в якому може зберігатися значення. Атрибути класу перераховуються в секції атрибутів. У загальному випадку опис атрибуту має такий синтаксис:

видимість ІМ'Я кратність: тип = початкове_значення {властивості}

Видимість, як завжди, позначається знаками `+, - # ~`. Ще раз підкреслимо, що якщо видимість не вказана, то ніякого значення видимості за умовчанням не мається на увазі.

Якщо ***ім'я атрибуту*** підкреслене, то це означає, що зоною дії цього атрибуту є клас, а не екземпляр класу, як завжди.

Підкреслення опису атрибуту відповідає описувачу **`static`**, вживаному у багатьох об'єктно-орієнтованих мовах програмування.

Кратність, якщо вона є присутньою, визначає цей атрибут як масив (певної або невизначеної довжини).

Тип атрибуту – це або примітивний (вбудований) тип, або тип, визначений користувачем.

Початкове значення має очевидний сенс: при створенні екземпляра цього класу атрибут набуває вказаного значення. Наприклад, в інформаційній системі відділу кадрів клас **Person**, швидше за все, повинен мати атрибут, що зберігає ім'я співробітника.

Операції і методи. ***Операція*** **–** це специфікація дії з об'єктом: зміна значення його атрибутів, обчислення нового значення за інформацією, що зберігається в об'єкті і так далі. Оголошення конкретної операції в класі має на увазі наявність методу в цьому ж класі.

Метод **–** *це реалізація операції, тобто алгоритм, що виконується. Виконання дій, що визначають операцією, ініціюється* ***викликом методу*****.**

При виклику методу можуть, у свою чергу, бути викликані методи цього ж, а також інших класів.

Описи операцій класу перераховуються в секції операцій і мають такий синтаксис:

видимість ІМ'Я (параметри): тип {властивості}

Тут слово параметри означає послідовність описів параметрів операції, кожне з яких має наступний формат:

напрям ПАРАМЕТР: тип = значення

Видимість, як завжди, позначається за допомогою знаків `+, -, #, ~` або за допомогою ключових слів **private, public, protected, package**.

Всі разом (ім'я операції, параметри і тип результату) зазвичай називають ***сигнатурою*** операції.

Інтерфейси і типи даних. У UML є декілька окремих випадків класифікаторів, які, подібно до класів, призначені для моделювання структури,

але мають ряд специфічних особливостей. Найважливішими з них є інтерфейси і типи даних.

Інтерфейс **–** це іменований набір складових, що описує контракт між постачальниками і споживачами послуг. Тобто, інтерфейс – це абстрактний клас, в якому усі складові (атрибути і операції) абстрактні. Абстрактні атрибути інтерфейсу – це атрибути, які обов'язково повинні з'явитися в класі, що реалізовує інтерфейс.

Тип даних – це сукупність множини значень і кінцевої множини операцій, застосованих до цих значень.

Шаблони. ***Шаблон*** **– це сутність (найчастіше класифікатор) з параметрами.** Параметром може бути будь-який елемент опису класифікатора – тип складової, кратність атрибуту і так далі. На діаграмі шаблон зображається за допомогою стандартної нотації класифікатора – прямокутника, до якого в правому верхньому кутку приєднаний пунктирний прямокутник з параметрами шаблону.

Сам по собі шаблон не може безпосередньо використовуватися в моделі. Для того щоб на основі шаблону отримати конкретний екземпляр класифікатора, треба вказати явні значення аргументів. Така вказівка називається **зв'язуванням** (binding). У UML застосовуються два способи зв'язування:

- явне зв'язування – залежність із стереотипом «bind»;
- неявне зв'язування – визначення класу, ім'я якого має формат

ім'я_класифікатора: ім'я_шаблону < аргументи >

Розглянемо приклад, пов'язаний з інформаційною системою відділу кадрів. Припустимо, від нас вимагається вказати, що для зберігання різних видів даних ми використовуватимемо класи, отримані з шаблонного класу (template class) **Array.** На рис. 17.3 визначений шаблон **Array** 1, що має два параметри: n і T. Цей шаблон застосовується для створення масивів певної довжини n, що містять елементи певного типу T. В даному випадку за допомогою явного 2 і неявного 3 зв'язування показані два еквівалентні способи визначення класу Positions у вигляді масиву з 256 елементів типу Position.

Призначення і сфера застосування шаблонів зрозумілі – шаблони потрібні, щоб визначити деяку загальну параметричну конструкцію класифікатора один раз, і потім використати її багаторазово, підставляючи конкретні значення аргументів. Явне зв'язування наочніше, неявне зв'язування менш наочне, зате записується коротше.

Використання шаблонів – самостійна парадигма, яка підтримується UML разом з об'єктно-орієнтованою.

На цьому ми закінчуємо обговорення сутностей на діаграмі класів. Усе що нам залишилося зробити – навести діаграму метамоделі для основних структурних сутностей (рис. 17.4).

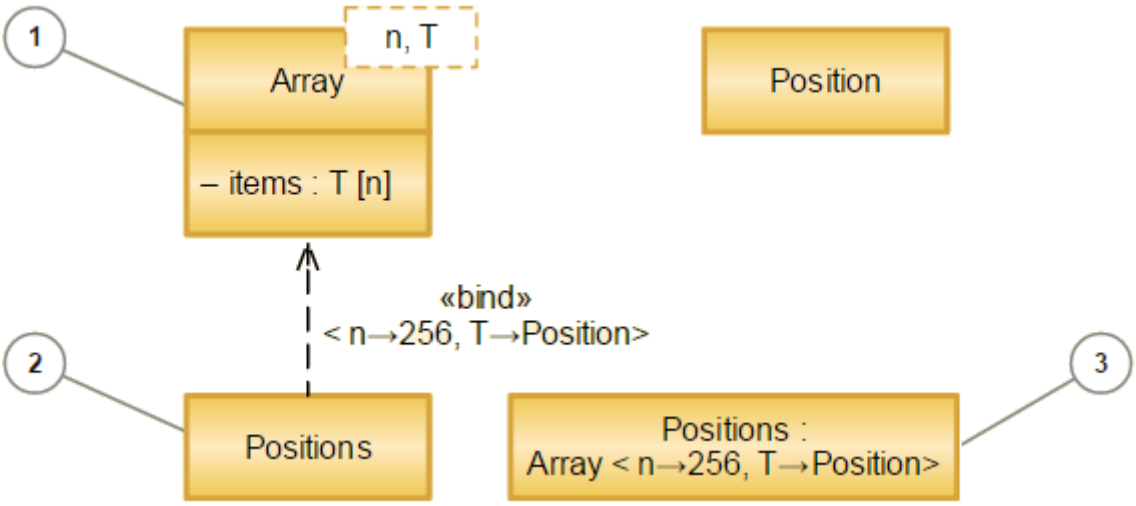


Рисунок 17.3. Явне і неявне зв’язування шаблону

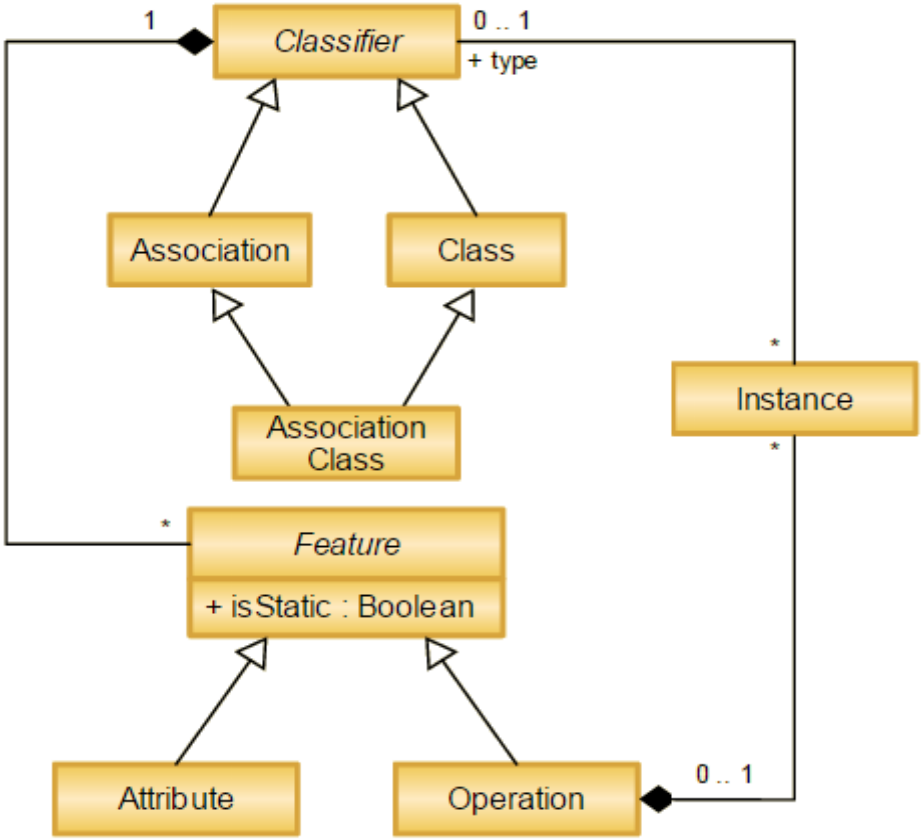


Рисунок 17.4 – Метамодель структурних сутностей

17.2. Відношення на діаграмі класів

Сутності на діаграмах класів зв’язуються головним чином відношеннями **асоціації** (у тому числі агрегації і композиції) і **узагальнення.** Відношення **залежності** і **реалізації** на діаграмах класів застосовуються рідше, але, проте, вони також застосовуються, і ми розпочнемо з них, як з простіших.

Відношення залежності і реалізації. Всього в UML визначена досить велика кількість стандартних стереотипів відношення залежності, які можна розділити на декілька груп:

– між класами і об’єктами на діаграмі класів;

– між пакетами;
– між варіантами використання.

Тут розглядаються деякі залежності першої групи, які перераховані в таблиці. 17.1.

Таблиця 17.1 – Стандартні стереотипи залежностей на діаграмі класів

Стереотип	Опис
«bind»	Підстановка параметрів в шаблон. Незалежною суттю є шаблон (клас з параметрами), а залежною – клас, який виходить з шаблону завданням аргументів.
«call»	Вказує залежність між двома операціями: операція залежного класу викликає операцію незалежного класу.
«derive»	Буквально означає «може бути обчислений за».
«use»	Залежність найзагальнішого вигляду, що показує, що залежний клас яким-небудь чином використовує незалежний клас.

Розглянемо відношення реалізації між інтерфейсами і іншими класифікаторами, зокрема, класами. На діаграмі класів застосовуються два відношення:

– класифікатор (зокрема, клас) використовує інтерфейс – це показується за допомогою залежності із стереотипом «call»;
– класифікатор (зокрема, клас) реалізує інтерфейс – це показується за допомогою відношення реалізації.

Розглянемо наш приклад з ІС відділу кадрів. Припустимо, що клас **Department** для реалізації операцій пов'язаних з рухом кадрів, використовує операції класу **Position,** що дозволяють займати і звільняти посаду – інші операції класу **Position** класу **Department** не потрібні. Для цього, як показано на рис. 17.5, можна визначити відповідний інтерфейс **IPosition** і зв'язати його відношеннями з цими класами.

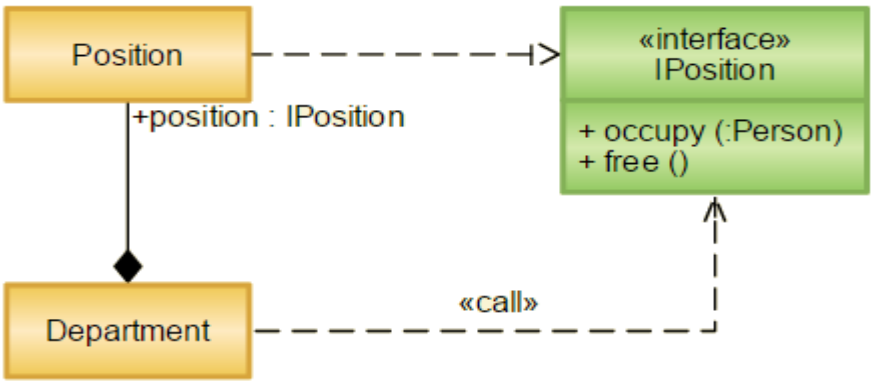


Рисунок 17.5 – Відношення реалізації і використання інтерфейсів

Використовуючи *нотацію «чупа-чупс»*, що з'явилася в UML 2, цю ж модель можна зображувати лаконічно, симетрично і просто (рис. 17.6).

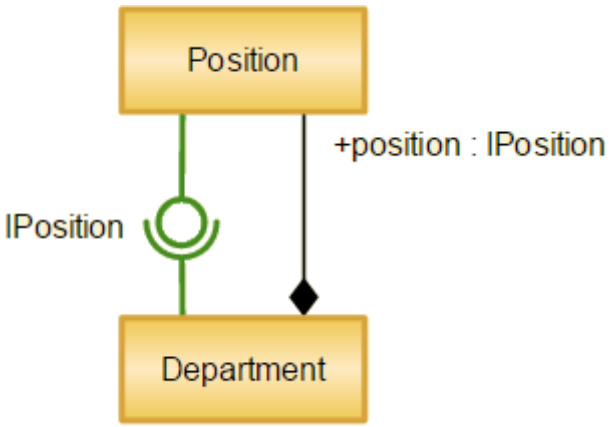


Рисунок 17.6 – Використання нотації «чупа-чупс»

Відношення узагальнення. Відношення узагальнення часто застосовується на діаграмі класів. Як правило, між классамы є щось загальне, і це загальне доцільно виділити в окремий клас. При цьому загальні складові, зібрані в суперкласі, автоматично наслідують підкласами. Таким чином, скорочується сумарна кількість описів, тобто зменшується ймовірність допустити помилку.

При узагальненні виконується **принцип підстановки**, що фактично означає збільшення гнучкості і універсальності програмного коду, при одночасному збереженні надійності, що забезпечується контролем типів.

Принцип підстановки полягає в тому, що екземпляр підкласу може використовуватися скрізь, де використовується екземпляр його суперкласу, підтримуючи таким чином контракт, пропонований суперкласом. Іншими словами, підклас забезпечує той же інтерфейс що і суперклас.

Розглянемо приклад.

ЗМІНИ В ТЕХНІЧНОМУ ЗАВДАННІ

Кожна структурна одиниця підприємства (підрозділ, посада) повинна мати свою назву.

У інформаційній системі відділу кадрів ми раніше виділили класи **Position, Department** і **Person.** Для усіх цих класів може бути вказаний атрибут, що містить власне ім’я об’єкту, що виділяє його у ряді однорідних. Для простоти покладемо, що такий атрибут має тип String. У такому разі можна визначити суперклас, відповідальний за зберігання цього атрибуту і роботу з ним, а інші класи зв’язати з суперкласом відношенням узагальнення.

Проте пильніший аналіз предметної області наводить на думку, що робота з власним ім’ям для виділених класів виконується не зовсім однаково. Дійсно, призначення і зміна власних імен підрозділам і посадам знаходиться в межах відповідальності ІС відділу кадрів, але призначення (тим більше зміна) власного імені співробітника явно виходить за ці межі. Виходячи з цих міркувань, ми приходимо до структури узагальнень, представленої на рис. 17.7.

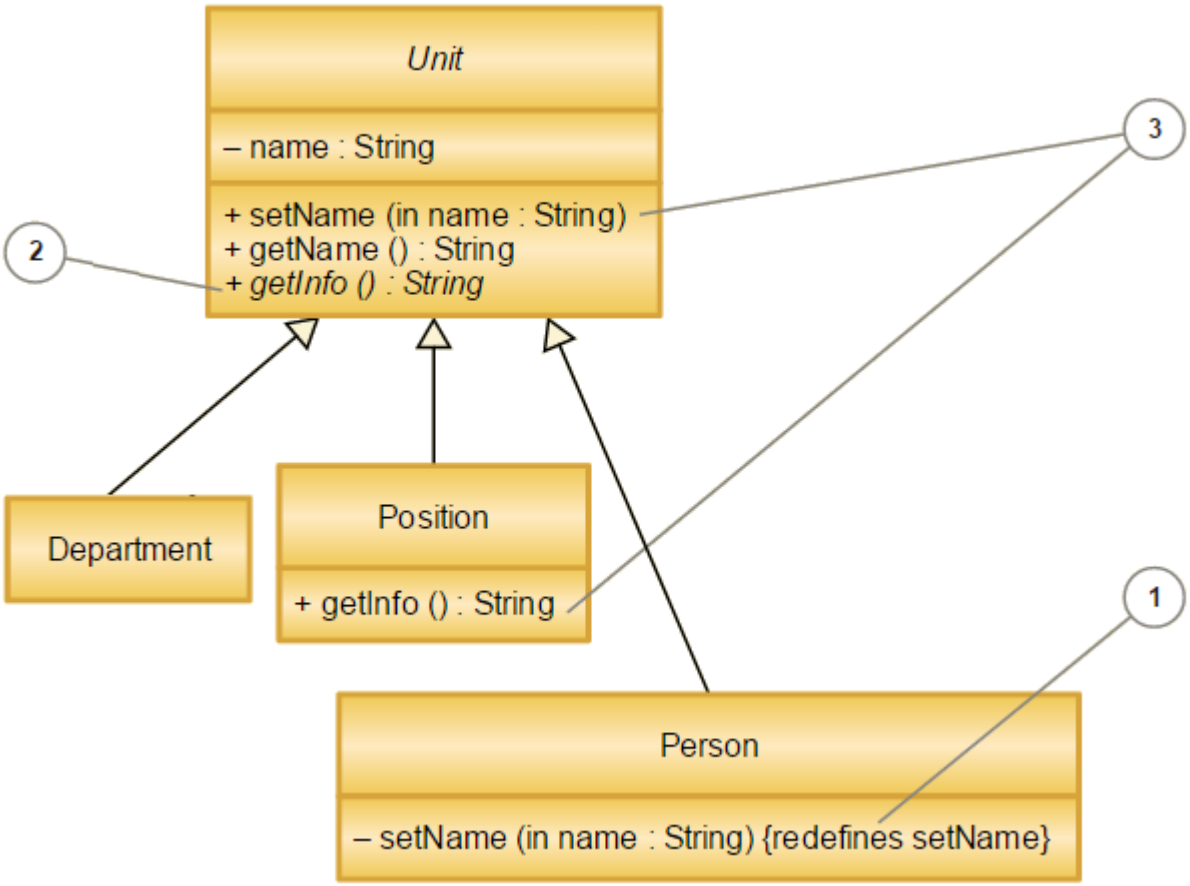


Рисунок 17.7 – Відношення узагальнення

Операція setName(), оголошена в класі **Unit** перевизначена для класу **Person.** На це вказує поміщене у фігурні дужки доповнення redefines 1, що йде за визначенням операції,. Перевизначення полягає в тому, що значення видимості для операції setName() змінене з «відкрита» на «закрита».

Зверніть увагу, що суперклас **Unit** визначений як абстрактний, тобто такий, що не має можливості мати безпосередні екземпляри, оскільки в системі не передбачається мати об'єкти цього класу. Екземпляри існують для конкретних підкласів **Department, Position** і **Person.** Клас **Unit** в даному випадку потрібний тільки для того, щоб звести описи одного атрибуту і двох операцій в одному місці і не повторювати їх двічі.

У об'єктно-орієнтованому програмуванні поширено поняття, згідно з яким кореневі інтерфейси (класи) ієрархій визначають абстрактну операцію, перевизначувану в підкласах, яка під час виконання повертає деяку інформацію про конкретний екземпляр створеного підкласу. Цю інформацію можна використати, наприклад, для тестування або в операціях безпечного приведення типів.

Введемо подібну операцію в абстрактному класі **Unit,** переслідуючи при цьому тільки одну мету – продемонструвати відношення між поняттями абстрактна операція 2 і операція з методом 3.

У UML існує можливість виділяти у множині узагальнень підмножини узагальнень і задавати обмеження для них.

Розглянемо такий (дещо штучний) приклад для інформаційної системи відділу кадрів. Припустимо, що для екземплярів класу **Person** вимагається змоделювати такі характеристики, як стать – **Gender** і службовий стан – **Employment Status.** Підкласи **Male Person** і **Female Person** описуватимуть стать співробітника**,** а **Employer** і **Employee**, відповідно, його службовий стан. Тоді цю ситуацію можна описати за допомогою такої діаграми (рис. 17.8).

Класифікація за статтю є завершеною і роздільною (disjoint – неспільність). Класифікація за службовим станом, навпаки, не є ні завершеною, ні диз'юнктивною (тобто, власник компанії (Employer) може бути і працівником компанії (Employee)).

Імена підмножин узагальнень, вказуються після двокрапки поряд з узагальненням, а у разі, якщо потрібно охопити декілька узагальнень, то застосовується нотація у вигляді пунктирної лінії, як це зроблено для підмножини Employment Status. Окрім імені підмножини можна вказати обмеження (чи їх комбінацію), що накладаються на підмножину. Можливі варіанти обмежень наведені в таблицю. 17.2.

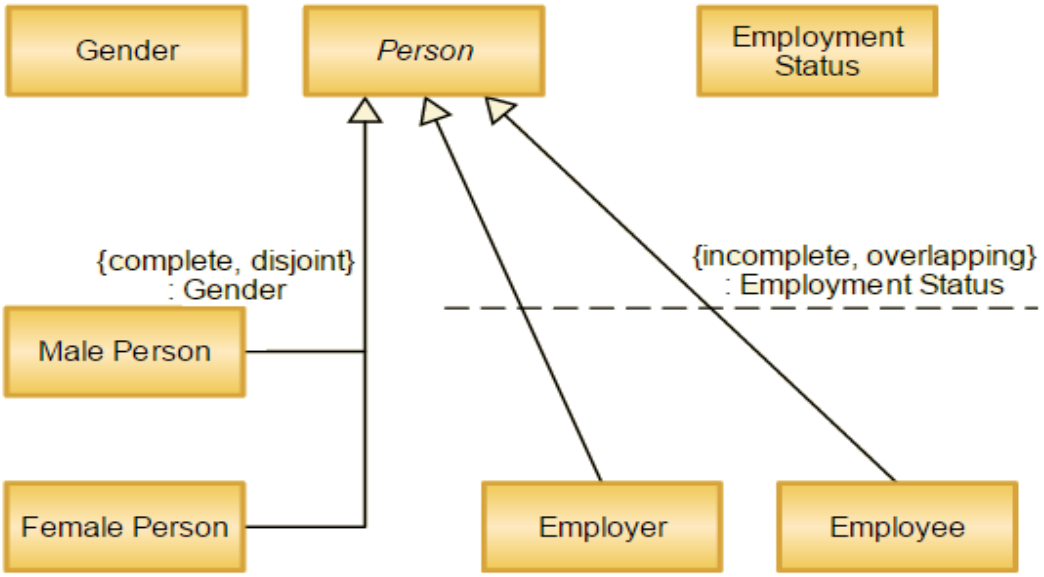


Рисунок 17.8 – Підмножини узагальнень

Як видно з опису, наведеного вище, пари {compete} – {incomplete} і {disjoint} – {overlapping}є взаємовиключними, тобто не може бути одночасно множина і {complete}, і {incomplete}. Значення обмежень за замовчуванням – {incomplete, disjoint}.

Таблиця 17.2 – Обмеження на підмножину узагальнень

Обмеження	**Застосування**
{complete} повнота	Множина узагальнень, що входять в підмножину, є повною, тобто визначає усі можливі підтипи для цієї характеристики суперкласифікатора.
{incomplete} неповнота	Множина узагальнень, що входять в підмножину, не є повною, тобто визначає тільки частину можливих підкласифікаторів для цієї характеристики

	суперкласифікатора.
{disjoint} неспільність	Області значень підкласифікаторів, що входять в цю підмножину не перетинаються, тобто є взаємовиключними.
{overlapping} спільність	Області значень підкласифікаторів можуть перетинатися, тобто вони не є взаємовиключними.

Асоціації і їх доповнення. Відношення асоціації є, мабуть, найважливішим на діаграмі класів. У загальному випадку асоціація, нотація якої – суцільна лінія, що сполучає класи, означає, що екземпляри одного класу пов'язані з екземплярами іншого класу. У UML асоціація є класифікатором, екземпляри якого називаються зв'язками.

Зв'язок (link) – це екземпляр асоціації (чи з'єднувача), який є впорядкованим набором (кортеж, tuple) посилань на екземпляри класифікаторів на полюсах асоціації.

Асоціація в UML має на увазі лише те, що пов'язані об'єкти мають достатню інформацію для організації взаємодії. Можливість взаємодії означає, що об'єкт одного класу може послати повідомлення об'єкту іншого класу, зокрема, викликати метод або ж прочитати або змінити значення відкритого атрибуту.

Для асоціації в UML передбачена найбільша кількість різних доповнень. Доповнення, як завжди, не є обов'язковими: їх використовують при необхідності, в різних ситуаціях по-різному. Якщо використати усі доповнення відразу, то діаграма стає настільки переобтяженою, що її важко читати. Отже, для асоціації визначені такі доповнення:

- ім'я асоціації (можливо, разом з напрямом читання);
- кратність полюса асоціації (кінець лінії асоціації);
- агрегації або композиція;
- можливість навігації для полюса асоціації;
- роль полюса асоціації;
- видимість полюса асоціації;
- змінність множини об'єктів на полюсі асоціації;
- обмеження subset і union полюса асоціації;
- клас асоціації.

Ім'я асоціації. Ім'я асоціації вказується у вигляді рядка тексту над (під, або поруч з) лінією асоціації. Ім'я не несе додаткового семантичного навантаження, а просто дозволяє розрізняти асоціації в моделі.

Наприклад, в ІС відділу кадрів, якщо співробітник обіймає посаду, то відповідні екземпляри класів **Person** і **Position** мають бути пов'язані, тобто між самими класами має бути відношення асоціації 1 і може бути ім'я 2 призначення, що пояснює її. Додатково можна вказати напрям читання імені асоціації 3. Фрагмент графічної моделі, наведений нижче (рис. 17.9), фактично можна прочитати вголос: *Person occupies Position*.

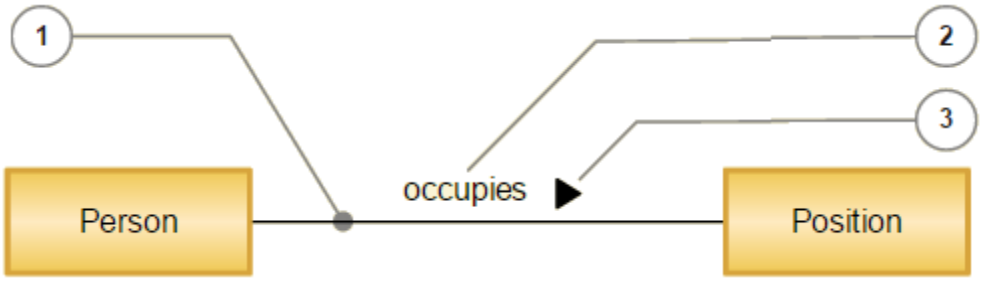


Рисунок 17.9 – Ім’я асоціації і напрям читання

Кратність полюса асоціації. Кратність полюса асоціації вказує, скільки об’єктів цього класу (з боку цього полюса) беруть участь у зв’язку. Кратність може бути задана як конкретне число, і тоді в кожному зв’язку з боку цього полюса бере участь рівно стільки об’єктів, скільки вказано. Поширеніший випадок, коли кратність вказується як діапазон можливих значень, і тоді число об’єктів, що беруть участь в зв’язку повинно знаходитися в межах вказаного діапазону. При вказівці кратності можна використати символ *, який означає невизначене число.

Наприклад, якщо в ІС відділу кадрів не передбачається дроблення ставок і поєднання посад, то працюючому співробітникові відповідає одна посада 1, а посаді відповідає один співробітник або жодного 2, тобто посада вакантна. (рис. 17.10).

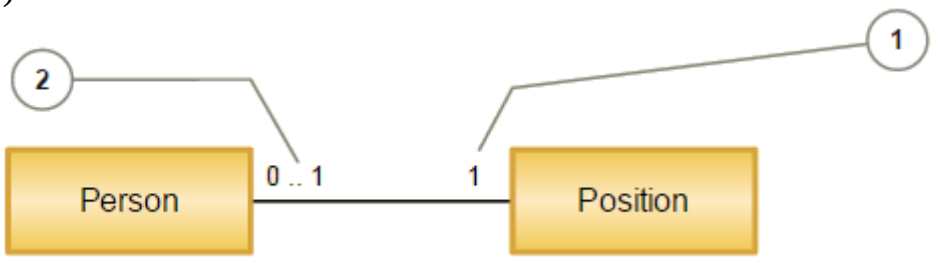


Рисунок 17.10 – Кратність полюсів асоціації

Складніші випадки також легко моделюються за допомогою кратності полюсів. Наприклад, якщо ми хочемо передбачити поєднання посад і зберігати інформацію навіть про непрацюючих співробітників, то діаграма набере вигляду (рис. 17.11). Запис * еквівалентний запису 0.*.

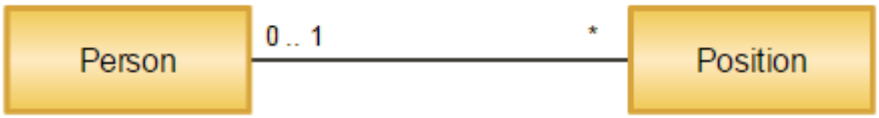


Рисунок 17.11 – Використання невизначеної кратності

Агрегація і композиція. У UML використовуються два приватних, але дуже важливих випадки відношення асоціації, які називаються ***агрегацією*** і ***композицією***. У обох випадках йдеться про моделювання відношення типу «частина – ціле». Відношення такого типу слід віднести до відношення асоціації, оскільки частини і ціле зазвичай взаємодіють.

Агрегація (aggregation) **–** це асоціація між класом **A** (частина) і класом **B** (ціле), яка означає, що екземпляри (один або декілька) класу **A** входять до

складу екземпляра класу **B**. Це відзначається за допомогою спеціального графічного доповнення: на полюсі асоціації, приєднаному до «цілого», зображається незафарбований ромб 1. Наприклад, на рисунку 17.12 вказано, що співробітник є членом робочої групи **Workgroup**.

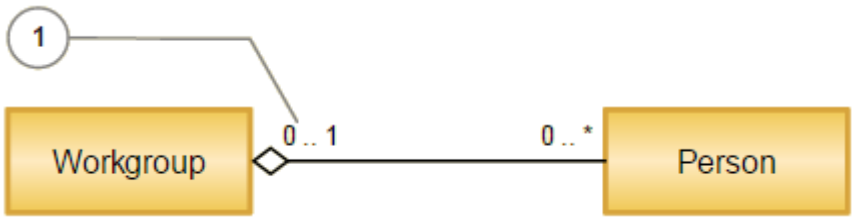


Рисунок 17.12 – Відношення агрегації

Композиція (composition) – це асоціація між класом A (частина) і класом B (ціле), яка додатково накладає сильніші обмеження порівняно з агрегацією: композиційно частина A може входити тільки в одне ціле B, частина існує, тільки доки існує ціле і припиняє своє існування разом з цілим. Графічно відношення композиції відображається зафарбованим ромбом 1 (рис. 17.13).

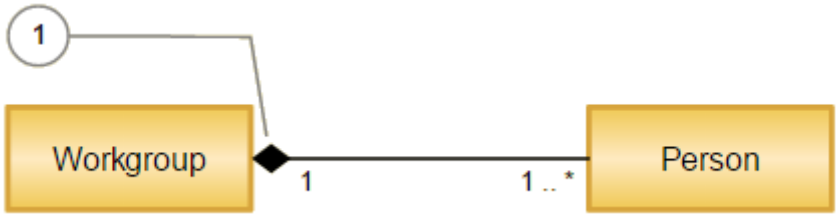


Рисунок 17.13 – Відношення композиції

При використанні відношення композиція, «ціле» часто називають ***композитом,*** а при використанні агрегації – ***агрегатом***.

У комбінації з вказівкою кратності, відношення асоціації, агрегації і композиції дозволяють лаконічно і повно відобразити структуру класів: що з чого полягає і як пов'язано. Тут наведений приклад одного з варіантів такої структури для інформаційної системи відділу кадрів (рис. 17.14).

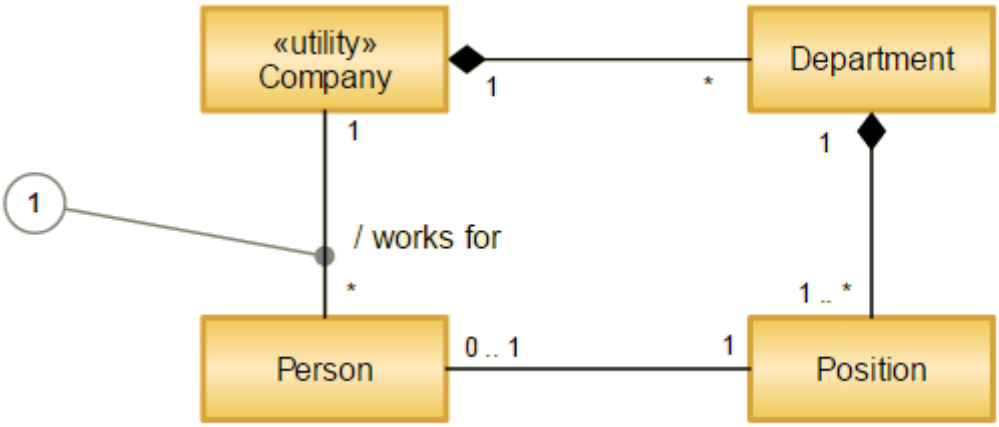


Рисунок 17.14 – Структура зв'язків класів інформаційної системи відділу кадрів

Зверніть увагу на асоціацію з ім'ям *works for* 1 на рисунку вище. Це похідна асоціація.

Похідний елемент (derived element) – це елемент, який можна обчислити або визначити за іншими елементами.

Формально похідний елемент зайвий, він вводиться в модель для ясності або наочності. Похідний елемент відзначається за допомогою знаку /, який ставиться перед його ім'ям. Похідним елементом в UML може бути атрибут, роль полюса або асоціація.

У яких випадках при моделюванні слід використати атрибути, а в яких – композицію? Відповідь на це питання залежить від того, які ще відношення між складовими частинами треба визначити. Якщо ніяких відношень немає, наприклад, якщо йдеться про частини, які мають примітивні типи (числа, рядки, і тому подібне), то слід використати атрибути. Якщо ж між частинами є відношення, зокрема, є взаємодія, то прийнятніше використати композицію, тому що нотація атрибутів не дає можливості відобразити ці відношення. Розглянемо приклад.

ЗМІНИ В ТЕХНІЧНОМУ ЗАВДАННІ

Інформаційна система відділу кадрів повинна підтримувати ієрархічну структуру підрозділів на підприємстві.

Простим і, як наслідок, не найкращим, є рішення, представлене нижче (рис. 17.15).

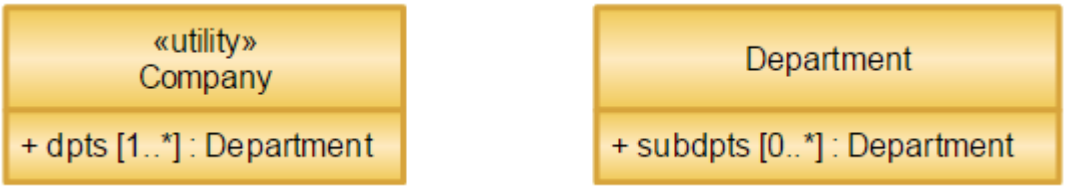


Рисунок 17.15 – Приклад використання атрибутів

Оптимальне рішення, яке дозволяє легко врахувати нову вимогу, наведене на рис. 17.16.

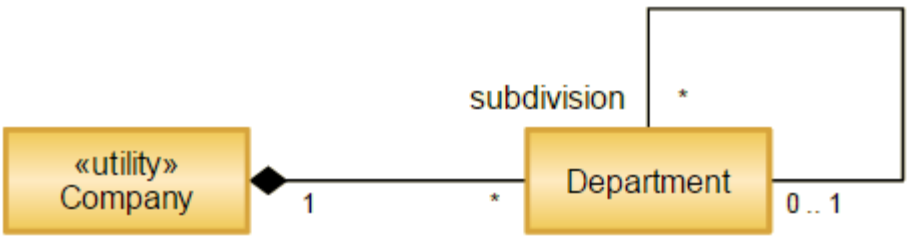


Рисунок 17.16 – Приклад використання композиції

З визначення композиції виходить, що композит управляє часом життя частин. Таким чином, йдеться про екземпляри класів, які є частинами для цього екземпляра композиту. Розглянемо приклад.

ЗМІНИ В ТЕХНІЧНОМУ ЗАВДАННІ

У підрозділі будь-якого рівня, у тому числі і на підприємстві в цілому, є єдина посада (начальник), яку система повинна трактувати особливим чином.

При реалізації цієї вимоги (див. рис. 17.17) першим спонуканням є ввести новий клас **Boss** (підклас класу **Position**) і провести композиції до класів **Company** 1 і **Department** 1**.** Як не дивно, але це не є синтаксичною помилкою. В цьому випадку мова піде про те, що що належить екземпляру класу **Company** екземпляр класу **Boss** не може в той же самий час бути частиною якого-небудь екземпляра класу **Department,** і навпаки (але самих екземплярів класу **Boss** може бути декілька).

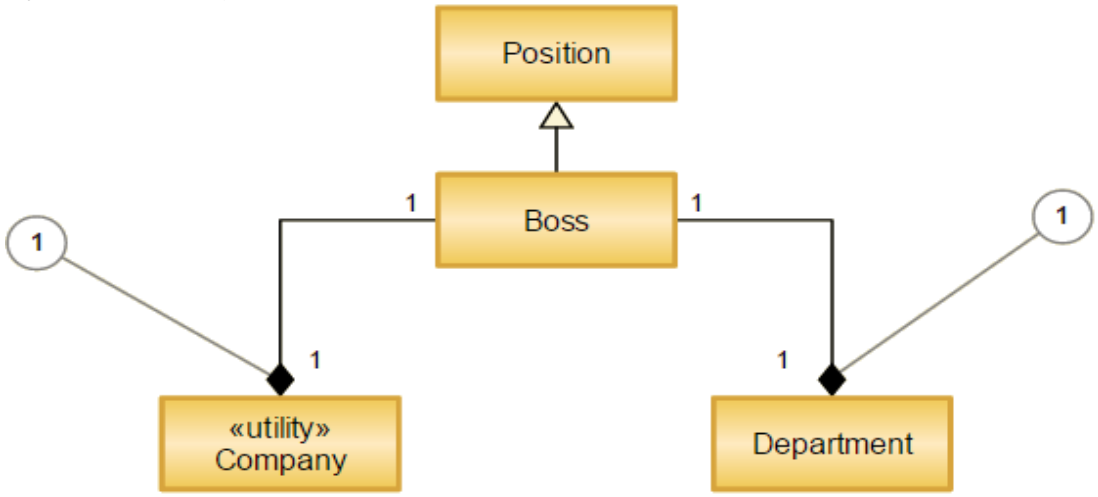


Рисунок 17.17 – Перший варіант реалізації складної композиції

Другий варіант рішення полягає в такому: якщо у групи класів є щось загальне, то можна завести абстрактний суперклас (клас **UnitWithBoss**) і встановити необхідну композицію з ним (рис. 17.18).

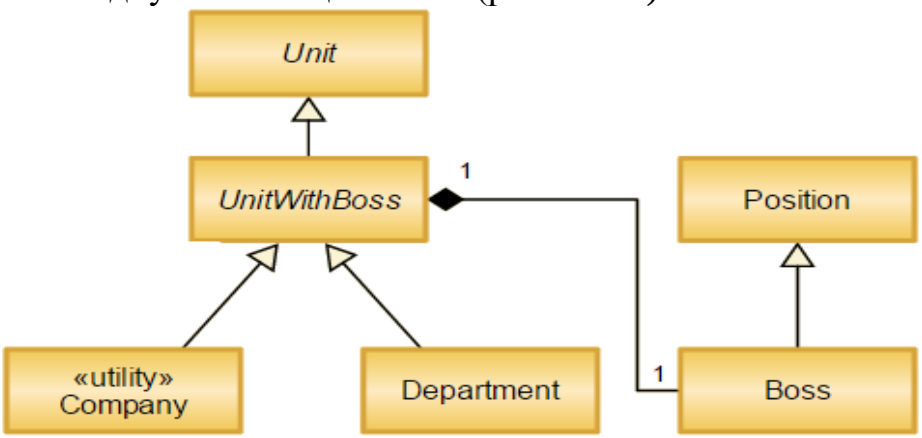


Рисунок 17.18 – Другий варіант реалізації складної композиції

Третій варіант, показаний на рис. 17.19, використовує ще один засіб UML – ***узагальнення асоціації.*** Клас **Boss** бере участь не в трьох композиціях, як може здатися на перший погляд, а в одній, з абстрактним класом Unit 1. Дві інші композиції до класів **Company** і **Department**, не є незалежними відношеннями, вони є окремими випадками одного відношення композиції, що показано стрілками узагальнення 2 і 3.

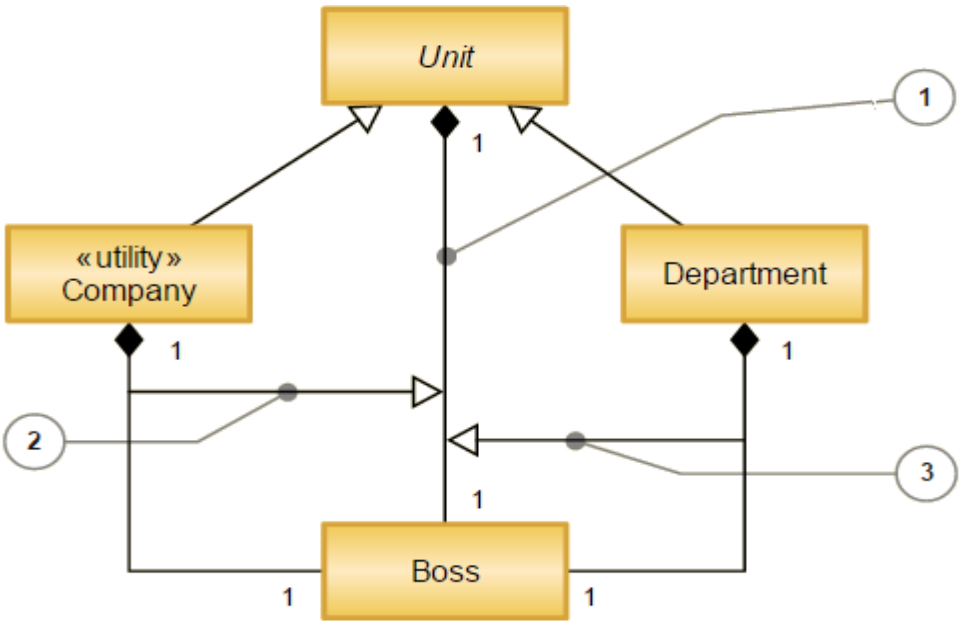


Рисунок 17.19 – Третій варіант реалізації складної композиції

Остання модель найгнучкіша, вона дозволяє врахувати різні додаткові вимоги, наприклад, різну кратність полюсів в різних окремих випадках асоціації або різні ролі, які грають класифікатори на полюсах цієї асоціації.

На рис. 17.20 роль полюса класу **Boss** в асоціації з **Unit** носить ім'я **Leader.** Спеціалізації цієї асоціації 1 не лише перевизначають ім'я цієї ролі **(CEO** і **Top Manager),** але також і додають тип **(INoReport** і **IReport**, відповідно).

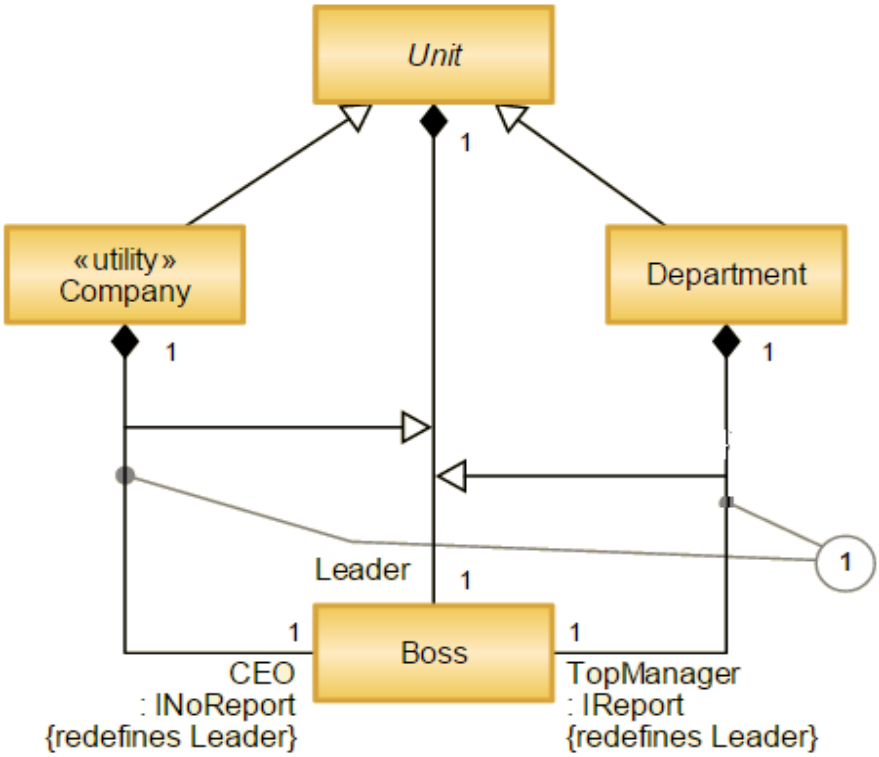


Рисунок 17.20 – Використання redefines для полюсів асоціації

Роль полюса асоціації. Багатополюсна асоціація. Полюс асоціації – це точка зіткнення лінії асоціації з прямокутником класу. Саме поблизу цієї точки розташовуються численні доповнення полюсів асоціації.

Роль полюса асоціації, що називається також ***специфікатором інтерфейсу*** **–** це спосіб вказати, як саме класифікатор (приєднаний до цього полюс асоціації) бере участь в асоціації.

Нотація цього доповнення – текст, вказаний на полюсі асоціації. У загальному випадку роль полюса асоціації має такий синтаксис:

видимість ІМ'Я: тип

Ім'я є обов'язковим, воно називається ім'ям ролі і фактично є власним ім'ям полюса асоціації. Якщо розглядається одна асоціація, що сполучає два різні класи, то в іменах ролей немає потреби: полюси асоціації легко можна розрізнити за іменами класів, до яких вони приєднані. Наведемо приклад (рис. 17.21).

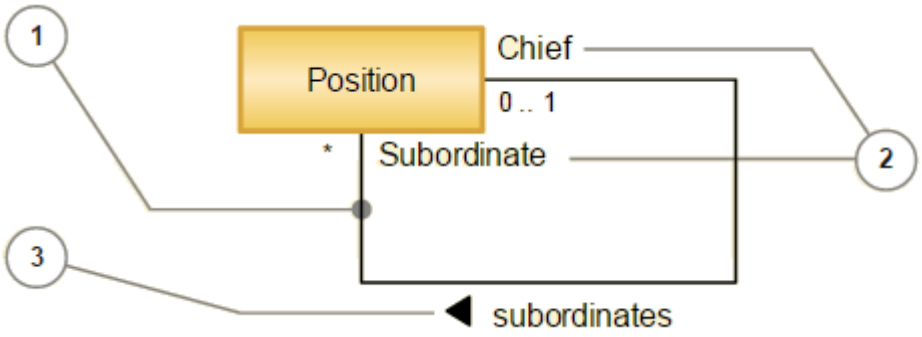


Рисунок 17.21 – Опис ієрархії посад

На рис. 17.21 зображена асоціація класу **Position** з самим собою 1. На полюсах асоціації вказані ролі 2. Значок, що показує напрям читання 3 – чорний трикутник – дозволяє прочитати цю асоціацію як **Chief subordinates Subordinate.** Ця асоціація покликана відбити наявність ієрархії підлеглості посад в організації. Проте на наведеному рисунку видно тільки, що об'єкти класу **Person** утворюють деяку ієрархію (кожен об'єкт пов'язаний з деякою кількістю об'єктів, що пролягають нижче в ієрархії, і не більше ніж з одним вищерозміщеним об'єктом).

Використовуючи ролі і, заразом, відношення реалізації, можна описати субординацію в інформаційній системі відділу кадрів досить точно.

Наприклад, на рис. 17.22 вказано, що в ієрархії субординації кожна посада може грати дві ролі. З одного боку, посада може розглядатися як начальницька 1 (chief), і в цьому випадку вона надає інтерфейс **IChief** 2 що має операцію **petition()** (начальникові можна подати службову записку). З іншого боку, посада може розглядатися як підлеглий 3 **(subordinate),** і в цьому випадку вона надає інтерфейс **ISubordinate** 4, що має операцію **report()** (від підлеглого можна зажадати звіт). У начальника може бути довільна кількість підлеглих 5, у тому числі і, у підлеглого може бути не більший за одного начальника 6.

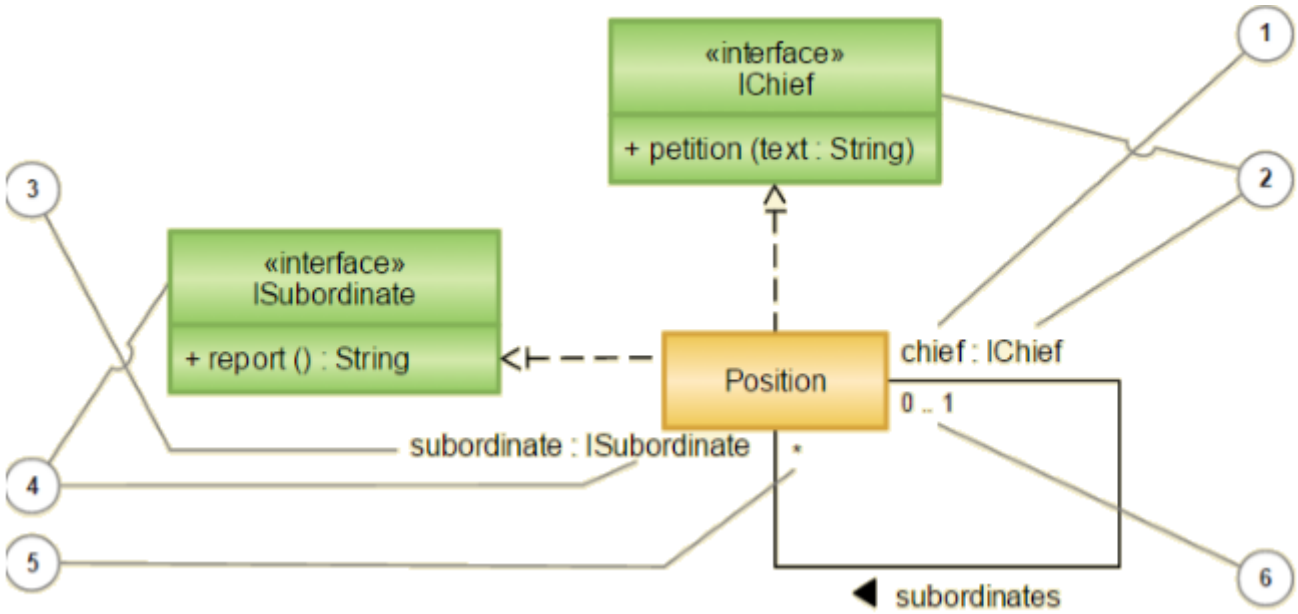


Рисунок 17.22 – Ролі полюсів асоціації

У організації застосовується сучасна організаційна форма управління і окрім ієрархії підрозділів і посад існує структура виконуваних проектів, що «пронизують» організацію[v]. Таку хитромудру (але дуже життєву) ситуацію дуже легко відобразити, використовуючи багатополюсні асоціації (рис. 17.23).

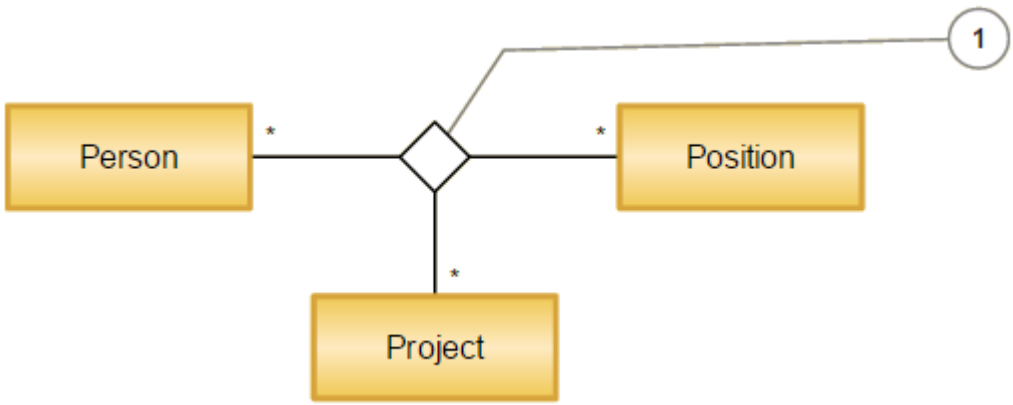


Рисунок 17.23 – Багатополюсна асоціація

Залишається сказати, що нотація багатополюсної асоціації є ромбом 1, до якого за допомогою ліній приєднуються усі класи, що беруть участь в асоціації.

Щоб повністю оцінити корисність багатополюсних асоціацій можна спробувати побудувати модель, семантично еквівалентну тільки що наведеній, але без використання багатополюсних асоціацій, в цьому випадку доведеться ввести додаткові сутності і відношення.

Таким чином, має місце складніше відношення між посадами, співробітниками і проектами, ніж ті, що наведені вище. А саме, допускається не лише поєднання посад (один співробітник може працювати на декількох посадах в різних проектах), але і дроблення ставок (одну посаду можуть обіймати декілька співробітників – півставки, чверть ставки і тому подібне).

Використовуючи ж розібрану нотацію асоціації, ми можемо констатувати, що між класами **Person, Position** і **Project** має місце асоціація «багато до багатьох». В даному випадку можна застосувати стандартний прийом нормалізації, який часто використовується при проектуванні схем баз даних: систематичним чином позбавитися від відношень «багато до багатьох» шляхом введення додаткової сутності 1 і трьох відношень «один до багатьох». Стосовно цього прикладу такий прийом дає рішення, наведене на рис. 17.24.

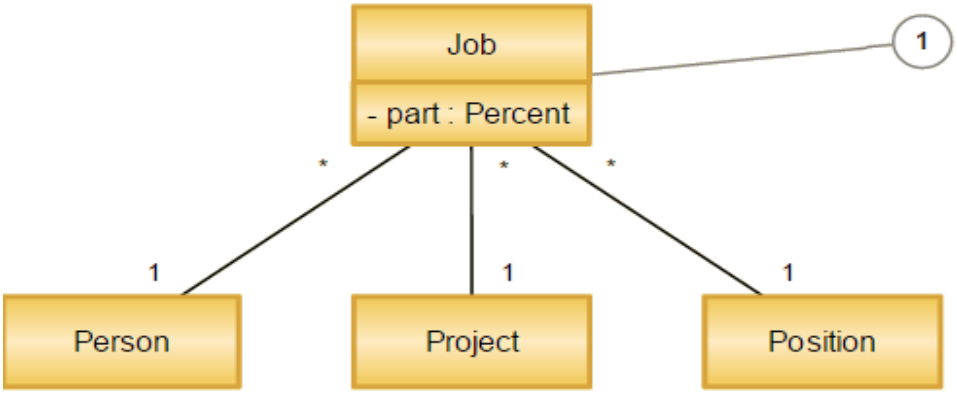


Рисунок 17.24 – Усунення відношення «багато до багатьом» за допомогою введення додаткової сутності

На цьому закінчимо обговорення діаграм класів – найважливішого засобу опису структури в UML– серією елементарних порад з практичного моделювання структури. Як видно з попередніх розділів, діаграми класів містять множину деталей. Для практично значимих систем діаграми класів зрештою виходять досить складними. Намагатися промальовувати складну діаграму класів відразу «на усю глибину» нераціонально – занадто великий ризик «потонути» в деталях. Вдала модель структури складної системи створюється за декілька (можливо, навіть за декілька десятків) ітерацій, в яких моделювання структури перемежається моделюванням поведінки. Ось ці поради.

1. Описувати структуру зручніше паралельно з описом поведінки.
2. Не обов'язково включати в модель усі класи відразу. На перших ітераціях досить ідентифікувати дуже невелику (10%) долю усіх класів системи.
3. Не обов'язково визначати усі складові класу відразу. Розпочніть з імені класу – операції і атрибути поступово виявляться в процесі моделювання поведінки.
4. Не обов'язково показувати на діаграмі усі складові класу і їх властивості.
5. Не обов'язково визначати усі відношення між класами відразу. Нехай клас на діаграмі «висить в повітрі» – нічого з ним не станеться.

Основна діаграма класів ІС ВК наведена на рис. 17.25.

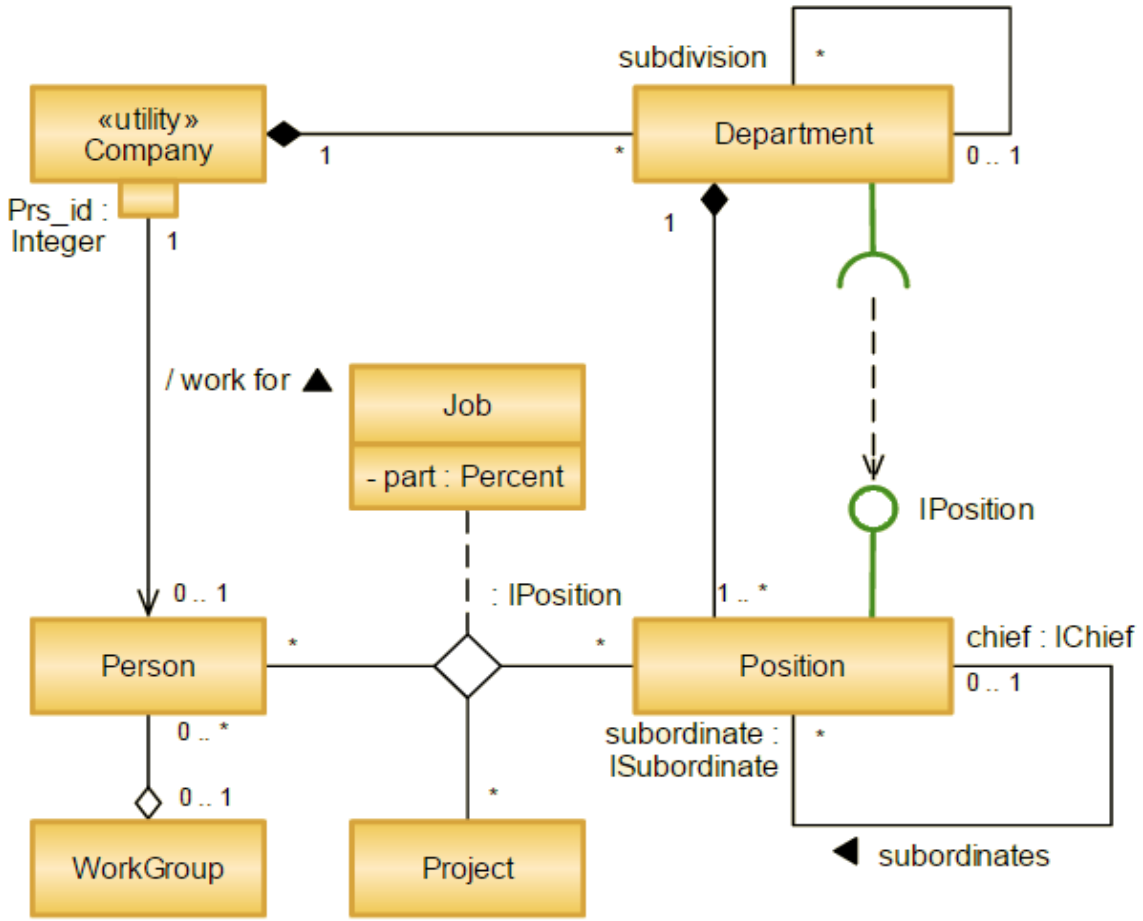


Рисунок 17.25 – Основна діаграма класів ІС ВК

17.3. Діаграми реалізації

Цей підрозділ присвячений відразу двом діаграмам: ***компонентів*** і ***розміщення,*** для яких можна використати узагальнювальну назву – **діаграми реалізації**. Пов'язано це з тим, що ці діаграми набувають особливої важливості на пізніших фазах розробки – на фазах реалізації і постачання.

З точки зору реалізації система, що проектується, складається з компонентів (представлених на **діаграмах компонентів),** розподілених за обчислювальними вузлами (представленими на **діаграмах розміщення**).

У UML 2 в порівнянні з UML 1 сталася значна зміна, а саме, поняття «компонент» було розділене на дві складові: логічну і фізичну. Логічна складова, що продовжує носити ім'я **компонент,** є елементом логічної моделі системи, тоді як фізична складова, що називається **артефактом,** втілює фізичний елемент системи, що проектується, і розміщується на **обчислювальному вузлі** (node).

Діаграми компонентів і розміщення мають багато спільного, об'єднуючи воєдино такі, найтіснішим чином пов'язані, елементы:

- структуру логічних елементів (компонентів);
- відображення логічних елементів (компонентів) на фізичні елементи (артефакти);

– структуру використовуваних ресурсів (вузлів) з розподіленими по них фізичними елементами (артефактами).

Тут ми окремо для кожної діаграми не розглядатимемо сутності, вживані на ній. Розглянемо усі сутності і відношення в одному розділі.

Інтерфейс. Поняттю інтерфейс і відношеннями між класифікаторами і інтерфейсами ми приділили достатньо уваги.

Тут можна виділити дві ролі (див. рис. 17.26), які грають інтерфейси 1 по відношенню до класифікаторів, наприклад, компонент 2. Інтерфейс може бути *забезпеченим і потрібним*.

Сам по собі інтерфейс – це просто опис контракту, а забезпеченим або потрібним він ставатиме залежно від того, як цей інтерфейс використовується:

– якщо класифікатор реалізує інтерфейс – то для цього класифікатора це ***забезпечений*** інтерфейс і цей факт показується за допомогою відношення реалізації 3;
– якщо класифікатор викликає операції інтерфейсу – то для цього класифікатора це ***необхідний*** інтерфейс і цей факт показується за допомогою відношення залежності 4.

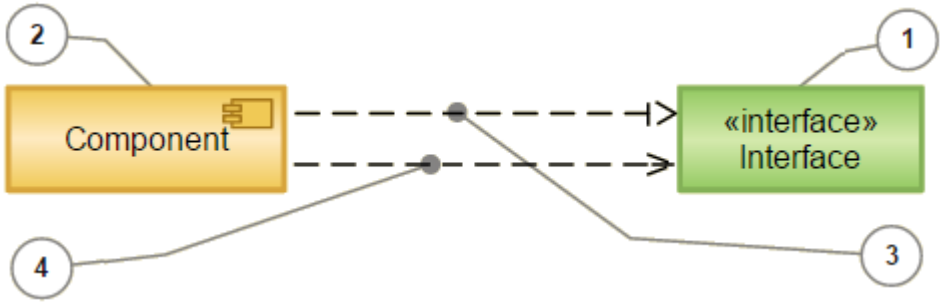


Рисунок 17.26 – Відношення між класифікаторами і інтерфейсами

Компоненти, артефакти і вузли. ***Компонент*** **–** це модульний фрагмент логічного представлення системи, взаємодія з яким описується набором забезпечених і необхідних інтерфейсів.

З поняттям «компонент» часто асоціюють компонентне або складальне програмування, проте для UML ця відповідність не правомірна. Компонент UML є частиною моделі, і описує логічну сутність, яка існує тільки під час *проектування (design time), хоча надалі її можна зв'язати з фізичною реалізацією (артефактом) часу виконання (*run time).

У UML для компонентів передбачені стереотипи (таблиця. 17.3).

Таблиця 17.3 – Стандартні стереотипи компонентів

Стереотип	Опис
<<buildComponent>>	Компонент, що служить для розробки додатка
<<entry>>	Інформаційний компонент, що постійно зберігається
<<service>>	Функціональний компонент без стану
<<subsystem>>	Одиниця ієрархічної декомпозиції великої системи

Артефакт – це будь-який створений штучно елемент програмної системи (виконувані файли, початкові тексти програм, веб-сторінки, довідкові файли, супровідні документи, файли з даними і т. д).

Для того щоб якось відбивати таку різноманітність типів артефактів в UML передбачені стандартні стереотипи, перераховані в таблиці. 17.4.

Таблиця 17.4 – Стандартні стереотипи артефактів

Стереотип	Опис
<<file>>	Файл будь-якого типу, що зберігається у файловій системі
<<document>>	Артефакт – файл (документ), який не є ні файлом початкових текстів, ні виконуваним файлом
<<executable>>	Здійснима програма будь-якого виду
<<library>>	Статична або динамічна бібліотека
<<script>>	Файл, що містить текст, що допускає інтерпретацію засобами, що відповідають програмним
<<source>>	Файл з початковим кодом програми

Найважливішим аспектом використання поняття артефакту в UML є те, що артефакт може брати участь відношенні маніфестації.

Маніфестація **–** це відношення залежності із стереотипом **«manifest»,** що зв'язує елемент моделі (наприклад, клас або компонент) і його фізичну реалізацію у вигляді артефакту.

Нижче представлений клас **Company,** який пов'язаний відношенням маніфестації з двома артефактами із стереотипом «source», які у свою чергу визначають артефакт часу виконання динамічну бібліотеку (із стереотипом «library») **Company (рис.** 17.27).

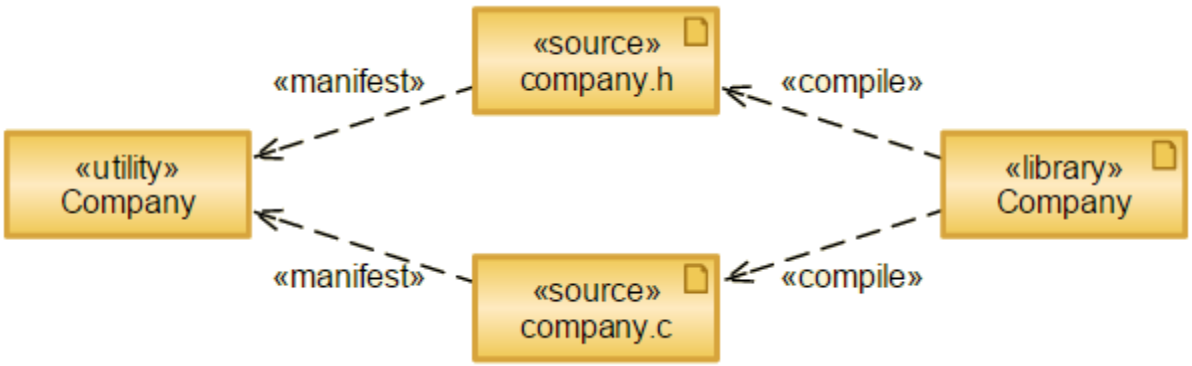


Рисунок 17.27 – Артефакти

Взагалі кажучи, відношення маніфестації – це відношення типу «багато до багатьох», один елемент моделі може бути реалізований багатьма артефактами, і один артефакт може брати участь в реалізації багатьох елементів моделі.

Маніфестацію графічно зображають відношенням залежності із стереотипом «manifest» від артефакту до сутнгості, що реалізовується. Оскільки маніфестація – це відношення типу «багато до багатьох», для повного опису

відношення маніфестації можуть знадобитися декілька відношень залежності в моделі.

Третя сутність, що розглядається в цьому розділі – вузол.

Вузол (node) – це фізичний обчислювальний ресурс, що бере участь в роботі системи.

У UML передбачено два стереотипи для вузлів «**executionEnvironment» і «device».** Вузол із стереотипом **«executionEnvironment»** дозволяє моделювати апаратно-програмну платформу, на якій відбувається виконання додатка. Прикладами середовища виконання є: операційна система, система управління базами даних і так далі

Вузол із стереотипом «device» також моделює апаратно-програмну платформу, але допускає можливість вкладення одного вузла в інший, як це показано на рис. 17.28.

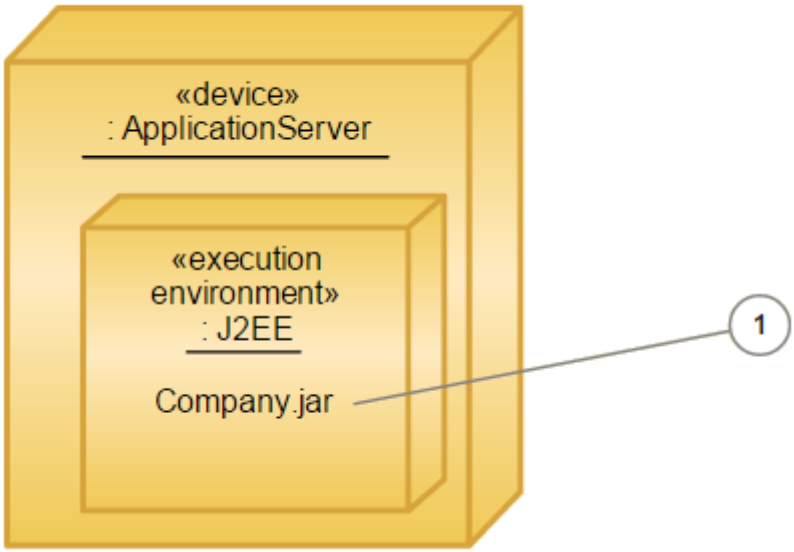


Рисунок 17.28 – Нотація вузла

Артефакти системи під час її роботи розміщуються на вузлах, що графічно виражається або їх перерахуванням усередині вузла 1 (див. рис. 17.27), або відношенням залежності із стереотипом «deploy» між артефактом і вузлом 1 (див. рис. 17.29), або зображенням артефакту усередині зображення вузла 2 (див. рис. 17.29). Усі варіанти нотації рівноправні.

Застосування діаграм компонентів і розміщення. Давайте спробуємо відповісти на питання, які інтерфейси, компоненти і артефакти можна виділити в інформаційній системі відділу кадрів, а також як доцільно розмістити розроблене програмне забезпечення на обчислювальних вузлах.

Основне призначення інформаційної системи, що проектується, – зберігати дані про кадри і виконувати за вказівкою користувача деякі операції з цими даними. Аналізуючи склад операцій, ми бачимо, що вони зводяться до створення, модифікації і видалення елементів даних, що зберігаються. Стандартним рішенням в таких ситуаціях є застосування готової СУБД (DBMS).

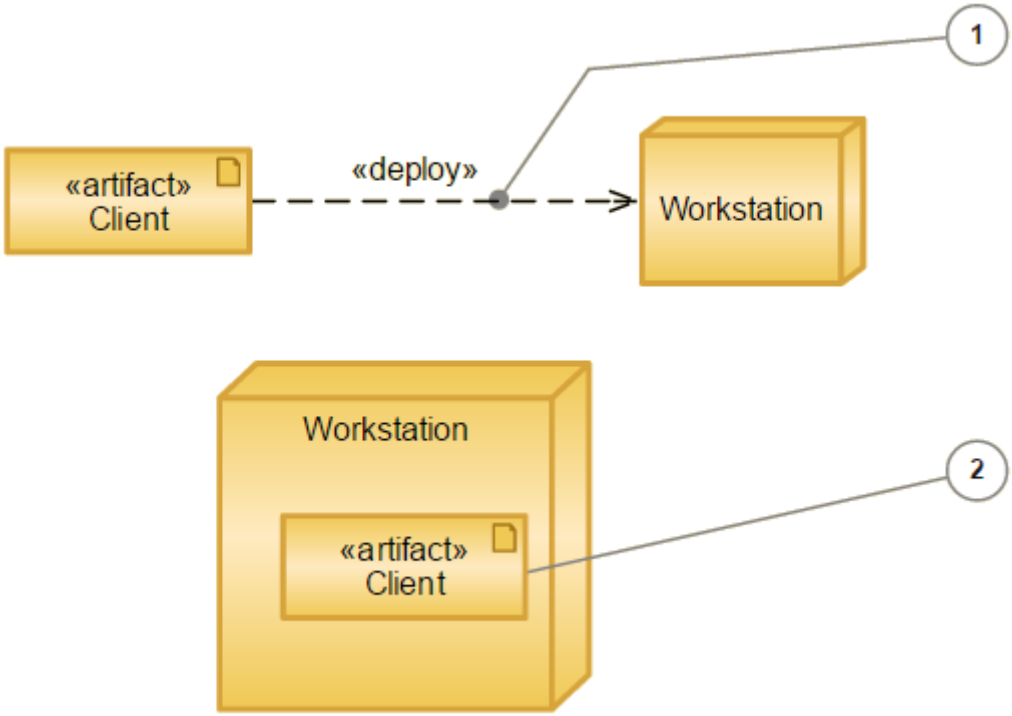


Рисунок 17.29 – Нотації розміщення артефакту на вузлі

З точки зору проектування інформаційної системи відділу кадрів доцільно вважати використовувану СУБД готовим компонентом із заздалегідь визначеними інтерфейсами і протоколом взаємодії.

СУБД візьме на себе усі функції з безпосереднього маніпулювання даними: створення, видалення і пошук записів в таблицях і так далі. Реалізація операцій інформаційної системи відділу кадрів зводиться до деякої послідовності елементарних операцій з даними.

Наприклад, операція переведенняу співробітника з однієї посади на іншу, мабуть, зажадає зміни в трьох елементах даних: у даних про співробітника і в даних про стару і нову посади. Проте чи доцільно вважати, що визначення і виконання самої послідовності елементарних операцій з даними також є прерогативою виділеного нами компонента – СУБД? Загальноприйнята практика відповідає на це питання негативно. З багатьох причин краще виділити це в окремий компонент, що зазвичай називається бізнес-логікою. Окрім цього, ми повинні припустити, що в нашій системі має бути компонент, відповідальний за призначений користувацький інтерфейс. Тобто, ми приходимо до структури компонентів, наведеної нижче, яка називається «Трирівнева архітектура» (рис. 17.30).

У версії UML 2 сталися такі зміни в нотації діаграм компонентів.

По-перше, компоненти, як і будь-який класифікатор, можна зображати однаково, у вигляді прямокутників, в яких вказується або стереотип «**component**» 1, або один із уточнюючих стереотипів, наведених в табл. 17.3 (Стандартні стереотипи компонент) 2, або відповідний значок в правому верхньому кутку прямокутника 3.

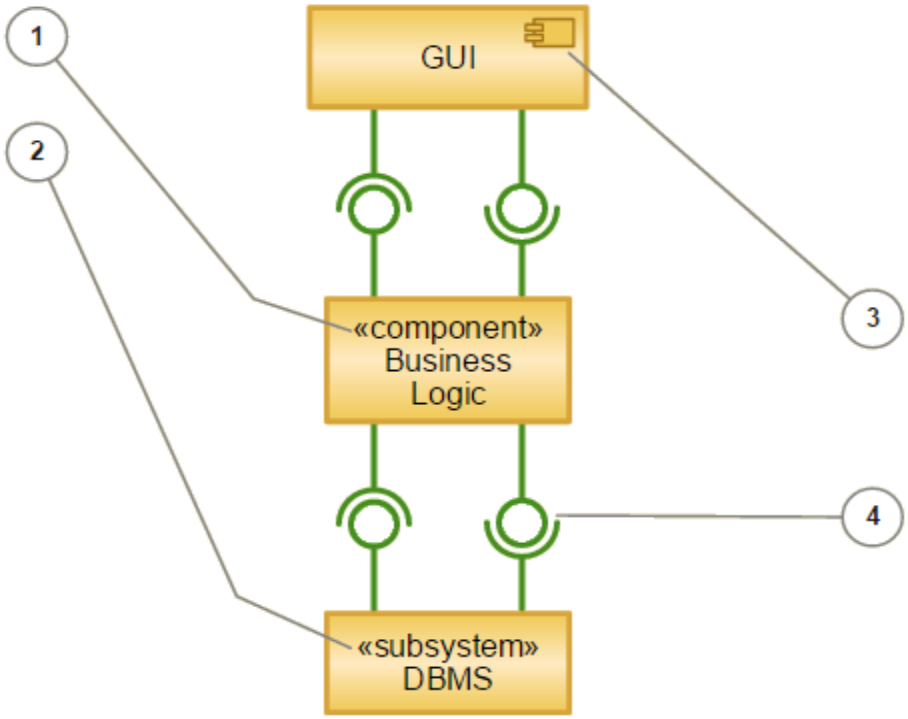


Рисунок 17.30 – Діаграма компонентів ІС ВК, виконана в нотації UML 2

По-друге, необхідні і забезпечені інтерфейси можна зображувати за допомогою нотації «чупа-чупс» 4, так що відношення взаємодії компонентів через деякий інтерфейс виглядає природним і симетричним.

У завершення дамо декілька порад з приводу того, в яких випадках слід застосовувати діаграми компонентів і розміщення.

Розпочнемо із уже висловленого елементарного міркування: у разі розробки «монолітного» настільного додатка діаграми розміщення не потрібні – вони виявляються тривіальними і ніякої корисної інформації не містять. Таким чином, діаграми розміщення застосовуються тільки при моделюванні багатокомпонентних застосувань.

РОЗДІЛ 18. МОДЕЛЮВАННЯ ПОВЕДІНКИ ІС ВІДДІЛУ КАДРІВ

Як вже говорилося вище, при створенні програмної системи недостатньо відповісти на питання «що робить система»? і «з чого вона складається»? – вимагається відповісти на питання «як працює система»?. Відповідь на це питання називається (в UML) *моделлю поведінки*.

Поведінка реальної програмної системи цілком і повністю визначається кодом її програми – як програма складена, так вона і виконується. Таким чином, ***модель поведінки*** – це опис алгоритму роботи системи. Для опису життєвого циклу конкретного об'єкту, поведінка якого залежить від історії цього об'єкту, або ж вимагає обробки асинхронних стимулів, використовується кінцевий автомат у формі **діаграми станів**.

Для опису потоку управління, тобто послідовності виконуваних елементарних кроків при виконанні окремої операції або реалізації складного варіанту використання, зручно використовувати **діаграми діяльності**.

Взаємодія декількох програмних об'єктів між собою описується **діаграмами взаємодії** в одній з двох еквівалентних форм **(діаграми кооперації** і **діаграми послідовності**).

18.1. Моделювання поведінки за допомогою кінцевих автоматів

18.1.1. Кінцеві автомати

Нескладна конструкція кінцевого автомата виявляється застосовною для адекватного опису дуже багатьох ситуацій. Наведемо невеликий приклад з нашої інформаційної системи відділу кадрів. Співробітник в організації, очевидно, може знаходитися в різних станах: спочатку він є кандидатом, в результаті виконання операції прийому на роботу він стає штатним співробітником. Штатний співробітник може бути переведений з однієї посади на іншу, залишаючись штатним співробітником. Нарешті, співробітник може бути звільнений. Життєвий цикл співробітника природно описати кінцевим автоматом, наприклад, у вигляді таблиці 18.1. У цій таблиці рядки поіменовані станами, стовпці – стимулами, а в осередках (ячейках) виписані процедура реакції і новий стан.

Таблиця 18.1 – Життєвий цикл співробітника

	Прийом	**Переклад**	**Звільнення**
Кандидат	Прийняти() У штаті	Помилка() Кандидат	Помилка() Кандидат
У штаті	Помилка() У штаті	Перевести() У штаті	Звільнити() Звільнений
Звільнений	Прийняти() У штаті	Помилка() Звільнений	Помилка() Звільнений

Якщо така таблиця здається недостатньо інформативною і наочною, то цю інформацію можна представити у формі діаграми станів-переходів. На рис. 18.1 наведена діаграма станів-переходів, що відповідає даному випадку, в нотації UML (Candidate – кандидат, Employee – службовець, Retired – звільнений, hire – прийом, move – переміщення, fire – звільнити).

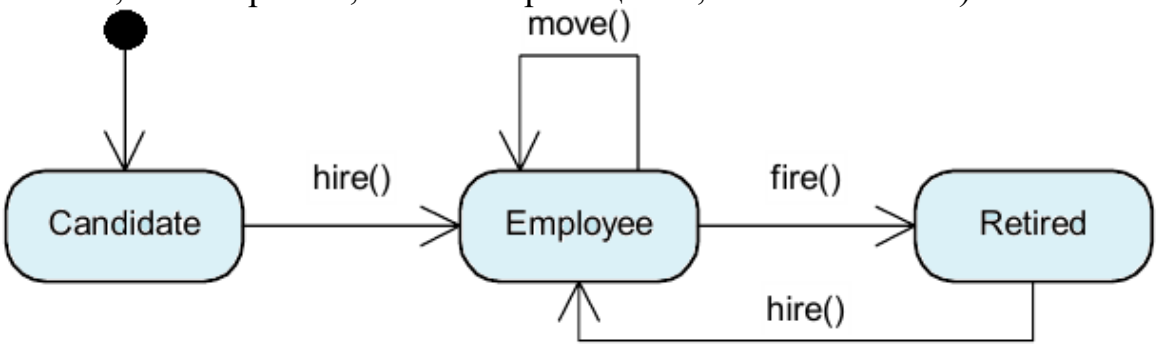


Рисунок 18.1 – Життєвий цикл співробітника в ІС ВК

18.1.2. Діаграми станів (автомата)

Діаграми станів (автомата) в UML є реалізацією основної ідеї використання кінцевих автоматів як засоби опису алгоритмів і, тим самим, моделювання поведінки.

Як ми вже знаємо, на діаграмах станів застосовується всього один тип сутностей – *стани*, і всього один тип відношень – *переходи.* Сукупність станів і переходів між ними утворює *машину станів*.

Таким чином, типів сутностей і відношень гранично мало, але підтипів, варіантів нотації і спеціальних випадків для них визначено багато. А саме, стани бувають:

- *прості;*
- *складені* двох видів: ортогональні (іноді їх називають *паралельними*) і ні;
- *спеціальні;*
- *посилальні.*

Переходи бувають *прості* і *складені* (подія, сторожова умова, дія). Розглянемо деякі з цих елементів на нашому прикладі.

Простий стан. Простий стан є в UML простим тільки по назві – він має таку структуру:

- ім’я;
- дія при вході;
- дія при виході;
- множина внутрішніх переходів;
- внутрішня активність;
- множина відкладених подій.

Ім’я стану є обов’язковим. Усі інші складові простого стану не є обов’язковими. Фактично, ім’я (простого) стану – це символ алфавіту станів Q.

Дія при вході (позначається за допомогою ключового слова entry) – це вказівка атомарної дії, яка повинна виконуватися під час переходу автомата в цей стан.

Дія при виході (позначається за допомогою ключового слова exit) – це вказівка атомарної дії, яка повинна виконуватися під час переходу автомата з даного стану.

Множина внутрішніх переходів – це множина простих переходів з цього стану в цей же стан (так званих переходів в себе). Внутрішній перехід відрізняється від простого переходу в себе тим, що дії при виході і вході не виконуються.

Внутрішня активність (позначається за допомогою ключового слова do) – це вказівка діяльності, яка починає виконуватися при переході в цей стан після виконання усіх дій, запропонованих переходом, включаючи дію на вході. Внутрішня активність або закінчується після закінчення, або уривається у разі виконання переходу (у тому числі і внутрішнього переходу).

Відкладена подія – це подія, для якої не визначено переходу в цьому стані, але яка, проте, не має бути втраченою, якщо воно станеться, поки автомат знаходиться в цьому стані.

Розглянемо наш приклад з інформаційної системи відділу кадрів. Співробітник в організації, очевидно, може знаходитися в різних станах: спочатку він є кандидатом (**Applicant),** в результаті виконання операції прийому **(hire())** на роботу він стає штатним співробітником **(Employed).** Штатний співробітник може бути переведений з однієї посади на іншу **(move()),** залишаючись штатним співробітником. Нарешті, співробітник може бути звільнений **(fire(), Unemployed**). Життєвий цикл співробітника природно описати кінцевим автоматом. Якщо нічого не відбувається, то за замовчуванням співробітник повинен виконувати свої основні обов'язки. На рис. 18.2 наведена діаграма станів-переходів, що відповідає даному випадку, в нотації UML.

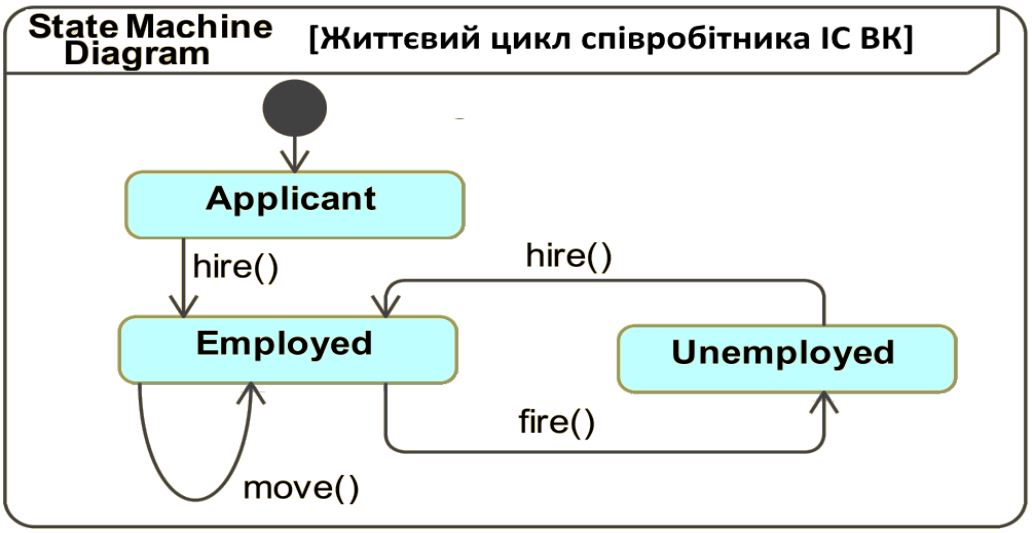


Рисунок 18.2 – Життєвий цикл співробітника ІС ВК

Вище ми відмітили, що багато елементів діаграм станів в UML введені тільки для зручності, без них можна в принципі обійтися. На підтвердження нашої тези наведемо один з можливих варіантів виключення додаткових елементів з моделі зі збереженням її семантики. Припустимо, що є простий стан

State1, описаний на рис. 18.3 з використанням таких засобів, як дія при вході, дія при виході і внутрішній перехід.

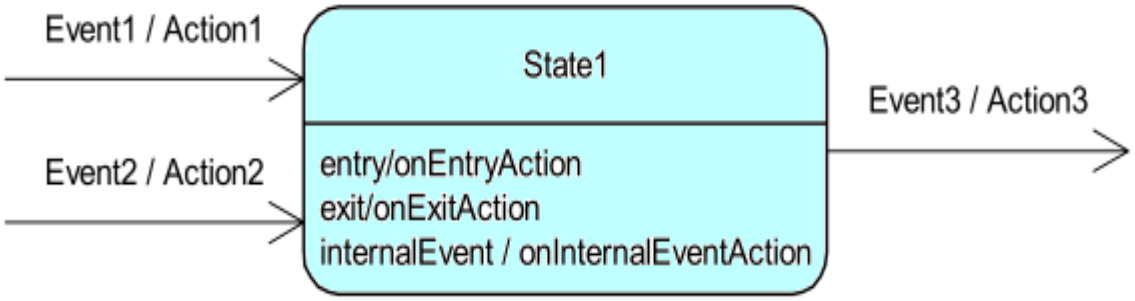


Рисунок 18.3 – Фрагмент моделі з простим станом до перетворення

В цьому випадку еквівалентну за поведінкою модель, що не використовує додаткових засобів, а тільки імена простих станів і прості переходи (класичний варіант кінцевого автомата), можна отримати, переносячи дії на вході і виході в в дії вхідних і вихідних переходів, відповідно, і моделюючи внутрішній перехід переходом в себе (рис. 18.4). Який стиль віддати перевазі – діло смаку.

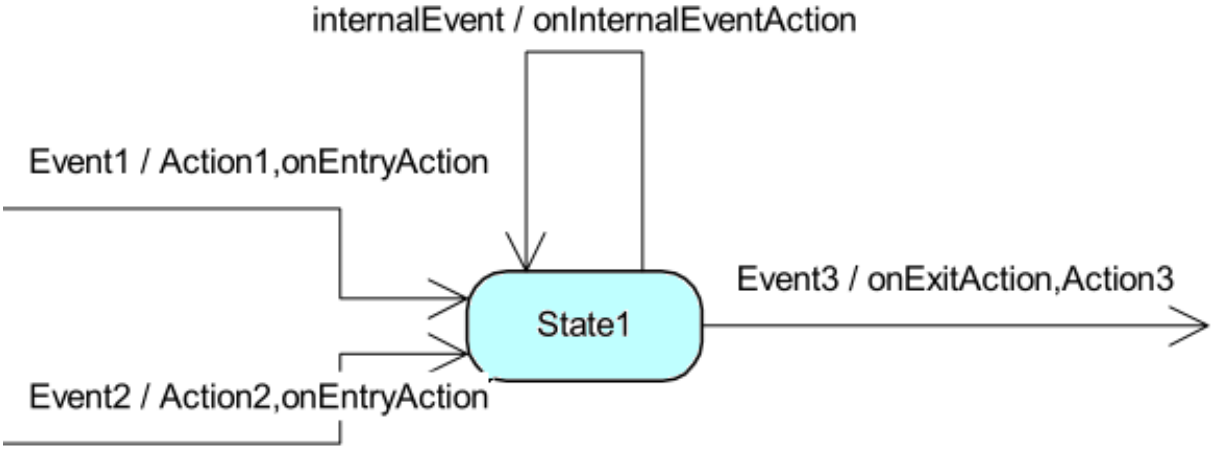


Рисунок 18.4 – Фрагмент моделі з простим станом після перетворення

Простий перехід. Простий перехід завжди веде з одного стану в інший стан. Існує декілька обмежень для спеціальних станів, наприклад, для початкового стану не може бути переходів, що входять, а для завершального – вихідних, а в іншому переходи між станами можуть бути визначені довільним чином.

Інші складові – подія переходу, сторожова умова і дії на переході не є обов'язковими. Якщо вони присутні, то зображаються у вигляді тексту в певному синтаксисі поряд із стрілкою, що зображає перехід. Синтаксис опису переходу такий:

Подія [Сторожова умова] / Дія

Наприклад, в інформаційній системі відділу кадрів перехід співробітника із стану «кандидат» (Applicant) в стан «в штаті» (Employed) може бути описаний за допомогою простого переходу, представленого на рис. 18.5.

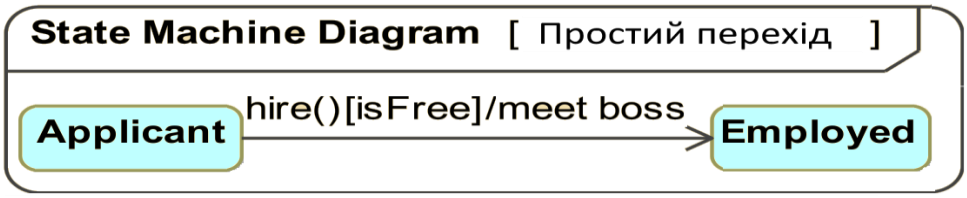


Рисунок 18.5 – Простий перехід

Подія переходу. ***Подія переходу*** **–** це той вхідний символ (стимул), який разом з поточним станом автомата визначає наступний стан. UML допускає наявність переходів без подій – такий перехід називається переходом після закінчення. *Перехід після закінчення* – це перехід без подій.

У UML передбачається декілька типів подій. Саме слово «подія» мимоволі викликає таку асоціацію: існує деякий зовнішній по відношенню до автомата світ, в якому час від часу відбуваються події, автомату стає відомо про подію, що сталася, і він реагує на подію шляхом переходу в певний стан. Ця асоціація цілком правомірна, якщо йдеться про моделювання життєвого циклу об'єкту в програмі, керованій подіями. Дійсно: в цьому випадку основний тип подій – це події виклику методів об'єкту. Об'єкт реагує на них, виконуючи тіла методів і міняючи значення своїх атрибутів (стан).

Автомату не важливе джерело подій: важлива послідовність, в якій події поступають на вхід автомата, змушуючи його реагувати.

Сторожова умова. ***Сторожова умова*** – цей логічний вираз, який повинен виявитися істинним для того, щоб збуджений перехід спрацював.

Для кожного збудженого переходу сторожова умова перевіряється рівно один раз, відразу після того, як перехід збуджений і до того, як в системі стануться які-небудь інші події. Якщо сторожова умова неправдива, то перехід не спрацьовує і подія втрачається. Навіть якщо згодом сторожова умова стане істинною, перехід зможе спрацювати тільки якщо повторно виникне подія переходу.

Сторожові умови можуть написані будь-якою мовою, зокрема, природною. Тому автоматично перевірити виконання цієї вимоги неможливо.

Як вже було сказано, можливі декілька витікаючих переходів з цього стану з однією і тією ж подією переходу, але з різними сторожовими умовами. Усі такі переходи збуджуються при настанні події переходу, але тільки один спрацьовує. Сторожові умови формулюються відносно значень аргументів події переходу і значень атрибутів об'єкту, поведінка якого моделюється.

Дія при переході. Останньою складовою простого переходу є дія.

Дія – це безперервне ззовні атомарне обчислення, чий час виконання дуже малий.

Автори мови мали на увазі, що інструменти моделювання зв'язуватимуть з поняттям дії в моделі UML поняття дії в цільовій мові програмування. Наприклад, для звичайних мов програмування діями є обчислення значення виразу і привласнення його змінної, виклик процедури тощо. У UML передбачені декілька типів дій, схожих за семантикою на дії в найпоширеніших мовах програмування. Проте UML не є мовою програмування і, тим самим, не

претендує на те, щоб бути універсальною мовою опису дій. Тому поняття дії в UML свідомо недовизначене – залишена свобода, яка необхідна інструментам для несуперечливого розширення семантики дій UML до семантики дій конкретної мови програмування.

У контексті обговорення машини станів UML варто підкреслити дві обставини.

1. **Дія є атомарною і безперервною**. При виконанні дії на переході або в стані не можуть відбуватися події, що переривають виконання дії. Точніше кажучи, подія може статися, але система зобов'язана затримати її обробку до закінчення виконання дії.
2. **Дія є безальтернативною і завершуваною**. Раз почавшись, дія виконується до кінця. Вона не може «роздумувати» виконуватися або виконуватися невизначено довго.

Сегментовані переходи. Сторожові умови в автоматах – річ зручна, але що вимагає від того, що моделює певних розумових зусиль. Тому в UML передбачені синтаксичні засоби, що до деякої міри полегшують семантично правильну побудову сторожових умов за рахунок наочнішого їх зображення. Такими є:

- *сегментовані переходи*, що реалізуються за допомогою перехідних станів і станів вибору;
- предикат *else* .

Лінія переходу може бути розбита на частини, що називаються ***сегментами***.

Розбиваючими елементами є такі фігури:

- *перехідний стан* (зображається у вигляді невеликого кружка);
- *стан вибору (галуження*, зображається у вигляді ромба);
- *дії посилки і прийому сигналу* (зображаються у вигляді прапорців).

Таке дерево сегментованих переходів семантично еквівалентне множині простих переходів, яке вийде, якщо розглянути усі шляхи з початкового стану в цільові стани, вважаючи сторожові умови, що зустрічаються на шляху, сполученими кон'юнкцією.

Покажемо це на прикладі з інформаційної системи відділу кадрів. Припустимо, що вимагається проводити більше диференційовану кадрову політику і розрізняти три різні стани звільнених з підприємства:

- **non grata** – скандаліст, нероба і порушник трудової дисципліни, звільнений з ініціативи адміністрації, якого ні за яких обставин не можна наймати на роботу;
- **welcome back** – хороший працівник, з яким адміністрації довелося розстатися зважаючи на тимчасові труднощі, що переживаються підприємством, і якого при першій нагоді слід запросити назад;
- **retired** – працівник звільнився за власним бажанням, його повторний прийом повинен проходити на загальних підставах.

Це, зрозуміло, дуже груба класифікація, але для демонстраційних цілей цілком достатня. На рис. 18.6 представлений відповідний фрагмент діаграми станів, в якому використано дерево сегментованих переходів і перехідні стани.

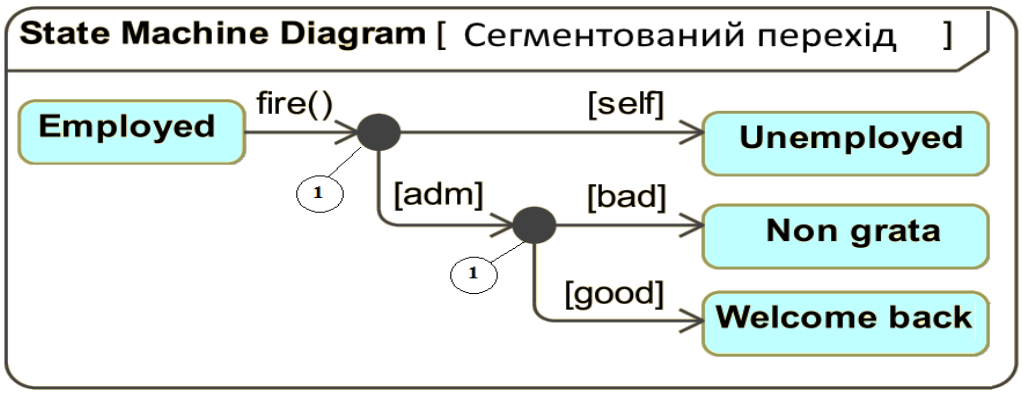


Рисунок 18.6 – Дерево сегментованих переходів

Перехідні стани у даному контексті мають той же сенс і можуть бути використані замість фігур галуження зі збереженням семантики діаграми (див. нижче рис. 18.7).

Продовжимо розгляд нашого прикладу. Відомо, що умови звільнення **adm** (адміністрація) і **self** (особисті інтереси) взаємно виключають один одного і одне з них при звільненні обов'язково має місце. Але це знаємо тільки ми – інструмент моделювання цього не знає і перевірити повноту і диз'юнктивність системи умов не зможе. Але він може допомогти позначити наше бажання наділити систему умов потрібними властивостями.

Для цього використовується ключове слово **else**, яке означає умову, що вважається істинною в усіх випадках, коли неправдиві всі інші умови, приписані до сегментів, що виходять з цього галуження (рис. 18.7).

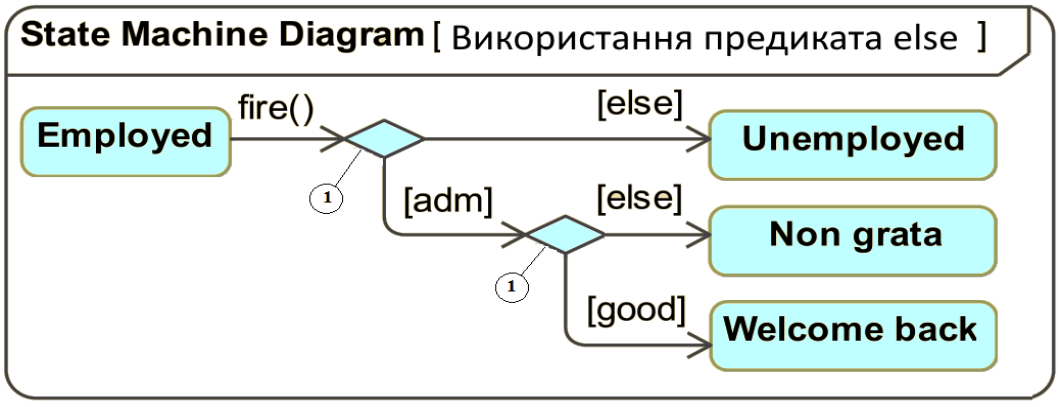


Рисунок 18.7 – Використання предиката else

Щоб підкреслити альтернативність умов, ми використали стан вибору 1 (рис. 18.7). В результаті опис складної системи умов стає наочнішим і надійнішим. Оскільки такий предикат єдиний, його можна не писати, маючи на увазі за замовчуванням. Таким чином, фрагмент діаграми станів на рис. 18.7 семантично еквівалентний фрагменту на рис. 18.6.

Підводячи підсумок обговоренню сторожових умов, ще раз підкреслимо, що сегментовані переходи і галуження нічого не додають (і не зменшують) в семантиці моделі: це просто синтаксичні позначення, що введені для зручності і наочності. Ту ж саму семантику, яку мають фрагменти діаграм станів на рис. 18.6 і рис. 18.7, можна передати за допомогою фрагмента, наведеного на рис. 18.8.

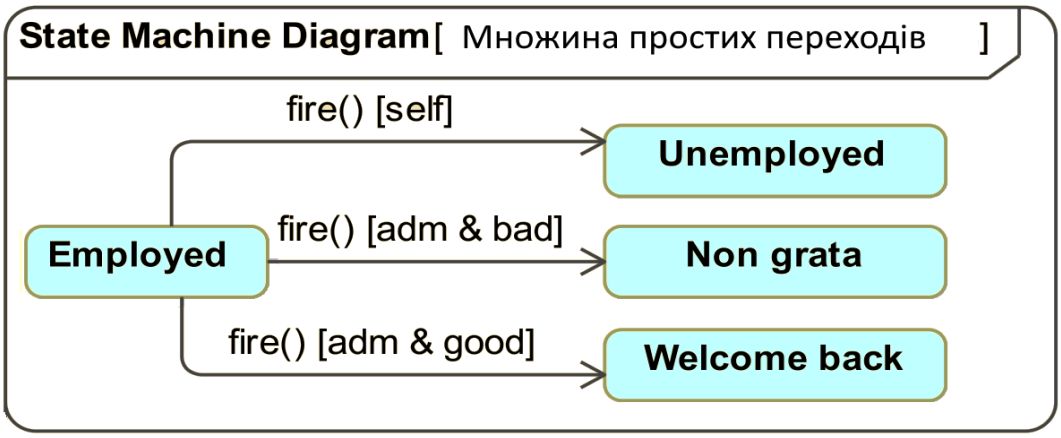


Рисунок 18.8 – Множина простих переходів з однією подією переходу і різними сторожовими умовами

Складені стани. Ми розглянули прості стани, що відповідають станам звичайної моделі кінцевого автомата. Розглянемо інше поняття, близьке до поняття стану, але специфічне для UML, – складений стан (composite state).

Складений стан – це стан, в який вкладена машина станів. Якщо вкладена тільки одна машина, то стан називається ***послідовним,*** якщо декілька ***– паралельним (ортогональним)***.

18.2. Діаграми діяльності

Основна сутність на діаграмі діяльності є частковим випадком простого стану (стан діяльності), а основне відношення – частковим випадком простого переходу (перехід після закінчення). У той же час всіляких додаткових позначень і варіантів нотації на діаграмі діяльності ще більше, ніж на діаграмі станів. Тому, щоб не загубитися в деталях, в наступному підрозділі ми обговоримо зміст основних понять, а вже в подальших підрозділах перейдемо до прикладів і основного нашого прикладу – інформаційної системи відділу кадрів.

18.2.1. Дія і діяльність

У параграфі 18.1.2 ми використали поняття дії, постулював, що дія є атомарною, безперервною ззовні, безумовною і завешуваною. Тут ми детальніше розглянемо різні типи дій в UML і на основі протиставлення визначимо поняття діяльності, стани дії і стани діяльності, які є основними елементами в діаграмах діяльності. Це дозволить нам виявити як схожість, так і відмінності діаграм станів і діяльності.

У таблиці. 18.2. наведені основні відомості про дії в UML.

Таблиця 18.2 – Дії в UML

Тип дії	Ключове слово
Присвоєння значення	**:=**
Виклик операції	**call**
Створення об’єкту	**create**
Знищення об’єкту	**destroy**
Повернення значення	**return**
Посилка сигналу	**send**
Останов	**terminate**
Дія, що не інтерпретується	будь-який текст

Згадаємо ще один засіб, що належить до дій UML, який називається повторювач. ***Повторювач*** – це вираз, що приписує виконати дію кілька разів (можливо, нуль разів). Синтаксично повторювач записується подібно до сторожової умови, в квадратних дужках, перед якими ставиться зірочка:

*** [повторювач] дія**

Діяльність (activity) в UML – це опис поведінки у формі графа діяльності. Діяльність в UML моделює те ж, що і дія, тобто у цьому сенсі діяльність подібна до дії, але діяльність протиставляється дії з усіх характеристичних ознак.

18.2.2. Граф діяльності

Усі сутності і відношення, які застосовуються на діаграмі діяльності, перераховані в параграфі 12.8.2 цього посібника. Тут же наведемо приклад діаграми станів (у стилі UML 1) звільнення співробітника в нашій ІС відділу кадрів як бізнес-процес, що реалізовує відповідний варіант використання. Приведена на рис 18.9 блок схема буквально відтворює текстовий опис сценарію.

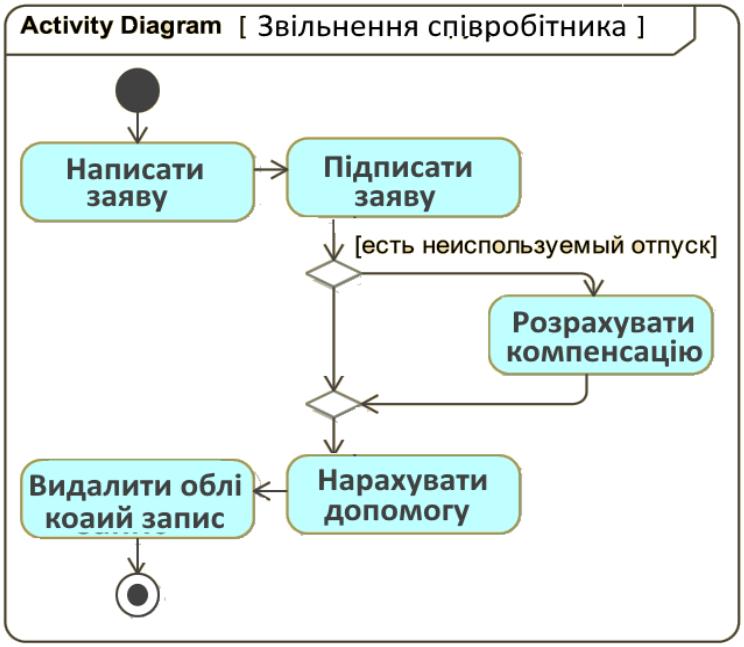


Рисунок 18.9 – Діаграма діяльності зі звільнення співробітника як блок-схема

Ніяких пояснень, як саме виконується, наприклад, діяльність **Написати заяву** тут немає, але бізнес-процес описаний абсолютно ясно.

Продовжимо приклад про звільнення співробітника. Тепер ми розглянемо цей приклад не як бізнес-процес високого рівня, а як операцію (процедуру, AdmFire) информаційної системи (рис. 18.10).

Виконання цього графа діяльності відбувається таким чином.

Початковий стан 7 (рис. 18.10) готовий передати маркер управління, а параметр діяльності 1 готовий передати маркер даних. Таким чином, усі дуги діяльності **Get Person Info,** що входять, готові передати маркери, і діяльність виконується. В результаті виконання з'являються три маркери даних в контактах **pos, fnd, dpt** відповідно. Маркер даних **fnd** має булевий тип, і його значення визначає, чи є присутнім у базі даних об'єкт **p.** Якщо це так, то **pos** і **dpt** містять посаду і підрозділ того, що звільняється, інакше вони порожні.

Маркер даних **pos** негайно вирушає в сховище даних 3, оскільки воно має єдину дугу, що входить, і зберігається там для подальшого використання.

Маркер даних **fnd** вирушає у вузол управління «розгалуження» 8, де перевіряється значення цього маркера даних. Якщо виконується умова **fnd = true**, то маркер даних вирушає у вузол управління «розвилки» 9, і виконується наступний крок. Інакше виконання графа діяльності закінчується.

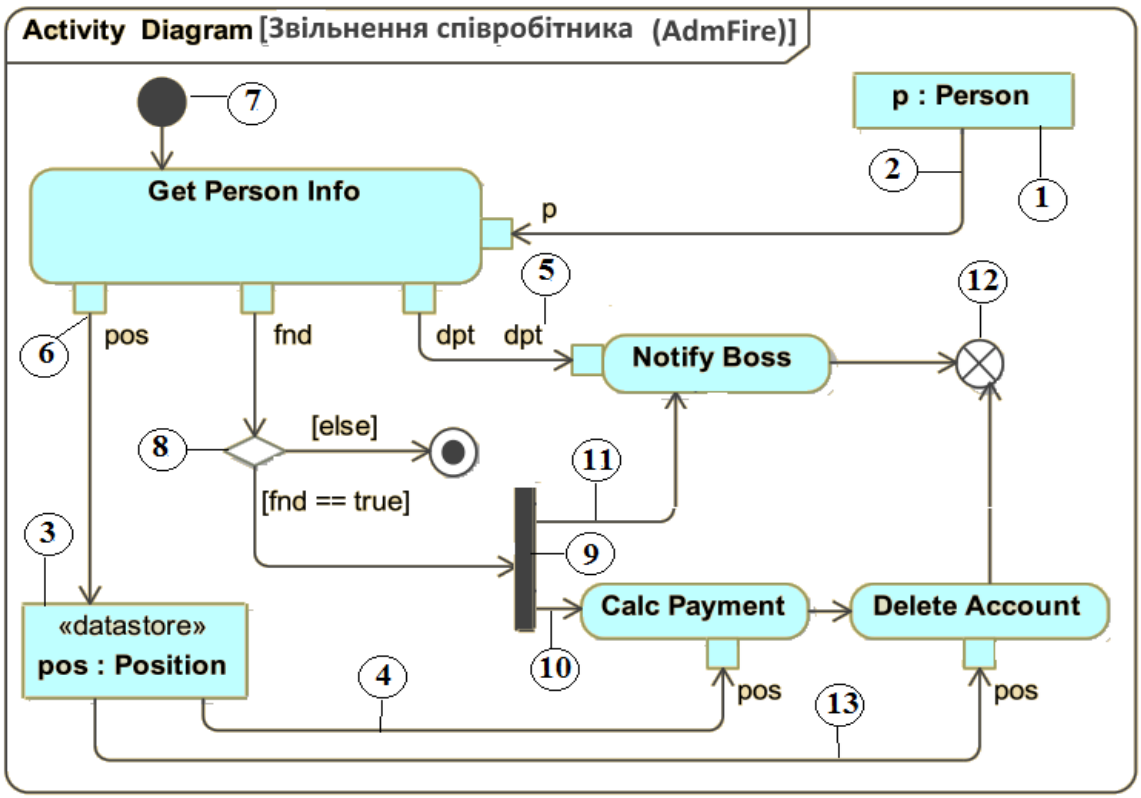


Рисунок 18.10 – Діаграма діяльності зі звільнення співробітника (AdmFire)

Вузол управління «розвилки» 9 розмножує отриманий маркер на два і відправляє їх далі, запускаючи два паралельні потоки управління 10 і 11. Відмітимо, що в даному випадку утримування маркерів втрачається, оскільки приймаються вони не через контакти, а безпосередньо, як маркери управління.

Діяльність **Notify Boss** готова прийняти маркер управління по переходу 11 і маркер даних **dpt** 5 від діяльності **Get Person Info**. Ця діяльність запускається, а після завершення відправляє маркер управління в завершальний стан потоку 12, де він поглинається, і виконання цього потоку управління завершується.

Діяльність **Cale Payment** готова прийняти маркер управління по переходу 10 і маркер даних pos з сховища. Ця діяльність запускається, а після завершення відправляє маркер управління діяльності **Delete Account**.

Діяльність **Delete Account** готова прийняти маркер управління від попередньої діяльності і маркер даних **pos** зі сховища по переходу 13. Ця діяльність запускається, а після завершення відправляє маркер управління в завершальний стан потоку 12, де він поглинається, і виконання цього потоку управління завершується.

18.2.3. Доріжки і розбиття

У UML 1 є своєрідний графічний засіб, який називається доріжкою.

Доріжка – це графічний коментар, що дозволяє класифікувати сутності за деякою ознакою. Доріжки зазвичай використовуються на діаграмах класів або на діаграмах діяльності.

У UML 2 ситуація дещо змінилася: доріжки, що перейменовані в ***розбиття***, переведені з розряду графічних коментарів у розряд сутностей метамоделі мови, і інструменти можуть (але не зобов'язані!) використати цю обставину.

Доріжки часто застосовуються при моделюванні бізнес-процесів в організаціях, звідки вони і були запозичені в UML. Розглянемо бізнес-процес прийому співробітника на роботу в нашій інформаційній системі відділу кадрів.

Припустимо, що нас цікавить проходження усього процесу в цілому, включаючи як ті кроки, які виконуються системою, так і ті кроки, які (поки) передбачається виконувати вручну. Користь такого опису очевидна: якщо ми знатимемо необхідні у бізнес-процесі кроки, у тому числі виконувані вручну, ми зможемо скласти надійніший план його поетапної автоматизації. Навіть у найпростішому випадку прийому на роботу етапу оформлення документів передують інші етапи: збір інформації про кандидата, перевірка і обробка цієї інформації, ухвалення рішення і, нарешті, власне прийом або відмова в прийомі.

На рис. 18.11 представлена діаграма діяльності, що відбиває простий бізнес-процес найму на роботу. Ми вважаємо, що наш процес включає чотири діяльності:

1. **Інтерв'ю (Interview)** – збір інформації.
2. **Аналіз (Analysis)** – аналіз зібраної інформації і ухвалення рішення.
3. **Заповнення бланків (Fill Forms)** – заповнення документів для найму на роботу.
4. **Відмова (Refuse)** – відмова в наймі.

На діаграмі рис. 18.11 немає ніяких доріжок – усі діяльності рівноправні і однорідні.

Припустимо тепер, що діяльності, в які найманий залучений безпосередньо (**Інтерв'ю, Заповнення бланків, Відмова),** відбуваються в одному місці і, так би мовити, у нього на очах, а важлива діяльність (про яку найманий може не здогадуватися) з аналізу інформації і ухвалення рішення **(Аналіз**) здійснюється у іншому місці і, можливо, іншими дійовими особами (технічними фахівцями, керівниками підрозділів і тощо).

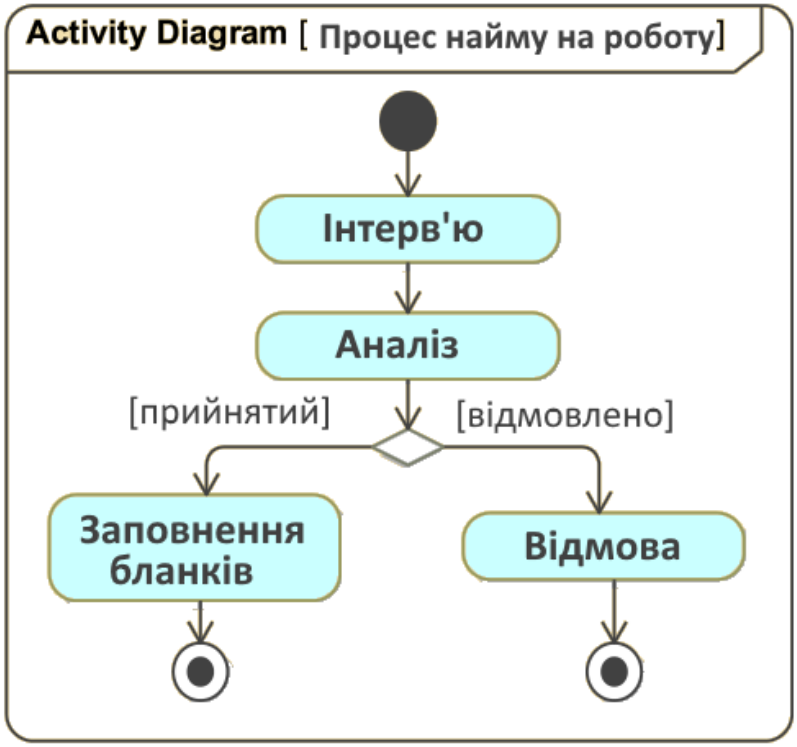


Рисунок 18.11 – Діаграма діяльності процесу найму на роботу

Ця важлива інформація не є частиною моделі, тобто не має відношення до поведінки системи, але ми можемо відобразити її на діаграмі за допомогою доріжок (рис. 18.12).

У даному випадку ми маємо на увазі, що доріжка з назвою **Відділ кадрів** (HR Department) містить діяльності, що виконуються в приймальні відділу кадрів, а доріжка з назвою **Цільовий відділ** (Target Department) містить діяльності, що виконуються в тому підрозділі, куди передбачається прийняти кандидата.

Графічно доріжки зображаються у вигляді прямокутників з назвами.

Автори UML стверджують, що зображення, подібне до наведеного на рис. 18.12, нагадує плавальні доріжки у басейні, звідки і сталася назва цієї графічної конструкції. Завершимо цей нескладний параграф ще одним прикладом застосування спадкоємців доріжок – розбиття в UML 2.

У даному прикладі ми накладемо на дії з прийому співробітника дві ортогональні (паралельні) класифікації. Перша співпадає зі вже розібраною класифікацією за місцем дії, а друга вказує, як виконується дія (діяльності Інтерв'ю або Аналіз) – усно (словесно, вербально) або інакше (рис. 18.13).

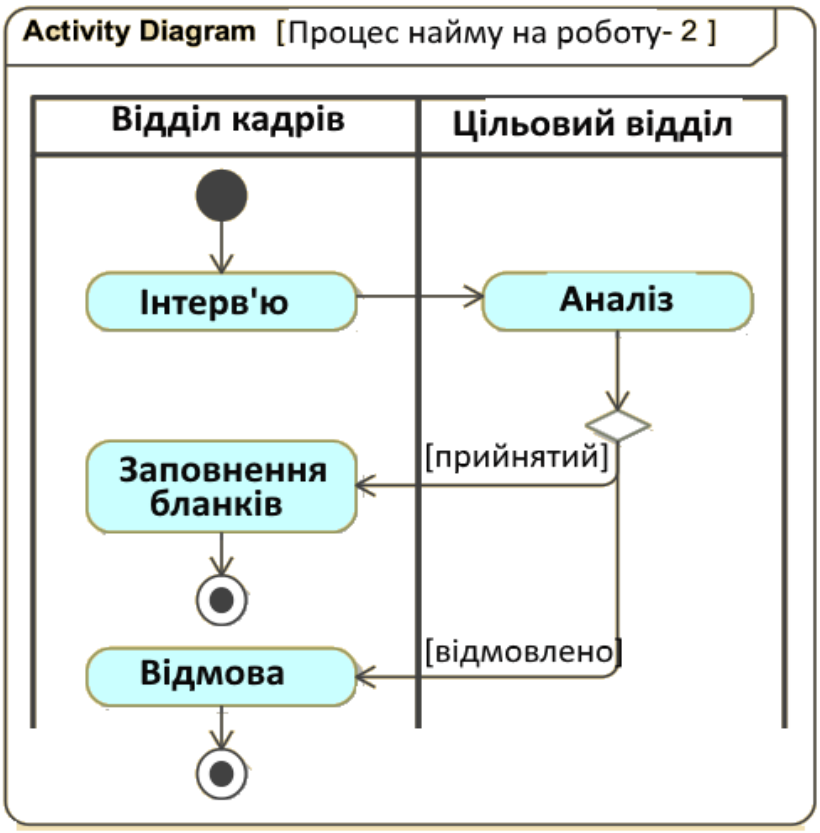


Рисунок 18.12 – Діаграма діяльності процесу найму на роботу (доріжки)

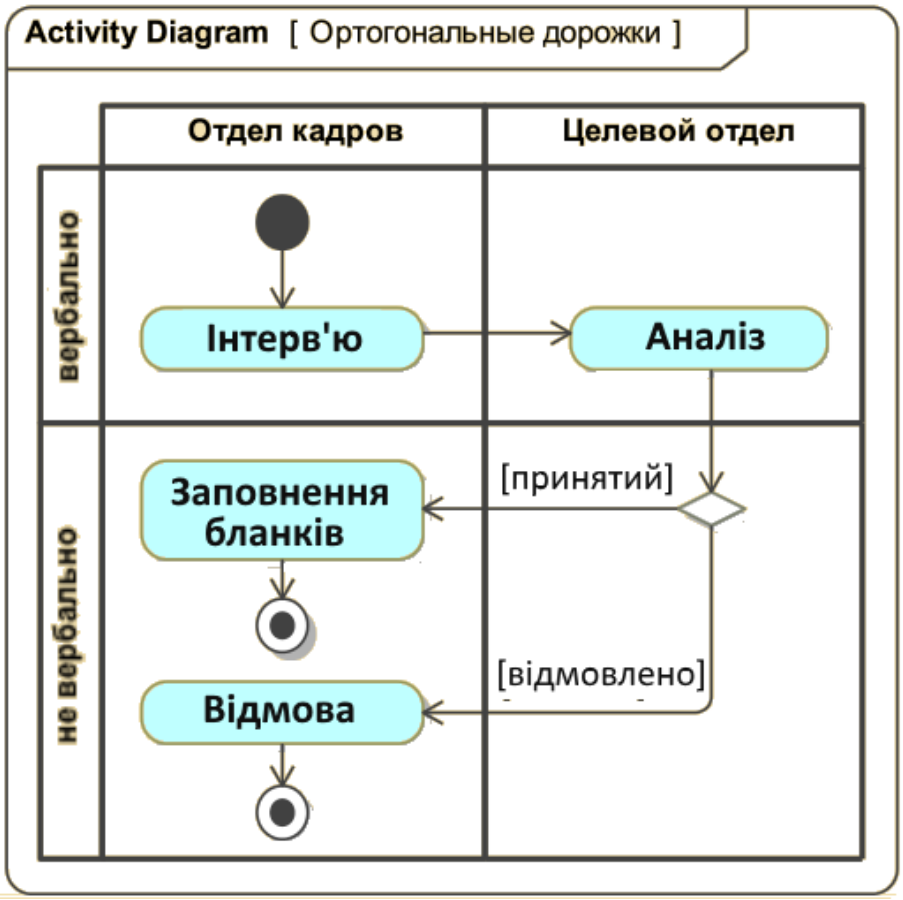


Рисунок 18.13 – Ортогональні доріжки

18.2.4. Траєкторія об’єкту і потік даних

Діаграми діяльності UML дозволяють моделювати поведінку, визначаючи не лише потік управління, як в наведених вище прикладах, але і потік даних.

У програмі, написаній звичайною процедурною мовою програмування, порядок виконання операторів (кроків алгоритму) визначається порядком розташування операторів в тексті програми і структурами управління. Фактично звичайна програма задає потік управління. Але при виконанні кроків алгоритму виконуються перетворення даних: змінні набувають і міняють свої значення. Послідовність зміни даних називається ***потоком даних***.

При **об’єктно-орієнтованому підході** до моделювання потік даних – це зміна станів об’єктів в часі, і опис такої зміни істотним чином характеризує поведінку.

Для опису цієї характеристики поведінки в UML використовуються поняття «Траєкторія об’єкту» і «об’єкт в стані».

Об’єкт в стані (object in state) – це екземпляр деякого класу, про який відомо, що він знаходиться в певному стані в цій точці обчислювального процесу

Синтаксично об’єкт в стані зображається, як завжди, у вигляді прямокутника, і його ім’я підкреслюється (не обов’язково), але додатково після імені об’єкту в квадратних дужках пишеться ім’я стану, в якому в цій точці обчислювального процесу знаходиться об’єкт.

Траєкторія об’єкту – це перехід особливого роду, початковим або цільовим станом яким є об’єкт в стані.

Траєкторія об’єкту зображається у вигляді пунктирної стрілки (на відміну від суцільної стрілки звичайного переходу). Семантично траєкторія об’єкту, що проведена від стану діяльності до об’єкту в стані, означає, що результатом діяльності є перехід вказаного об’єкту в цей стан (чи, можливо, створення нового об’єкту у вказаному стані, що є частковим випадком зміни стану). Траєкторія об’єкту, що проведена від об’єкту в стані до стану діяльності або стану дії, означає, що об’єкт в цьому стані є необхідним вхідним параметром указаної діяльності.

Таким чином, об’єкти в стані і траєкторії дозволяють показати на діаграмі діяльності не лише залежність з управління, але і залежність за даними між станами діяльності на діаграмі. А саме, щоб показати, що одна діяльність використовує результати іншої діяльності, досить показати траєкторію об’єкту, що передається.

Розглянемо це на прикладі діаграми діяльності, що описує процес найму співробітника в інформаційній системі відділу кадрів (рис. 18.14). Рисунок 18.14 в основному повторює рисунок 18.12, але тут представлена траєкторія об’єкту класу **Person,** що зберігає дані про співробітника, що приймається. На діаграмі добре видно, що саме є вхідними і вихідними даними кожного із станів діяльності: в результаті діяльності **Interview** створюється новий об’єкт, який далі обробляється, міняючи свій стан. Таким чином, траєкторія об’єкту показує, що результат однієї діяльності є вхідними даними іншої діяльності.

З діаграми на рисунку 18.14 витікає, що діяльність **Analysis** виконується після діяльності **Interview,** причому це вказано двічі: один раз за допомогою переходу після закінчення 1 з Interview в Analysis і другий раз за допомогою траєкторії об’єкту 2, яка показує, що для виконання діяльності **Analysis** потрібний об’єкт, що створюється діяльністю **Interview**.

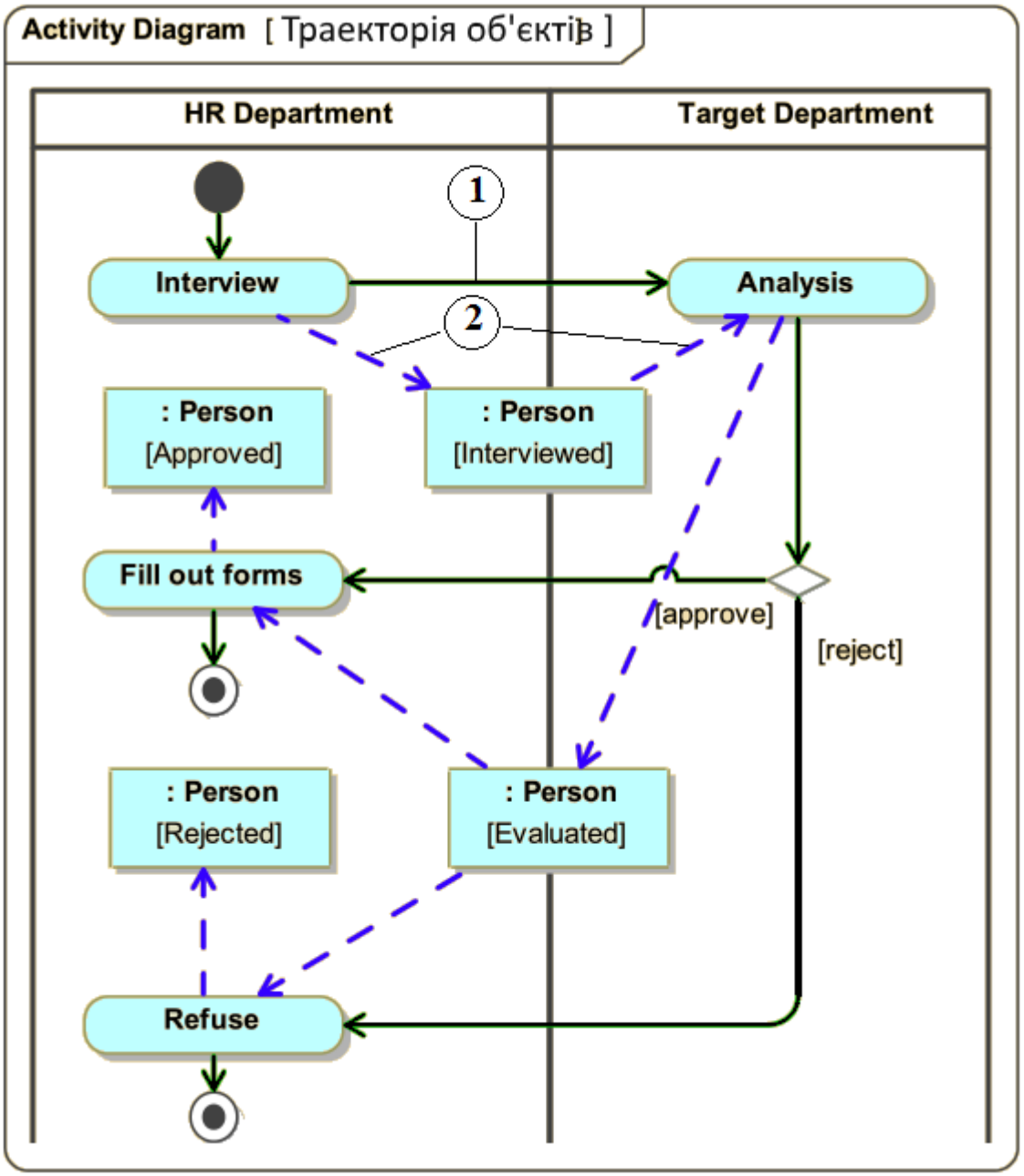


Рисунок 18.14 – Траєкторія об’єкту найму співробітника

Діаграма на рис. 18.5 описує ту ж саму поведінку, що і діаграма на рис. 18.14. Тобто, якщо потік управління однозначно відновлюється з траєкторій об’єктів, то вказівку потоку управління можна опустити.

Нотація UML 2 дозволяє використати для опису поведінки потік даних (траєкторію об’єкту) ще ширше, оскільки у визначенні семантики графа діяльності маркери даних і маркери управління практично рівноправні.

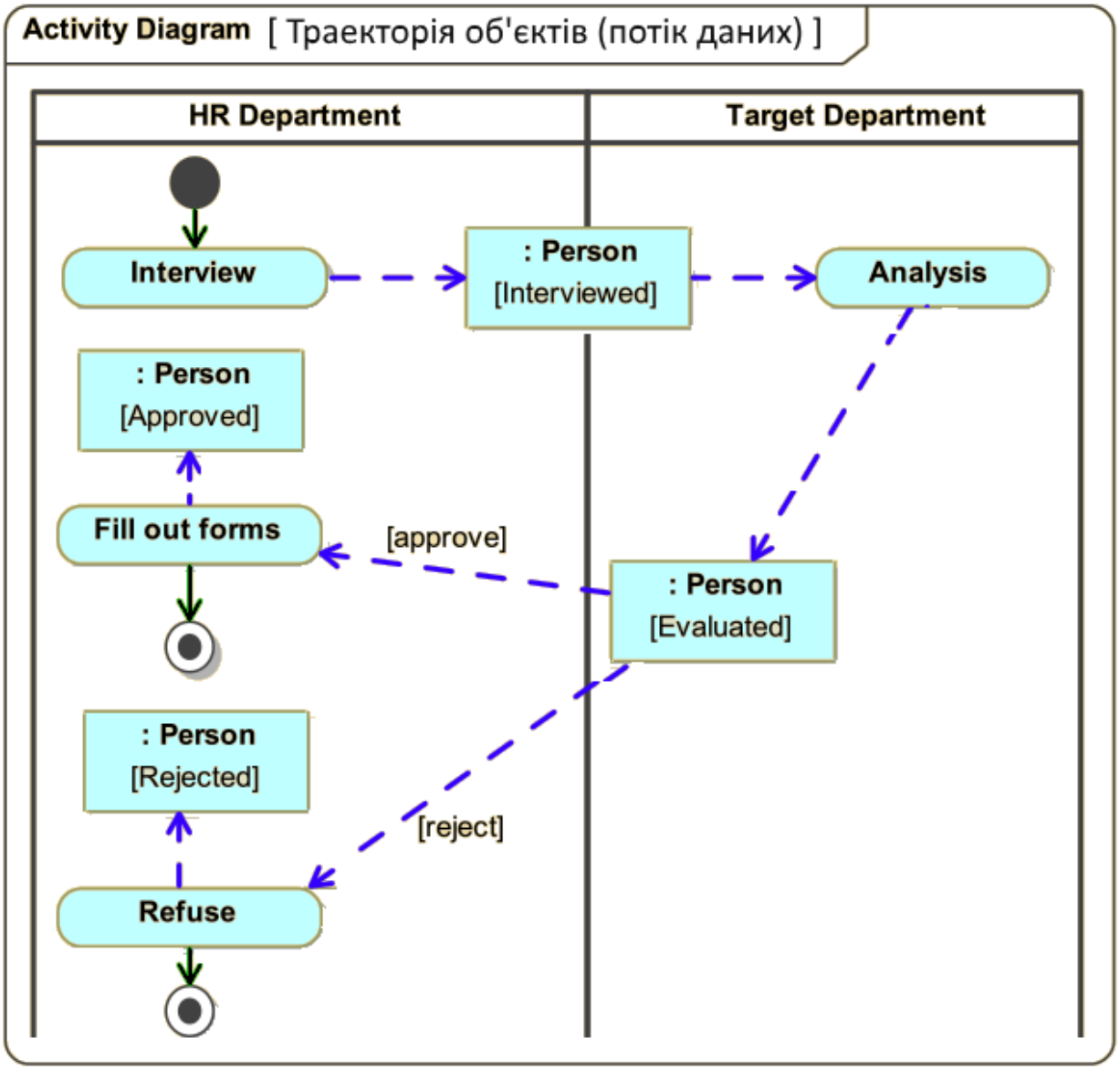


Рисунок 18.15 – Траєкторія об’єкту (потік даних)

На рис. 18.16 приведена ще одна діаграма діяльності, що описує процес прийому на роботу, але в нотації UML 2 із застосуванням контактів і параметрів діяльності.

Контакти – це вузли даних, що показують вхідні і вихідні параметри дій.

Вузол даних, поміщений на **рамці діяльності** є параметром діяльності в цілому.

Підводячи підсумки опису поведінки діаграмами діяльності в UML, обговоримо сфери застосування цих діаграм.

Таким чином, діаграми діяльності – загальний і потужний засіб, – який можна застосовувати як для опису алгоритмів (для зображення блок-схем як засіб візуального структурного програмування, так і для опису поведінки на найвищому рівні абстракції (у вигляді діаграм діяльності).

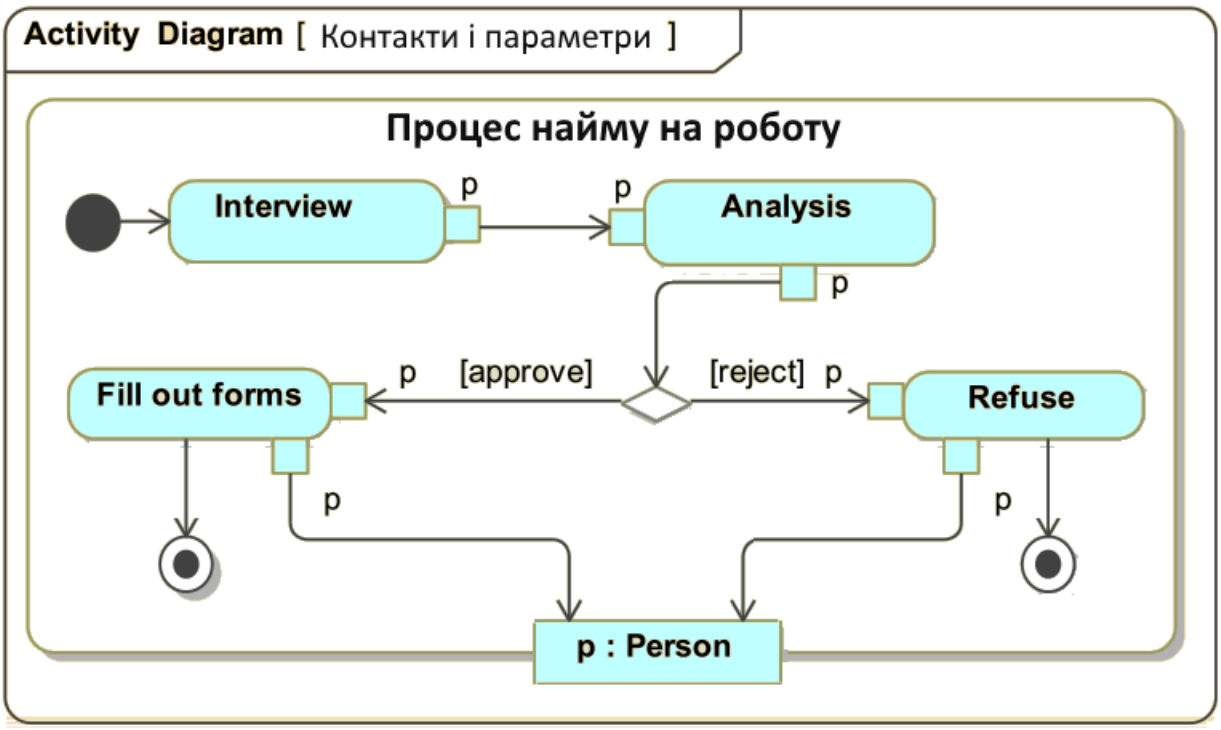


Рисунок 18.16 – Застосування контактів і параметрів діяльності

18.3. Діаграми взаємодії

Діаграми взаємодії призначені для моделювання поведінки шляхом обміну повідомленнями між взаємодіючими об’єктами. Цей тип діаграм дозволяє описувати не лише взаємодію програмних об’єктів (екземплярів класів), але і взаємодію екземплярів інших класифікаторів: дійових осіб, варіантів використання, компонентів та ін.

Діаграми взаємодії зображаються в декількох різних графічних формах, з яких найважливішими є **діаграми послідовності** і **діаграми комунікації**.

Ми вже відмічали, що діаграми послідовності і діаграми комунікації семантично еквівалентні (тому що описують одну і ту ж послідовність передачі повідомлень між об’єктами в процесі їх взаємодії), хоча графічно виглядають зовсім по-різному (тому що в діаграмі послідовності графічно підкреслюється впорядкованість в часі повідомлень, що передаються, тоді як в діаграмі комунікації на передній план висувається структура зв’язків між об’єктами, якими передаються повідомлення).

18.3.1. Повідомлення

Повідомлення (message) – це передача управління і даних від одного об’єкту (відправника) до іншого (одержувача), але якщо одержувачів декілька, то таке повідомлення називається ***широкомовним*** (broadcast). Повідомлення може бути ***умовним***. При виконанні певної сторожової умови повідомлення відправляється, інакше нічого не відбувається. Сторожова умова записується, як завжди, в квадратних дужках.

Відправка повідомлення є **дією,** а отримання повідомлення **– подією.** Нагадаємо, що в UML такі дії пов'язані з передачею інформації і відправкою повідомлень: виклик методу (call); створення об'єкту (create); знищення об'єкту (destroy); повернення значення (return); посилка сигналу (send).

Дія записується у вигляді тексту над (чи поряд з) стрілкою, що символізує повідомлення. Синтаксис виклику методу має відмінності в UML 1 і в UML 2. У UML 1 має місце такий синтаксис:

змінні := ІМ'Я (аргументи)

У UML 2 використовується дещо інший синтаксис:

атрибути – ІМ'Я (аргументи) : змінні

Атрибути зліва від знаку «=» це атрибути об'єктау, що відправляє повідомлення, призначені для зберігання значень, що повертаються. Змінні праворуч від знаку «:» – це локальні змінні взаємодії для набуття значень, що повертаються. Можна не використати ні атрибути, ні змінні, можна використати або те, або інше, а можна використати і те, і інше.

У UML розрізняється декілька типів передачі управління за допомогою повідомлення. Щоб відрізнити тип передачі повідомлення, в UML застосовується спеціальна графічна нотація, а саме розрізняються види стрілок, якими позначаються повідомлення. Хоча на діаграмах комунікації і послідовності повідомлення позначаються по-різному, принципи зображення однакові і перераховані в таблиці 12.2 (підрозділ 12.9).

18.3.2. Діаграми послідовності

Діаграма послідовності призначена для моделювання поведінки у формі опису протоколу сеансу обміну повідомленнями між взаємодіючими екземплярами класифікаторів під час виконання одного з можливих сценаріїв.

Вісь часу і час життя об'єктів. На діаграмі послідовності вважається виділеним один напрям, що відповідає плину часу. За замовчуванням вважається, що час тече зверху вниз, але це не обов'язково, наприклад, можна вважати, що час тече зліва направо, обумовивши це спеціальним коментарем. У наших прикладах використовується виключно нотація за замовчуванням: час завжди тече зверху вниз. Саму вісь часу не відображають.

Повідомлення зображаються прямими стрілками різного виду (таблиця. 18.3). Якщо передача повідомлення вважається миттєвою (тобто час передачі дуже малий), то стрілка горизонтальна (тобто перпендикулярна осі часу). Якщо ж треба відобразити затриману доставку повідомлення, то стрілку трохи нахиляють, так щоб кінець стрілки був нижчий початку.

Лінія життя і стрілки повідомлень. Паралельно осі часу від усіх об'єктів, що беруть участь у взаємодії, відходить пряма пунктирна лінія, яка називається лінією життя (lifeline). Якщо стрілка відходить від лінії життя об'єкта, то це означає, що цей об'єкт відправляє повідомлення, а якщо стрілка повідомлення входить в лінію життя, то це означає, що цей об'єкт отримує повідомлення. Якщо ж стрілка перетинає лінію життя об'єкта, то це нічого не означає – повідомлення пролетіло мимо. Якщо в процесі взаємодії об'єкт

закінчує своє існування, то лінія життя обривається і в цьому місці ставиться косий хрест. Над стрілкою повідомлення вказується текстова частина.

Сутності і відношення на діаграмі послідовності. Розглянемо взаємодію, що виникає при одному з простих сценаріїв в інформаційній системі відділу кадрів, а саме, створення підрозділу. Ця взаємодія ініціюється зовнішньою дійовою особою – менеджером штатного розпису, який відкриває відповідну форму і запускає виконання операції створення підрозділу, після чого закриває не потрібну йому форму. Загальна схема взаємодії для цього сценарію представлена на рис. 18.17. Зверніть увагу, що окрім повідомлень, які відповідають викликам методів (наприклад, 1), на діаграмі є присутнім повідомлення, що відповідає виклику конструктора 2.

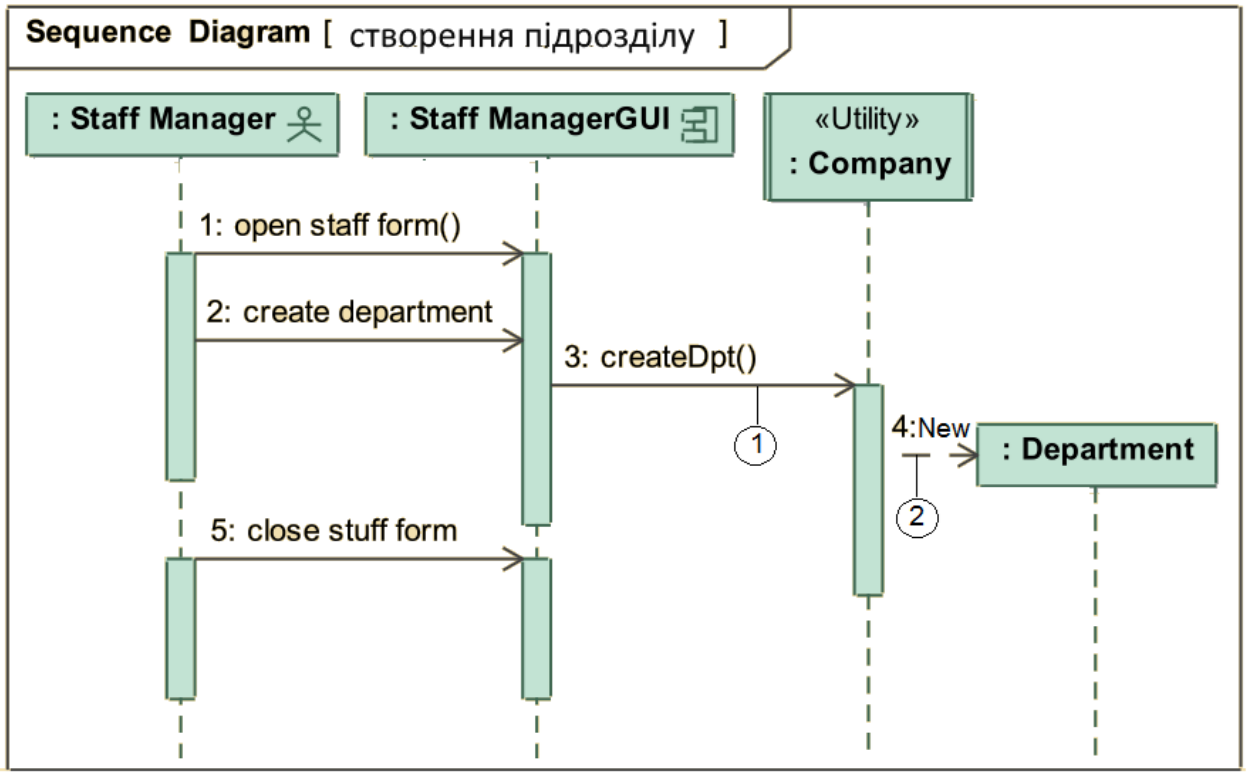


Рисунок 18.17 – Діаграма послідовності

Мітка часу. Якщо треба в явному вигляді вказати обмеження за часом, наприклад, вказати, що час затримки доставки повідомлення має бути обмежений згори, то на діаграму в потрібному місці (має значення тільки положення по вертикалі) поряд з початком або кінцем стрілки повідомлення поміщають довільні ідентифікатори, які називаються *мітками часу* (time observation), і додають часові обмеження, які задають потрібні умови на значення міток часу.

Мітка часу – це іменована точка на лінії життя. Перед міткою часу ставиться символ «@».

Припустимо, що інформаційна система відділу кадрів призначена для організації, що має віддалену філію. У цій філії є і працює свій екземпляр інформаційної системи, який, очевидно, має бути проінформований, що в

головній організації створений новий підрозділ. Можливо, ця інформація про зміни в головній організації дійде у філію з деякою затримкою, оскільки для зв'язку використовується повільний канал передачі даних. Таку ситуацію можна промоделювати діаграмою, наведеною на рис. 18.18, де мітка часу 1 в сукупності з часовим обмеженням 2, описують затриману доставку повідомлення notyfy().

Лінії 3 і 4 – це графічні коментарі, які потрібні для прив'язки мітки часу і часового обмеження до необхідних точок лінії життя.

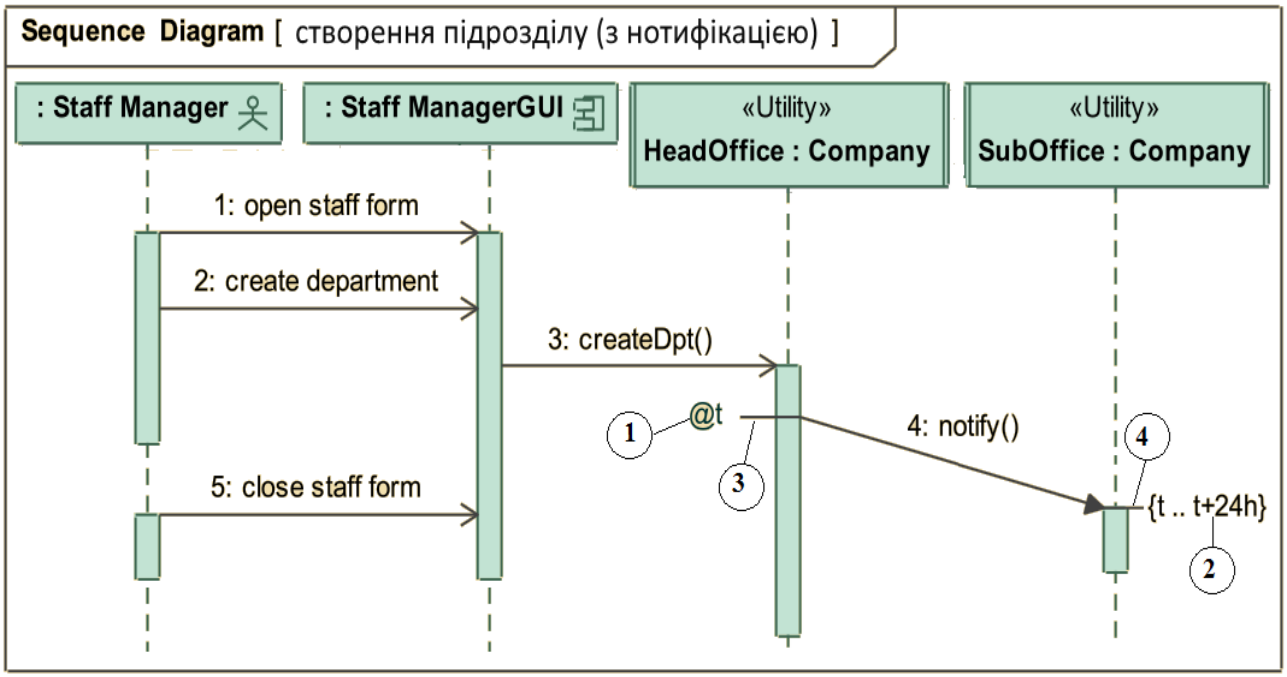


Рисунок 18.18 – Мітки часу і затримана доставка

Повернення управління. Для наочності на діаграмі послідовності можна показати в явному вигляді повернення управління (і, можливо, повернене значення), хоча це не обов'язково: повернення управління мається на увазі при використанні повідомлення типу виклик методу. Більше того, якщо використати полоски для явної вказівки активації об'єктів, стрілки повернення не потрібні: їх легко можна подумки відновити.

Розглянемо застосування цієї групи позначень на прикладі з інформаційної системи відділу кадрів. Припустимо, при створенні нового підрозділу негайно виконується метод createBossPos() зі створення нової посади (начальника) в цьому підрозділі і ця вакансія заповнюється, а після успішного створення підрозділу можна показати менеджерові штатного розпису змінену організаційну діаграму компанії (рис. 18.19).

Тут ми використовуємо як активації 1, у тому числі вкладені 2, так і повернення 3, щоб показати застосування усіх засобів, хоча, можливо, це трохи перевантажує діаграму.

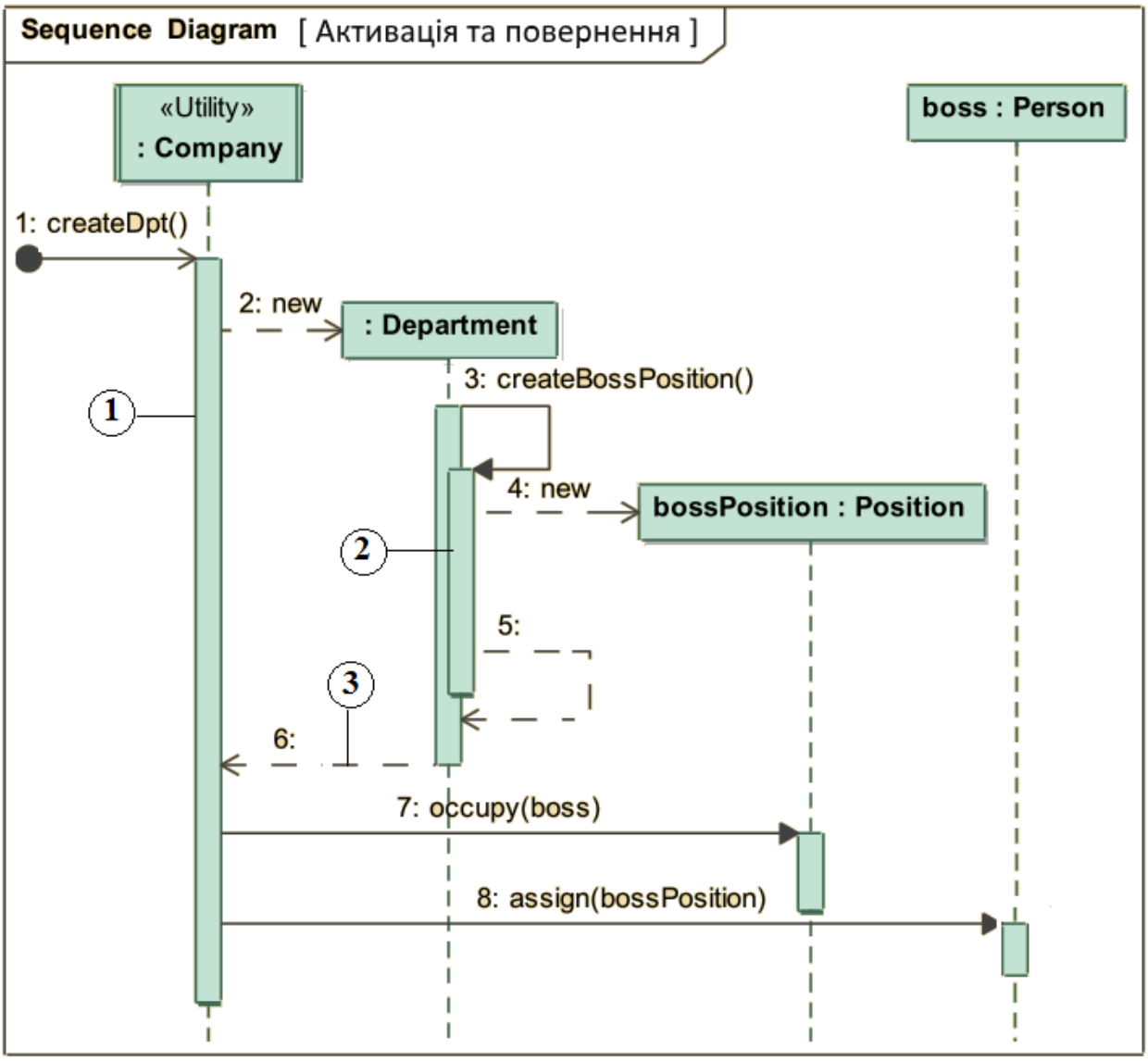


Рисунок 18.19 – Діаграми послідовності (активація та повернення)

18.3.3. Діаграми комунікації

Один і той же об’єкт може брати участь в різних взаємодіях, граючи в них різні ролі. Таким чином, взаємодія завжди відбувається в певному контексті, який визначається множиною об’єктів, що беруть участь у взаємодії, і зв’язків. Діаграма комунікації (так само як і діаграма послідовності) описує поведінку як взаємодію, тобто як протокол обміну повідомлень між об’єктами.

Сполучення з меншими номерами передують сполученням з великими номерами. Якщо ж використовуються вкладені потоки управління, тобто повідомлення типу виклику операції, то повідомлення нумеруються складнішим чином.

Розглянемо наш приклад з інформаційної системи відділу кадрів (створення підрозділу). Рис. 18.20 семантично еквівалентний рис. 18.17 – ці діаграми описують одну і ту ж взаємодію.

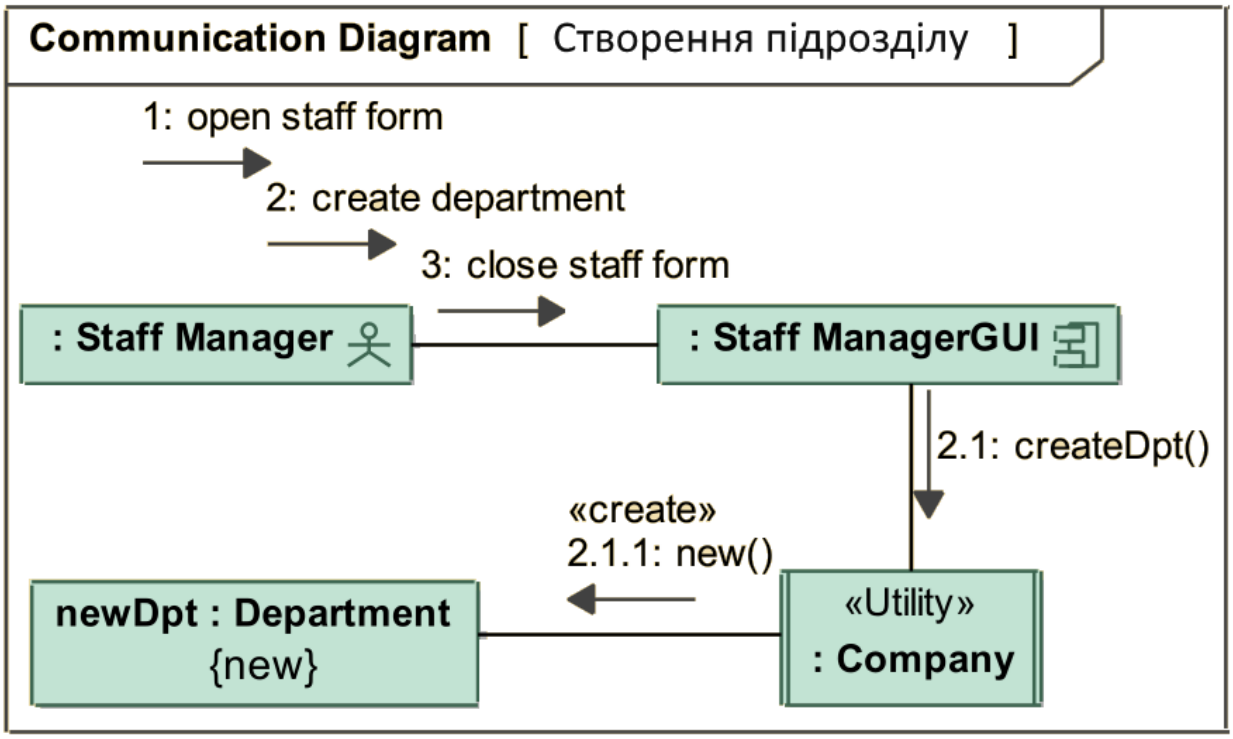


Рисунок 18.20 – Діаграма комунікації

РОЗДІЛ 19. ВИКОНАННЯ НАВЧАЛЬНОГО ПРОЕКТУ В СЕРЕДОВИЩІ RATIONAL ROSE

19.1. Постановка завдання

Перед керівником інформаційної служби університету ставиться завдання розробки нової клієнт-серверної **системи реєстрації студентів** замість старої системи на мейнфреймі. Нова система повинна дозволяти студентам реєструватися на курси і переглядати свої табелі успішності з персональних комп'ютерів, підключених до локальної мережі університету. Професори (викладачі університету) повинні мати доступ до онлайнової системи, щоб вказати курси, які вони читатимуть, і проставляти оцінки за курси.

Через нестачу засобів університет не в змозі замінити відразу усю існуючу систему. Залишається функціонувати в колишньому виді база даних, що містить усю інформацію про курси (каталог курсів). Ця база даних підтримується реляційною СУБД. Нова система працюватиме з існуючою БД в режимі доступу, без оновлення.

На початку кожного семестру студенти можуть запросити каталог курсів, що містить список курсів, пропонованих в цьому семестрі. Інформація про кожен курс повинна включати ім'я професора, найменування кафедри і вимоги до попереднього рівня підготовки (прослуханих курсів).

Нова система повинна дозволяти студентам вибирати 4 курси в майбутньому семестрі. Додатково кожен студент може вказати 2 альтернативні курси на той випадок, якщо який-небудь з вибраних ним курсів виявиться вже заповненим або скасованим. На кожен курс може записатися не більше 10 і не менше 3 студентів (якщо менше 3, то курс буде скасований). У кожному семестрі існує період часу, коли студенти можуть змінити свої плани. В цей час студенти повинні мати доступ до системи, щоб додати або видалити вибрані курси. Після того, як процес реєстрації певного студента завершений, система реєстрації направляє інформацію в розрахункову систему, щоб студент міг внести плату за семестр. Якщо курс виявиться заповненим у процесі реєстрації, студент має бути сповіщений про це до остаточного формування його особистого учбового плану.

У кінці семестру студенти повинні мати доступ до системи для перегляду своїх електронних табелів успішності. Оскільки ця інформація конфіденційна, система повинна забезпечувати її захист від несанкціонованого доступу.

Професори повинні мати доступ до онлайнової системи, щоб вказати курси, які вони читатимуть, і проглянути список студентів, що записалися на їх курси. Окрім цього, професори повинні мати можливість проставити оцінки за курси.

19.2. Складання глосарію проекту

Глосарій призначений для опису термінології предметної області. Він може бути використаний як неформальний словник даних системи (таблиця. 19.1).

Таблиця 19.1 – Глосарій предметної області

Курс	Навчальний курс, що пропонується університетом
Конкретний курс (Course Offering)	Конкретне читання цього курсу в конкретному семестрі (один і той же курс може вестися в декількох паралельних сесіях). Включає точні дні тижня і час.
Каталог курсів	Повний каталог усіх курсів, що пропонуються університетом.
Розрахункова система	Система обробки інформації про плату за курси.
Оцінка	Оцінка, що отримана студентом за конкретний курс.
Професор	Викладач університету.
Табель успішності (Report Card)	Усі оцінки за усі курси, що отримані студентом в цьому семестрі.
Студент	Особа, що проходить навчання в університеті.
Навчальний графік (Schedule)	Курси, що вибрані студентом в поточному семестрі.

19.3. Опис додаткових специфікацій

Призначення додаткових специфікацій – визначити вимоги до системи реєстрації курсів, які не відбиті в моделі варіантів використання. Разом вони утворюють повний набір вимог до системи.

Додаткові специфікації визначають нефункціональні вимоги до системи, такі, як надійність, зручність використання, продуктивність, супроводжуваність, а також ряд функціональних вимог, що є загальними для декількох варіантів використання.

Функціональні можливості. Система повинна забезпечувати режим роботи, який розрахований на багато користувачів.

Якщо конкретний курс виявляється заповненим в той час, коли студент формує свій навчальний графік, що включає цей курс, то система повинна сповістити його про це.

Зручність використання. Призначений для користувача інтерфейс має бути сумісним з Windows XP/7.

Надійність. Система має бути в працездатному стані 24 години в день 7 днів в тиждень, час простою – не більше 10%.

Продуктивність. Система повинна підтримувати до 500 користувачів, що одночасно працюють з центральною базою, і до 100 користувачів, що одночасно працюють з локальними серверами.

Безпека. Система не повинна дозволяти студентам змінювати будь-які навчальні графіки, окрім своїх власних, а також не повинна дозволяти професорам модифікувати конкретні курси, вибрані іншими професорами.

Тільки професори мають право ставити студентам оцінки. Тільки реєстратор може змінювати будь-яку інформацію про студентів.

Проектні обмеження. Система має бути інтегрована з існуючою системою каталогу курсів, що функціонує на основі реляційної СУБД.

19.4. Створення моделі варіантів використання

Дійові особи:

1. Student (Студент) – записується на курси;
2. Professor (Професор) – вибирає курси для викладання;
3. Registrar (Реєстратор) – формує навчальний план і каталог курсів, веде усі дані про курси, професорів і студентів;
4. Billing System (Розрахункова система) – отримує від цієї системи інформацію щодо плати за курси;
5. Course Catalog (Каталог курсів) – передає в систему інформацію з каталогу курсів, пропонованих університетом.

Вправа 1. Створення дійових осіб в середовищі Rational Rose

Щоб помістити дійову особу у браузер:

1. Клацніть правою кнопкою миші на пакеті представлення варіантів використання у браузері.
2. Виберіть в меню, що відкрилося, пункт New > Actor
3. У браузері з’явиться нова дійова особа під назвою NewClass. Зліва від його імені ви побачите піктограму дійової особи UML.
4. Виділивши нову дійову особу, введіть її ім’я.
5. Після створення дійових осіб збережіть модель під ім’ям coursereg(analysis) за допомогою пункту меню File > Save.

Результат виконання вправи показаний на рис. 19.1.

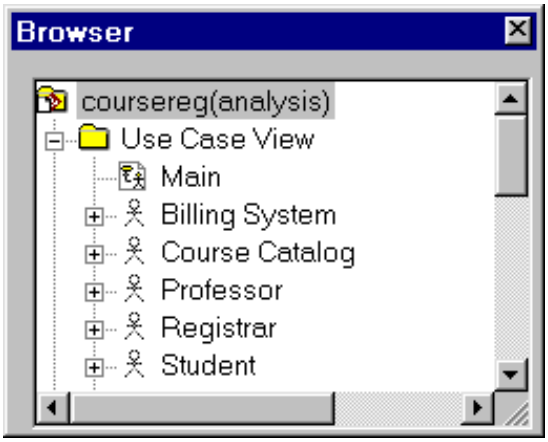


Рисунок 19.1 – Представлення дійових осіб у браузері

Варіанти використання:

Виходячи з потреб дійових осіб, виділяються такі варіанти використання:

1. Login (Увійти в систему);
2. Register for Courses (Зареєструватися на курси);
3. View Report Card (Проглянути табель успішності);
4. Select Courses to Teach (Вибрати курси для викладання);
5. Submit Grades (Проставити оцінки);
6. Maintain Professor Information (Вести інформацію про професорів);
7. Maintain Student Information (Вести інформацію про студентів);
8. Close Registration (Закрити реєстрацію).

Вправа 2. Створення варіантів використання в середовищі Rational Rose

Щоб помістити варіант використання у браузер:

1. Клацніть правою кнопкою миші на пакеті представлення варіантів використання у браузері.
2. Виберіть в меню, що з'явилося, пункт New > Use Case
3. Новий варіант використання під назвою NewUseCase з'явиться у браузері. Зліва від нього буде видна піктограма варіанту використання UML.
4. Виділивши новий варіант використання, введіть його назву.

Результат виконання вправи показаний на рис. 19.2.

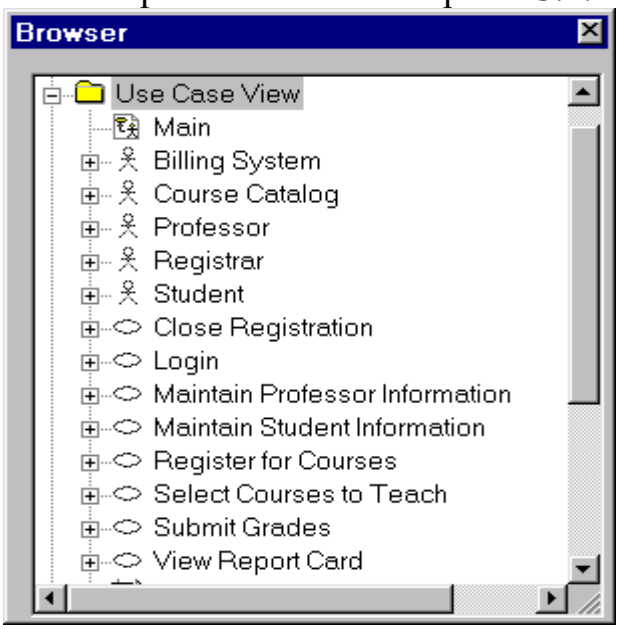


Рисунок 19.2 – Представлення варіантів використання у браузері

Діаграма варіантів використання:

Створіть діаграму варіантів використання для системи реєстрації. Потрібні для цього дії детально перераховані далі. Готова діаграма варіантів використання повинна виглядати як на рис. 19.3.

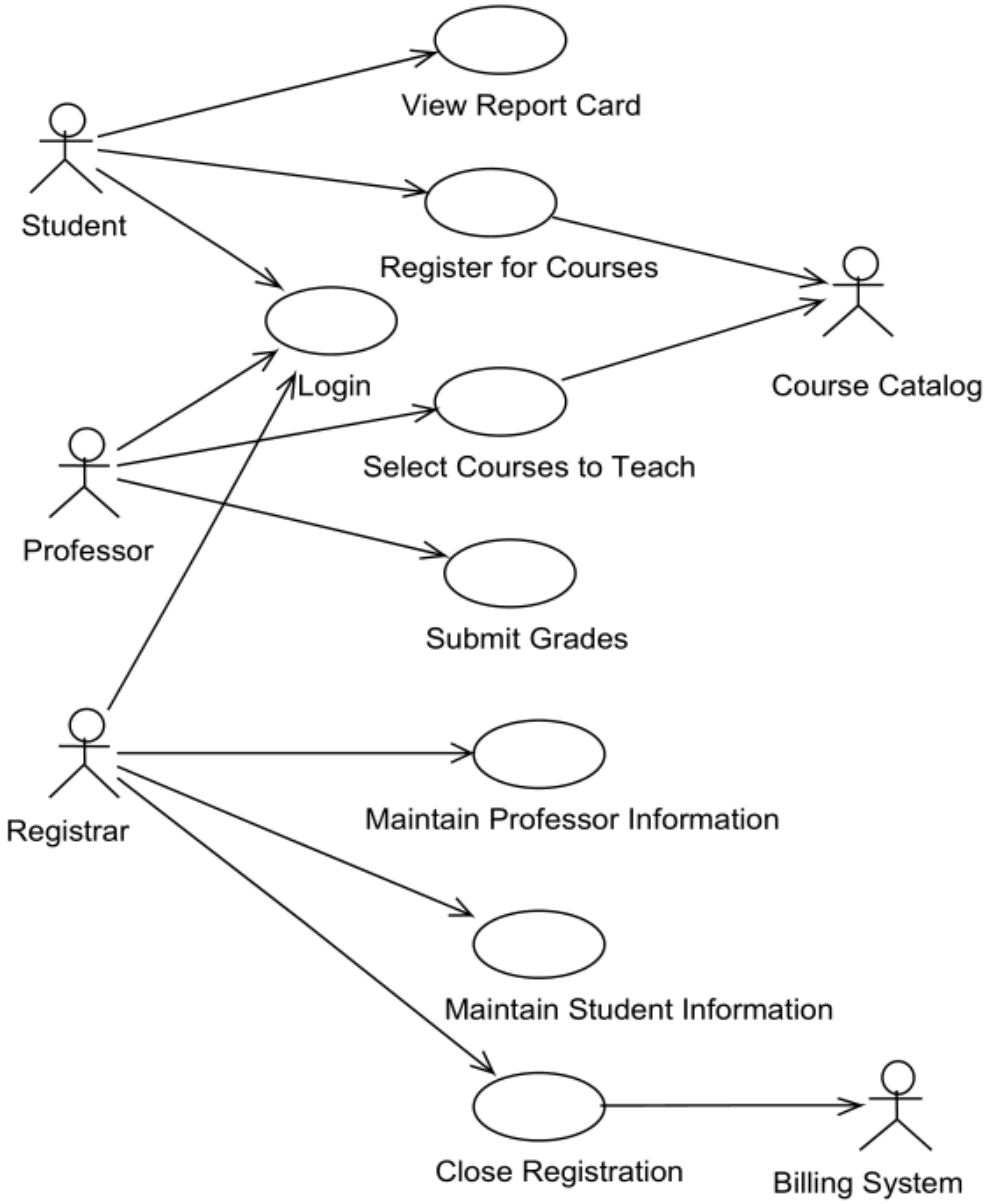


Рисунок 19.3 – Діаграма варіантів використання для системи Реєстрації

У середовищі Rose діаграми варіантів використання створюються в уявленні варіантів використання. Головна діаграма (Main) пропонується за замовчуванням. Для моделювання системи можна потім розробити стільки додаткових діаграм, скільки необхідно.

Щоб отримати доступ до головної діаграми варіантів використання:

1. Поряд з представленням варіантів використання у браузері клацніть на значку « + «, це приведе до відкриття цього представлення.
2. Двічі клацніть на головній діаграмі Main, щоб відкрити її. Рядок заголовка зміниться, включивши фразу [Use Case Diagram: Use Case view / Main].

Для створення нової діаграми варіантів використання:

1. Клацніть правою кнопкою миші на пакеті представлення варіантів використання у браузері.
2. Із спливаючого меню виберіть пункт New > Use Case Diagram.

3. Виділивши нову діаграму, введіть її ім’я.
4. Двічі клацніть на назві цієї діаграми у браузері, щоб відкрити її.

Вправа 3. Побудова діаграми варіантів використання

1. Відкрийте діаграму варіантів використання Main.
2. Перетягніть її мишею з браузеру на діаграму варіантів використання.
3. За допомогою кнопки Unidirectional Association (Однонапрямлена асоціація) панелі інструментів намалюйте асоціації між дійовими особами і варіантами використання.

Наявність загального варіанту використання Login для трьох дійових осіб дозволяє узагальнити їх поведінку і ввести нову дійову особу Any User (см Модифікована діаграма варіантів використання, рис. 19.4).

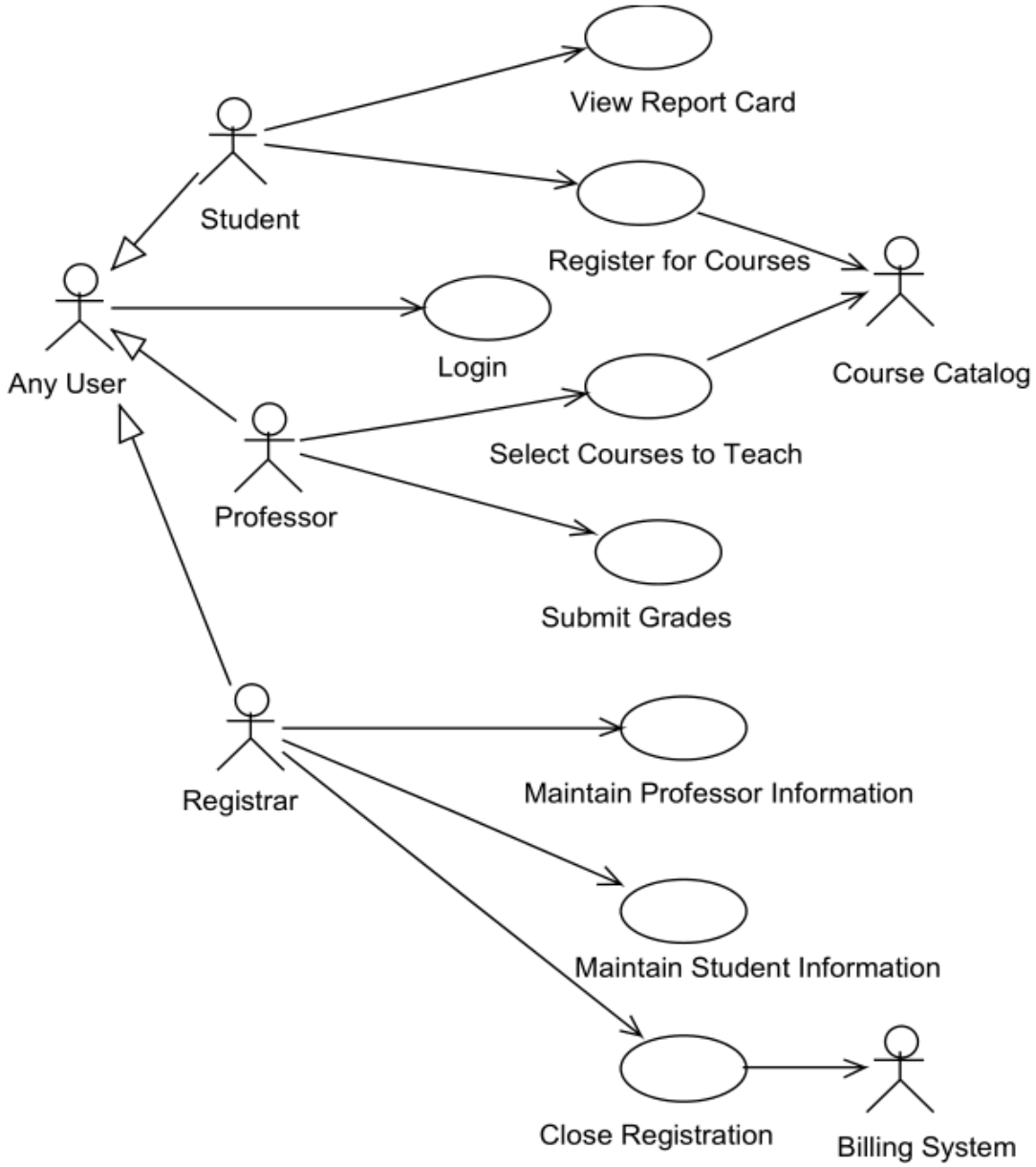


Рисунок 19.4 – Модифікована діаграма варіантів використання

Вправа 4. Додавання описів до варіантів використання

1. Виділіть у браузері варіант використання «Register for Courses».
2. У вікні документації введіть наступний опис до цього варіанту використання: «This use case allows a student to register for courses in the current semester» (Цей варіант використання дає студентові можливість зареєструватися на курси в поточному семестрі).
3. Створіть за допомогою MS Word три текстові файли з описами варіантів використання Login (Увійти до системи), Register for Courses (Зареєструватися на курси) і Close Registration (Закрити реєстрацію).

Варіант використання Login:

Короткий опис

Цей варіант використання описує вхід користувача в систему реєстрації курсів.

Основний потік подій

Цей варіант використання починає виконуватися, коли користувач хоче увійти до системи реєстрації курсів.

1. Система просить ім'я користувача і пароль.
2. Користувач вводить ім'я і пароль.
3. Система перевіряє ім'я і пароль, після чого відкривається доступ в систему.

Альтернативні потоки

Неправильне ім'я/пароль. Якщо під час виконання **Основного потоку** виявиться, що користувач ввів неправильне ім'я і/або пароль, система виводить повідомлення про помилку. Користувач може повернутися до початку **Основного потоку** або відмовитися від входу в систему, при цьому виконання варіанту використання завершується.

Передумови

Відсутні.

Постумови

Якщо варіант використання виконаний успішно, користувач входить в систему. Інакше стан системи не змінюється.

Варіант використання Register for Courses:

Короткий опис

Цей варіант використання дозволяє студентові зареєструватися на конкретні курси в поточному семестрі. Студент може змінити свій вибір (відновити або видалити курси), якщо зміна виконується у встановлений час на початку семестру. Система каталогу курсів надає список усіх конкретних курсів поточного семестру.

Основний потік подій

Цей варіант використання починає виконуватися, коли студент хоче зареєструватися на конкретні курси або змінити свій графік курсів.

1. Система просить необхідну дію (створити графік, відновити графік, видалити графік).
2. Коли студент вказує дію, виконується один з підпорядкованих потоків (створити, відновити, видалити або прийняти графік).

Створити графік.

1. Система виконує пошук в каталозі курсів доступних конкретних курсів і виводить їх список.
2. Студент вибирає зі списку 4 основні курси і 2 альтернативні курси.
3. Після вибору система створює графік студента.
4. Виконується підпорядкований потік «Прийняти графік».

Відновити графік.

1. Система виводить поточний графік студента.
2. Система виконує пошук в каталозі курсів доступних конкретних курсів і виводить їх список.
3. Студент може відновити свій вибір курсів, видаляючи або додаючи конкретні курси.
4. Після вибору система оновлює графік.
5. Виконується підпорядкований потік «Прийняти графік».

Видалити графік.

1. Система виводить поточний графік студента.
2. Система просить у студента підтвердження видалення графіку.
3. Студент підтверджує видалення.
4. Система видаляє графік. Якщо графік включає конкретні курси, на які записався студент, він має бути видалений зі списків цих курсів.

Прийняти графік. Для кожного вибраного, але ще не «зафіксованого» конкретного курсу в графіці система перевіряє виконання студентом попередніх вимог (проходження певних курсів), факт відкриття конкретного курсу і відсутність конфліктів графіку. Потім система додає студента в список вибраного конкретного курсу. Курс фіксується в графіці і графік зберігається в системі.

Альтернативні потоки

Зберегти графік. У будь-який момент студент може замість прийняття графіку зберегти його. В цьому випадку крок «Прийняти графік» замінюється на наступний:

1. «Незафіксовані» конкретні курси позначаються в графіці як «вибрані».
2. Графік зберігається в системі.

Не виконані попередні вимоги, курс заповнений або мають місце конфлікти графіку. Якщо під час виконання підпорядкованого потоку «Прийняти графік» система виявить, що студент не виконав необхідні попередні вимоги, або вибраний ним конкретний курс заповнений, або мають місце конфлікти графіку, то видається повідомлення про помилку. Студент може або вибрати інший конкретний курс і продовжити виконання варіанту використання, або зберегти графік, або відмінити операцію, після чого основний потік розпочнеться з початку.

Графік не знайдений. Якщо під час виконання підпорядкованих потоків «Відновити графік» або «Видалити графік» система не може знайти графік студента, то видається повідомлення про помилку. Після того, як студент підтвердить це повідомлення, основний потік розпочнеться з початку.

Система каталогу курсів недоступна – це повідомлення завершення варіанту використання. Якщо виявиться, що неможливо встановити зв'язок з системою каталогу курсів, то буде видано повідомлення про помилку.

Реєстрація на курси закінчена. Якщо на самому початку виконання варіанту використання виявитьтся, що реєстрація на поточний семестр закінчена, буде видано повідомлення і варіант використання завершиться.

Видалення скасоване. Якщо під час виконання підпорядкованого потоку «Видалити графік» студент вирішить не видаляти його, видалення відміняється, і основний потік розпочнеться з початку.

Передумови

Перед початком виконання цього варіанту використання студент повинен увійти до системи.

Постумови

Якщо варіант використання завершиться успішно, графік студента буде створений, оновлений або видалений. Інакше стан системи не зміниться.

Варіант використання Close Registration:

Короткий опис

Цей варіант використання дозволяє реєстраторові закривати процес реєстрації. Конкретні курси, на які не записалося достатньої кількості студентів (менше трьох), відміняються. У розрахункову систему передається інформація про кожного студента по кожному конкретному курсу, щоб студенти могли внести плату за курси.

Основний потік подій

Цей варіант використання починає виконуватися, коли реєстратор просить припинення реєстрації.

1. Система перевіряє стан процесу реєстрації. Якщо реєстрація ще виконується, видається повідомлення і варіант використання завершується.
2. Для кожного конкретного курсу система перевіряє, чи веде його який-небудь професор, і чи записалося на нього не менше трьох студентів. Якщо ці умови виконуються, система фіксує конкретний курс в кожному графіку, який включає цей курс.
3. Для кожного студентського графіку перевіряється наявність в ньому максимальної кількості основних курсів; якщо їх недостатньо, система намагається доповнити альтернативними курсами зі списку цього графіку. Вибирається перший доступний альтернативний курс. Якщо таких курсів немає, то ніяке доповнення не відбувається.
4. Система закриває усі конкретні курси. Якщо в якому-небудь конкретному курсі виявляється менше трьох студентів (з урахуванням доповнень, зроблених в п.3), система відміняє його і виключає з кожного графіку.

5. Система розраховує плату за навчання для кожного студента в поточному семестрі і направляє інформацію в розрахункову систему. Розрахункова система направляє студентам рахунки для оплати з копією їх остаточних графіків.

Альтернативні потоки

Конкретний курс ніхто не веде. Якщо під час виконання основного потоку виявляється, що деякий конкретний курс не ведеться ніяким професором, то цей курс відміняється. Система виключає цей курс з кожного графіку, що містить його.

Розрахункова система недоступна. Якщо неможливо встановити зв'язок з розрахунковою системою, через деякий встановлений час система знову спробує зв'язатися з нею. Спроби повторюватимуться до тих пір, поки зв'язок не встановиться.

Передумови

Перед початком виконання цього варіанту використання реєстратор повинен увійти до системи.

Постумови

Якщо варіант використання завершиться успішно, реєстрація закривається. Інакше стан системи не зміниться.

Вправа 5. Прикріплення файлу до варіанту використання

1. Клацніть правою кнопкою миші на варіанті використання.
2. У меню, що відкрилося, виберіть пункт Open Specification
3. Перейдіть на вкладку файлів.
4. Клацніть правою кнопкою миші на білому полі і з меню, що відкрилося, виберіть пункт Insert File.
5. Вкажіть створений раніше файл і натисніть на кнопку Open, щоб прикріпити файл до варіанту використання.

У результаті представлення варіантів використання в браузері прийме наступний вигляд (рис. 19.5).

Видалення варіантів використання і дійових осіб. Існує два способи видалити елемент моделі – з однієї діаграми або з усієї моделі. Щоб видалити елемент моделі з діаграми:

1. Виділіть елемент на діаграмі.
2. Натисніть на клавішу Delete.
3. Зверніть увагу, що, хоча елемент і видалений з діаграми, він залишився у браузері і на інших діаграмах системи.

Щоб видалити елемент з моделі:

1. Виділіть елемент на діаграмі.
2. Виберіть пункт меню Edit > Delete from Model або натисніть поєднання клавіш CTRL + D.

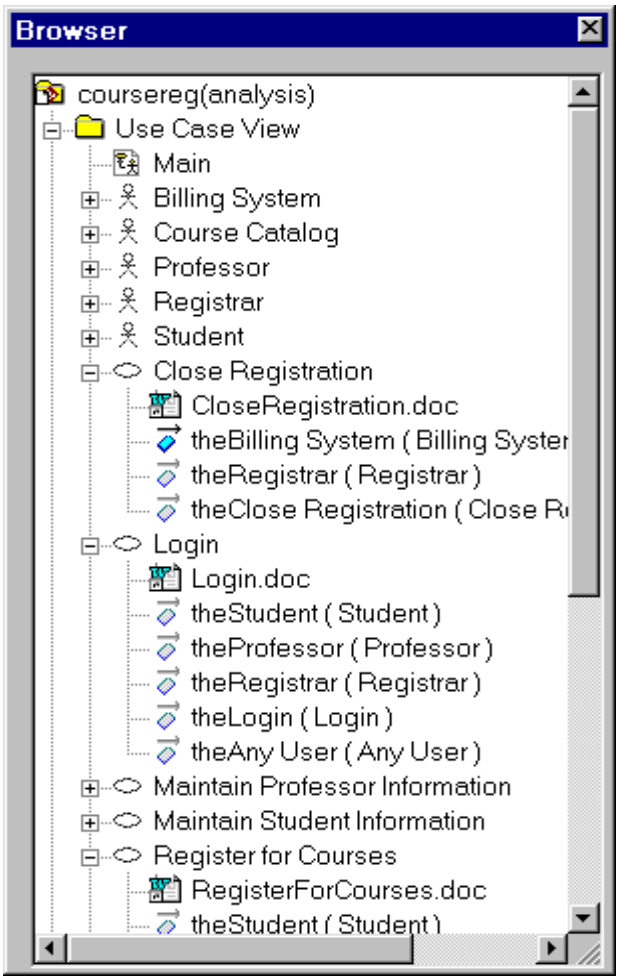


Рисунок 19.5 – Представлення варіантів використання у браузері

19.5. Аналіз системи

19.5.1. Архітектурний аналіз

Прийняття угод з моделювання. Включає:

- використовувані діаграми і елементи моделі;
- правила їх застосування;
- угоди з іменування елементів;
- організацію моделі (пакети).

Приклад угод з моделювання:

Імена варіантів використання мають бути короткими дієслівними фразами. Для кожного варіанту використання має бути створений пакет Use – Case Realization, що включає:

1. Принаймні одну реалізацію варіанту використання.
2. Діаграму «View Of Participating Classes» (VOPC).
3. Імена класів мають бути іменниками, що відповідають, по можливості, поняттям предметної області.
4. Імена класів повинні розпочинатися із заголовної букви.
5. Імена атрибутів і операцій повинні розпочинатися з рядкової букви.

6. Складені імена мають бути суцільними, без підкреслень, кожне окреме слово повинне розпочинатися із заголовної букви.

Реалізація варіанту використання (Use – Case Realization):

Описує реалізацію конкретного варіанту використання в термінах взаємодіючих об'єктів і представляється за допомогою набору діаграм: діаграм класів, що реалізовують варіант використання, і діаграм взаємодії (діаграм послідовності і кооперативних діаграм), об'єктів, що відбивають взаємодію, в процесі реалізації варіанту використання (рис. 19.6).

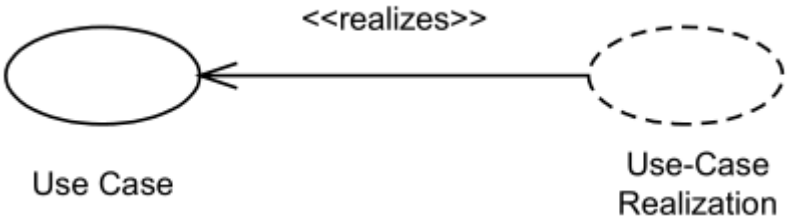


Рисунок 19.6 – Реалізація варіанту використання

Ідентифікація ключових абстракцій. Полягає в попередньому визначенні класів системи (класів аналізу). Джерела – знання предметної області, вимоги до системи, глосарій. Класи аналізу для системи реєстрації показані на рис. 19.7.

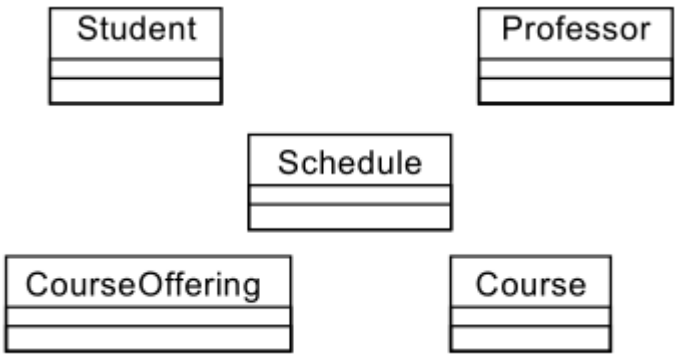


Рисунок 19.7 – Класи аналізу системи реєстрації

Вправа 6. Створення структури моделі і класів аналізу відповідно до вимог архітектурного аналізу

Структура логічного представлення браузера повинна мати такий вигляд (рис. 19.8):

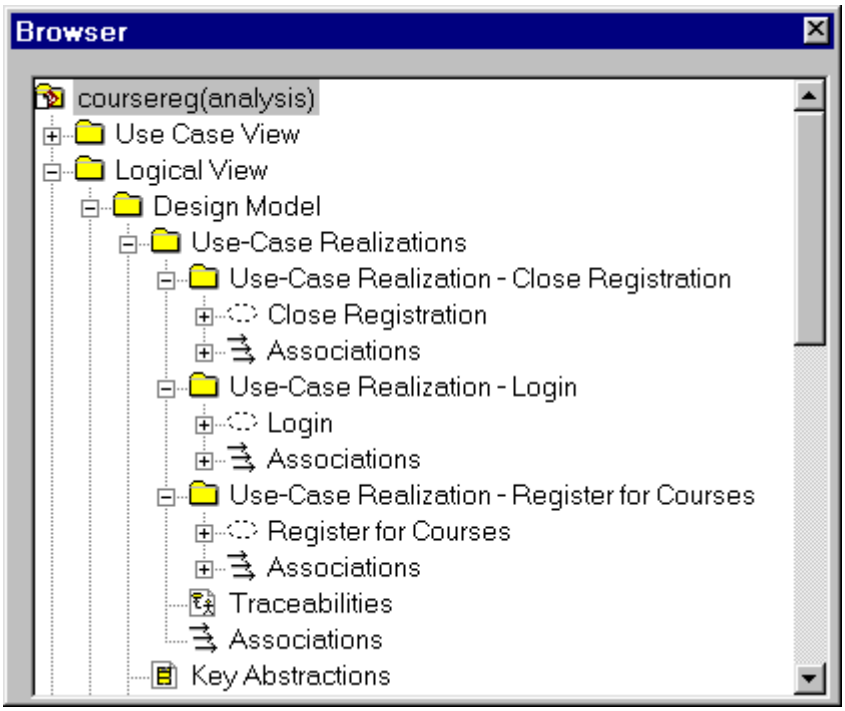


Рисунок 19.8 – Структура логічного представлення браузера

Створення пакетів і діаграми Traceabilities:

1. Клацніть правою кнопкою миші на логічному представленні браузерау.
2. У меню, що відкрилося, виберіть пункт New > Package
3. Назвіть новий пакет Design Model.
4. Створіть аналогічним чином пакети Use – Case Realizations, Use – Case Realization – Close Registration, Use – Case Realization – Login і Use – Case Realization – Register for Courses.
5. У кожному з пакетів типу Use – Case Realization створіть відповідні кооперації Close Registration, Login і Register for Courses (кожна кооперація є варіантом використання із стереотипом «use – case realization», який задається в специфікації варіанту використання).
6. Створіть в пакеті Use – Case Realizations нову діаграму варіантів використання з назвою Traceabilities і побудуйте її відповідно до рис. 19.9.

Створення класів аналізу і відповідної діаграми Key Abstractions:

1. Клацніть правою кнопкою миші на пакеті Design Model.
2. Виберіть в меню, що відкрилося, пункт New > Class. Новий клас під назвою NewClass з’явиться у браузері.
3. Виділіть його і введіть ім’я Student.
4. Створіть аналогічним чином класи Professor, Schedule, Course і CourseOffering.
5. Клацніть правою кнопкою миші на пакеті Design Model.
6. У меню, що відкрилося, виберіть пункт New > Class Diagram.

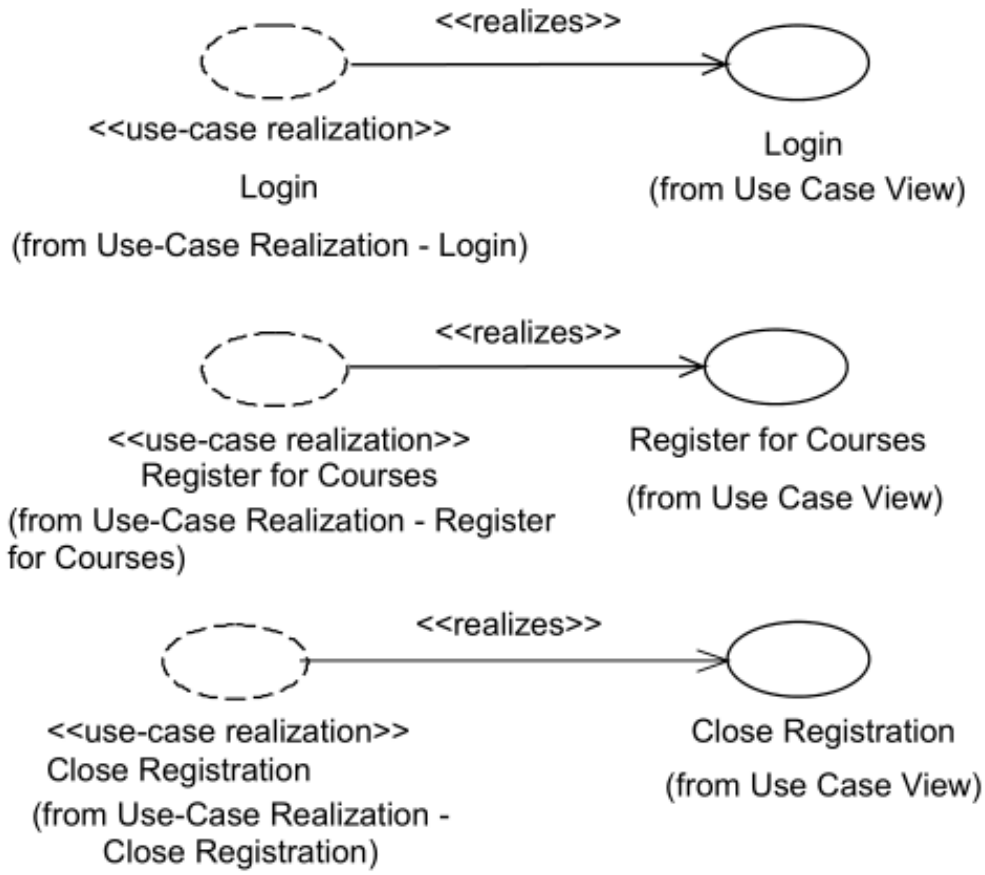


Рисунок 19.9 – Діаграма Traceabilities

7. Назвіть нову діаграму класів Key Abstractions.
8. Щоб розташувати знову створені класи на діаграмі класів, відкрийте її і перетягніть класи на відкриту діаграму мишею. Діаграма класів повинна виглядати як на рис. 19.7.

19.5.2. Аналіз варіантів використання

Ідентифікація класів, що беруть участь у реалізації потоків подій варіанту використання. У потоках подій варіанту використання виявляються класи трьох типів:

1. **Граничні класи (Boundary) –** служать посередниками при взаємодії зовнішніх об'єктів з системою. Як правило, для кожної пари «дійова особа – варіант використання» визначається один граничний клас. Типи граничних класів: призначений для користувача інтерфейс (обмін інформацією з користувачем, без деталей інтерфейсу – кнопок, списків, вікон), системний інтерфейс і апаратний інтерфейс (використовувані протоколи, без деталей їх реалізації).
2. **Класи-сутності (Entity)** – є ключові абстракції (поняття) системи, що розробляється. Джерела виявлення класів-сутностей: ключові абстракції, створені в процесі архітектурного аналізу, глосарій, опис потоків подій варіантів використання.
3. **Класи (Control)**, що управляють, – забезпечують координацію поведінки об'єктів в системі. Можуть бути відсутніми в деяких варіантах

використання, що обмежуються простими маніпуляціями з даними, що зберігаються. Як правило, для кожного варіанту використання визначається один клас, що управляє. Приклади класів, що управляють: менеджер транзакцій, координатор ресурсів, обробник помилок.

Приклад набору класів, що беруть участь у реалізації варіанту використання Register for Courses, наведений на рис. 19.10.

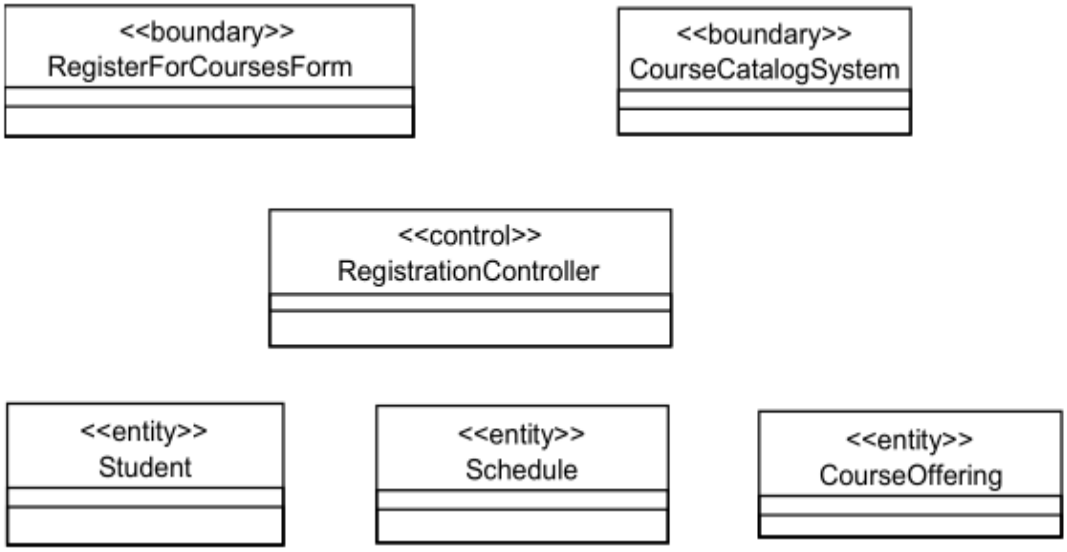


Рисунок 19.10 – Класи, що беруть участь в реалізації варіанту використання Register for Courses

Вправа 7. Створення класів, що беруть участь в реалізації варіанту використання Register for Courses, і діаграми класів «View Of Participating Classes» (VOPC)

1. Клацніть правою кнопкою миші на пакеті Design Model.
2. Виберіть в меню, що відкрилося, пункт New > Class. Новий клас під назвою NewClass з'явиться в браузері.
3. Виділіть його і введіть ім'я RegisterForCoursesForm.
4. Клацніть правою кнопкою миші на класі RegisterForCoursesForm.
5. У меню, що відкрилося, виберіть пункт Open Specification.
6. У полі стереотипу виберіть Boundary і натисніть на кнопку Ок.
7. Створіть аналогічним чином класи CourseCatalogSystem із стереотипом Boundary і RegistrationController із стереотипом Control.
8. Призначте класам Schedule, CourseOffering і Student стереотип Entity.
9. Клацніть правою кнопкою миші на кооперації Register for Courses в пакеті Use – Case Realization – Register for Courses.
10. У меню, що відкрилося, виберіть пункт New > Class Diagram.
11. Назвіть нову діаграму класів VOPC (classes only).
12. Відкрийте її і перетягніть класи на відкриту діаграму відповідно до рис. 19.10.

Розподіл поведінки, що реалізується варіантом використання, між класами. Реалізується за допомогою діаграм взаємодії (діаграм послідовності і

кооперативних діаграм). Для кожного альтернативного потоку подій будується окрема діаграма. Приклади:

- обробка помилок;
- контроль часу виконання;
- обробка неправильних даних, що вводяться.

Недоцільно описувати тривіальні потоки подій.

Вправа 8. Створення діаграм взаємодії

Створимо діаграми послідовності і кооперативні діаграми для основного потоку подій варіанту використання Register for Courses. Готові діаграми послідовності повинні мати вигляд (рис. 19.11-19.15).

Налаштування

1. У меню моделі виберіть пункт Tools > Options.
2. Перейдіть на вкладку діаграм.
3. Контрольні перемикачі Sequence Numbering, Collaboration Numbering мають бути помічені, а Focus of Control – ні. Натисніть Ок.

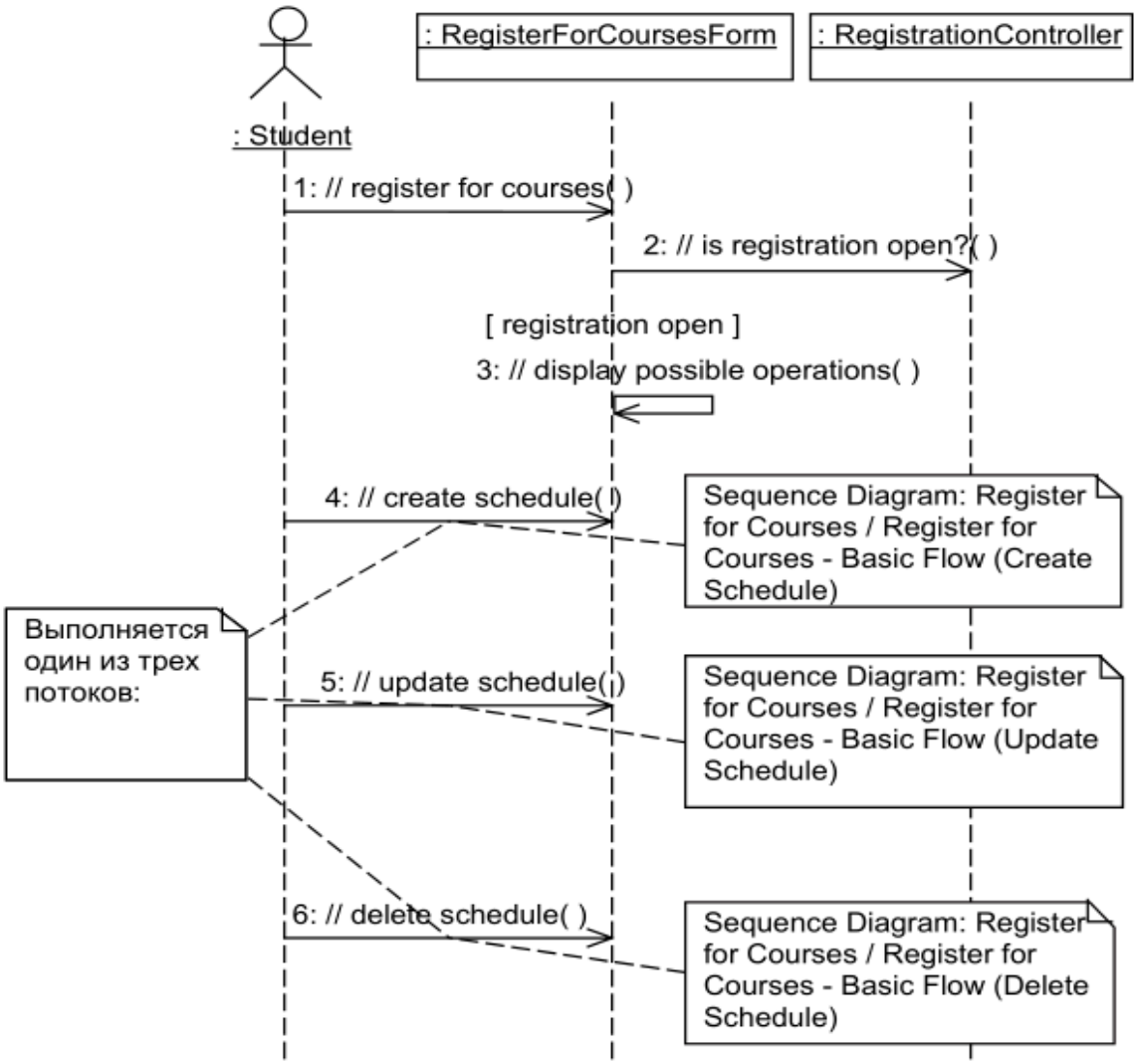


Рисунок 19.11 – Діаграма послідовності Register for Courses – Basic Flow

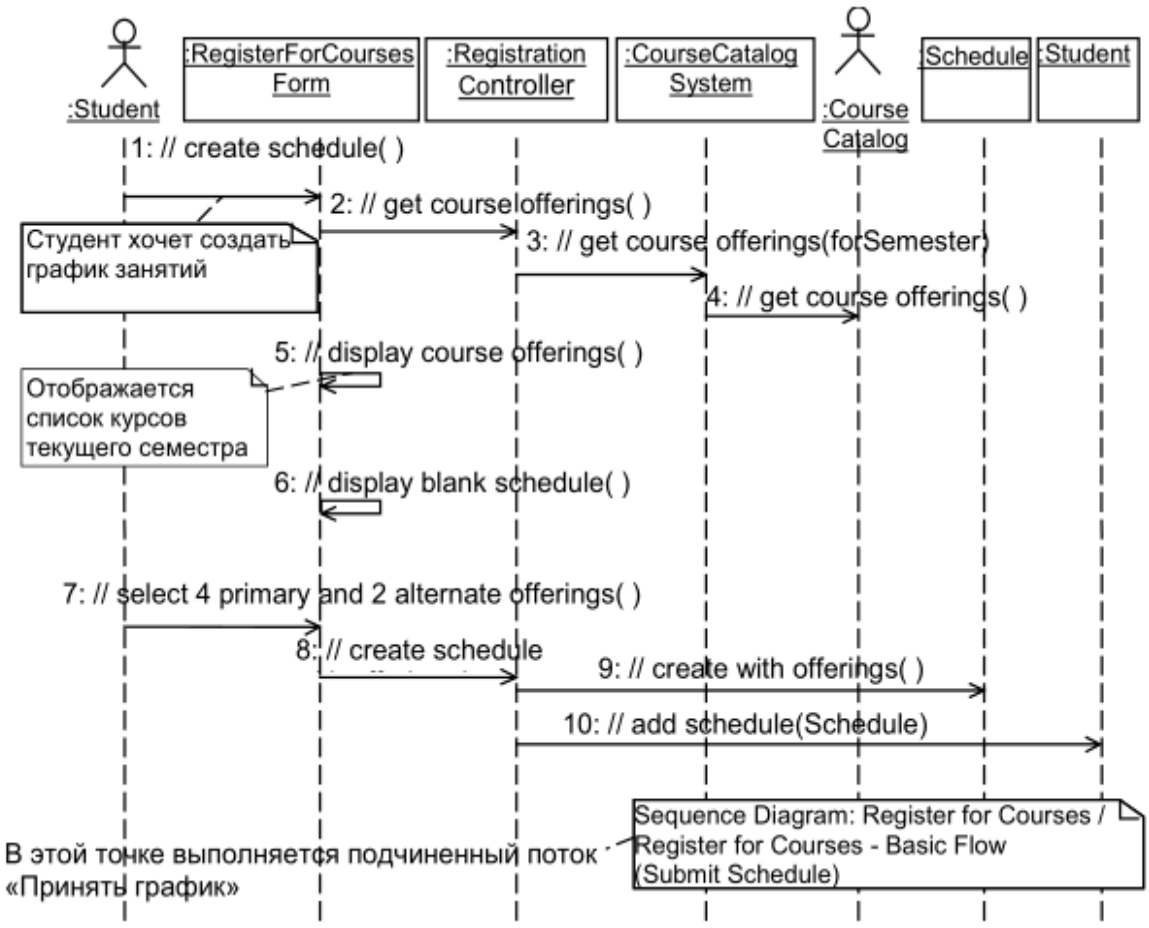


Рисунок 19.12 – Діаграма послідовності Register for Courses – Basic Flow (Create Schedule)

Створення діаграми послідовності

1. Клацніть правою кнопкою миші на кооперації Register for Courses в пакеті Use – Case Realization – Register for Courses.
2. У меню, що відкрилося, виберіть пункт New > Sequence Diagram.
3. Назвіть нову діаграму Register for Courses – Basic Flow.
4. Двічі клацніть на ній, щоб відкрити її.

Додавання на діаграму дійової особи, об’єктів і повідомлень

1. Перетягніть дійову особу Student з браузеру на діаграму.
2. Перетягніть класи RegisterForCoursesForm і RegistrationController з браузеру на діаграму.
3. На панелі інструментів натисніть кнопку Object Message (Повідомлення об’єкту).
4. Проведіть мишею від лінії життя дійової особи Student до лінії життя об’єкту RegisterForCoursesForm.
5. Виділивши повідомлення, введіть його ім’я: // register for courses.
6. Повторіть дії 3 – 5, щоб помістити на діаграму інші повідомлення, як показано на рис. 19.11 (для повідомлення рефлексії 3 використовується кнопка Message to Self).

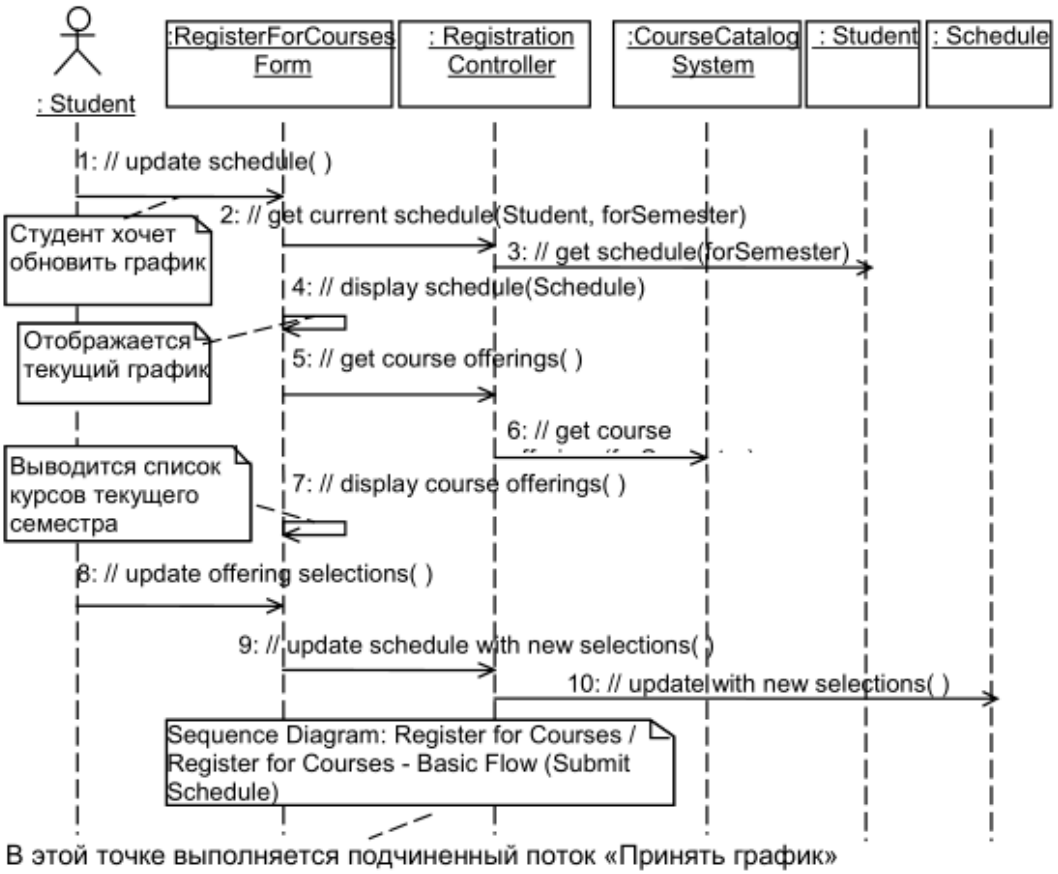


Рисунок 19.13 – Діаграма послідовності Register for Courses – Basic Flow (Update Schedule)

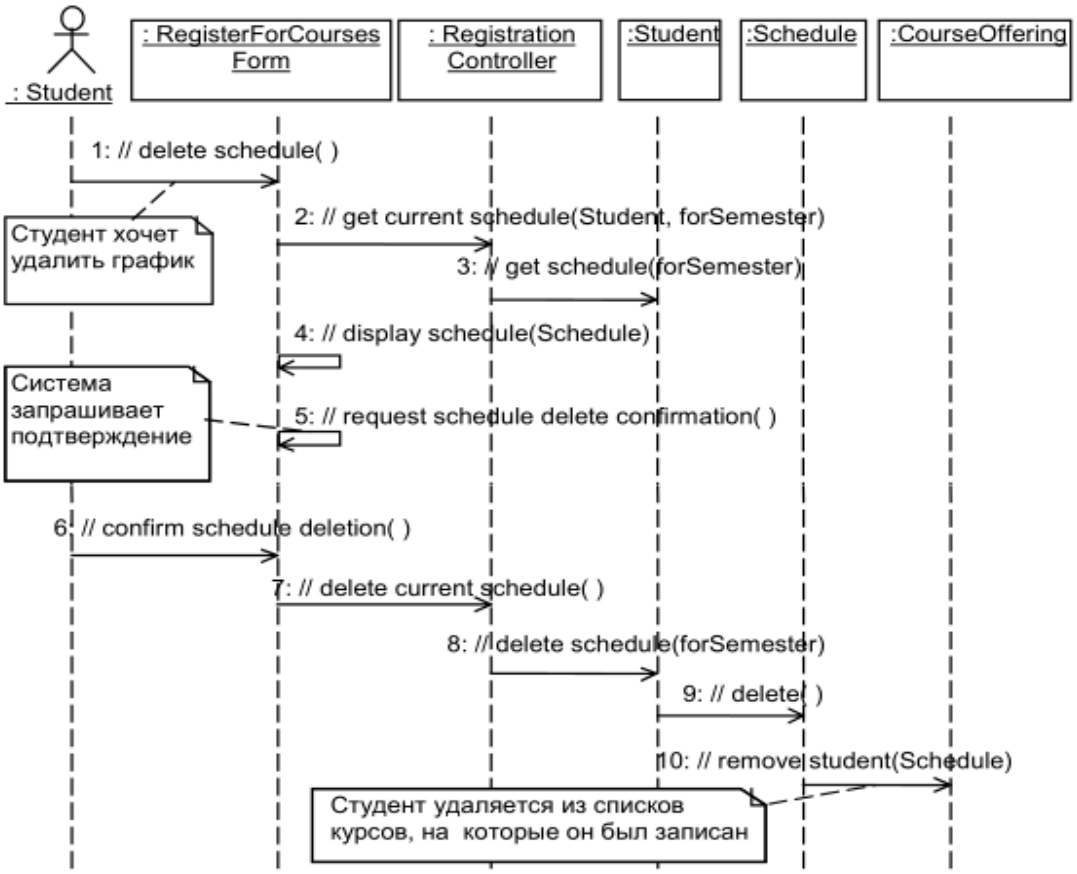


Рисунок 19.14 – Діаграма послідовності Register for Courses – Basic Flow (Delete Schedule)

Співвідношення сполучень з операціями

1. Клацніть правою кнопкою на повідомленні 1, // register for courses.
2. У меню, що відкрилося, виберіть пункт <new operation>. З'явиться вікно специфікації операції.
3. У полі імені залиште ім'я повідомлення – // register for courses.
4. Натисніть на кнопку Ок, щоб закрити вікно специфікації операції і повернутися на діаграму.
5. Повторіть дії 1 – 4, поки не співвіднесете з операціями усі інші повідомлення.

Виконайте аналогічні дії для створення діаграм послідовності, показаних на рис. 19.12 – 19.15. Зверніть увагу, що на діаграмі рис. 19.15 з'явився об'єкт нового класу PrimarySheduleOfferingInfo (класу асоціацій, що описує зв'язок між класами Shedule і OfferingInfo), який треба заздалегідь створити.

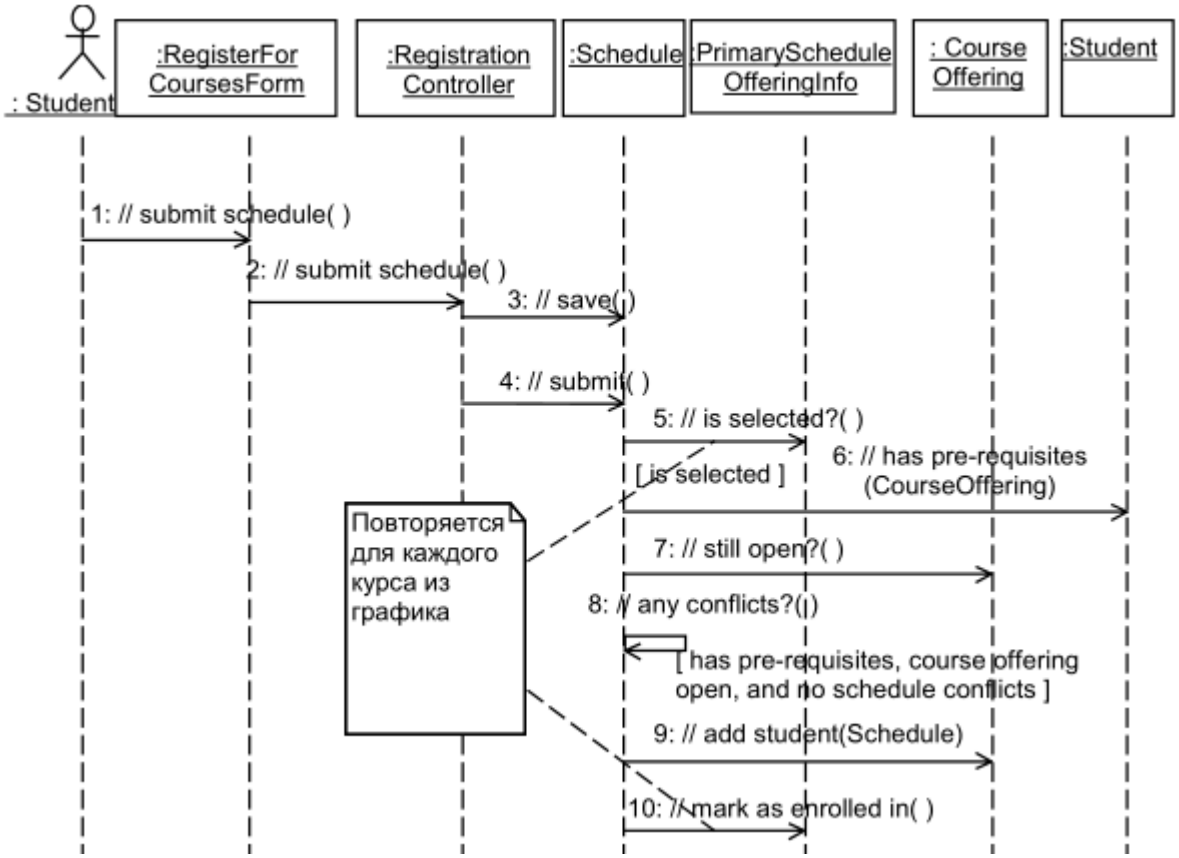


Рисунок 19.15 – Діаграма послідовності Register for Courses – Basic Flow (Submit Schedule)

Створення приміток

Щоб помістити на діаграму примітку:

1. Натисніть на панелі інструментів кнопку Note.
2. Клацніть мишею в тому місці діаграми, куди збираєтеся
3. Щоб прикріпити примітку до елементу діаграми, на панелі інструментів натисніть кнопку Anchor Notes To Item (Прикріпити примітки до елементу).

4. Натиснувши ліву кнопку миші, проведіть покажчик від примітки до елементу діаграми, з яким воно буде пов’язано. Між приміткою і елементом виникне штрихова лінія.
5. Щоб створити примітку-посилання на іншу діаграму, створіть порожню примітку (без тексту) і перетягніть на нього з браузеру потрібну діаграму.

Окрім приміток, на діаграму можна помістити також і текстову область. З її допомогою можна, наприклад, додати до діаграми заголовок.

Щоб помістити на діаграму текстову область:
1. На панелі управління натисніть кнопку Text Box.
2. Клацніть мишею усередині діаграми, щоб помістити туди текстову область.
3. Виділивши цю область, введіть в неї текст.

Створення кооперативної діаграми

Для створення кооперативної діаграми досить відкрити діаграму послідовності і натиснути клавішу F5.

Визначення обов’язків (responsibilities), атрибутів і асоціацій класів. Обов’язок (responsibility) – дія, яку об’єкт зобов’язаний виконувати за запитом інших об’єктів. Обов’язок перетвориться в одну або більше за операції класу на кроці проектування. Обов’язки визначаються, виходячи з повідомлень на діаграмах взаємодії, і документуються в класах у вигляді операцій «аналізу», які з’являються там автоматично в процесі побудови діаграм взаємодії (співвідношення сполучень з операціями).

Так, діаграма класів VOPC (рис. 19.10) після побудови діаграм взаємодії у вправі 8 повинна набрати вигляду, зображеного на рис. 19.16.

Атрибути класів аналізу визначаються, виходячи зі знань про предметну область, вимог до системи і глосарію.

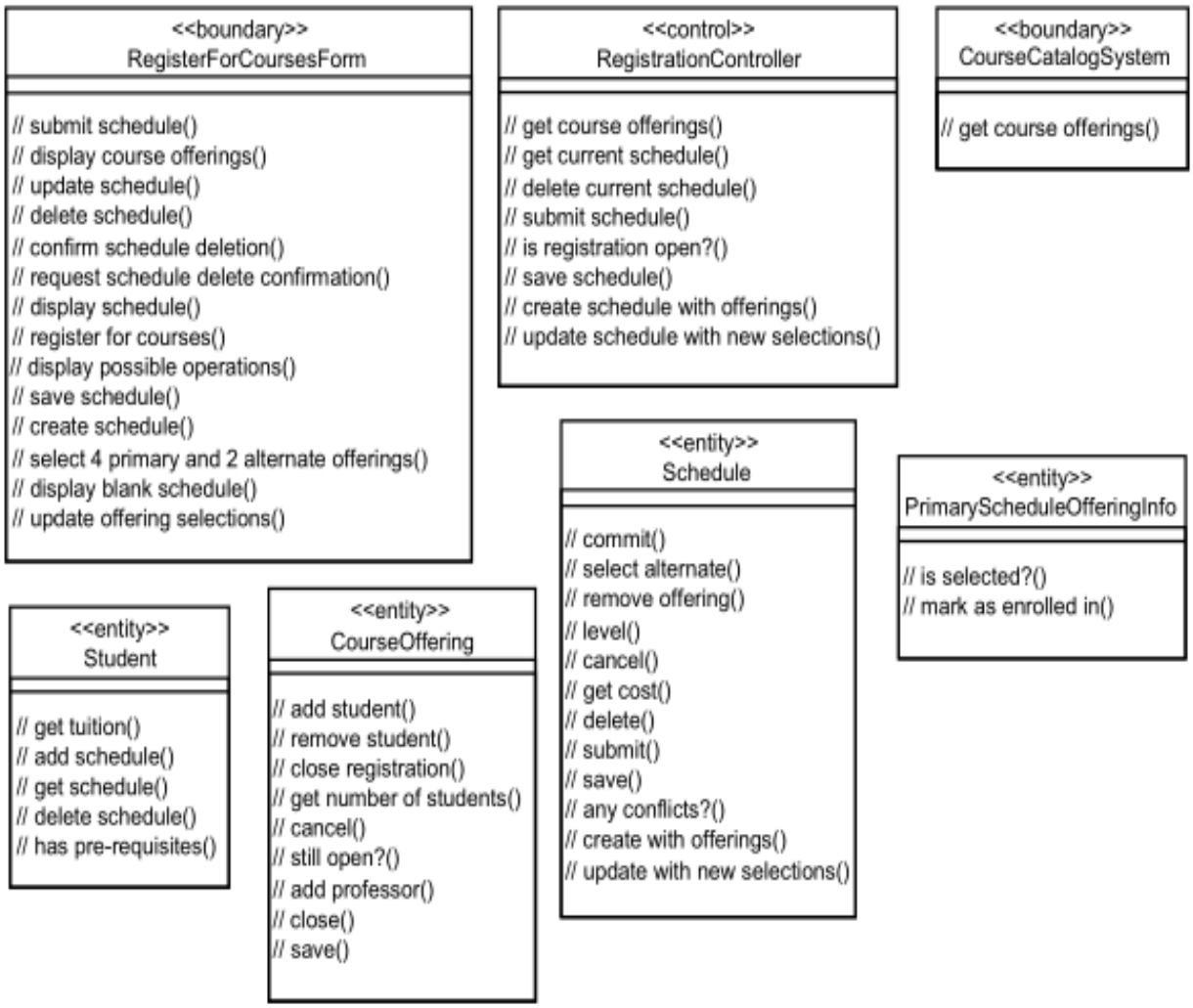


Рисунок 19.16 – Діаграма класів VOPC (classes only) з операціями «аналізу»

Вправа 9. Додавання атрибутів до класів

Налаштування

1. У меню моделі виберіть пункт Tools > Options.
2. Перейдіть на вкладку Diagram.
3. Переконайтеся, що перемикач Show All Attributes помічений.
4. Переконайтеся, що перемикачі Suppress Attributes і Suppress Operations не помічені.

Додавання атрибутів

1. Клацніть правою кнопкою миші на класі Student.
2. У меню, що відкрилося, виберіть пункт New Attribute.
3. Введіть новий атрибут address
4. Натисніть клавішу Enter.
5. Повторіть кроки 1 – 4, додавши атрибути name і studentID.
6. Додайте атрибути до класів CourseOffering, Shedule і PrimaryScheduleOfferingInfo, як показано на рис. 19.17.

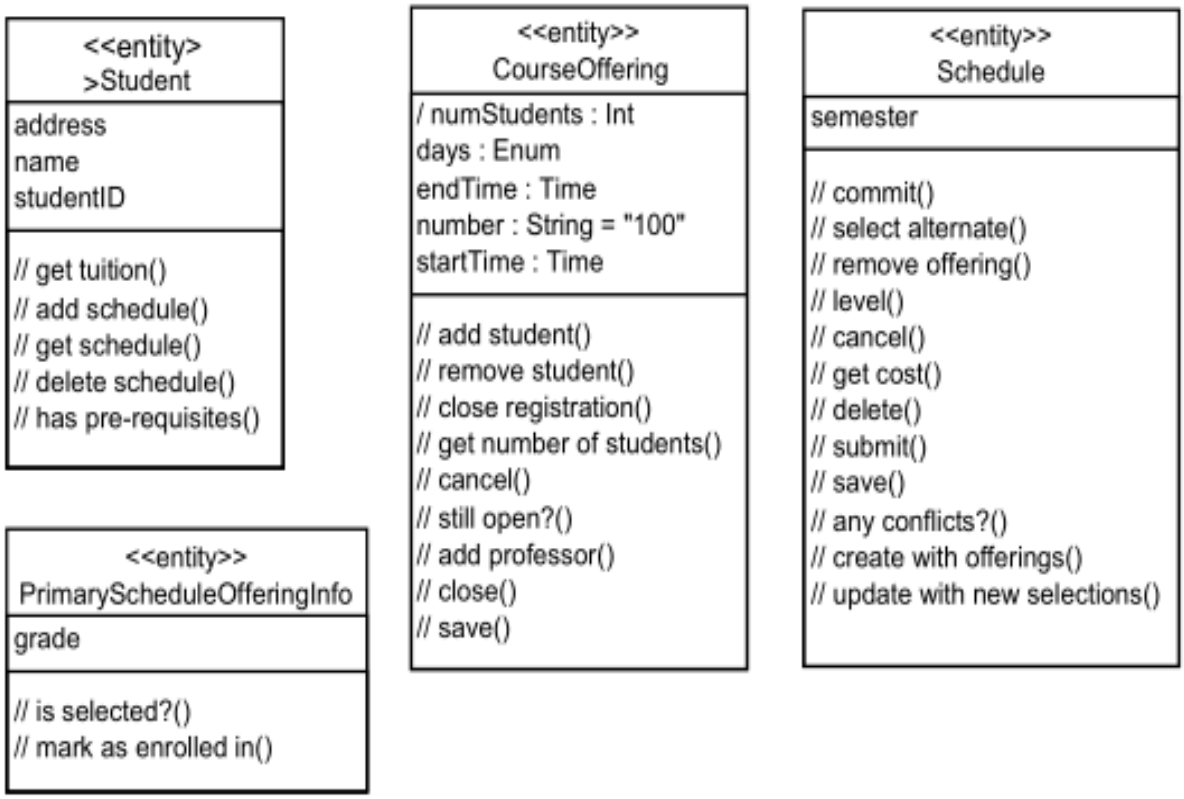


Рисунок 19.17 – Класи з операціями «аналізу» і атрибутами

Зв’язки між класами (асоціації) визначаються на основі діаграм взаємодії. Якщо два об’єкти взаємодіють (обмінюються повідомленнями), між ними повинен існувати зв’язок (шлях взаємодії). Для асоціацій задаються множинність і, можливо, напрям навігації. Можуть використовуватися множинні асоціації, агрегації і класи асоціацій.

Вправа 10. Додавання зв’язків

Додамо зв’язки до класів, що беруть участь у варіанті використання Register for Courses. Для відображення зв’язків між класами побудуємо три нових діаграм класів в кооперації Register for Courses пакету Use – Case Realization – Register for Courses (рис. 19.18 – 19.20).

Додані два нові класи – підкласи FulltimeStudent (студент очного відділення) і ParttimeStudent (студент вечірнього відділення).

На цій діаграмі показані класи асоціацій, зв’язки, що описують, між класами Schedule і CourseOffering і доданий суперклас ScheduleOfferingInfo. Дані і операції, що містяться в цьому класі (status – курс включений в графік або скасований), належать як до основних, так і до альтернативних курсів, тоді як оцінка (grade) і остаточне включення курсу в графік можуть мати місце тільки для основних курсів.

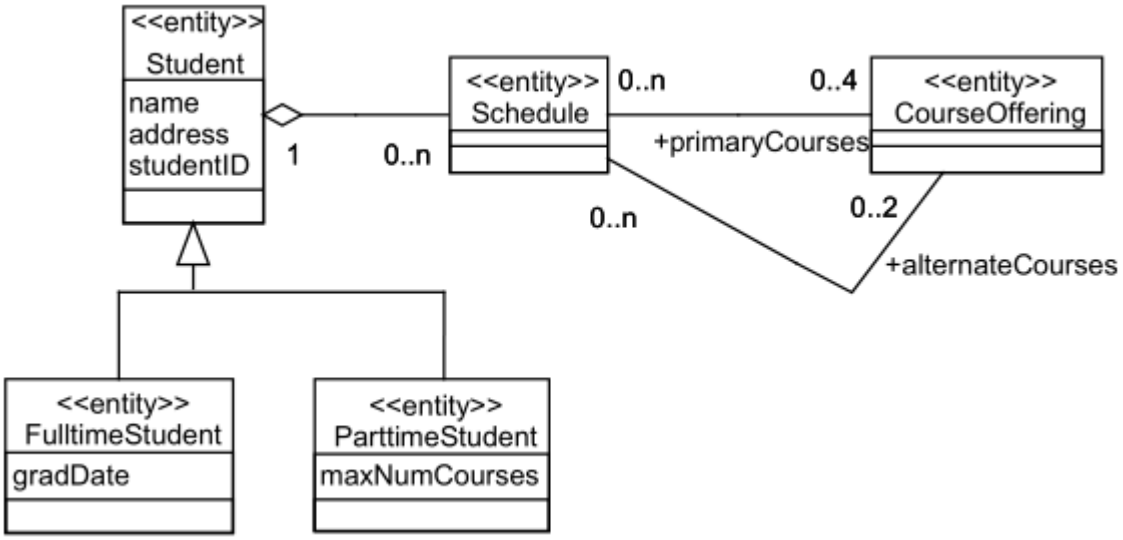


Рисунок 19.18 – Діаграма Entity Classes (класи-сутності)

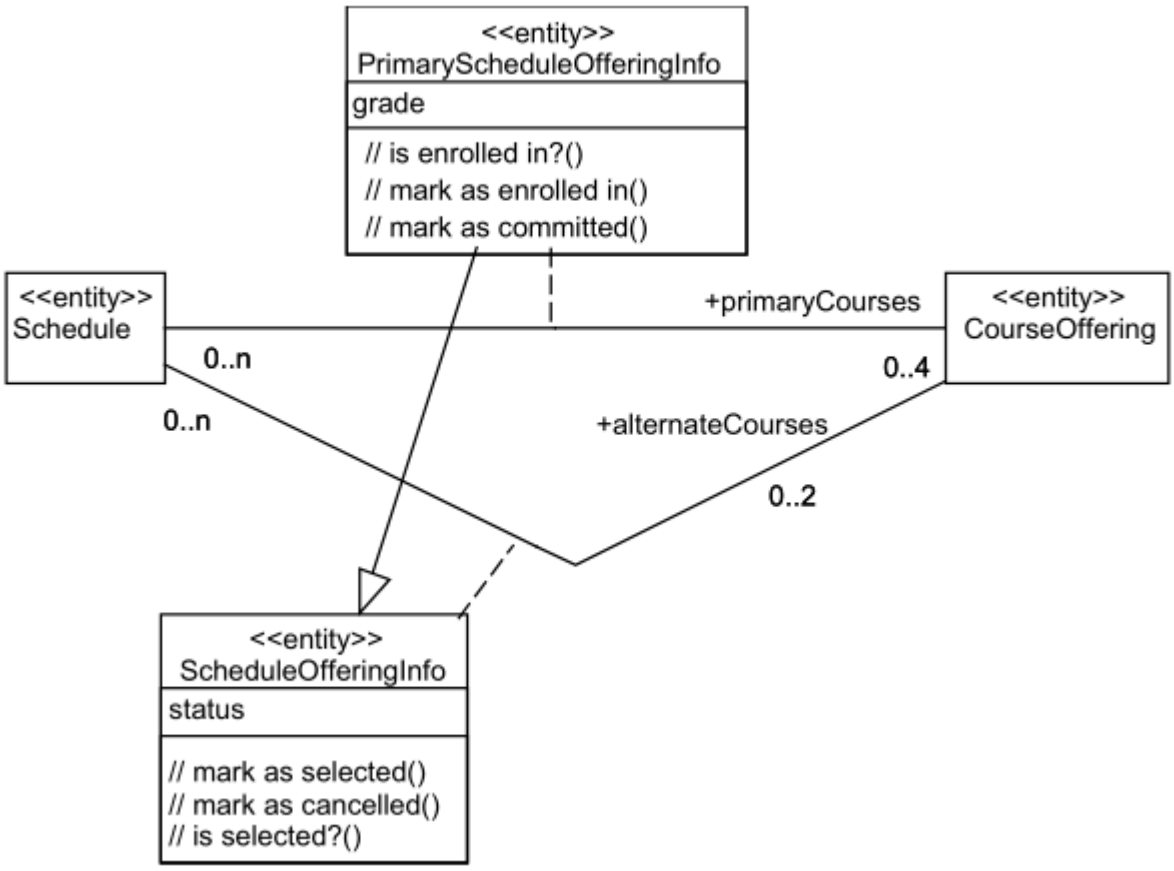


Рисунок 19.19 – Діаграма CourseOfferingInfo

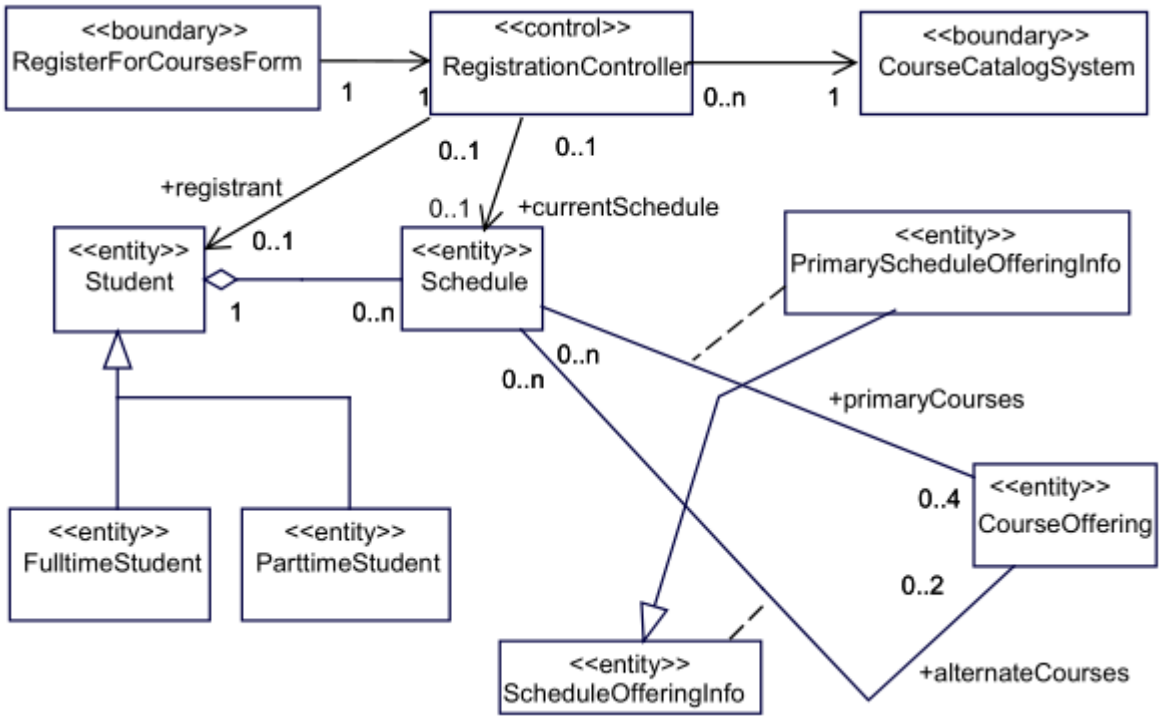


Рисунок 19.20 – Повна діаграма класів VOPC (без атрибутів і операцій)

Створення асоціацій

Асоціації створюють безпосередньо на діаграмі класів. Панель інструментів діаграми класів містить кнопки для створення як одно-, так і двонаправлених асоціацій. Щоб на діаграмі класів створити асоціацію:

1. Натисніть на панелі інструментів кнопку Association.
2. Проведіть мишею лінію асоціації від одного класу до іншого.

Щоб задати можливості навігації за асоціацією:

1. Клацніть правою кнопкою миші на зв’язку з того кінця, на якому хочете показати стрілку.
2. У меню, що відкрилося, виберіть пункт Navigable.

Щоб створити асоціацію рефлексії:

1. На панелі інструментів діаграми натисніть кнопку Association.
2. Проведіть лінію асоціації від класу до якого-небудь місця поза класом.
3. Відпустіть кнопку миші.
4. Проведіть лінію асоціації назад до класу.

Створення агрегацій

1. Натисніть кнопку Aggregation панелі інструментів.
2. Проведіть лінію агрегації від класу-частини до цілого.

Щоб помістити на діаграму класів агрегацію рефлексії:

1. На панелі інструментів діаграми натисніть кнопку Aggregation.
2. Проведіть лінію агрегації від класу до якого-небудь місця поза класом.
3. Відпустіть кнопку миші.
4. Проведіть лінію агрегації назад до класу.

Створення узагальнень

При створенні узагальнення може потрібно перенести деякі атрибути або операції з одного класу в інший. Якщо, наприклад, знадобиться перенести їх з підкласу в суперклас Employee, у браузері для цього досить просто перетягнути атрибути або операції з одного класу в інший. Не забудьте видалити іншу копію атрибуту з другого підкласу, якщо він є.

Щоб помістити узагальнення на діаграму класів:

1. Натисніть кнопку Generalization панелі інструментів.
2. Проведіть лінію узагальнення від підкласу до суперкласу.

Специфікації зв'язків

Специфікації зв'язків стосуються імен асоціацій, ролевих імен, множинності і класів асоціацій.

Щоб задати множинність зв'язку:

1. Клацніть правою кнопкою миші на одному кінці зв'язку.
2. У меню, що відкрилося, виберіть пункт Multiplicity.
3. Вкажіть потрібну множинність.
4. Повторіть те ж саме для іншого кінця зв'язку.

Щоб задати ім'я зв'язку:

1. Виділіть потрібний зв'язок.
2. Введіть його ім'я.

Щоб задати зв'язку ролеве ім'я:

1. Клацніть правою кнопкою миші на асоціації з потрібного кінця.
2. У меню, що відкрилося, виберіть пункт role Name.
3. Введіть ролеве ім'я.

Щоб задати елемент зв'язку (клас асоціацій):

1. Відкрийте вікно специфікації необхідного зв'язку.
2. Перейдіть на вкладку Detail.
3. Задайте елемент зв'язку в полі Link Element.

Завдання для самостійної роботи

Виконати аналіз варіанту використання Close Registration і побудувати відповідні діаграми взаємодії і класів.

19.6. Проектування системи

19.6.1. Проектування архітектури

Цілі проектування архітектури системи:

- аналіз взаємодій між класами аналізу, виявлення підсистем і інтерфейсів;
- уточнення архітектури з урахуванням можливостей повторного використання;

– ідентифікація архітектурних рішень і механізмів, необхідних для проектування системи.

Вводяться глобальні пакети:

– базисні (foundation) класи (списки, черги і так далі);
– обробники помилок (error handling classes);
– математичні бібліотеки;
– утиліти;
– бібліотеки інших постачальників.

Визначаються проектні класи (design classes):

– клас аналізу відображається в проектний клас, якщо він простий або представляє єдину логічну абстракцію;
– складний клас аналізу може бути розбитий на декілька класів, перетворений в пакет або в підсистему.

Приклади можливих підсистем:

– класи, що забезпечують складний комплекс послуг (наприклад, забезпечення безпеки і захист);
– граничні класи, що реалізовують складний призначений для користувача інтерфейс або інтерфейс із зовнішніми системами;
– різні продукти: комунікаційне ПЗ (middleware, підтримка COM/CORBA), доступ до баз даних, типи і структури даних (стеки, списки, черги), загальні утиліти (математичні бібліотеки), різні прикладні продукти.

Ухвалення рішення про перетворення класу в підсистему визначається досвідом і знаннями архітектора проекту.

Угоди з проектування інтерфейсів:

1. Ім’я інтерфейсу: коротке (одно-два слова), відбиваюче його роль в системі.
2. Опис інтерфейсу: повинно відображати його обов’язки (розмір – невеликий абзац).
3. Опис операцій: ім’я, що відображати результат операції, ключові алгоритми, значення, що повертається, параметри з типами.
4. Документування інтерфейсу: характер використання операцій і порядок їх виконання (показується за допомогою діаграм послідовності), тестові плани і сценарії і так далі. Уся ця інформація об’єднується в спеціальний пакет із стереотипом <<subsystem>>, який містить елементи, що утворюють підсистему, діаграми послідовності і/або кооперативні діаграми, що описують взаємодію елементів при реалізації операцій інтерфейсу, і інші діаграми.
5. Клас <<subsystem proxy>> безпосередньо реалізує інтерфейс і управляє реалізацією його операцій.
6. Усі інтерфейси мають бути повністю визначені в процесі проектування архітектури, оскільки вони служитимуть в якості точок синхронізації при паралельній розробці.

Виділення архітектурних рівнів:

1. Application Layer – містить елементи прикладного рівня (призначений для користувача інтерфейс);
2. Business Services Layer – містить елементи, що реалізовують бізнес-логіку додатків (найбільш стійка частина системи);
3. Middleware Layer – забезпечує сервіси, незалежні від платформи.

Приклад виділення архітектурних рівнів і об'єднання елементів моделі в пакети для системи реєстрації наведений на рис. 19.21.

Для того щоб помістити клас в пакет, досить просто перетягнути його у браузері на потрібний пакет.

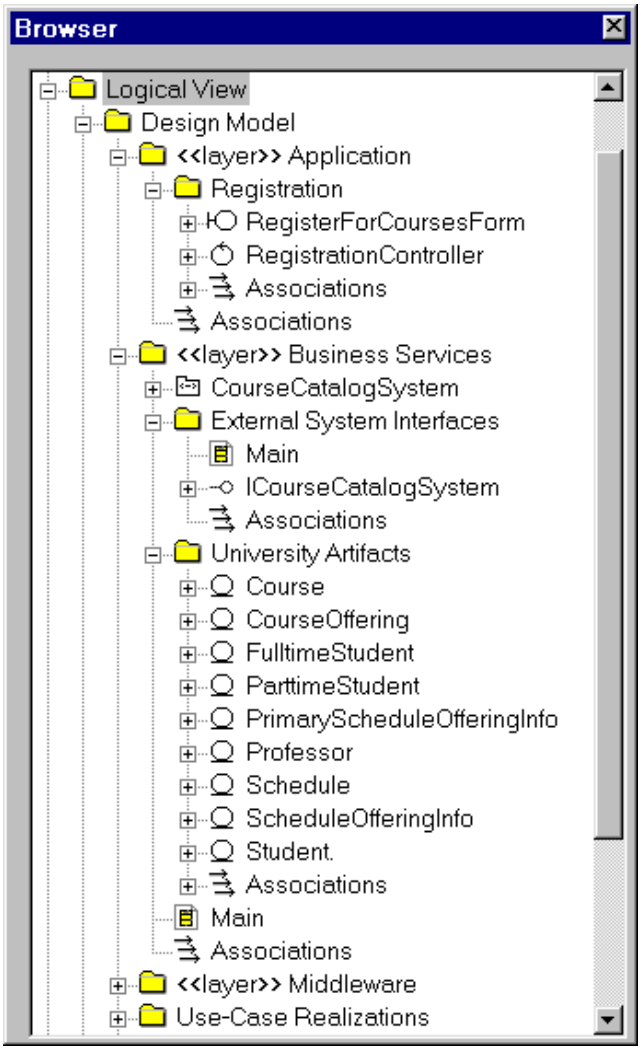


Рисунок 19.21 – Структура логічного представлення моделі на кроці проектування

Це представлення відбиває такі рішення, прийняті архітектором:

1. Виділені три архітектурні рівні (створені три пакети із стереотипом <<layer>>).
2. У пакеті Application створений пакет Registration, куди включені граничні класи, що управляють.
3. Граничний клас CourseCatalogSystem перетворений в підсистему (пакет CourseCatalogSystem із стереотипом <<subsystem>>).
4. У пакет Business Services, окрім підсистеми CourseCatalogSystem, включені ще два пакети: пакет External System Interfaces включає

інтерфейс з підсистемою CourseCatalogSystem (клас ICourseCatalogSystem із стереотипом <<Interface>>), а пакет University Artifacts – усі класи-сутності.

Структура і діаграми пакету (підсистеми) CourseCatalogSystem показана на рис. 19.22 – 19.26.

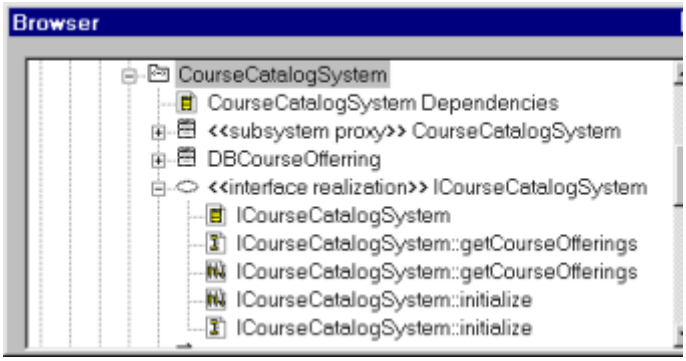


Рисунок 19.22 – Структура пакету CourseCatalogSystem

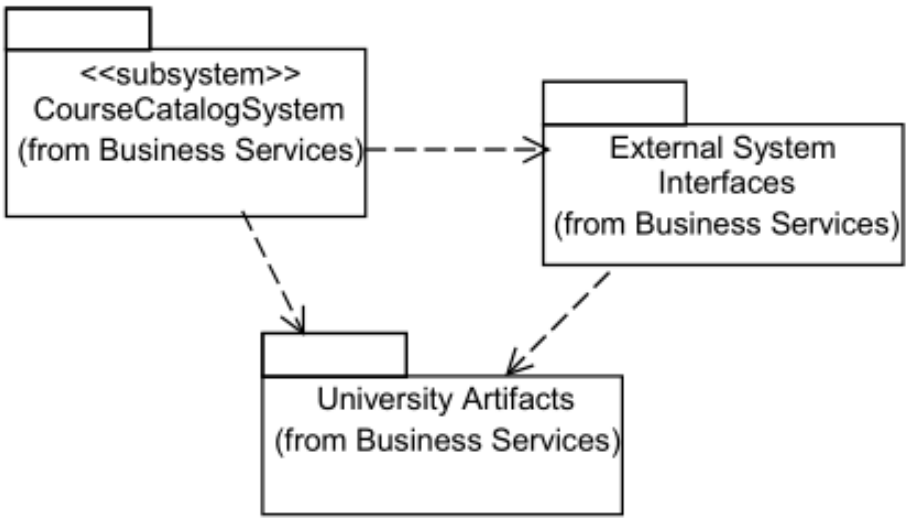


Рисунок 19.23 – Залежності між підсистемою і іншими пакетами (діаграма класів CourseCatalogSystem Dependencies)

Щоб помістити залежність між пакетами на діаграму класів:

1. Натисніть кнопку Dependency панелі інструментів.
2. Проведіть лінію залежності від залежного пакету до того, від якого він залежить.

Клас DBCourseOfferring відповідає за взаємодію з БД каталогу курсів (рис. 19.25, 19.26).

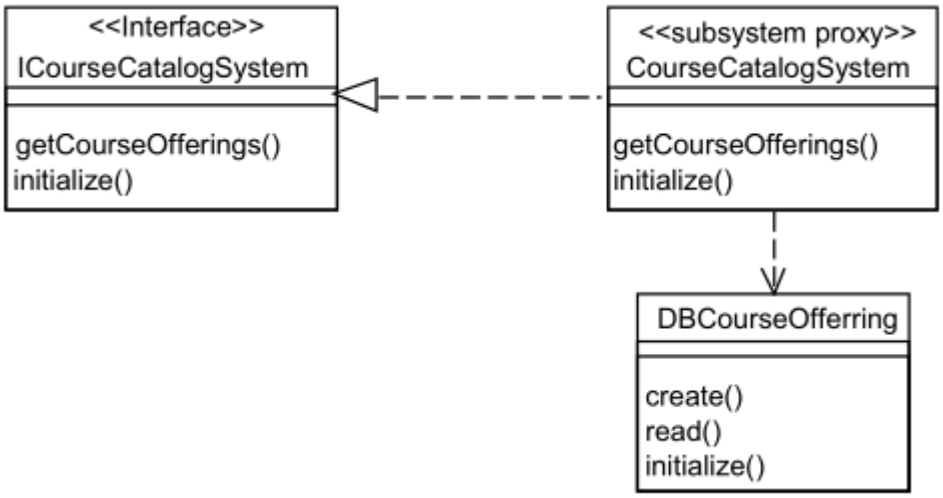


Рисунок 19.24 – Класи, що реалізовують інтерфейс підсистеми (діаграма класів ICourseCatalogSystem)

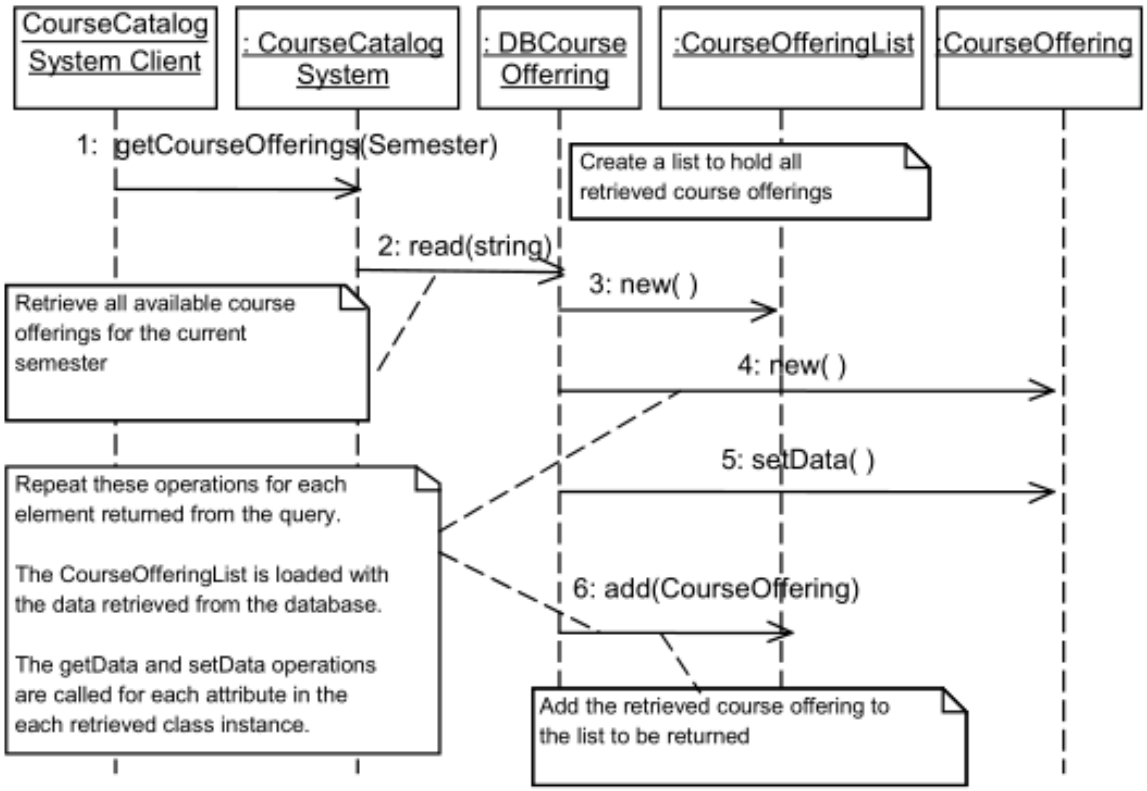


Рисунок 19.25 – Діаграма послідовності ICourseCatalogSystem::getCourse Offerings, що описує взаємодію елементів при реалізації операції інтерфейсу getCourseOfferings

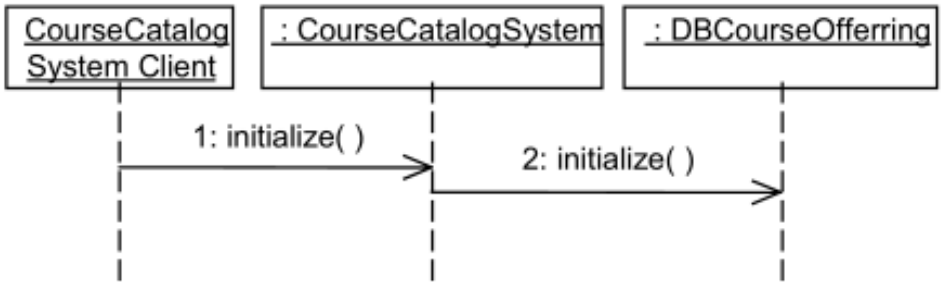


Рисунок 19.26 – Діаграма послідовності ICourseCatalogSystem: :initialize, що описує взаємодію елементів при реалізації операції інтерфейсу initialize

19.6.2. Моделювання розподіленої конфігурації системи

Розподілена конфігурація системи моделюється за допомогою діаграми розміщення. Її основні елементи:

- вузол (node) – обчислювальний ресурс (процесор або інший пристрій (дискова пам'ять, контроллери різних пристроїв тощо). Для вузла можна задати процеси, що виконуються на ньому;
- з'єднання (connection) – канал взаємодії вузлів (мережа).

Приклад: мережева конфігурація системи реєстрації (без процесів, рис. 19.27).

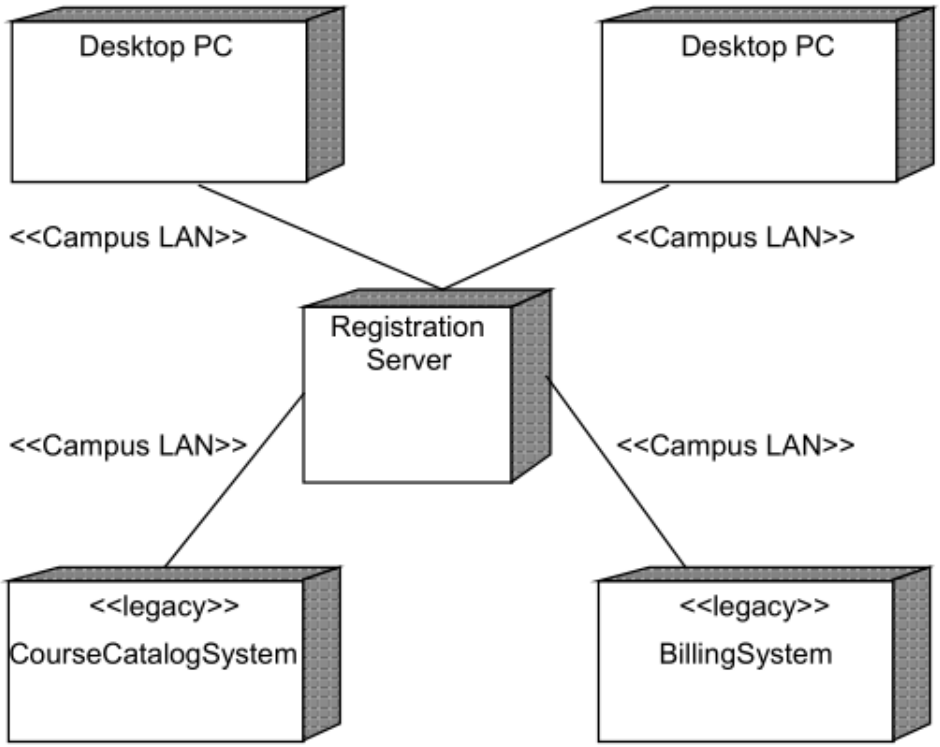


Рисунок 19.27 – Мережева конфігурація системи реєстрації

Розподіл процесів по вузлах мережі робиться з урахуванням таких чинників:

- використовувані зразки розподілу (триланкова клієнт-серверна конфігурація, «товстий клієнт», «тонкий клієнт», рівноправні вузли (peer – to – peer) і так далі);

– час відгуку;
– мінімізація мережевого трафіку;
– потужність вузла;
– надійність устаткування і комунікацій.

Приклад розподілу процесів по вузлах наведений на рис. 19.28.

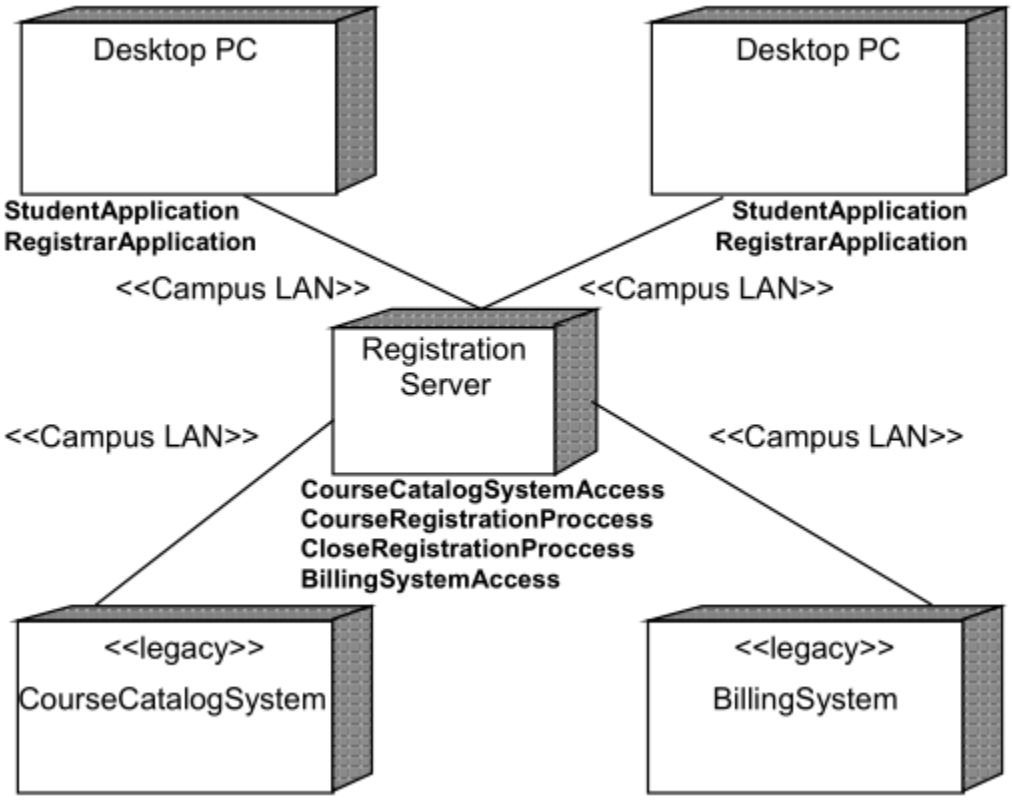


Рисунок 19.28 – Мережева конфігурація системи реєстрації з розподілом процесів по вузлах

Вправа 11. Створення діаграми розміщення системи реєстрації

Щоб відкрити діаграму розміщення, потрібно двічі клацнути мишею на представленні Deployment View (представленні розміщення) у браузері.

Щоб помістити на діаграму процесор:

1. На панелі інструментів діаграми натисніть кнопку Processor.
2. Клацніть на діаграмі розміщення в тому місці, куди хочете його помістити.
3. Введіть ім’я процесора.

У специфікаціях процесора можна ввести інформацію про його стереотип, характеристики і планування. Стереотипи застосовуються для класифікації процесорів (наприклад, комп’ютерів під управлінням UNIX або ПК).

Характеристики процесора – цей його фізичний опис. Він може, зокрема, включати швидкість процесора і об’єм пам’яті.

Поле планування (scheduling) процесора містить опис того, як здійснюється планування його процесів:

1. **Preemptive (з пріоритетом)**. Високопріоритетні процеси мають перевагу перед низькопріоритетними.
2. **Non preemptive (без пріоритету).** У процесів немає пріоритету. Поточний процес виконується до його завершення, після чого починається наступний.
3. **Cyclic (циклічний).** Управління передається між процесами по кругу. Кожному процесу дається певний час на його виконання, потім управління переходить до наступного процесу.
4. **Executive (виконавчий).** Існує деякий обчислювальний алгоритм, який і управляє плануванням процесів.
5. **Manual (вручну).** Процеси плануються користувачем.

Щоб призначити процесору стереотип:

1. Відкрийте вікно специфікації процесора.
2. Перейдіть на вкладку General.
3. Введіть стереотип в поле Stereotype.

Щоб ввести характеристики і планування процесора:

1. Відкрийте вікно специфікації процесора.
2. Перейдіть на вкладку Detail.
3. Введіть характеристики в поле характеристик.
4. Вкажіть один з типів планування.

Щоб показати планування на діаграмі:

1. Клацніть правою кнопкою миші на процесорі.
2. У меню, що відкрилося, виберіть пункт Show Scheduling.

Щоб додати зв’язок на діаграму:

1. На панелі інструментів натисніть кнопку Connection.
2. Клацніть на вузлі діаграми.
3. Проведіть лінію зв’язку до іншого вузла.

Щоб призначити зв’язку стереотип:

1. Відкрийте вікно специфікації зв’язку.
2. Перейдіть на вкладку General.
3. Введіть стереотип в поле Stereotype (Стереотип).

Щоб додати процес:

1. Клацніть правою кнопкою миші на процесорі у браузері.
2. У меню, що відкрилося, виберіть пункт New > Process.
3. Введіть ім’я нового процесу.

Щоб показати процеси на діаграмі:

1. Клацніть правою кнопкою миші на процесорі.
2. У меню, що відкрилося, виберіть пункт Show Processes.

19.6.3. Проектування класів

Класи аналізу перетворяться в проектні класи:

- проектування граничних класів – залежить від можливостей середовища розробки призначеного для користувача інтерфейсу (GUI Builder);

- проектування класів-сутностей – з урахуванням міркувань продуктивності (виділення в окремі класи атрибутів з різною частотою використання);
- проектування класів, що управляють, – видалення класів, що реалізовують просту передачу інформації від граничних класів до сутностей;
- ідентифікація стійких (persistent) класів, що містять інформацію, що зберігається.

Обов'язки класів, визначені в процесі аналізу, перетворяться в операції. Кожній операції привласнюється ім'я, що характеризує її результат. Визначається повна сигнатура операції: operationName(parameter: class,.): returnType. Створюється короткий опис операції, включаючи сенс усіх її параметрів. Визначається видимість операції: public, private, protected. Визначається зона дії (scope) операції: екземпляр або класифікатор.

Визначаються (уточнюються) атрибути класів:

- окрім імені, задається тип і значення за умовчанням (необов'язкове): attributeName: Type = Default;
- враховуються угоди по іменуванню атрибутів, прийняті в проекті і мові реалізації;
- задається видимість атрибутів: public, private, protected;
- при необхідності визначаються похідні (обчислювані) атрибути.

Приклад визначення операцій і атрибутів показаний на рис. 19.29.

Вправа 12. Визначення атрибутів і операцій для класу Student

Щоб задати тип даних, значення за умовчанням і видимість атрибуту:

1. Клацніть правою кнопкою миші на атрибуті у браузері.
2. У меню, що відкрилося, виберіть пункт Open Specification.
3. Вкажіть тип даних в списку, що розкривається, типів або введіть власний тип даних.
4. У полі Initial Field (Первинне значення) введіть значення атрибуту за умовчанням.
5. У полі Export Control виберіть видимість атрибуту: Public, Protected, Private або Implementation. За умовчанням видимість усіх атрибутів відповідає Private (рис. 19.29).

Щоб змінити нотацію для позначення видимості:

1. У меню моделі виберіть пункт Tools > Options.
2. Перейдіть на вкладку Notation.
3. Помітьте контрольний перемикач Visibility as Icons, щоб використати нотацію Rose, або зніміть позначку, щоб використати нотацію UML.

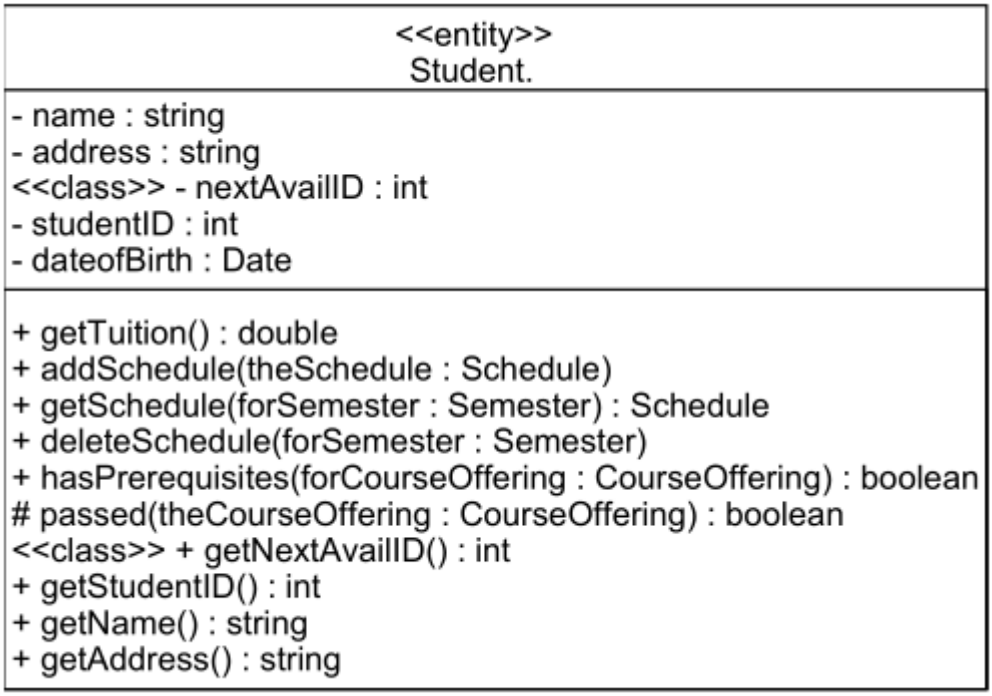


Рисунок 19.29 – Клас Student з повністю певними операціями і атрибутами

Примітка. Зміна значення цього параметра приведе до зміни нотації тільки для нових діаграм і не торкнеться вже існуючих діаграм.

Щоб задати тип значення, що повертається, стереотип і видимість операції:

1. Клацніть правою кнопкою миші на операції у браузері.
2. Відкрийте вікно специфікації класу цієї операції.
3. Вкажіть тип значення, що повертається, в списку, що розкривається, або введіть свій тип.
4. Вкажіть стереотип у відповідному списку, що розкривається, або введіть новий.
5. У полі Export Control вкажіть значення видимості операції: Public, Protected, Private або Implementation. За умовчанням видимість усіх операцій встановлена в public.

Щоб додати до операції аргумент:

1. Відкрийте вікно специфікації операції.
2. Перейдіть на вкладку Detail.
3. Клацніть правою кнопкою миші в області аргументів, в меню, що відкрилося, виберіть Insert.
4. Введіть ім’я аргументу.
5. Клацніть на колонці Data type і введіть туди тип даних аргументу.
6. Якщо потрібно, клацніть на колонці default і введіть значення аргументу за замовчуванням.

Визначення станів для класів моделюється за допомогою діаграм станів.

Діаграми станів створюються для опису об’єктів з високим рівнем динамічної поведінки.

Як приклад розглянемо поведінку об’єкту класу CourseOffering (рис. 19.30). Він може знаходитися у відкритому стані (можливе додавання нового студента) або в закритому стані (максимальна кількість студентів вже записалася на курс). Таким чином, конкретний стан залежить від кількості студентів, пов’язаних з об’єктом CourseOffering. Розглядаючи кожен варіант використання, можна виділити ще два стани: ініціалізація (до початку реєстрації студентів на курс) і відміна (курс виключається з розкладу).

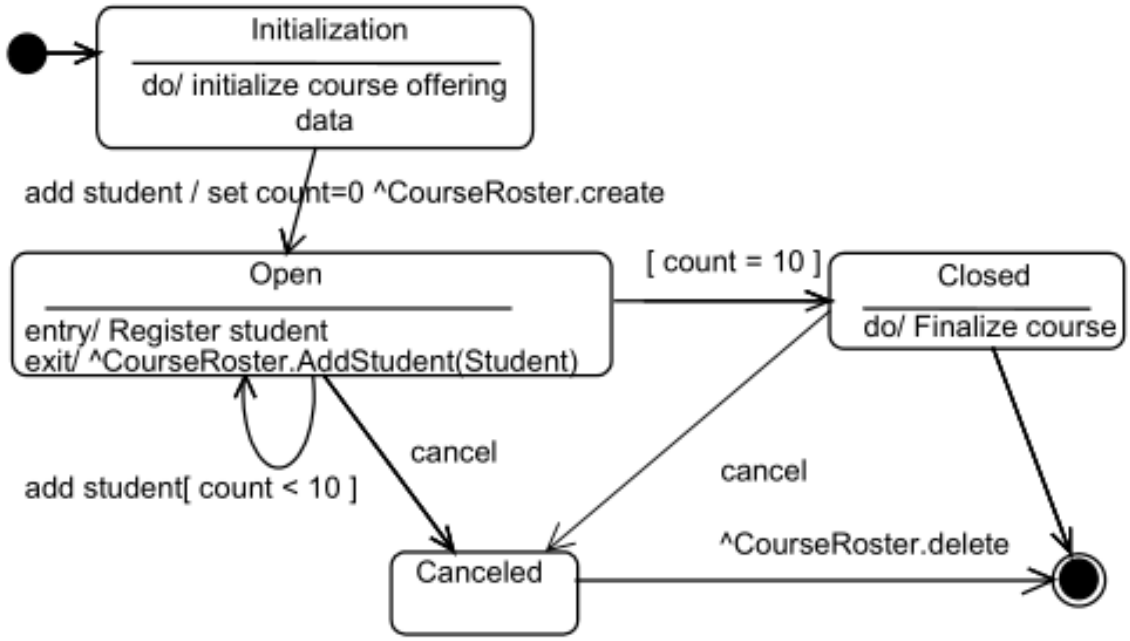


Рисунок 19.30 – Діаграма станів для класу CourseOffering

Вправа 13. Створення діаграми станів для класу CourseOffering

Для створення діаграми станів:

1. Клацніть правою кнопкою миші у браузері на потрібному класі.
2. У меню, що відкрилося, виберіть пункт New > Statechart Diagram.

Щоб додати стан:

1. На панелі інструментів натисніть кнопку State
2. Клацніть мишею на діаграмі станів в тому місці, куди хочете його помістити.

Усі елементи стану можна додати за допомогою вкладки Detail вікна специфікації стану.

Щоб додати діяльність:

1. Відкрийте вікно специфікації необхідного стану.
2. Перейдіть на вкладку Detail.
3. Клацніть правою кнопкою миші на вікні Actions.
4. У меню, що відкрилося, виберіть пункт Insert.
5. Двічі клацніть на новій дії.
6. Введіть дію в поле Actions.
7. У вікні When вкажіть Do, щоб зробити нову дію діяльністю.

Щоб додати вхідну дію, у вікні When вкажіть On Entry.

Щоб додати вихідну дію, у вікні When вкажіть On Exit.

Щоб послати подію:
1. Відкрийте вікно специфікації необхідного стану.
2. Перейдіть на вкладку Detail.
3. Клацніть правою кнопкою миші на вікні Actions.
4. У меню, що відкрилося, виберіть пункт Insert.
5. Двічі клацніть на новій дії.
6. В якості типу дії вкажіть Send Event.
7. У відповідні поля введіть подію (event), аргументи (arguments) і цільовий об'єкт (Target).

Щоб додати перехід:
1. Натисніть кнопку Transition панелі інструментів.
2. Клацніть мишею на стані, звідки здійснюється перехід.
3. Проведіть лінію переходу до того стану, де він завершується.

Щоб додати перехід рефлексії:
1. Натисніть кнопку Transition to Self панелі інструментів.
2. Клацніть на тому стані, де здійснюється перехід рефлексії.

Щоб додати подію, його аргументи, умову, що захищає, і дію:
1. Двічі клацніть на переході, щоб відкрити вікно його специфікації.
2. Перейдіть на вкладку General.
3. Введіть подію в поле Event.
4. Введіть аргументи в поле Arguments.
5. Введіть умову, що захищає, в поле Condition.
6. Введіть дію в поле Action.

Щоб відправити подію:
1. Двічі клацніть на переході, щоб відкрити вікно його специфікації.
2. Перейдіть на вкладку Detail.
3. Введіть подію в поле Send Event.
4. Введіть аргументи в поле Send Arguments.
5. Задайте мету в полі Send Target.

Для вказівки початкового або кінцевого стану:
1. На панелі інструментів натисніть кнопку Start State або End State.
2. Клацніть мишею на діаграмі станів в тому місці, куди хочете помістити стан.

Уточнення асоціацій: деякі асоціації (семантичні, структурні, стійкі зв'язки за даними) можуть бути перетворені в залежності (неструктурні, тимчасові зв'язки, відбивають видимість), а агрегації – в композиції (рис.19.31).

Для перетворення агрегації в композицію:
1. Клацніть правою кнопкою миші на тому кінці агрегації, який упирається в клас-частину (см рис.19.31 – Schedule).
2. У меню, що відкрилося, виберіть пункт Containment.
3. Вкажіть метод включення By Value.

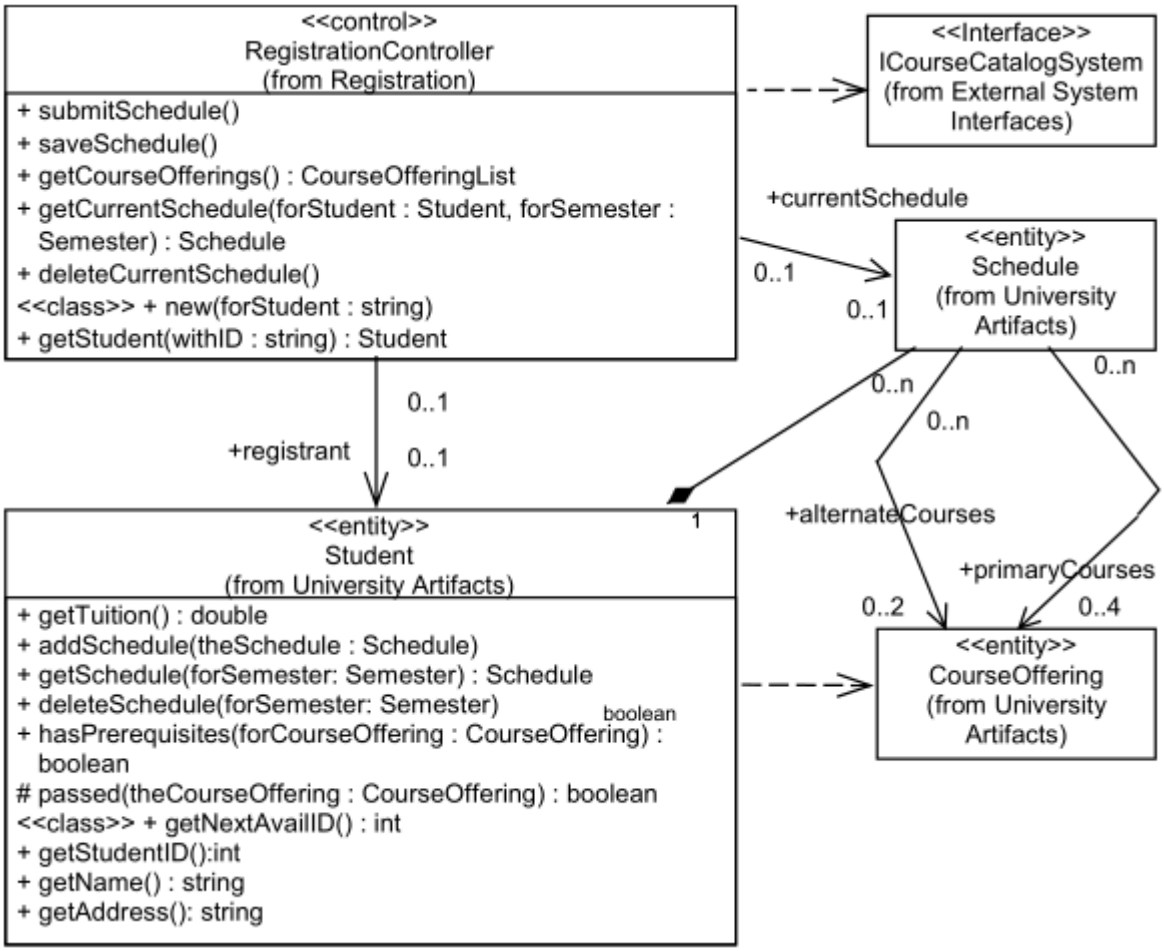


Рисунок 19.31 – Приклад перетворення асоціацій і агрегацій

Примітка. Значення By Value припускає, що ціле і частина створюються і руйнуються одночасно, що відповідає композиції. Агрегація (By Reference) припускає, що ціле і частина створюються і руйнуються в різний час.

Уточнення узагальнень: у разі ситуації з міграцією підкласів (студент може переходити з очної форми навчання на вечірню) ієрархія спадкоємства реалізується так, як показано на рис. 19.32. Таке рішення підвищує стійкість системи (не треба модифікувати опис об’єкту).

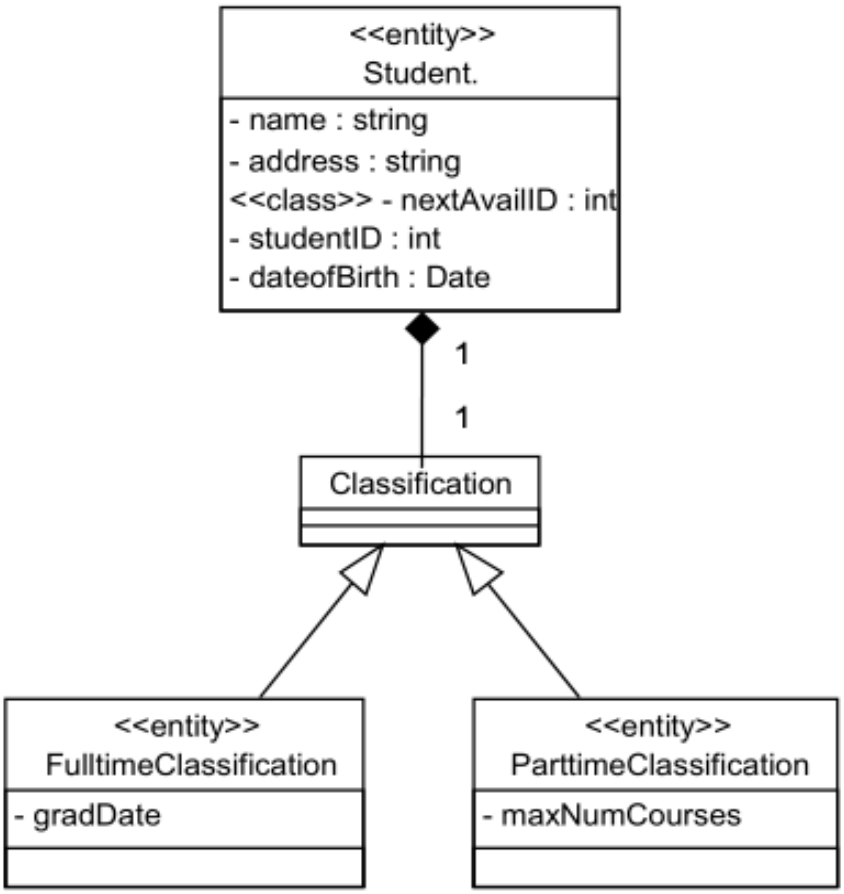


Рисунок 19.32 – Перетворення узагальнення

19.6.4. Проектування баз даних

Проектування реляційних баз даних виконується з використанням засобу Data Modeler. Його робота грунтується на відомому механізмі відображення об'єктної моделі в реляційну. Результатом є побудова діаграми «сутність-зв'язок» і подальша генерація опису БД на SQL.

Вправа 14. Проектування реляційної бази даних

Проектування БД складається з наступних кроків:

Створення нового компонента – бази даних:

1. Клацніть правою кнопкою миші на представленні компонентів.
2. У меню, що відкрилося, виберіть пункт Data Modeler > New > Database.
3. Відкрийте вікно специфікації знову створеного компонента DB_0 і в списку Target виберіть Oracle 8.x.

Визначення стійких (persistent) класів:

1. Відкрийте вікно специфікації класу Student в пакеті University Artifacts.
2. Перейдіть на вкладку Detail.
3. Встановіть значення перемикача Persistence в Persistent.
4. Виконайте такі ж дії для класів Classification, FulltimeClassification і ParttimeClassification.
5. Відкрийте клас Student у браузері, натиснувши « + «.

6. Клацніть правою кнопкою миші на атрибуті studentID.
7. У меню, що відкрилося, виберіть пункт Data Modeler > Part of Object Identity (вказівка атрибуту в якості частини первинного ключа).

Примітка. Кроки 5, 6 і 7 можна виконувати в Rational Rose, починаючи з версії 2001.

Створення схеми БД:
1. Клацніть правою кнопкою миші на пакеті University Artifacts.
2. У меню, що відкрилося, виберіть пункт Data Modeler > Transform to Data Model.
3. У вікні, що з'явилося, в списку Target Database вкажіть DB_0 і натисніть OK. В результаті в логічному уявленні з'явиться новий пакет Schemas.
4. Відкрийте пакет Schemas і клацніть правою кнопкою миші на пакеті <<Schema>> S_0.
5. У меню, що відкрилося, виберіть пункт Data Modeler > New > Data Model Diagram.
6. Відкрийте пакет, потім відкрийте знову створену діаграму «сутність-зв'язок» NewDiagram і перенесіть на неї усі класи-таблиці, що знаходяться в пакеті <<Schema>> S_0. Діаграма, що вийшла, показана на рис. 19.33.

Генерація опису БД на SQL:
1. Клацніть правою кнопкою миші на пакеті <<Schema>> S_0.
2. У меню, що відкрилося, виберіть пункт Data Modeler > Forward Engineer.
3. У вікні майстра Forward Engineering Wizard, що відкрилося, натисніть Next.
4. Відмітьте усі прапорці генерації DDL і натисніть Next.
5. Вкажіть ім'я і розташування текстового файлу з результатами генерації і натисніть Next.
6. Після завершення генерації відкрийте створений текстовий файл і прогляньте результати.

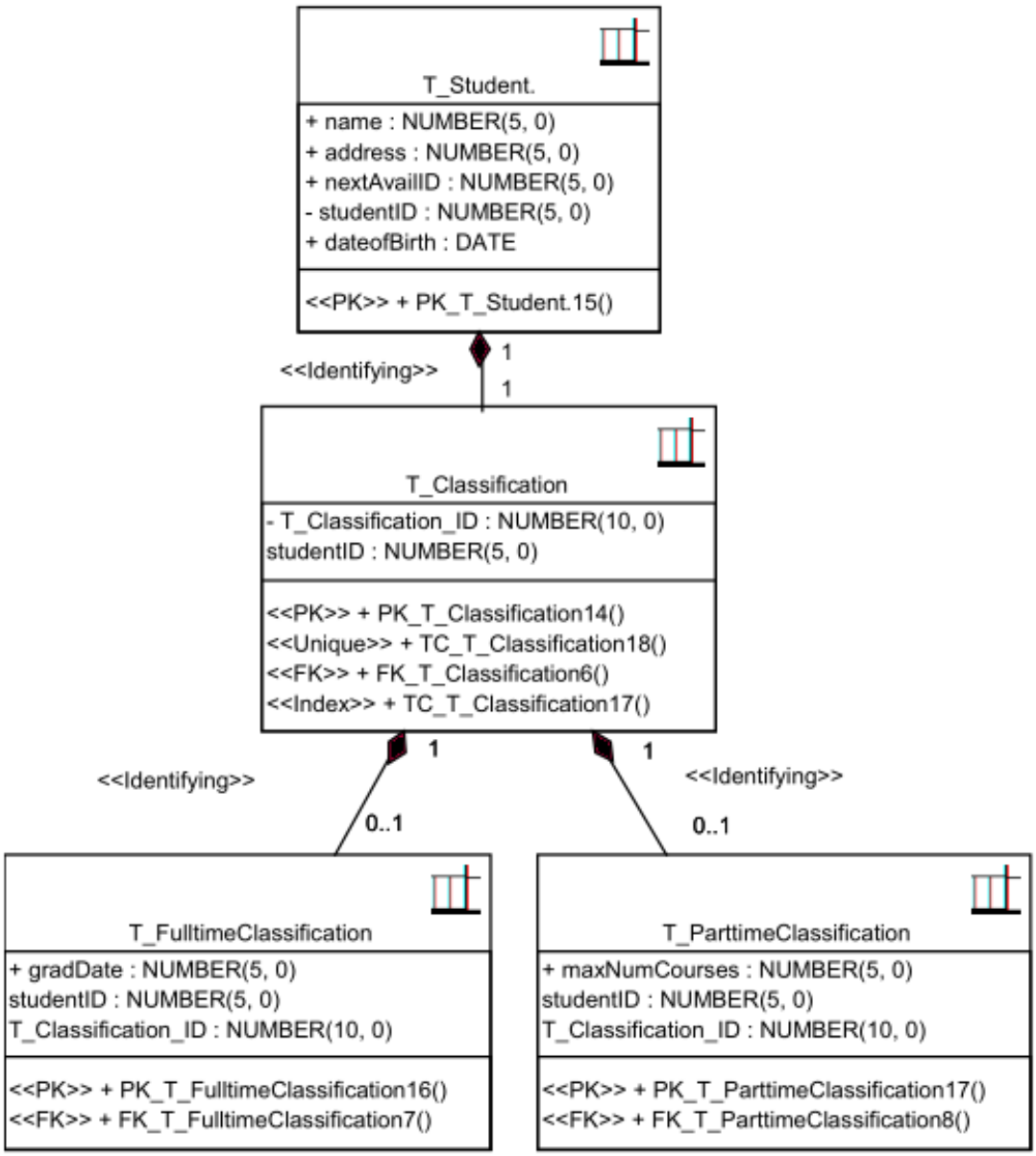


Рисунок 19.33 – Діаграма «сутність-зв’язок»

19.7. Реалізація системи

19.7.1. Створення компонентів

У Rational Rose діаграми компонентів створюються в уявленні компонентів системи. Окремі компоненти можна створювати безпосередньо на діаграмі, або перетягувати їх туди з браузеру.

Вправа 15. Створення компонентів

Виберемо в якості мови програмування C++ і для класу Student створимо ті компоненти, що відповідають цій мові.

Створення діаграми компонентів:

1. Двічі клацніть мишею на головній діаграмі компонентів в уявленні компонентів.

2. На панелі інструментів натисніть кнопку Package Specification.
3. Помістіть специфікацію пакету на діаграму.
4. Введіть ім’я специфікації пакету Student і вкажіть у вікні специфікації мову C++.
5. На панелі інструментів натисніть кнопку Package Body.
6. Помістіть тіло пакету на діаграму.
7. Введіть ім’я тіла пакету Student і вкажіть у вікні специфікації мову C++.
8. На панелі інструментів натисніть кнопку Dependency.
9. Проведіть лінію залежності від тіла пакету до специфікації пакету.

Результат показаний на рис. 19.34.

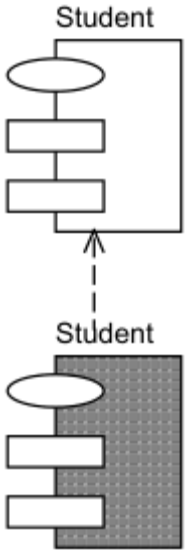


Рисунок 19.34 – Діаграма компонентів

Співвідношення класів з компонентами:
1. У логічному уявленні браузеру знайдіть клас Student.
2. Перетягніть цей клас на специфікацію пакету компонента Student в уявленні компонентів браузеру. В результаті клас Student буде співвіднесений із специфікацією пакету компонента Student.
3. Перетягніть клас Student на тіло пакету компонента Student в уявленні компонентів браузеру. В результаті клас Student буде співвіднесений з тілом пакету компонента Student.

19.7.2. Генерація коду

Процес генерації коду складається з чотирьох основних кроків:
1. Перевірка коректності моделі.
2. Установка властивостей генерації коду.
3. Вибір класу, компонента або пакету.
4. Генерація коду.

Для перевірки моделі:
1. Виберіть в меню Tools > Check Model.
2. Проаналізуйте усі знайдені помилки у вікні журналу.

До найпоширеніших помилок належать такі, наприклад, як повідомлення на діаграмі послідовності або кооперативній діаграмі, не співвіднесені з операцією, або об’єкти цих діаграм, не співвіднесені з класом.

За допомогою пункту меню Check Model можна виявити велику частину неточностей і помилок в моделі. Пункт меню Access Violations дозволяє виявляти порушення правил доступу, що виникають тоді, коли існує зв’язок між двома класами різних пакетів, але зв’язку між самими пакетами немає.

Щоб виявити порушення правил доступу:

1. Виберіть в меню Report > Show Access Violations.
2. Проаналізуйте усі порушення правил доступу у вікні.

Можна встановити декілька параметрів генерації коду для класів, атрибутів, компонентів і інших елементів моделі. Цими властивостями визначається спосіб генерації програм. Для кожної мови в Rose передбачений ряд певних властивостей генерації коду. Перед генерацією коду рекомендується аналізувати ці властивості і вносити необхідні зміни.

Для аналізу властивостей генерації коду виберіть Tools > Options, а потім вкладку відповідної мови. У вікні списку можна вибрати клас, атрибут, операцію і інші елементи моделі. Для кожної мови в цьому списку вказані свої власні елементи моделі. При виборі різних значень на екрані з’являються різні набори властивостей.

Будь-які зміни, що вносяться в набір властивостей у вікні Tools > Options, впливають на усі елементи моделі, для яких використовується цей набір.

Іноді треба змінити властивості генерації коду для одного класу, атрибуту, однієї операції і так далі. Для цього відрийте вікно специфікації елементу моделі. Виберіть вкладку мови (C++, Java, .) і змініть властивості тут. Усі зміни, що вносяться у вікні специфікації елементу моделі, роблять вплив тільки на цей елемент.

При генерації коду за один раз можна створити клас, компонент або цілий пакет. Код генерується за допомогою діаграми або браузера. При генерації коду з пакету можна вибрати або пакет логічного представлення на діаграмі класів, або пакет представлення компонентів на діаграмі компонентів. При виборі пакету логічного представлення генеруються усі класи цього пакету. При виборі пакету представлення компонентів генеруються усі компоненти цього пакету.

Після вибору класу або компонента на діаграмі виберіть в меню відповідний варіант генерації коду. Повідомлення про помилки, що виникають в процесі генерації коду, з’являтимуться у вікні журналу.

Під час генерації коду Rose вибирає інформацію з логічного і компонентного представлень моделі і генерує великий об’єм «скелетного» (skeletal) коду:

1. **Класи**. Генеруються усі класи моделі.
2. **Атрибути**. Код включає атрибути кожного класу, у тому числі видимість, тип даних і значення за умовчанням.

3. **Сигнатури операцій**. Код містить визначення операцій з усіма параметрами, типами цих параметрів і типом значення операції, що повертається.
4. **Зв'язки**. Деякі із зв'язків моделі викликають створення атрибутів при генерації коду.
5. **Компоненти**. Кожен компонент реалізується у вигляді відповідного файлу з початковим кодом.

Вправа 16. Генерація коду C++

1. Відкрийте діаграму компонентів системи.
2. Виберіть усі об'єкти на діаграмі компонентів.
3. Виберіть Tools > C++> Code Generation в меню.
4. Виконайте генерацію коду.
5. Прогляньте результати генерації (меню Tools > C++> Browse Header і Tools > C++> Browse Body).

РОЗДІЛ 20. ВАРІАНТИ ЗАВДАНЬ ДЛЯ САМОСТІЙНОЇ РОБОТИ

20.1. Постановка завдання на проектування ІС

У рамках лабораторних робіт студенти зможуть на практиці освоїти сучасні комп'ютерні технології, методи і засоби проектування і розробки порівняно простих, але дієвих інформаційних систем, набути досвіду в документуванні і випробуваннях програмних продуктів, познайомитися з сучасними міжнародними стандартами в програмуванні. Усе це внесе свій вклад у формування із студентів кваліфікованих фахівців в області інформаційних систем і прикладної інформатики.

Завданнями студента в ході лабораторних робіт є проектування, розробка і відладка програмного забезпечення автоматизованої інформаційної системи відповідно до завдання, створення комплекту проектної і експлуатаційної документації з модулями програм на машинних носіях.

У кожному із запропонованих варіантів вимагається за допомогою CASE-засобів (наприклад, Rational Rose або StarUML) побудувати модель програмного забезпечення. Процес створення моделі повинен проходити так, як це описано у розділі «Основні відомості про CASE-засіб Rational Rose».

Мають бути виконані такні дії:

1) складання глосарію проекту;
2) створення моделі варіантів використання;
3) аналіз варіантів використання;
4) проектування системи;
5) реалізація системи.

Після виконання третього етапу модель повинна задовольняти перерахованим нижче вимогам.

Глосарій проекту повинен повінен містити діаграму класів та мати вигляд таблиці і зберігатися в окремому файлі. На діаграмах варіантів використання кожна дійова особа (actor) і варіант використання повинні супроводжуватися описом. Ці описи мають бути складені українською мовою. Опис дійової особи повинен коротко (в один-два рядки) повідомляти про роль цієї особи.

Опис варіанту використання повинен включати пояснення, передумову, потоки подій (основний і альтернативні, якщо такі є) і постумову. Описи є або приєднаними текстовими файлами, або текстом, введеним в поле Documentation специфікації відповідного елементу діаграми. Діаграми взаємодії, що відповідають потокам подій варіантів використання, повинні містити необхідні пояснення.

При проектуванні системи вимагається:

- створити ієрархію класів системи;
- розмістити класи по пакетах (використати ділення: призначений для користувача інтерфейс – управління – дані; чи інше залежно від постановки завдання);

- зв’язати об’єкти з класами, повідомлення на діаграмах взаємодії – з операціями;
- кожен клас забезпечити описом, який повинен включати короткий опис (відповідальність класу), опис атрибутів у вигляді таблиці (ім’я, опис, тип), таблицю з описом операцій (ім’я, опис, сигнатура);
- для класів вказати стереотипи;
- побудувати діаграми класів системи, що відображають зв’язки між класами;
- для опису поведінки екземплярів окремих класів побудувати діаграми станів;
- розробити (якщо це передбачено варіантом завдання) схему бази даних і відобразити її на діаграмі «сутність – зв’язок».

При реалізації системи необхідно побудувати діаграми компонентів для кожного пакету і для системи в цілому. Також слід розробити діаграму розміщення. Залежно від варіанту завдання діаграма розміщення повинна показувати розташування компонентів в розподіленому застосуванні або зв’язки між вбудованим процесором і пристроями.

20.2. Варіанти завдань на проектування ІС

1. Страхова компанія

Опис предметної області.

Ви працюєте в страховій компанії. Вашим завданням є відстежування фінансової діяльності компанії.

Компанія має різні філії по всій країні. Кожна філія характеризується назвою, адресою і телефоном. Діяльність компанії організована таким чином: до Вас звертаються різні особи з метою укладення договору про страхування. Залежно від об’єктів, що приймаються на страхування, і страхованих ризиків, договір укладається по певному виду страхування (наприклад, страхування автотранспорту від викрадення, страхування домашнього майна, добровільне медичне страхування). При укладенні договору Ви фіксуєте дату укладення, страхову суму, вид страхування, тарифну ставку і філію, в якій укладено договір.

Класи об’єктів.

Договори (Номер договору, Дата укладення, Страхова сума, Тарифна ставка, Філія, Вид страхування).

Вид страхування (Вид страхування, Найменування).

Філія (Філія, Найменування філії, Адреса, Телефон).

2. Готель

Опис предметної області.

Ви працюєте в готелі. Вашим завданням є відстежування фінансової сторони роботи готелю.

Ваша діяльність організована таким чином: готель надає номери клієнтам на певний термін. Кожен номер характеризується місткістю, комфортністю

(люкс, напівлюкс, звичайний) і ціною. Вашими клієнтами є різні особи, про яких Ви збираєте певну інформацію (прізвище, ім'я, по батькові і деякий коментар). Здача номера клієнтові робиться за наявності вільних місць в номерах, відповідних клієнтові за вказаними вище параметрами. При поселенні фіксується дата поселення. При виїзді з готелю для кожного місця запам'ятовується дата звільнення.

Класи об'єктів.

Клієнти (Клієнт, Прізвище, Ім'я, По батькові, Паспортні дані, Коментар).

Номери (Номер, Кількість людина, Комфортність, Ціна).

Поселення (Клієнт, Номер, Дата поселення, Дата звільнення, Примітка).

3. Ломбард

Опис предметної області.

Ви працюєте в ломбарді. Вашим завданням є відстежування фінансової сторони роботи ломбарду.

Діяльність Вашої компанії організована таким чином: до Вас звертаються різні особи з метою отримання грошових коштів під заставу певних товарів. У кожного з клієнтів, що приходять до Вас, Ви просите прізвище, ім'я, по батькові і інші паспортні дані.

Після оцінювання вартості принесеного в якості застави товару Ви визначаєте суму, яку готові видати на руки клієнтові, а також свої комісійні. Крім того, визначаєте термін повернення грошей. Якщо клієнт згоден, то Ваші домовленості фіксуються у вигляді документу, гроші видаються клієнтові, а товар залишається у Вас. У випадку якщо в зазначений термін не відбувається повернення грошей, товар переходить у Вашу власність.

Класи об'єктів.

Клієнти (Клієнт, Прізвище, Ім'я, По батькові, Номер паспорта, Серія паспорта, Дата видачі паспорта).

Категорії товарів (Категорія товарів, Назва, Примітка).

Здача в ломбард (Категорія товарів, Клієнт, Опис товару, Дата здачі, Дата повернення, Сума, Комісійні).

4. Реалізація готової продукції

Опис предметної області.

Ви працюєте в компанії, що займається оптово-роздрібним продажем різних товарів. Вашим завданням є відстежування фінансової сторони роботи компанії.

Діяльність Вашої компанії організована таким чином: Ваша компанія торгує товарами з певного спектру. Кожен з цих товарів характеризується найменуванням, гуртовою ціною, роздрібною ціною і довідковою інформацією. У Вашу компанію звертаються покупці.

Для кожного з них Ви запам'ятовуєте у базі даних стандартні дані (найменування, адреса, телефон, контактна особа) і складаєте по кожній угоді документ, запам'ятовуючи разом з покупцем кількість купленого ним товару і дату купівлі.

Класи об'єктів.

Товари (Найменування, Гуртова ціна, Роздрібна ціна, Опис).

Покупці (Телефон, Контактна особа, Адреса).

Угоди (Дата угоди, Товар, Кількість, Покупець, Ознака оптового продажу).

5. Ведення замовлень

Опис предметної області.

Ви працюєте в компанії, що займається оптовим продажем різних товарів. Вашим завданням є відстежування фінансової сторони роботи компанії.

Діяльність Вашої компанії організована таким чином: Ваша компанія торгує товарами з певного спектру. Кожен з цих товарів характеризується ціною, довідковою інформацією і ознакою наявності або відсутності доставки. У Вашу компанію звертаються замовники.

Для кожного з них Ви запам'ятовуєте у базі даних стандартні дані (найменування, адреса, телефон, контактна особа) і складаєте по кожній угоді документ, запам'ятовуючи разом із замовником кількість купленого ним товару і дату купівлі.

Класи об'єктів.

Замовники (Найменування, Адреса, Телефон, Контактна особа).

Товари (Ціна, Доставка, Опис).

Замовлення (Замовник, Товар, Кількість, Дата).

6. Бюро з працевлаштування

Опис предметної області.

Ви працюєте у бюро з працевлаштування. Вашим завданням є відстежування фінансової сторони роботи компанії.

Діяльність Вашого бюро організована таким чином: Ваше бюро готове шукати працівників для різних працедавців і вакансії для тих фахівців різного профілю, що шукають роботу.

При зверненні до Вас клієнта-працедавця, його стандартні дані (назва, вид діяльності, адреса, телефон) фіксуються у базі даних. При зверненні до Вас клієнта-претендента, його стандартні дані (прізвище, ім'я, по батькові, кваліфікація, професія, інші дані) також фіксуються у базі даних. По кожному факту задоволення інтересів обох сторін складається документ. У документі вказуються претендент, працедавець, посада і комісійні (доход бюро).

Класи об'єктів.

Працедавці (Назва, Вид діяльності, Адреса, Телефон).

Угоди (Працедавець, Посада, Комісійні).

Претенденти (Прізвище, Ім'я, По батькові, Кваліфікація, Вид діяльності, Інші дані, Передбачуваний розмір заробітної плати).

7. Нотаріальна контора

Опис предметної області.

Ви працюєте в нотаріальній конторі. Вашим завданням є відстежування фінансової сторони роботи компанії.

Діяльність Вашої нотаріальної контори організована таким чином: Ваша фірма готова надати клієнтові певний комплекс послуг. Для наведення ладу Ви формалізували ці послуги, склавши їх список з описом кожної послуги. При зверненні до Вас клієнта, його стандартні дані (назва, вид діяльності, адреса, телефон) фіксуються у базі даних. По кожному факту надання послуги клієнтові складається документ. У документі вказуються послуга, сума угоди, комісійні (доход контори), опис угоди.

Класи об'єктів.

Клієнти (Назва, Вид діяльності, Адреса, Телефон).

Угоди (Клієнт, Послуга, Сума, Комісійні, Опис).

Послуги (Назва, Опис).

8. Фірма з продажу запчастин

Опис предметної області.

Ви працюєте у фірмі, що займається продажем запасних частин для автомобілів. Вашим завданням є відстежування фінансової сторони роботи компанії.

Основна частина діяльності, що знаходиться у Вашому веденні, пов'язана з роботою з постачальниками. Фірма має певний набір постачальників, по кожному з яких відомі назва, адреса і телефон. У цих постачальників Ви придбаєте деталі. Кожна деталь разом з назвою характеризується артикулом і ціною (вважаємо ціну постійною). Деякі з постачальників можуть поставляти однакові деталі (один і той же артикул). Кожен факт купівлі запчастин у постачальника фіксується у базі даних, причому обов'язковими для запам'ятовування є дата купівлі і кількість придбаних деталей.

Класи об'єктів.

Постачальники (Постачальник, Назва, Адреса, Телефон).

Деталі (Назва, Артикул, Ціна, Примітка).

Постачання (Постачальник, Деталь, Кількість, Дата).

9. Курси по підвищенню кваліфікації

Опис предметної області.

Ви працюєте в навчальному закладі і займаєтеся організацією курсів підвищення кваліфікації.

У Вашому розпорядженні є відомості про сформовані групи студентів. Групи формуються залежно від спеціальності і відділення. У кожній з них включена певна кількість студентів. Проведення занять забезпечує штат викладачів. Для кожного з них у Вас у базі даних зареєстровані стандартні анкетні дані (прізвище, ім'я, по батькові, телефон) і стаж роботи. В результаті розподілу навантаження Ви отримуєте інформацію про те, скільки годин занять проводить кожен викладач з відповідними групами. Крім того, зберігаються також зведення про вид занять (лекції, практика), що проводяться, предмет і плату за 1 годину.

Класи об'єктів.

Групи (Спеціальність, Відділення, Кількість студентів).

Викладачі (Прізвище, Ім'я, По батькові, Телефон, Стаж).

Навантаження (Викладач, Група, Кількість годин, Предмет, Тип заняття, Оплата).

10. Визначення факультативів для студентів

Опис предметної області.

Ви працюєте у вищому навчальному закладі і займаєтеся організацією факультативів.

У Вашому розпорядженні є відомості про студентів, що включають стандартні анкетні дані (прізвище, ім'я, по батькові, адреса, телефон). Викладачі Вашої кафедри повинні забезпечити проведення факультативних занять по деяких предметах. По кожному факультативу існує певна кількість годинника і вид занять (лекції, практика, лабораторні роботи), що проводяться. У результаті роботи із студентами у Вас з'являється інформація про те, хто з них записався на які факультативи. Існує деякий мінімальний об'єм факультативних предметів, які повинен прослухати кожен студент. По закінченню семестру Ви заносите інформацію про оцінки, отримані студентами на іспитах.

Класи об'єктів.

Студенти (Прізвище, Ім'я, По батькові, Адреса, Телефон).

Предмети (Назва, Об'єм лекцій, Об'єм практик, Об'єм лабораторних робіт).

Навчальний план (Студент, Предмет, Оцінка).

11. Розподіл навчального навантаження

Опис предметної області.

Ви працюєте у вищому навчальному закладі і займаєтеся розподілом навантаження між викладачами кафедри.

У Вашому розпорядженні є відомості про викладачів кафедри, що включають разом з анкетними даними відомостія про їх вчений ступінь, займану адміністративну посаду і стаж роботи. Викладачі Вашої кафедри повинні забезпечити проведення занять по деяких предметах. По кожному з них існує певна кількість годин. В результаті розподілу навантаження у Вас повинна вийти інформація такого роду: «Такий-то викладач проводить заняття по такому-то предмету з такою-то групою».

Класи об'єктів.

Викладачі (Прізвище, Ім'я, По батькові, Вчений ступінь, Посада, Стаж).

Предмети (Назва, Кількість годин).

Навантаження (Викладач, Предмет, Номер групи).

12. Розподіл додаткових обов'язків

Опис предметної області.

Ви працюєте в комерційній компанії і займаєтеся розподілом додаткових разових робіт. Вашим завданням є відстежування ходу виконання додаткових робіт.

Компанія має певний штат співробітників, кожен з яких отримує певний оклад. Час від часу, виникає потреба у виконанні деякої додаткової роботи, що не входить в коло основних посадових обов'язків співробітників. Для наведення ладу в цій сфері діяльності Ви проклассифицировали усі види додаткових робіт, визначившись з сумою оплати за фактом їх виконання. При виникненні додаткової роботи певного виду Ви призначаєте відповідального, фіксуючи дату початку роботи. За фактом закінчення роботи Ви фіксуєте дату і виплачуєте додаткову суму до зарплати з урахуванням Вашої класифікації.

Класи об'єктів.

Співробітники (Прізвище, Ім'я, По батькові, Оклад).

Види робіт (Опис, Плата за день).

Роботи (Співробітник, Вид робіт, Дата початку, Дата закінчення).

13. Технічне обслуговування верстатів

Опис предметної області.

Ваше підприємство займається ремонтом верстатів і іншого промислового устаткування. Вашим завданням є відстежування фінансової сторони діяльності підприємства.

Клієнтами Вашої компанії є промислові підприємства, оснащені різним складним устаткуванням. У разі поломок устаткування вони звертаються до Вас.

Ремонтні роботи у Вашій компанії організовані таким чином: усі верстати прокласифіковані за країнами-виробниками, роками випуску і марками. Усі види ремонту відрізняються назвою, тривалістю в днях, вартістю. Виходячи з цих даних, по кожному факту ремонту Ви фіксуєте вид верстата і дату початку ремонту.

Класи об'єктів.

Види верстатів (Країна, Рік випуску, Марка).

Види ремонту (Назва, Тривалість, Вартість, Примітки).

Ремонт (Вид верстата, Ремонт, Дата початку, Примітки).

14. Туристична фірма

Опис предметної області.

Ви працюєте в туристичній компанії. Ваша компанія працює з клієнтами, продаючи їм путівки. Вашим завданням є відстежування фінансової сторони діяльності фірми.

Робота з клієнтами у Вашій компанії організована таким чином: у кожного клієнта, що прийшов до Вас, збираються деякі стандартні дані – прізвище, ім'я, по батькові, адреса, телефон. Після цього Ваші співробітники з'ясовують у клієнта, куди він хотів би поїхати відпочивати. При цьому йому

демонструються різні варіанти, що включають країну проживання, особливості місцевого клімату, наявні готелі різного класу. Разом з цим, обговорюється можлива тривалість перебування і вартість путівки. У випадку якщо вдалося домовитися, і знайти для клієнта прийнятний варіант, Ви реєструєте факт продажу путівки (чи путівок, якщо клієнт купує відразу декілька путівок), фіксуючи дату відправлення. Іноді Ви вирішуєте надати клієнтові деяку знижку.

Класи об'єктів.

Маршрути (Країна, Клімат, Тривалість, Готель, Вартість).

Путівки (Маршрут, Клієнт, Дата відправлення, Кількість, Знижка).

Клієнти (Прізвище, Ім'я, По батькові, Адреса, Телефон).

15. Вантажні перевезення

Опис предметної області.

Ви працюєте в компанії, що займається перевезеннями вантажів. Вашим завданням є відстежування вартості перевезень з урахуванням заробітної плати водіїв.

Ваша компанія здійснює перевезення різними маршрутамих. Для кожного маршруту Ви визначили деяку назву, обчислили зразкову відстань і встановили деяку оплату для водія. Інформація про водіїв включає прізвище, ім'я, по батькові і стаж. Для проведення розрахунків Ви зберігаєте повну інформацію про перевезення (маршрут, водій, дати відправки і прибуття). За фактом деяких перевезень водіям виплачується премія.

Класи об'єктів.

Маршрути (Назва, Дальність, Кількість днів в дорозі, Оплата).

Водії (Прізвище, Ім'я, По батькові, Стаж).

Виконана робота (Маршрут, Водій, Дата відправки, Дата повернення, Премія).

16. Облік телефонних переговорів

Опис предметної області.

Ви працюєте в комерційній службі телефонної компанії. Компанія надає абонентам телефонні лінії для міжміських переговорів. Вашим завданням є відстежування вартості міжміських телефонних переговорів.

Абонентами компанії є юридичні особи, що мають телефонну точку, ідентифікаційний номер платника податків **(ІПН)**, розрахунковий рахунок у банку. Вартість переговорів залежить від міста, в яке здійснюється дзвінок, і часу доби (день, ніч). Кожен дзвінок абонента автоматично фіксується у базі даних. При цьому запам'ятовуються місто, дата, тривалість розмови і час доби.

Класи об'єктів.

Абоненти (Номер телефону, ИНН, Адреса).

Міста (Назва, Тариф денний, Тариф нічний).

Переговори (Абонент, Місто, Дата, Кількість хвилин, Час доби).

17. Облік внутрішньоофісних витрат

Опис предметної області.

Ви працюєте у бухгалтерії приватної фірми. Співробітники фірми мають можливість здійснювати дрібні покупки для потреб фірми, надаючи у бухгалтерію товарний чек. Вашим завданням є відстежування внутрішньоофісних витрат.

Ваша фірма складається з відділів. Кожен відділ має назву. У кожному відділі працює певна кількість співробітників. Співробітники можуть здійснювати покупки відповідно до видів витрат. Кожен вид витрат має назву, деякий опис і граничну суму коштів, які можуть бути витрачені по цьому виду витрат в місяць. При кожній купівлі співробітник оформляє документ, де вказує вид витрати, дату, суму і відділ.

Класи об'єктів.

Відділи (Назва, Кількість співробітників).

Види витрат (Назва, Опис, Гранична норма).

Витрати (Вид витрат, Відділ, Сума, Дата).

18. Бібліотека

Опис предметної області.

Ви є керівником бібліотеки. Ваша бібліотека вирішила заробляти гроші, видаючи напрокат деякі книги, наявні в невеликій кількості екземплярів. Вашим завданням є відстежування фінансових показників роботи бібліотеки.

У кожної книги, що видається в прокат, є назва, автор, жанр. Залежно від цінності книги Ви визначили для кожної з них заставну вартість (сума, що вноситься клієнтом при узятті книги напрокат) і вартість прокату (сума, яку клієнт платить при поверненні книги, отримуючи назад запоруку). У бібліотеку звертаються читачі. Усі читачі реєструються в картотеці, яка містить стандартні анкетні дані (прізвище, ім'я, по батькові, адреса, телефон). Кожен читач може звертатися у бібліотеку кілька разів. Усі звернення читачів фіксуються, при цьому по кожному факту видачі книги запам'ятовуються дата видачі і очікувана дата повернення.

Класи об'єктів.

Книги (Назва, Автор, Заставна вартість, Вартість прокату, Жанр).

Читачі (Прізвище, Ім'я, По батькові, Адреса, Телефон).

Видані книги (Книга, Читач, Дата видачі, Дата повернення).

19. Прокат автомобілів

Опис предметної області.

Ви є керівником комерційної служби у фірмі, що займається прокатом автомобілів. Вашим завданням є відстежування фінансових показників роботи пункту прокату.

У Ваш автопарк входить деяка кількість автомобілів різних марок, вартостей і типів. Кожен автомобіль має свою вартість прокату. У пункт прокату звертаються клієнти. Усі клієнти проходять обов'язкову реєстрацію, при якій про них збирається стандартна інформація (прізвище, ім'я, по батькові,

адреса, телефон). Кожен клієнт може звертатися в пункт прокату кілька разів. Усі звернення клієнтів фіксуються, при цьому по кожній угоді запам'ятовуються дата видачі і очікувана дата повернення.

Класи об'єктів.

Автомобілі (Марка, Вартість, Вартість прокату, Тип).

Клієнти (Прізвище, Ім'я, По батькові, Адреса, Телефон).

Видані автомобілі (Автомобіль, Клієнт, Дата видачі, Дата повернення).

20. Видача банком кредитів

Опис предметної області.

Ви є керівником інформаційно-аналітичного центру комерційного банку. Одним з істотних видів діяльності Вашого банку є видача кредитів юридичним особам. Вашим завданням є відстежування динаміки роботи кредитного відділу.

Залежно від умов отримання кредиту, процентної ставки і терміну повернення усі кредитні операції діляться на декілька основних видів. Кожен з цих видів має свою назву. Кредит може отримати юридична особа (клієнт), що при реєстрації надав такі відомості: назва, вид власності, адреса, телефон, контактна особа. Кожен факт видачі кредиту реєструється банком, при цьому фіксуються сума кредиту, клієнт і дата видачі.

Класи об'єктів.

Види кредитів (Назва, Умови отримання, Ставу, Термін).

Клієнти (Назва, Вид власності, Адреса, Телефон, Контактна особа).

Кредити (Вид кредитів, Клієнт, Сума, Дата видачі).

21. Платна поліклініка

Опис предметної області.

Ви є керівником служби планування платної поліклініки. Вашим завданням є відстежування фінансових показників роботи поліклініки.

У поліклініці працюють лікарі різних спеціальностей, що мають різну кваліфікацію. Щодня в поліклініку звертаються хворі. Усі хворі проходять обов'язкову реєстрацію, при якій у базу даних заносяться стандартні анкетні дані (прізвище, ім'я, по батькові, рік народження). Кожен хворий може звертатися в поліклініку кілька разів, потребуючи різної медичної допомоги. Усі звернення хворих фіксуються, при цьому встановлюється діагноз, визначається вартість лікування, запам'ятовується дата звернення.

Класи об'єктів.

Лікарі (Прізвище, Ім'я, По батькові, Спеціальність, Категорія).

Пацієнти (Прізвище, Ім'я, По батькові, Рік народження).

Звернення (Лікар, Пацієнт, Дата звернення, Діагноз, Вартість лікування).

22. Інтернет-магазин

Опис предметної області.

Ви є співробітником комерційного відділу компанії, що продає різні товари через інтернет. Вашим завданням є відстежування фінансової складової роботи компанії.

Робота Вашої компанії організована таким чином: на Інтернет-сайті компанії представлені (виставлені на продаж) деякі товари. Кожен з них має деяку назву, ціну і одиницю виміру (штуки, кілограми, літри). Для проведення досліджень і оптимізації роботи магазину Ви намагаєтеся збирати дані з Ваших клієнтів. При цьому для Вас визначальне значення мають стандартні анкетні дані, а також телефон і адреса електронної пошти для зв'язку. У разі придбання товарів на суму понад 1000 грн. клієнт переходить в категорію «постійних клієнтів» і отримує знижку на кожну купівлю у розмірі 2%. По кожному факту продажу Ви автоматично фіксуєте клієнта, товари, кількість, дату продажу, дату доставки.

Класи об'єктів.

Товари (Назва, Ціна, Одиниця виміру).

Клієнти (Прізвище, Ім'я, По батькові, Адреса, Телефон, email, Ознака постійного клієнта).

Продажі (Товар, Клієнт, Дата продажу, Дата доставки, Кількість).

23. Хімчистка

Опис предметної області.

Ви працюєте в хімчистці. Ваша хімчистка здійснює прийом у населення речей для виведення плям. Для наведення ладу Ви, в міру можливості, складаєте базу цих клієнтів, запам'ятовуючи їх анкетні дані (прізвище, ім'я, по батькові). Починаючи з 3-го звернення, клієнт переходить в категорію постійних клієнтів і отримує знижку в 3% при чищенні кожної подальшої речі. Усі послуги, що робляться Вами, підрозділяються на види, що мають назву, тип і вартість, залежну від складності робіт. Робота з клієнтом спочатку полягає у визначенні об'єму робіт, виду послуги і, відповідно, її вартості. Якщо клієнт згоден, він залишає річ (при цьому фіксується послуга, клієнт і дата прийому) і забирає її після обробки (при цьому фіксується дата повернення).

Класи об'єктів.

Види послуг (Назва, Тип, Вартість).

Клієнти (Прізвище, Ім'я, По батькові, Ознака постійного клієнта).

Послуги (Вид послуги, Клієнт, Дата прийому, Дата повернення).

СПИСОК ЛІТЕРАТУРИ

1. Грекул В.И., Денищенко Г.Н., Коровкина Н.Л. Проектирование информационных систем. Интернет-университет информационных технологий. – ИНТУИТ.ру, 2005. – www.intuit.ru.
2. Вендров А.М. Case-технологии. Современные методы и средства проектирования информационных систем. – М.: Финансы и статистика, 1998. – 176 с.
3. Вендров А.М. Проектирование программного обеспечения экономических информационных систем: Учебник. – 2-е изд. – М.: Финансы и статистика, 2006. – 544 с.
4. Якобсон А., Буч Г., Рамбо Дж.. Унифицированный процесс разработки программного обеспечения. – СПб.: Питер, 2002. – 496 с.
5. Бек К. Экстремальное программирование. – СПб.: Питер, 2002. – 224 с.
6. Алистэр Коуберн, Люди как нелинейные и наиболее важные компоненты в создании программного обеспечения, Humans and Technology, Октябрь, 1999.
7. Брукс Фредерик. Мифический человеко-месяц, или Как создаются программные комплексы. – СПб.: Символ-Плюс, 2006. – 304 с.
8. Друкер П. Задачи менеджмента в XXI веке. – М.: Вильямс, 2002. – 272 с.
9. Белбин Р.М. Типы ролей в командах менеджеров. – М.: HIPPO, 2003. – 232 с.
10. Орлов С.А. Технологии разработки программного обеспечения: Учебник для вузов. 3-е издание. – СПб.: Питер, 2004. – 528 с.
11. Гради Буч, Джеймс Рамбо, Ивар Якобсон Введение в UML от создателей языка. Пер: Н. Мухин – М.: ДМК Пресс, 2011г. – 496с.
12. Новиков Ф.А. Анализ и проектирование на UML. – СПб.: Спб. госуниверситет информационных технологий, 2007. – 286 с.
13. Соммервилл И. Инженерия программного обеспечения. – М.: Вильямс, 2002. – 624 с.
14. Грекул В.И., Денищенко Г.Н., Коровкина Н.Л. Проектирование информационных систем. Интернет-университет информационных технологий. – ИНТУИТ.ру, 2005. – www.intuit.ru.
15. Кен Ауер, Рой Миллер. Экстремальное программирование: постановка процесса. С первых шагов и до победного конца. – Спб.: Питер, 2004. –368 с.
16. Буч Г. Объектно-ориентированный анализ и проектирование с примерами приложений на C++. 2-е изд. / Пер. с англ. – М.: «Издательство Бином», СПб.: «Невский диалект», 2000. – 560 с.
17. Якобсон А, Буч Г., Рамбо Дж. Унифицированный процесс разработки программного обеспечения. – Спб.: Питер, 2002. – 496 с.

18. Ларман Крэг. Применение UML 2.0 и шаблонов проектирования. Введение в объектно-ориентированный анализ, проектирование и итеративную разработку. 3-е издание. – М.: Вильямс, 2013. – 737 с.
19. Фаулер М. UML. Основы. Краткое руководство по стандартному языку объектного моделирования / Пер. с англ. – СПб.: Символ-Плюс, 2011. – 192 с.
20. Литвинов В.В., Голуб С.В. Об'єктно-орієнтоване моделювання при проектуванні вбудованих систем і систем реального часу. – Черкаси: ЧНУ ім. Богдана Хмельницького, 2011. – 376 с.
21. Вигерс К.И. Разработка требований к программному обеспечению / Пер. с англ. – М.: Издат.-торговый дом «Русская редакция», 2004. – 576 с.
22. Иванов Д.Ю., Новиков Ф.А. Моделирование на UML. – СПб.: Наука и техника, 2010. – 640 с.
23. Ларман К. Применение UML 2.0 и шаблонов проектирования. Введение в объектно-ориентированный анализ, проектирование и итеративную разработку / Пер. с англ. – 3-е изд. – М.: ИД «Вильямс», 2013. – 736 с.
24. Мацяшек, Лешек А. Анализ и проектирование информационных систем с помощью UML 2.0, 3-е изд. – М.: ООО «И.Д. Вильямс», 2008. – 816 с.
25. Маклафлин Б., Поллайс Г., Уэст Д. Объектно-ориентированный анализ и проектирование. – СПб.: Питер, 2013. – 608 с.
26. Леоненков А.В. Самоучитель UML 2. – СПб.: БХВ-Петербург, 2007. – 576 с.
27. Р. Фатрелл, Д. Шафер, Л. Шафер. Управление программными проектами: достижение оптимального качества при минимуме затрат. – М.: Вильямс, 2003. – 1136 с.
28. Ройс У. Управление проектами по созданию программного обеспечения. – М.: Лори, 2002. – 424 с.
29. Орлов С.А., Цилькер Б.Я. Технологии разработки программного обеспечения: Учебник для вузов. 4-е изд. Стандарт третьего поколения. – СПб.: Питер, 2012. – 608 с.
30. Розенберг Д., Кендалл С. Применение объектного моделирования с использованием UML и анализ прецедентов. – М.: ДМК Пресс, 2002. –160 с.
31. Мюллер Р. Д. Базы данных и UML: Проектирование. – М.: Лори, 2002. – 420 с.
32. Нейбург Э. Д., Максимчук Р. А. Проектирование баз данных с помощью UML. – М.: Вильямс, 2002. – 288 с.
33. Макконнелл С., Профессиональная разработка программного обеспечения, – М.: Символ-Плюс, 2007. –240 с.
34. Мартин Р. Быстрая разработка программ. Принципы, примеры, практика. М.: Вильямс, 2004. – 746 с.
35. Книберг Х., Скарин М. Scrum и XP: заметки с передовой. Интернет-ресурс http://scrum.org.ua/wp-content/uploads/2008/12/scrum_xp-from-the-trenches-rus-final.pdf (05.02.15, на русском языке).
36. Фаулер М. UML. Основы, 3-е изд. – СПб: Символ-Плюс, 2004. – 192 с.

37. Ambler. S. W. Agile Database Techniques: Effective Strategies for the Agile Software Develo per.John Wiley & Sons, 2003. – 416 pp.
38. Рудаков А.В., Федорова Г.Н. Технология разработки программных продуктов. Практикум. 4-е изд. – М.: ИЦ «Академия», 2014. – 192 с.
39. Пущин М.Н. Проектирование информационных систем: Учебное пособие. – М.: Изд-во МИЭТ, 2008. – 234 с.
40. Рамбо Дж., Блаха М. UML 2.0. Объектно-ориентированное моделирование и разработка. 2-е изд. – СПб.: Питер, 2007. – 544 с.
41. Клевцов С.И. Анализ и формирование требований к ПО информационных систем сбора и обработки данных: Учебное пособие. – Таганрог: Изд-во ТТИ ЮФУ, 2007. – 100 с.
42. Маслов А.В. Проектирование информационных систем в экономике: учебное пособие / А.В. Маслов. – Томск: Изд-во Томского политехнического университета, 2008. – 216 с.
43. Золотов С.Ю. Проектирование информационных систем : учебное пособие / С.Ю.Золотов. – Томск : Эль Контент, 2013. – 88 с.
44. Гвоздева Т.В. Проектирование информационных систем : учебное пособие / Т.В. Гвоздева, Б.А. Баллод. – Ростов н/Д : Феникс, 2009. – 508 с.
45. Brian Dobing, Jeffrey Parson. How UML is Used // CACM, vol. 49, #5,2006.
46. John Erickson, Keng Siau. Can UML Be Simplified? Practitioner Use of UML in Separate Domains, 2007.
47. Stanislaw Wrycza, Bartosz Marcinkowski. Towards a Light Version of UML2.X: Appraisal and Model, 2007.
48. Леоненков А.В. Самоучитель UML. 2-е изд. – СПб.: БХВ-Петербург, 2004. – 432 с.

Навчальне видання

Авраменко Валентин Семенович
Авраменко Артем Сергійович

ПРОЕКТУВАННЯ ІНФОРМАЦІЙНИХ СИСТЕМ

Навчальний посібник

Загальний редактор *Г.В. Косенюк*
Комп'ютерне верстання *А.С. Авраменко*

Підписано до друку __.__.2017. Формат 60х84/16.
Ум. друк. арк. 20,28. Тираж 300 пр. Зам. № __

Видавець і виготовлювач
Черкаський національний університет імені Богдана Хмельницького
Адреса: бульвар Шевченка, 81, м. Черкаси, Україна, 18031
Тел. (0472) 37-13-16, факс (0472) 37-22-33,
e-mail: vydav@cdu.edu.ua, http://www.cdu.edu.ua
Свідоцтво про внесення до державного реєстру
суб'єктів видавничої справи ДК №3427 від 17.03.2009 р.