Технології програмування

Зміст

Передмова

На ранніх етапах розвитку програмування, коли програми писалися як послідовностей машинних команд, будь-якої технології програмування була відсутня. Після досягнення спочатку непереборного рівня складності виникла інженерія програмування.
До кінця 70-х років програмування, як правило, було роботою окремих обдарованих людей. Через недосконаліть перших методик програмування навіть відносно короткі програми (довжиною близько 600 рядків) створювалися протягом кількох місяців.
Початок 80-х років відповідало широкому запровадження у практику програмування методів проектування, запозичених із техніки. Наприклад, за прикладом техніки, впроваджується стандарти, що регламентує стадії та етапи програмних розробок. Цей стандарт входить до групи стандартів єдиної системи програмної документації (ЄСПД). ЄСПД відіграла значну позитивну роль у практиці вітчизняного програмування та пережила без значних змін вже кілька нових технологій програмування, наприклад, технологію структурного програмування та технологію об'єктно-орієнтованого програмування.
Технологія програмування - це наукова та практично апробована стратегія розробки програм, що містить опис сукупності методів та засобів розробки програм, а також порядок застосування цих методів та засобів.
На сьогодні поняття процесу програмування якісно змінилися. Виробництво програм набуло масового характеру, суттєво збільшився їх обсяг та складність. Розробка програмних комплексів вимагає значних зусиль великих колективів фахівців. Програми перестали бути лише обчислювальними та почали виконувати найважливіші функції з управління та обробки інформації в різних галузях науки, техніки, економіки та ін.
З появою систем автоматизованого проектування (САПР) у 80-х роках було зроблено узагальнення теорії проектування технічних систем та пристроїв із виявленням інваріантів у вигляді проектних процедур, особливо евристичних. Були намічені шляхи та зроблено перші спроби їхньої автоматизації. Найбільш високу складність є автоматизація ранніх етапів проектування. На цих етапах для задоволення потреби подолання дискомфорту необхідно синтезувати ідеї реалізації систем та пристроїв.
Паралельний розвиток теорії програмування та теорії проектування зробив актуальним їх системне дослідження. Мета досліджень, полягала у досягненні позитивного подальшого взаємного проникнення цих теорій.
Перша тема містить відомості з основ теорії проектування, необхідні для ознайомлення з термінологією проектування взагалі та основними принципами проведення програмних проектів. Даються такі методологічні поняття проектування, як елементи системного підходу, а також одного з найважливіших його методів — блочно-ієрархічного підходу. У розділі пояснюється місце стандартів у програмуванні. Вводяться поняття життєвого циклу програмного виробу, і навіть стадій та етапів проведення програмних розробок. Розкриваються основні поняття моделювання систем та роль моделювання розробки проектів програмних систем, проводяться приклади моделей.
У другій темі розглядаються методи активізації мислення на ранніх етапах проектування програмних виробів, що дозволяє вирішити завдання вибору найкращого варіанту з безлічі допустимих проектних рішень, які задовольняють пред'явленим вимогам. Методи пошукового конструювання, запозичені з техніки, адаптуються стосовно програм. Надаються приклади видів діалогів програм, що дозволяє підвищити ефективність розробки зовнішніх функціональних специфікацій. Для повного освоєння низки положень глави може знадобитися кілька років. Але ж треба колись починати ставати системним аналітиком.
У третій темі викладається інженерний технологічний підхід до розробки програм, згідно з яким досягається скорочення термінів розробки програмних продуктів завдяки комбінації етапів та видів робіт, орієнтованої на різні класи програмного забезпечення та на особливості колективу розробників.

Четверта тема розкриває поняття фізичної та логічної структури даних програм. У розділі розглядається набір операцій над структурами даних програм, наводиться класифікація логічних структур даних, розбираються базові структури даних, динамічні та динамічно пов'язані структури даних, і навіть файлові структури даних. Розглядаються методи документування структур даних.

П'ята тема містить опис методики розробки структурованих алгоритмів у формі проектної процедури розробки функціональних описів. Надаються рекомендації щодо використання проектної процедури стосовно областей, що знаходяться поза сферою програмування: техніки, організаційного забезпечення.

У шостій темі вводиться поняття архітектури програмної системи, наводяться відомості з низки способів об'єднання окремих програм у єдиний програмний комплекс.

Сьома тема містить опис технології структурного програмування, що вважається застарілою, але нині ще використовується як самостійно, і у гібридних объектно-ориентированных проектах. Ряд фундаментальних ідей цієї технології сприйняли сучасними технологіями.

У восьмій темі розглядається технологія об'єктно-орієнтованого проектування. Розбираються основні поняття технології. Даються кроки етапів виконуваних робіт. Розглядаються приклади виконання проектів малої та середньої складності.

Дев'ята тема Містить поняття технологій візуального програмування. Ця технологія дозволяє в діалоговому режимі створювати «скелет» програми.

У десятій темі розкривається поняття САПР програмних розробок, заснованих на CASE-засобах, що дозволяють у наочній формі моделювати предметну область, аналізувати цю модель на всіх етапах розробки та супроводу програмного проекту та розробляти програми відповідно до інформаційних потреб користувачів. У розділі розглядається CASE-засіб Rational Rose фірми "Software Corporation" (США), призначене для автоматизації етапів аналізу та проектування програмних систем, а також для генерації кодів різними мовами та випуску проектної документації.

Одинадцята тема присвячена тестуванню програм, що дозволяють досягти заданого рівня найважливішого критерію якості програмних виробів – надійності. У розділі викладаються аксіоми тестування, прийоми налагодження, різні підходи до тестування програм.

У дванадцятій темі описуються основні засади менеджменту програмних розробок. Даються принципи організації колективу розробників програмних виробів, посадові обов'язки та функції окремих працівників.

Додаток 1 необхідний розуміння стадій і етапів розробки програм за ДСТУ, але він замінює, можливо, змінився текст стандарту.

Додаток 2 містить приклад виконання навчального технічного завдання. Цей приклад розкриває принципи складання технічного завдання, але також замінює стандарт.

Додаток 3 дає уявлення про фонд евристичних прийомів проектування програм.

Додаток 4 містить опис елементів мови програмування Object Pascal, він необхідний кращого розуміння тем. 8 та 9.

Додаток 5 розкриває основні терміни та визначення, що використовуються.

Вступ.

В даний час програмування трансформувалося у цілу індустрію виробництва програмних виробів. Тому вже мало знати лише мову програмування та операційний підхід до складання алгоритмів.

Професійний розробник програмних виробів повинен мати теорію проектування, методи активізації мислення. Йому потрібне вміння оперування моделями, методами генерації рішень та вибору їх оптимальних варіантів.

Створення програмних виробів колективом розробників зумовило необхідність уміння планування робіт та їхнього розподілу між окремими учасниками проекту.

У сучасному програмуванні потрібно активне володіння дедуктивним мисленням, що не досягається шкільною та вузівською освітою. Курс містить теоретичні знання, необхідні як програмістам-кодувальникам програм, так і системним аналітикам. У ній викладаються методики оволодіння дедуктивним мисленням.

На ранніх етапах розвитку програмування, коли програми писалися як послідовностей машинних команд, будь-якої технології програмування була відсутня. Після досягнення спочатку непереборного рівня складності виникла інженерія програмування. До кінця 70-х років програмування, як правило, було роботою окремих обдарованих людей. Через недосконаліть перших методик програмування навіть відносно короткі програми (довжиною близько 600 рядків) створювалися протягом кількох місяців. Початок 80-х років відповідало широкому запровадження у практику програмування методів проектування, запозичених із техніки. Наприклад, за прикладом техніки, впроваджується ДСТУи, що регламентує стадії та етапи програмних розробок. Стандарт належить до групи стандартів єдиної системи програмної документації (ЕСПД). ЕСПД відіграла значну позитивну роль у практиці програмування та пережила без значних змін вже кілька нових технологій програмування, наприклад, технологію структурного програмування та технологію об'єктно-орієнтованого програмування.

Технологія (Матеріал з Вікіпедії – вільної енциклопедії)

Технологія(відін.-грец.τέχνη «мистецтво, майстерність, вміння» +λόγος«слово; думка, сенс, поняття») – сукупність методів і інструментівдля досягнення бажаного результату [w1]; у широкому розумінні – застосування наукового знання для вирішення практичних завдань[w1][w2]. Технологія включає способи роботи, її режим, послідовність дій[w3].

Технологія є порівняно новим, багатогранним терміном, точне визначення якого вислизає через постійний розвиток сенсу цього поняття, як самого по собі, так і взятого у відносинах з іншими такими ж широкими поняттями: культура, суспільство, політика, релігія, природа[w4]. На початку XX століття термін «технологія» охоплював сукупність засобів, процесів та ідей на додаток до інструментів та машин. До середини XX століття поняття визначалося такими фразами як «засоби або діяльність, за допомогою яких людина змінює своє довкілля і маніпулює нею»[w5].

Технологія програмування - це наукова та практично апробована стратегія розробки програм, що містить опис сукупності методів та засобів розробки програм, а також порядок застосування цих методів та засобів.

На сьогодні поняття процесу програмування якісно змінилися. Виробництво програм набуло масового характеру, суттєво збільшився їх обсяг та складність. Розробка програмних комплексів вимагає значних зусиль великих колективів фахівців. Програми перестали бути лише обчислювальними і почали виконувати найважливіші функції з управління та обробки інформації в різних галузях науки, техніки, економіки та ін. З появою систем авоматизованого проектування (САПР) у 80-х роках були зроблені узагальнення теорії проектування технічних систем та пристроїв виявленням інваріантів як проектних процедур, особливо евристичних. Були намічені шляхи та зроблено перші спроби їхньої автоматизації. Найбільш високу складність є автоматизація ранніх етапів проектування. На цих етапах для задоволення потреби подолання дискомфорту необхідно синтезувати ідеї реалізації систем та пристроїв. Паралельний розвиток теорії програмування та теорії проектування зробив актуальним їх системне дослідження. Мета досліджень полягала у досягненні позитивного подальшого взаємного проникнення цих теорій.

У курсі розглядаються такі теми:

Перша темамістить відомості з основ теорії проектування, необхідні для ознайомлення з термінологією проектування взагалі та основними принципами проведення програмних проектів. Розглядаються такі методологічні поняття проектування як елементи системного підходу, а також одного з його найважливіших методів — блочно-ієрархічного підходу. У темі пояснюється місце стандартів програмування. Вводяться поняття життєвого циклу програмного виробу, і навіть стадій та етапів проведення програмних розробок. Розкриваються основні поняття моделювання систем та роль моделювання розробки проектів програмних систем, проводяться приклади моделей.

У другій темі розглядаються методи активізації мислення на ранніх етапах проектування програмних виробів, що дозволяє вирішити задачу вибору найкращого варіанта з множини допустимих проектних рішень, які задовольняють пред'явленим вимогам. Методи пошукового конструювання, запозичені з техніки, адаптуються стосовно програм. Розглядаються приклади видів діалогів програм, що дозволяє підвищити ефективність розробки зовнішніх функціональних специфікацій.

У третій темі викладається інженерний технологічний підхід до розробки програм, згідно з яким досягається скорочення термінів розробки програмних продуктів завдяки комбінації етапів та видів робіт, орієнтованої на різні класи програмного забезпечення та на особливості колективу розробників.

Четверта тема розкриває поняття фізичної та логічної структури даних програм. У темі розглядається набір операцій над структурами даних програм, наводиться класифікація логічних структур даних, розбираються базові структури даних, динамічні та динамічно пов'язані структури даних, і навіть файлові структури даних. Розглядаються методи документування структур даних.

П'ята тема містить опис методики розробки структурованих алгоритмів у формі проектної процедури розробки функціональних описів. Розглядаються рекомендації щодо використання проектної процедури стосовно областей, що знаходяться поза сферою програмування: техніки, організаційного забезпечення.

У шостій темі вводиться поняття архітектури програмної системи, наводяться відомості з низки способів об'єднання окремих програм у єдиний програмний комплекс.

Сьома тема містить опис технології структурного програмування, що вважається застарілою, але нині ще використовується як самостійно, і у гібридних объектно-ориентированных проектах. Ряд фундаментальних ідей цієї технології ефективно використовуються сучасними технологіями.

У восьмій темі розглядається технологія об'єктно-орієнтованого проектування. Розбираються основні поняття технології. Розглядаються кроки етапів виконуваних робіт. Розглядаються приклади виконання проектів малої та середньої складності.

Дев'ята тема Містить поняття технологій візуального програмування. Ця технологія дозволяє в діалоговому режимі створювати «скелет» програми.

У десятій темі розкривається поняття САПР програмних розробок, заснованих на CASE-засобах, що дозволяють у наочній формі моделювати предметну область, аналізувати цю модель на всіх етапах розробки та супроводу програмного проекту та розробляти програми відповідно до інформаційних потреб користувачів. У розділі розглядається CASE-засіб IBM Rational Rose, призначений для автоматизації етапів аналізу та проектування програмних систем, а також для генерації кодів різними мовами та випуску проектної документації.

Одинадцята тема присвячена тестуванню програм, що дозволяють досягти заданого рівня найважливішого критерію якості програмних виробів – надійності. У розділі викладаються аксіоми тестування, прийоми налагодження, різні підходи до тестування програм.

У дванадцятій темі описуються основні засади менеджменту програмних розробок. Розглядаються принципи організації колективу розробників програмних виробів, посадові обов'язки та функції окремих працівників.

Додаток 1необхідно для розуміння стадій і етапів розробки програм за ДСТУами, але воно не замінює, можливо, текст стандарту, що змінився.

Додаток 2містить приклад виконання навчального технічного завдання. Цей приклад розкриває принципи складання технічного завдання, але також замінює стандарт.

Додаток 3дає уявлення про фонд евристичних прийомів проектування програм.

Додаток 4містить опис елементів мови програмування Object Pascal, він необхідний кращого розуміння тем 8 і 9.

Додаток 5розкриває основні терміни та визначення, що використовуються в цьому тексті.

Тема 1 МЕТОДОЛОГІЧНІ ОСНОВИ ПРОЕКТУВАННЯ ПРОГРАМ

1.1. ЗАГАЛЬНІ ПОЛОЖЕННЯ ТЕОРІЇ ПРОЕКТУВАННЯ

Як без оформленого проекту цілком можна побудувати шпаківню, але неможливе будівництво висотної будівлі або комплексу космодрому з будівельною індустрією, житловими, стартовими та виробничими комплексами, так і без проекту можна реалізувати лише невелику програму, але не автоматизоване робоче місце фахівця, а тим більше автоматизовану систему управління великого підприємства.

Що ж роблять програмісти? Програмісти виробляють програмний продукт. У термінах автоматизованих систем програмісти створюють програмне забезпечення.

Програмний продукт— програма, яку можна запускати, тестувати, виправляти та розвивати. Така програма має бути написана в єдиному стилі, ретельно відтестована до необхідного рівня надійності, супроводжена докладною документацією та підготовлена для тиражування.

Програмний виріб- Програма на носії даних, що є продуктом промислового виробництва. Термін затверджено Державним стандартом.

Програмне забезпечення автоматизованих систем- Сукупність програм на носіях даних та програмних документів, призначена для налагодження, функціонування та перевірки працездатності автоматизованих систем.

Автоматизована система (АС)- Організаційно-технічна система, що забезпечує вироблення рішень на основі автоматизації інформаційних процесів у різних сферах діяльності (управління, проектування, виробництво тощо) або їх поєднаннях, система, що складається з персоналу та комплексу засобів автоматизації його діяльності, що реалізує інформаційну технологію виконання встановлених функцій.

Необхідність у проекті викликана складністю завдання.

Наступний приклад показує нелінійну залежність зростання складності завдання від її розміру. Необхідно в думці скласти числа 4 і 3. Відповідь, зрозуміло, - 7. Необхідно в думці перемножити числа 7 і 9. Відповідь, звичайно, - 63. Але якщо не знаєте таблицю множення, то треба виконати нестандартне перетворення у вигляді багаторазового додавання. Чи важко воно для вас? Необхідно в думці перемножити числа 289 і 347. Якщо ви не феноменальний лічильник, то чи вистачить у вашій голові оперативної пам'яті? А чи зможете ви перемножити на думці шестизначні числа? Але якщо декомпозувати це завдання на обчислення ряду творів одного з співмножників на окремі цифри іншого співмножника і

потім знайдені твори скласти (при цьому записувати на папері всі проміжні результати), то з цим завданням може справитися пересічна людина.

Ще приклад, що показує один із шляхів зниження складності завдання за рахунок її декомпозиції на доступні для огляду частини. Звичайна нормальна людина із середніми здібностями може одночасно у своїй голові втримати не більше семи думок. У школі завдання з шістьма діями вважаються завданнями підвищеної складності та позначаються символом «*». В арміях різних країн, часів і народів проводилося поділ на десятки, сотні, тисячі. У командирів у підпорядкуванні перебувало або десятьох воїнів, або десятьох молодших командирів.

Програма - дуже складний об'єкт, що містить до сотень тисяч і навіть кількох мільйонів думок. Складність програмного продукту - аж ніяк не випадкова властивість, скоріше необхідна. Його складність визначається чотирма основними причинами: складністю завдання, складністю управління процесом розробки, складністю опису поведінки окремих підсистем, складністю забезпечення гнучкості кінцевого програмного продукту.

У табл. 1.1 наведено п'ять ознак складної системи разом із прикладами. Ці ознаки інваріантні як для відчутної системи реального світу «музичний центр», так програмної системи — текстового редактора.

Таблиця 1.1.

Приклади музичного центру та текстового редактора як складних систем

Ознаки	Музичний центр	Текстовий редактор
1. Складність часто представляється як ієрархії. Складна система зазвичай складається з взаємозалежних підсистем, які також можуть бути розділені на підсистеми і т. д., аж до найнижчих рівнів абстракції	Складається з шести підсистем: підсилювача, блоку цифрового керування системою, програвача компакт-дисків, касетної деки, радіоприймача, динаміків. Кожна з підсистем може бути розділена на підсистеми. Підсилювач поділяється на фільтри, попередні каскади посилення та підсилювач потужності. Блок цифрового управління системою поділяється на процесор, панель кнопок, панель індикації та цифро аналогові, аналого-цифрові перетворювачі. Програвач компакт-дисків - на лазер, пристрій керування лазером, цифро аналоговий перетворювач і т.д.	Складається з файлів: опис глобальних констант і змінних, бібліотеки модулів підтримки дисплея, бібліотеки модулів підтримки клавіатури, бібліотеки модулів підтримки головного «меню», набору модулів самого редактора. Бібліотека модулів підтримки клавіатури у свою чергу включає модуль рядкового редактора, який використовує такі модулі файлу бібліотеки підтримки дисплея, як відображення рядка на екрані та переміщення курсору в задану позицію, а також цілий ряд внутрішніх модулів
2. Вибір нижчого рівня абстракції є значною мірою довільним і більшою мірою визначається спостерігачем	Як нижчий рівень абстракції можна вибрати вузли, що виконують закінчені функції обробки електронних або звукових сигналів: підсилювальні каскади — посилюють сигнали, фільтри — забезпечують виключення перешкод відповідних частот тощо. , Т. е. Операційні	Системні аналітики як нижчий рівень абстракції в програмах використовують модулі. Кодувальники, що реалізують модулі, як нижчий рівень абстракції використовують алгоритмічні структури (оператори) мови високого рівня та структури даних

	підсилювачі, транзистори, діоди та ін.	
3. Внутрі елементні зв'язки зазвичай міцніші за між елементні зв'язки. Тому взаємодії елементів усередині елементів системи виявляються природно відокремленими від взаємодії між самими елементами. (Відмінність між внутрішньо- та між елементними взаємодіями обумовлює поділ системи на абстрактні автономні частини, які можна вивчати окремо.)	Кожен вузол, як правило, має або один (керуючий), або два входи (керуючий та сигнальний) і лише один вихід (оброблений сигнал). Зв'язки між вузлами забезпечуються з'єднанням входів та виходів різних вузлів. Вузол працює як «чорна скринька», внутрішньо елементні зв'язки якої «не видно» ззовні. Кількість внутрішньо елементних зв'язків суттєво більша, ніж між елементних	Зв'язки між модулями реалізовані за допомогою аргументів (у кількості від 0 до 10) функцій та невеликої кількості глобальних змінних. Внутрішньо модульні зв'язки реалізовані за допомогою загальних для модуля змінних (зазвичай від 10 до кількох десятків). Оскільки змінні доступні з будь-якої точки модуля, такий зв'язок є зв'язком типу «все з усіма»
4. Ієрархічні системи зазвичай складаються з кількох підсистем різного типу, реалізованих у різному порядку та у різноманітних комбінаціях	Кожен з електронних вузлів пристрою виконаний, зрештою, з тих самих типових елементів: напівпровідникових приладів (транзисторів і діодів), опорів, конденсаторів різних номіналів і способів виготовлення. Розрізняються порядок та комбінації використання цих елементів у різних вузлах	Кожен модуль являє собою набір одних і тих же обчислювальних структур (операторів) і стандартних функцій, що по-різному взаємодіють один з одним через загальні дані в кожному з модулів
5. Складна система, що працює, неминуче виявляється результатом розвитку працюючої простої системи. Складна система, розроблена від початку до кінця на папері, ніколи не працює і не можна змусити її заробити. Зазвичай спочатку створюють просту працюючу систему, яку розвивають у наступних версіях на основі нових ідей, отриманих під час експлуатації	Прототипи музичного центру: радіо, касетний магнітофон, програвач компакт-дисків. Музичний центр є комбінацією та подальшим розвитком цих систем: покращено підсистеми посилення та фільтрації звуку, покращено динаміки, додано цифровий процесор для обробки звуку.	Спочатку з'явилися найпростіші текстові редактори як малі, так і екранні для набору та коригування текстів в режимі друкарської машинки. Потім з'явилися текстові процесори, що форматують текст та здійснюють перевірку орфографії. Далі з'явилися інтегровані системи, що включають процесори: текстові, графічні, електронних таблиць, баз даних та ділової графіки

Проектування— це розробка проекту, процес створення специфікації, яка потрібна на побудови в заданих умовах ще неіснуючого об'єкта з урахуванням первинного описи цього об'єкта. Результатом проектування є проектне рішення чи сукупність проектних рішень, які відповідають заданим вимогам. Задані вимоги обов'язково мають включати форму подання рішення.

Специфікаціяу сфері проектної діяльності - це якийсь опис у точних термінах.

Проектним документом називають документ, виконаний за заданою формою, у якому представлено будь-яке проектне рішення. У програмуванні проектні рішення оформлюються

як програмної документації. Розрізняють зовнішню програмну документацію, яка узгоджується із замовником, та внутрішню проміжну документацію проекту, яка необхідна самим програмістам для їхньої роботи.

Проект(Від лат. Projectus - кинутий вперед) - сукупність проектних документів відповідно до встановленого переліку, що представляє результат проектування.

Проектною ситуацією називають реальність (ситуацію), у якій ведеться проектування. Паровоз та електровоз проектувалися у різних проектних ситуаціях, визначених рівнем знань людства. Саме тому в XIX ст. став віком паровоза.

Будь-яке завдання характеризується необхідністю перетворення деякої вихідної ситуації на ситуацію, звану рішенням. Говорячи про будь-яке завдання, завжди маємо її інформаційні елементи:

- інформація про умову (умова задачі) - що задано;
- інформація про рішення (ознаки вихідної ситуації) - що потрібно отримати;
— інформація про технологію перетворення умови на рішення — як вирішити.

Проектне завдання(англ. Engineering Task) характеризується невизначеністю апріорі інформації: що потрібно отримати, що поставлено. Більш того, спосіб розв'язання задачі є об'єктом проектування. І нарешті, розв'язання проектної задачі має бути знайдено в рамках обмежень зовнішнього середовища проектування: доступних коштів, заздалегідь заданих термінів, можливостей технічних засобів та інструментарію програмування, наукових знань, програмних заділів тощо.

Проектні завдання під силу тільки тим, хто здатний сприймати явище цілком і в найдрібніших деталях одночасно, дотепно пов'язуючи ці деталі між собою. Саме таких людей завжди називали інженерами, та й сам термін походить від латинського ingenium, що означає природний розум, а також винахідливість. Інженер-програміст - спеціаліст з вирішення проектних завдань. Інженер-системотехнік - інженер інженерів, фахівець із вирішення проектних завдань створення таких особливо складних штучних систем, як автоматизовані системи.

Джерелом, першопричиною будь-якої проектної діяльності є суб'єкт — людина чи група людей, які мають дискомфорт у існуючій ситуації.

«Лежачи на теплій печі» (перебуваючи у комфортній ситуації), можна мріяти про вирішення світових проблем і нічого не робити. Однак страх перед майбутнім дискомфортом (замерзання, голод) поверне мрійника в реальну ситуацію і вимагатиме знаходження способу вирішення і вирішення проблеми подальшого його існування (заготівля дров і продуктів).

Дискомфорт суб'єкта може бути конкретизований як потреби, задоволення якої знімає його. Для задоволення потреби потрібен певний об'єкт проектування (у разі програмний продукт), наявність якого, його і стан задовольняють потребам суб'єкта.

Відповівши питанням, які властивості має володіти об'єкт, підготуємо вихідні дані для наступного питання: як має бути влаштований об'єкт, щоб мати такі властивості? Для вирішення цього завдання також необхідно розкрити вихідну ситуацію. Причому таке розкриття потрібно різних рівнях конкретизації об'єкта. Отже, виходить, що процедура розкриття проектної ситуації може повторюватися багаторазово і різних етапах вирішення спільної проектної завдання. При цьому всі рішення взаємопов'язані: рішення, прийняті на одному етапі, мають бути враховані під час виконання інших. Якимось чином формалізувати і врахувати цей вплив важко, тому процес рішення має ітераційний характер.

Для задоволення потреби має бути реалізована деяка діяльність, кінцевим результатом якої (метою) і буде створення об'єкта та (або) приведення його в бажаний (цільовий) стан. Ця діяльність також є об'єктом, що вимагає проектування. Стосовно неї також має вирішуватися аналогічне завдання. Інакше висловлюючись, сам процес проектування є об'єктом проектування.

Метод(від грецьк. methodos — спосіб дослідження чи пізнання, теорія чи вчення) — прийом чи система прийомів практичного здійснення чогось у будь-якій предметної області, сукупність прийомів чи операцій практичного чи теоретичного освоєння дійсності,

підпорядкованих вирішенню конкретних завдань. Метод включає засоби - за допомогою чого здійснюється дія та способи - яким чином здійснюється дія.
Методика (Від грец. methodike) - Упорядкована сукупність методів практичного виконання чого-небудь.
Методики проектування викладаються у вигляді описів проектних процедур та проектних операцій.
Під проектною процедурою розуміють формалізовану сукупність дій, виконання яких закінчується проектним рішенням. Наприклад, проектною процедурою є процедури розкриття проектної ситуації та розробки структури програми.
Дія чи формалізовану сукупність дій, що становлять частину проектної процедури, алгоритм яких залишається незмінним для низки проектних процедур, називають проектною операцією. Наприклад, креслення схеми, диференціювання функції.
Проектні процедури можуть включати інші проектні процедури тощо до проектних операцій. Проектні процедури можуть бути алгоритмами (тільки для тривіальних нетворчих операцій) і евроритми (якими викладаються евристичні операції).
Алгоритм- суворо однозначно визначена для виконавця послідовність дій, що призводять до вирішення завдань.
Сучасне значення слова «алгоритм» багато в чому аналогічно до таких понять, як рецепт, процес, методика, спосіб. Згідно з Д. Кнутом [17], алгоритм має п'ять важливих властивостей.
Кінцівка. Алгоритм завжди повинен закінчуватися після виконання кінцевого числа кроків.
Визначеність. Кожен крок алгоритму має бути точно визначено.
Наявність вхідних даних. Алгоритм має деяку кількість вхідних даних, що задаються до початку роботи або визначаються динамічно під час виконання.
Наявність вихідних даних. Алгоритм має одне або кілька вихідних даних, що мають певний зв'язок із вхідними даними.
Ефективність. Алгоритм зазвичай вважається ефективним, якщо його оператори досить прості для того, щоб їх можна було виконати за допомогою олівця і паперу протягом кінцевого проміжку часу.
Термін "евроритм" науки евристика утворений від легендарного вигуку Архімеда "Еврика!", що в перекладі з грецької означає "знайшов, відкрив". Алгоритм у процесі виконання не змінюється бездумним виконавцем (процесором). На відміну від алгоритму евроритм виконується людиною, яка мислить, яка може вдосконалити порядок своєї роботи в процесі її виконання. Евроритм може включати алгоритми. Наприклад, інструкція користування програмою — це евроритм, особливо якщо одні й самі дії можна виконати різними способами (через пункт меню чи натисканням кнопки).
Евристика наука, що розкриває природу розумових операцій людини під час вирішення конкретних завдань незалежно від своїх конкретного змісту. У вужчому сенсі евристика — це припущення, засновані на досвіді розв'язання родинних завдань.
Інженерія програмування(англ. software engineering, у термінах автоматизованих систем – розробка програмного забезпечення) – інженерна справа, творча технічна діяльність. Інженерія спирається на специфічні методи та методики, у тому числі евристичні. Інженерія [20] вивчає різні методи та інструментальні засоби з погляду певних цілей, тобто має очевидну практичну спрямованість. Основна ідея інженерії програмування полягає в тому, що розробка програмного забезпечення є формальним процесом, який можна вивчати, виражати в методиках та вдосконалювати.
Інженерія програмування має чітку певну дату народження — 1968 р. Причина її появи — реакція на так звану «кризу програмного забезпечення», викликану досягненням непереборного рівня складності. Характерні питання та завдання інженерії програмування, викладені Фредеріком Бруксом, актуальні й до сьогодні.
Як проектувати та будувати програми, що утворюють системи?

Як проектувати та будувати програми та системи, які є надійним, налагодженим, документованим та супроводжуваним продуктом?
Як здійснювати інтелектуальний контроль за умов великої складності?
Інженерна діяльність базується на сукупності загальнонаукових методів системного підходу, аналітико-синтетичному методі блочно-ієрархічного підходу до проектування складних систем, аналітико-синтетичному методі стадії та етапи розробки. Додатково інженери використовують методи та методики, спеціалізовані стосовно об'єкта проектування або виготовлення.
Важливими розробки процесів проектування є такі поняття, як стратегія і тактика.
Стратегія(від грец. Stratos - військо і ago - веду) - наука, мистецтво генерації найбільш істотних загальних довДСТУрокових цілей і найбільш загального плану досягнення переваги, курсу дій та розподілу ресурсів ще до виконання реальних дій.
Тактика(від грецьк. taktika — мистецтво упорядковувати) — фіксована у своїй послідовності сукупність засобів і прийомів задля досягнення наміченої мети і мистецтво її застосування, способи дії, зорієнтовані досягнення конкретних цілей, є ланками реалізації стратегічних цілей. Метою застосування способу дії є здійснення оптимальних дій, заздалегідь не передбачених стратегічним планом ситуаціях, вже у процесі виконання реальних дій.
Стратегія охоплює теорію і практику підготовки до виконання проекту, а також загальне планування тактик ведення проектів. Стратегія визначає, куди, в якому напрямку рухатись, куди тримати курс ще до початку проекту. А тактика визначає, як, яким способом рухатися, які конкретні дії робити при труднощі під час виконання проекту.
Стратегія виконання конкретного проекту описується у програмному документі – технічному завданні.
Методологія(Від грецьк. methodos і logos - слово, вчення про методи) - система принципів і способів організації та побудови теоретичної та практичної діяльності, а також вчення про цю систему.
Методологія програмування вивчає методи з погляду основ побудови. Це об'єднана єдиним філософським підходом сукупність методів, що застосовуються у процесі розробки програмних продуктів. Будь-яка методологія створюється на основі вже накопичених у предметній галузі емпіричних фактів та практичних результатів.
Технологія(від грец. techne - мистецтво, майстерність, уміння і logos - слово, вчення) - сукупність виробничих процесів у певній галузі виробництва, а також науковий опис способів виробництва, сукупність прийомів, що застосовуються у будь-якій справі, майстерності, мистецтві.
Сучасна методологія проектування дозволила довести методи проектування до технологій із набором методик.
Технологія програмування як наука вивчає технологічні процеси та порядок їх проходження (з використанням знань, методів та засобів). Технологічний процес - послідовність спрямованих на створення заданого об'єкта дій (технологічних процедур і операцій), кожна з яких заснована на будь-яких природних процесах і людської діяльності. Знання, методи та засоби можуть використовуватись у різних процесах і, отже, у технологіях.
Технологія програмування— для інженера це наукова та практично апробована стратегія створення програм, що містить опис сукупності методів та засобів розробки програм, а також порядок застосування цих методів та засобів.
У реальних проектних ситуаціях потрібний синтез раціональної стратегії кожного конкретного проекту. Інженери часто цей синтез здійснюють на основі однієї, двох та навіть трьох технологій.
Інженерна справа (діяльність зі створення та використання технологій) охоплює не тільки проектування та виробництво, а й структури організацій із взаємодією людей. У термінах інженерного відносини технологія — інструментарій інженера, інтелектуальний чи відчужений як штучних систем.

Головна різниця між технологією програмування та програмною інженерією полягає у способі розгляду та систематизації матеріалу. У технології програмування акцент робиться на вивченні процесів розробки програм (технологічних процесів) у порядку їх проходження – методи та інструментальні засоби розробки програм використовуються у цих процесах (їх застосування та утворюють технологічні процеси). У програмній інженерії вивчаються насамперед методи та інструментальні засоби розробки програм з погляду досягнення певних цілей — вони можуть використовуватись у різних технологічних процесах (і у різних технологіях програмування). Як ці методи та засоби утворюють технологічні процеси — питання другорядне.

Не слід плутати технологію програмування з методологією програмування, хоча у обох випадках вивчаються методи. У технології програмування методи розглядаються «згори» — з погляду організації технологічних процесів, методології — «знизу» — з погляду основ їх побудови.

Перед нами стоїть питання, яке вже згадувалося: «Як визначити, чи досягнута мета, чи привела діяльність до бажаного результату: чи той об'єкт створений, який нам потрібен, чи має потрібні властивості?». І тому використовуються показники якості (іноді їх називають критеріями) — величини, властивості, поняття, що характеризують систему з погляду суб'єкта, дозволяють оцінити рівень задоволення його потреб. На початковому етапі проектування аналіз потреб дозволяє визначити вид об'єкта; його функції (що об'єкт виконує при функціонуванні), властивості та стан, у яких задовольняються потреби, і навіть якість об'єкта.

При дослідженні систем вирішуються завдання аналізу та синтезу.

Аналіз(Від грец. Analysis - розкладання, розчленування) - прийом розумової діяльності, пов'язаний з уявним (або реальним) розчленуванням на частини предмета, явища або процесу.

Синтез (від грец. synthesis - поєднання, поєднання, складання) - метод наукового дослідження явищ дійсності в їх єдності та цілісності, у взаємодії їх частин, узагальнення, зведення в єдине ціле.

Синтез нерозривно пов'язані з аналізом і немає окремо від цього, і навіть пов'язані з іншими розумовими процесами. Без синтезу неможливе виконання процедур узагальнення, систематизації, порівняння (вибору), разом із якими він залишає логічний апарат мислення.

Теоретично проектування використовуються такі поняття аналізу та синтезу.

Аналіз -процес визначення функціонування за заданим описом системи.

Синтез- Процес побудови опису системи по заданому функціонуванню.

Одним з визначень задачі оптимізації розробки програм є знаходження розумного компромісу між досягнутою метою та витраченими на це ресурсами, що може призводити як до перегляду цілей розробки, так і до зміни ліміту ресурсів. Вирішення цього завдання особливо важливе, оскільки програми є дорогим продуктом, і, правильно провівши розробку, можна досягти значного її здешевлення.

За своєю природою програма (тобто набір інструкцій) набагато ближче до технології (точніше, до опису технологічного процесу перетворення вхідної інформації у вихідну інформацію), ніж виробу. Це означає, що з оцінки продуктивність праці програміста не потрібно шукати спосіб оцінки кількості продукції, яку він випускає, оскільки ніяка фізична продукція немає і, отже, немає її обсягу. При використанні стандартного терміна «програмний виріб» виникають методологічні, правові та суто технічні складності. Так, наприклад, програмісту не складе ніяких труднощів вставити в програму будь-яку кількість модулів з будь-яким обсягом незадіяних операторів. Товарні властивості програмного продукту – не товарні властивості цього виробу, а товарні властивості технології. Причому технології це теж об'єкти, які традиційно проектуються інженерами.

Програмний продукт є розробленою програмістом інформаційною технологією, яка матеріалізується у замовника у вигляді виробу, стаючи автоматизованими системами та

інструментами їх обслуговування. Це пояснення, мабуть, знімає багато правових проблем, а також проблеми ціноутворення.

1.2. ЗАГАЛЬНІ ПРИНЦИПИ РОЗРОБКИ ПРОГРАМ

Програми розрізняються за призначенням, виконуваним функцій, форм реалізації. Однак можна вважати, що існують деякі загальні принципи, які слід використовувати для розробки програм.

Частотний принцип. Принцип заснований на виділенні в алгоритмах та даних спеціальних груп за частотою використання. Для дій, що найчастіше зустрічаються під час роботи програм, створюються умови для їх швидкого виконання. До даних, що часто використовуються, забезпечується найбільш швидкий доступ. «Часті» операції намагаються робити коротшими. Слід зазначити, що не більше 5% операторів програми надають відчутний вплив на швидкість виконання програми. Цей факт дозволяє значну частину операторів програми кодувати без урахування швидкості обчислень, звертаючи основну увагу при цьому на «красу» та наочність текстів.

Принцип модульності. Під модулем в даному контексті розуміють функціональний елемент системи, що розглядається, має оформлення, закінчене і виконане в межах вимог системи, і засоби сполучення з подібними елементами або елементами вищого рівня даної або іншої системи. Методи відокремлення складових програм в окремі модулі можуть відрізнятися значно. Значною мірою поділ системи на модулі визначається методом проектування програм, що використовується.

Принцип функціональної вибірковості. Цей принцип є логічним продовженням частотного та модульного принципів та використовується при проектуванні програм. У програмах виділяється деяка частина важливих модулів, які мають бути готові для ефективної організації обчислювального процесу. Цю частину у програмах називають ядром чи монітором. При формуванні складу монітора потрібно врахувати дві суперечливі вимоги. До складу монітора, крім чисто керуючих модулів, повинні увійти модулі, що найчастіше використовуються. Кількість модулів повинна бути такою, щоб обсяг пам'яті, яку займає монітор, був не надто великим. Програми, що входять до складу монітора постійно зберігаються в оперативній пам'яті. Інші частини програм постійно зберігаються у зовнішніх пристроях, що запам'ятовують, і завантажуються в оперативну пам'ять тільки при необхідності, перекриваючи один одного також при необхідності.

Принцип генерованості. Основне положення цього принципу визначає такий спосіб вихідного представлення програми, який би дозволяв здійснювати налаштування на конкретну конфігурацію технічних засобів, коло проблем, умови роботи користувача.

Принцип функціональної надмірності. Цей принцип враховує можливість проведення однієї й тієї роботи різними засобами. Особливо важливий облік цього принципу при розробці інтерфейсу для видачі одних і тих же даних різними способами виклику через психологічні відмінності в сприйнятті інформації.

Принцип "за замовчуванням". Застосовується полегшення організації зв'язків із системою як у стадії генерації, і під час роботи з вже готовими програмами. Принцип заснований на зберіганні в системі деяких базових описів структур, модулів, конфігурацій обладнання та даних, що визначають умови роботи із програмою. Цю інформацію програма використовує як задану за умовчанням, якщо користувач забуде або свідомо не конкретизує її.

1.3. СИСТЕМНИЙ ПІДХІД І ПРОГРАМУВАННЯ

Системний підхід- загальнонауковий узагальнений евроритм, що передбачає всебічне дослідження складного об'єкта з використанням компонентного, структурного, функціонального, параметричного та генетичного видів аналізу.

Компонентний аналіз- Розгляд об'єкта, що включає в себе складові елементи і входить, у свою чергу, в систему вищого рангу.

Структурний аналіз виявлення елементів об'єкта та зв'язків між ними.

Функціональний аналіз- Розгляд об'єкта як комплексу виконуваних ним корисних і шкідливих функцій.
Параметричний аналіз- встановлення якісних меж розвитку об'єкта - фізичних, економічних, екологічних та ін. Що стосується програм параметрами можуть бути: час виконання якого-небудь алгоритму, розмір займаної пам'яті і т. д. При цьому виявляються ключові технічні протиріччя, що заважають подальшому розвитку об'єкта, та ставиться завдання їх усунення з допомогою нових технічних рішень.
Генетичний аналіз дослідження об'єкта з його відповідність законам розвитку програмних систем. У процесі аналізу вивчається історія розвитку (генеза) досліджуваного об'єкта: конструкції аналогів та можливих частин, технології виготовлення, обсяги тиражування, мови програмування тощо.
При блочно-ієрархічному підході (приватному єврориtмі системного підходу, який використовується часто в техніці та програмуванні) процес проектування та уявлення про об'єкт розчленовується на рівні. На найвищому рівні використовується найменш детальне уявлення, що відображає найзагальніші риси та особливості проектованої системи. На кожному новому послідовному рівні розробки рівень подробиці розгляду зростає, у своїй система розглядається над цілому, а окремими блоками.
Методологія блочно-ієрархічного підходу базується на трьох концепціях: розбиття та локальної оптимізації, абстрагування, повторюваності.
Концепція розбиття дозволяє складне завдання проектування об'єкта чи системи звести до вирішення більш простих завдань з урахуванням їхнього взаємозв'язку.
Локальна оптимізація передбачає поліпшення параметрів усередині кожного простого завдання.
Абстрагованість полягає у побудові моделей, що відображають лише значущі в цих умовах властивості об'єктів.
Повторюваність- У використанні існуючого досвіду проектування.
Блочно-ієрархічний підхід дозволяє кожному рівні вирішувати завдання прийнятної складності. Розбиття на блоки має бути таким, щоб документація на будь-якому рівні була доступна для огляду та сприймання однією людиною.
Головним недоліком блочно-ієрархічного підходу і те, що у верхніх рівнях мають справу з неточними моделями об'єкта, і рішення приймаються за умов недостатньої інформації. Отже, при цьому підході висока ймовірність проектних помилок.

1.4. ЗАГАЛЬНОСИСТЕМНІ ПРИНЦИПИ СТВОРЕННЯ ПРОГРАМ

При створенні та розвитку програмного забезпечення (ПЗ) рекомендується застосовувати такі загальносистемні принципи:
1) принцип включення, що передбачає, що вимоги до створення, функціонування та розвитку ПЗ визначаються з боку більш складної, що включає його в себе системи;
2) принцип системної єдності, що полягає в тому, що на всіх стадіях створення, функціонування та розвитку ПЗ його цілісність забезпечуватиметься зв'язками між підсистемами, а також функціонуванням підсистеми управління;
3) принцип розвитку, що передбачає в ПЗ можливість його нарощування та вдосконалення компонентів та зв'язків між ними;
4) принцип комплексності, що полягає в тому, що ПЗ забезпечує пов'язаність обробки інформації як окремих елементів, так і всього обсягу даних в цілому на всіх стадіях обробки;
5) принцип інформаційної єдності, тобто у всіх, підсистемах, засобах забезпечення та компонентах ПЗ використовуються єдині терміни, символи, умовні позначення та способи подання;
6) принцип сумісності, що полягає в тому, що мова, символи, коди та засоби програмного забезпечення узгоджені, забезпечують спільне функціонування всіх підсистем та зберігають відкриту структуру системи в цілому;

7) принцип інваріантності, що визначає, що підсистеми та компоненти ПЗ інваріантні до оброблюваної інформації, тобто є універсальними або типовими.

1.5. ОСОБЛИВОСТІ ПРОГРАМНИХ РОЗРОБОК

Томас Кун 1977 р. визначив термін «парадигма» як зведення норм наукового мислення. Парадигма це правило (modus operandi) розвитку наукового знання. Воно протягом певного часу дає науковій спільноті модель постановки проблем та їх вирішення.
Коли та чи інша методологія застосовується під час стадії кодування (реалізації), часто її називають парадигмою програмування — способом мислення у програмуванні.
У програмуванні існують різні концепції мов (парадигми), які при написанні програм можуть призводити як до тих самих, і радикально різним підходам. Понад те, для низки мов необхідний «свій» тип мислення, особливі технології розробки, спеціальна школа навчання. Більшість програмістів використовують у роботі одну-дві мови програмування у межах однієї парадигми. Іноді програмісту буває важко зрозуміти чиюсь програму, реалізовану в незвичній йому парадигмі. На противагу зміні мети проекту під мову, що використовується, у ряді проектних випадків раціонально обрати іншу мову програмування.
Прийоми та способи програмування конкретного програміста визначаються мовою, що використовується. Часто осторонь залишаються альтернативні підходи до мети, отже, не використовуються оптимальні рішення у виборі парадигми, відповідної задачі. Нижче наведено список основних парадигм програмування разом із властивими їм видами абстракцій:
- Процедурно-орієнтовані - алгоритми;
- Об'єктно-орієнтовані - класи та об'єкти;
- Логічно-орієнтовані - цілі, виражені в обчисленні предикатів;
— орієнтовані правила — правила «якщо..., то...»;
— орієнтовані обмеження — інваріантні співвідношення;
- Паралельне програмування - потоки даних. Існують та інші парадигми. Чому ж їх стільки? Почасти тому, що програмування – порівняно нова
дисципліна, а частково через бажання людей вирішувати різні завдання. Крім того, найпопулярніша в даний момент комп'ютерна архітектура не є єдиною. В даний час проводиться велика кількість експериментів з машинами, що мають нестандартні архітектури, багато з яких розраховані на застосування інших парадигм програмування, наприклад, числа Фібоначчі. Загальна природа цифрових машин дозволяє з більшою чи меншою ефективністю моделювати одну архітектуру за допомогою іншої. З архітектур найбільш вдалі ті, у яких за рахунок апаратури та програмного забезпечення досягнуто найвищої швидкості та простоти використання.
Неможливо назвати будь-яку парадигму найкращою у всіх галузях практичного застосування. Наприклад, для проектування баз знань більш придатна парадигма, орієнтована правила. Об'єктно-орієнтована парадигма є найбільш прийнятною для кола завдань, пов'язаних з великими промисловими системами, в яких основною проблемою є складність.

1.6. СТАНДАРТИ І ПРОГРАМУВАННЯ

Стандарти давно використовуються в техніці та програмуванні. Створення складної системи неможливо без стандартів. Вони потрібні для боротьби з хаосом і плутаниною, але водночас стандарт не повинен бути надто «вузьким» і заважати технічному прогресу.
Державні стандарти відстежують тенденції розвитку програмування та дають обов'язкові рекомендації щодо їх дотримання. Крім державних стандартів (ДСТУ), діють галузеві стандарти (ОСТ), стандарти підприємств (СТП).
Група стандартів ДЕРЖСТАНДАРТ «Єдина система програмної документації» (ЄСПД) зазнала мало змін з моменту її створення, пережила кілька поколінь ЕОМ та революційних змін технологій розробки програм. При цьому вона досі ніколи не ускладнювала новацій.

Крім вищевикладених стандартів де-юре є стандарти де-факто. Ряд стандартів встановлюється де-факто провідними фірмами-розробниками програм та обчислювальної техніки. Стандарти де-факто з'являються з урахуванням ідей якоїсь широко відомої розробки. Вигідно робити продукти у стилі розробки якоїсь фірми, оскільки користувачі вже мають навички роботи з меню у стилі Lotus, електронними таблицями, текстовими редакторами. Зазвичай стандартом де-факто визначаються операційні системи, транслятори з мов програмування, організація файлів і середній рівень якості, що досягається після закінчення тестування програм. Конкретному розробнику вигідно дотримуватися таких стандартів.

У галузі програмування загальновизнаною провідною організацією з розробки стандартів є ANSI (Американський національний інститут стандартів). Цей інститут є лідером із встановлення стандартів мов програмування, кодових таблиць клавіш і символів, що виводяться на екран, та ще багатьох інших. Необхідно також наголосити на стандартах ISO. На жаль, найблагородніша справа стандартизації — досягнення загальної уніфікації та взаємозамінності — може стати гальмом розвитку. Вводячи новий стандарт, треба враховувати наслідки введення, особливо якщо стандарт є випереджаючим та випереджає практику розвитку або якщо стандарт є надто «вузьким» і гальмує еволюцію прогресу. В усьому світі керуються наступним ставленням до стандартів: або повністю за ними, або роби свій власний стандарт. Стандарти надають додаткові обмеження.

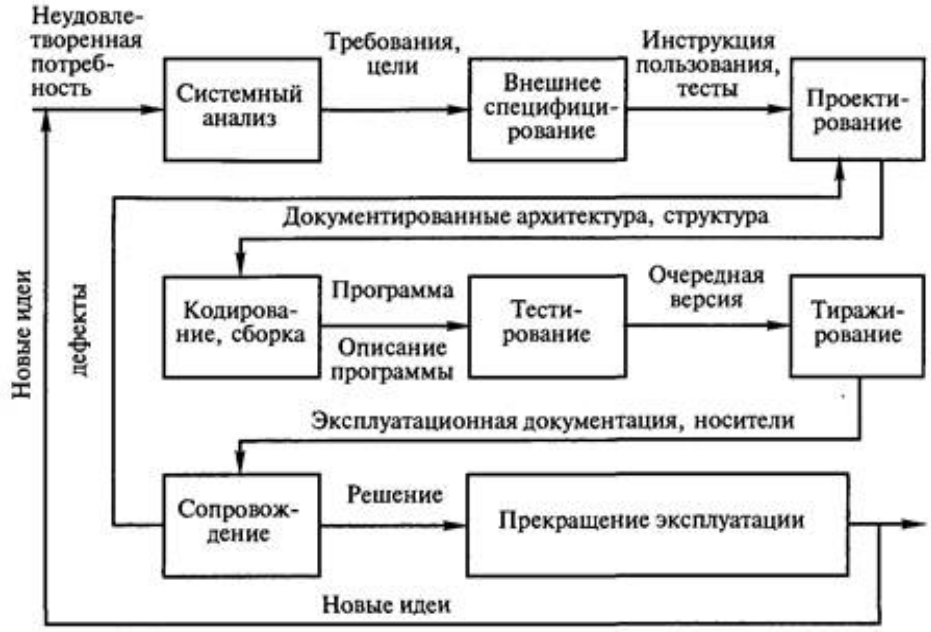
Програміст повинен вміти як використовувати готові стандарти, а й розробляти нові. Так, наприклад, правила однотипного оформлення вихідного тексту програми визначаються стандартом проекту, який може бути змінений на початку розробки нового проекту. Однак протягом виконання одного проекту оформлення всіх частин програми має бути однотипним. Тому найчастіше перед початком нового проекту конкретним програмістам слід розробляти свої стандарти, які не порушують ДСТУ, ОСТ та СТП та діють у межах конкретного проекту.

1.7. ОПИС ЦИКЛУ ЖИТТЯ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

Програми створюються, експлуатуються та розвиваються в часі. Як і будь-які штучні системи вони мають свій життєвий цикл.

*Життєвий цикл*сукупність взаємозалежних процесів створення та послідовної зміни стану продукції від формування до неї вихідних вимог до закінчення її експлуатації чи споживання.

Програми з мінімальною тривалістю життєвого циклу створюються для разового вирішення наукових та інших завдань. Їхній життєвий цикл — від кількох днів до кількох місяців. Раніше такі програми не мали зручного інтерфейсу, оскільки витрати на його розробку ще недавно в кілька разів перевершували витрати на розробку обчислювальної частини. Життєвий цикл програмних виробів показано на рис. 1.1.



Мал. 1.1. Життєвий цикл програмних виробів

Кожна програма починається з якоїсь незадоволеної потреби і, усвідомивши її, необхідно провести системний аналіз для виявлення цілей майбутнього програмного виробу та вимог до нього. Наступним етапом буде зовнішня специфікація, призначена для створення «ідеології» програми — загальної спрямованості у подальшому проектуванні, аж до зовнішнього вигляду програми та інструкції користування програмою. На етапі проектування програмний виріб специфікується у повному обсязі від постановки завдання до робочого проекту з описом внутрішньої структури програми та плану розробки частин програми. Потім відбувається кодування та тестування, внаслідок чого виходить готова версія програми. Програма випускається у тираж та супроводжується виробником. Супровід полягає як в усуненні помилок і випуску виправлених версій, що виявляються в процесі експлуатації, так і в удосконаленні базової версії програми, що часто призводить до перепроектування програми та випуску радикально оновлених версій. Закінчення життєвого циклу зумовлюється припиненням експлуатації розробки. Однак ідеї, висунуті в процесі експлуатації програми, зазвичай використовуються при розробці наступного, більш досконалого та сучасного виробу.
Припинення експлуатації зазвичай не одномоментний акт знищення програми в комп'ютері, а період часу, коли деякі організації або деякі користувачі ще продовжують використовувати стару розробку.

1.8. СТАДІЇ ТА ЕТАПИ РОЗРОБКИ ПРОГРАМ

ДСТУ 19.102-77 регламентує стадії та етапи програмних розробок протягом усього життєвого циклу. Даний стандарт сформувався на основі аналізу вдалих та невдалих програмних розробок та містить основні рекомендації щодо проведення нових розробок. Стандарт уже пережив кілька технологій програмування. При цьому, практично не змінюючись, він не був гальмом прогресу. Крім найменувань стадій та етапів проектування ДСТУ 19.102-77 фактично містить опис аналітико-синтетичного евроритму (алгоритма дій проектувальника з використанням методів аналізу та синтезу) за тимчасовими етапами проекту.
Стадія проекту- Одна з частин процесу створення програми, встановлена нормативними документами і закінчується випуском проектної документації, що містить опис повної, в рамках заданих вимог, моделі програми на заданому для даної стадії рівні або виготовленням

програм. По досягненні закінчення стадії замовник має можливість розглянути стан проекту та прийняти рішення щодо подальшого продовження проектних робіт. Наприклад, замовник може ухвалити рішення про продовження робіт за одним із узгоджених варіантів.

Етап проекту зазвичай частина стадії проекту, виділена з міркувань єдності характеру робіт і (або) завершального результату або спеціалізації виконавців. Іноді виділяють етапи (фази), що охоплюють кілька стадій. Наприклад, етап проектування програми включає стадії ЕП та ТП. Описи етапів регламентують порядок виконання окремих видів робіт задля досягнення стадії. Одні й самі види робіт можуть тривати низку етапів.

Стадії та етапи розробки програм за ДСТУ 19.102-77 на момент написання книги дано у додатку 1.

Програмний документ «Технічне завдання» (ТЗ) крім основних вимог до програмного виробу містить проект порядку взаємодії замовника та виконавця після закінчення конкретних етапів, тобто перелік необхідних стадій та етапів та вимог до їх виконання. ТЗ може одразу не встановлювати всіх вимог, які можуть бути уточнені та узгоджені із замовником на наступних стадіях. Проте сама можливість зміни вимог має закладатися у ТЗ. У ТЗ визначається стратегія виконання проекту.

"Ескізний проект" (ЕП), як правило, необхідний для розробки кількох альтернативних варіантів реалізації майбутнього виробу та уточнення вимог на основі їх аналізу. Ступінь опрацювання при цьому має бути достатньою лише для досягнення можливості порівняння варіантів.

"Технічний проект" (ТП) виконується для отримання однозначного опису кінцевого (оптимального) варіанта побудови програмного виробу та порядку його реалізації.

«Робочий проект» (РП) необхідний реалізації виробу відповідно до раніше наміченим планом.

Стадія "Впровадження" необхідна для розмноження програмної документації в потрібній кількості, навчання користувачів, допомоги в освоєнні програми, супроводження програми.

Науково-дослідницька робота (НДР) може бути самостійним етапом. НДР переважно проводиться виявлення останніх наукових досягнень з метою їх використання у проекті, перевірки реалізованості вироби та уточнення окремих його характеристик.

Відповідно до ДСТУ 19.102-77 допускається виключати стадію ЕП, а в технічно обґрунтованих випадках - стадії ЕП та ТП. Допускається об'єднувати, виключати етапи робіт та (або) їх зміст, а також запроваджувати інші етапи робіт за погодженням із замовником. Це дозволяє розумно збудувати проект конкретної розробки (хід проекту також є об'єктом проектування).

*приклад 1.*Розробка наукомісткої підпрограми може вестись за такими стадіями:

• ТЗ (ТЗ головне плюс ТЗ на окрему НДР);

• очікування на результати НДР, що виконується в іншій організації фахівцями-математиками (термін від місяця до декількох років);

• РП (близько місяця);

• Використання.

*приклад 2.*Потрібно розробити програмний виріб середньої чи великої складності. При середній складності виробу необхідне проведення ТП, а за великої складності - ЕП і ТП. На відміну від прикладу 1, у цьому випадку ТЗ може не містити закінчених вимог.

*приклад 3.*Потрібно створити програмні засоби, які автоматизують окремі види робіт. Розробка такого проекту може проводитись за такими стадіями:

• ТЗ;

• ЕП з НДР щодо дослідження існуючих програмних засобів, що автоматизують виконання окремих видів робіт;

• РП з розробки лише документації без реалізації будь-яких програм, якщо НДР показала, що можна обійтися лише існуючими програмними засобами;

• Використання.

*приклад 4.*Розробка таких інформаційних систем, як САПР або АСУ, повинна здійснюватися відповідно до відповідних стандартів. ТП САПР чи АСУ може містити технічні завдання розробки окремих програмних виробів. Зазвичай, такі ТЗ дуже конкретні. На етапі РП САПР або АСУ спочатку ведеться контроль над розробкою програмних виробів за всіма необхідними для цього стадіями розробки програмних виробів, потім проводиться спільна перевірка всіх розроблених програм.

Зазвичай підставою для укладення договору між замовником та виконавцем є гарантійний лист замовника. З гарантійного листа укладається договір. Обов'язковим додатком договору є ТЗ.

Деякі вітчизняні та зарубіжні джерела пропонують виділяти такі етапи:

1) аналіз вимог до системи (системний аналіз). (Зазвичай проводиться на основі первинного дослідження потоків інформації при традиційному проведенні робіт з фіксацією видів цих робіт та їх послідовності.);
2) визначення цілей, що досягаються розроблюваними програмами;
3) виявлення аналогів, які забезпечують досягнення подібних цілей, їх переваг та недоліків;
4) постановка задачі на розробку нових програм, визначення зовнішніх специфікацій (тобто описів вхідної та вихідної інформації, а іноді та їх форм) та способів (алгоритмів, методів) обробки інформації;
5) оцінка досягнення цілей розробки (Далі, за необхідності, етапи 1-5 можуть бути ітеративно повторені до досягнення задовільного вигляду виробу з описом виконуваних ним функцій та деякою ясністю реалізації його функціонування.);
6) розгляд можливих варіантів структурної побудови програмного виробу: або у вигляді кількох програм, або кількох частин однієї програми; результатом цієї роботи є варіанти архітектури програмної системи та (або) вимоги до структури окремих програмних компонентів; організація файлів для між програмного обміну даними;
7) розробка остаточного варіанта архітектури системи та розробка остаточної структури програмних компонентів;
8) складання та перевірка специфікацій модулів;
9) складання описів логіки модулів;
10) складання остаточного плану реалізації програм;
11) кодування та тестування окремих модулів та сукупності готових модулів до отримання готової програми;
12) комплексне тестування;
13) розробка експлуатаційної документації на програму;
14) проведення приймально-здавальних та інших випробувань;
15) коригування програм за результатами випробувань;
16) остаточне здавання програмного виробу замовнику;
17) тиражування програмного виробу;
18) супровід програми.

Сучасні технології проектування програмного забезпечення (ПЗ) спрямовані на часткову автоматизацію етапів і суміщення їх у часі з метою скорочення термінів виконання проектів. У літературних джерелах застосовуються найменування етапів, які охоплюють ряд наведених етапів і за часом охоплюють кілька стадій. Наприклад, етап розробки програми.

1.9. ТИПОВІ ПОМИЛКИ НАВЧАЛЬНИХ ПРИ СКЛАДАННІ ТЕХНІЧНОГО ЗАВДАННЯ

У додатку 2 наведено приклад виконання навчального технічного завдання. У прикладі опущені: лист затвердження, титульний лист та додатки.

Головним, що відрізняє одне ТЗ від іншого, є сенс вимог. Усі вимоги оформлюються пропозиціями з використанням оборотів «має», «потрібно забезпечити», «необхідно виконати». Рекомендується оформляти вимоги у формі нумерованих абзаців. Це дозволить давати на них посилання.

Типовими помилками є написання неконкретних вимог, які нікого нічого не зобов'язують. Вимоги не повинні бути суперечливими. Тому дублювання тих самих за змістом вимог різними словами в різних місцях тексту часто призводить до неможливості реалізації виробу через неоднозначність їх розуміння замовником та виконавцем.
Нижче наведено типові помилки учнів при написанні вимог до функціональних характеристик:
• нерозуміння терміну «функціональні характеристики» (Сенс цього терміну характеризується такою пропозицією: «Проектований завод у нормальному режимі роботи повинен забезпечити випуск за одну 8-годинну зміну не менше 40 просапних тракторів». Ця фраза містить призначення проектованого об'єкта — заводу, так і те, що випускає завод і при яких обмеженнях стосовно програм, для правильного написання вимог до функціональних характеристик необхідно сторонніми очима майбутнього користувача розглянути, що робить корисний програмний виріб і при яких обмеженнях.);
• опис вимог до функціональних характеристик загального, універсального об'єкта (якщо за конкретними вимогами можна реалізувати цілий спектр виробів, вони не конкретні);
• написання вимог, що свідомо не реалізуються.
У наведеному у додатку 2 прикладі виконання навчального технічного завдання відсутні програми. Саме технічне завдання містить вимоги до дуже простого програмного виробу, тому ряд розділів технічного завдання написані як складнішого виробу.

1.10. МОДЕЛЮВАННЯ І ПРОГРАМУВАННЯ. ПОНЯТТЯ СПЕЦИФІКАЦІЙ

Один об'єкт чи система може у ролі моделі іншого об'єкта чи системи, якщо з-поміж них встановлено схожість у сенсі. Модель системи (або будь-якого іншого об'єкта або явища) може бути формальний опис системи, в якому виділені основні об'єкти, що становлять систему, і відносини між цими об'єктами.
Побудова моделей - поширений спосіб вивчення складних об'єктів і явищ. У моделі опущені численні деталі, що ускладнюють розуміння. Моделювання широко поширене у науці, техніці та програмуванні.
Моделі допомагають перевірити працездатність системи, що розробляється на ранніх етапах її розробки; спілкуватися із замовником системи, уточнюючи його вимоги до системи; вносити (у разі потреби) зміни у проект системи (як на початку її проектування, так і на інших фазах її життєвого циклу); передавати проект іншим виконавцям, наприклад кодувальникам, оскільки сам проект є моделлю проектованої програми.
Збираючись зробити прибудову до будинку, ви, мабуть, не почнете з того, що купите купу дощок і збиватимете їх разом різними способами, поки не отримаєте щось приблизно підходяще. Для роботи вам знадобляться проекти та схеми, так що ви швидше за все почнете з планування та структурування прибудови. Зверніть увагу, що ваш результат буде довговічнішим. Ви не хочете, щоб робота зруйнувалася від невеликого дощу.
У світі програмного забезпечення те саме роблять для нас моделі. Вони складають проекти систем. Проект будинку дозволяє планувати його до початку безпосередньої роботи, модель дозволяє спланувати систему до того, як ви приступите до її створення. Це гарантує, що проект вдасться, вимоги буде враховано і система зможе витримати ураган наступних змін.
Будь-які досить великі програми є складними системами. Проблема складності долається шляхом декомпозиції задачі. Базова парадигма у підході до будь-якої великої задачі ясна: необхідно «розділяти та панувати».
У разі декомпозиції використовується абстрагування для отримання абстрактних моделей. Абстракція - уявне відволікання, відокремлення від тих чи інших сторін, властивостей або зв'язків предметів і явищ для виявлення суттєвих ознак. Абстрагування від проблеми передбачає ігнорування ряду подробиць для того, щоб звести завдання до більш простого завдання.
Мозок людини оперує даними через асоціації, створюючи павутину з ланцюжків, у яких залучені клітини мозку. Характерною особливістю людського мислення є мала, без

особливих тренувань можливість абстрагування від предметів навколишнього світу, т. е. якщо людина справляє дію, він зазвичай робить його над конкретним предметом.
Окремі види абстракцій визначають найбільш ефективний спосіб декомпозиції стосовно конкретних цілей. Правильно впізнавши абстракції, можна отримати моделі, доступні розуміння як окремими людьми, і колективом учасників проекту.
Завдання абстрагування та подальшої декомпозиції типові для процесу створення програм. Декомпозиція використовується для розбиття програми на компоненти, які можуть бути об'єднані, дозволяючи вирішити основне завдання. Абстрагування ж передбачає продуманий вибір моделей майбутніх компонентів.
Кожна стадія проекту завершується затвердженням програмних документів. Виняток може становити розробка нехитрих програм з коротким (до півроку) життєвим циклом та з трудомісткістю не більше одного людино-місяця. Повне документування таких програм є економічно недоцільним. Програмна документація складається з урахуванням розроблених у ході проекту специфікацій.
Під специфікацією розуміється досить повний і точний опис розв'язуваної задачі на етапах проекту. Специфікація є моделлю проектованого об'єкта (програми).
Специфікація може описувати угоду між програмістами та замовниками (користувачами). Програміст береться написати програму, а користувач погоджується не покладатися знання у тому, як саме ця програма реалізована, т. е. не припускати нічого такого, що було зазначено у специфікації. Така угода дозволяє розділити аналіз реалізації від використання програми. Специфікації дають можливість створювати логічні засади, що дозволяють успішно «розділяти та панувати».
Складність проектованих систем призвела до створення спеціальних абстрактних мов, графічних нотацій і автоматизованих систем, що підтримують їх, що полегшують процес створення специфікацій.
Первинні специфікації становлять термінах розв'язуваної завдання, а чи не програми. У ході виконання проекту специфікації послідовно зазнають змін до програмних документів стадій і до документації, яка необхідна для експлуатації та супроводу програми.
Вже в первинних специфікаціях можна виділити дві частини: функціональну та експлуатаційну.
*Первинна функціональна специфікація*насамперед описує:
- Об'єкти, що беруть участь у завданні (що робить програма і що робить людина, яка працює з цією програмою);
- процеси та дії - евроритми для людини, алгоритми методів вирішення задачі в машині із зазначенням суті та порядку обробки інформації (з займаною інформацією та програмою розміром оперативної пам'яті);
- Вхідні та вихідні дані, їх організацію (наприклад, сценарій діалогу з екранними формами, організація файлів із зазначенням довжин полів записів та граничної кількості інформації у файлах).
Тобто, спочатку фіксуються зовнішні функціональні специфікації, а потім і внутрішні.
Загалом зовнішні функціональні специфікації включають:
- Опис того, що робить програма;
- визначення, що робить людина, а що машина (за якими евроритмами працює людина, звідки вона бере інформацію і як її готує до введення в ЕОМ);
- Специфікації вхідних та вихідних даних;
- Реакції на виняткові ситуації.
Тут бажана розробка інструкції користування майбутньою програмою, тобто все, що побачив би користувач, отримавши готову програму. До тестування добре складених зовнішніх специфікацій можна залучити потенційних користувачів до розробки внутрішніх специфікацій. Користувачеві можна показувати макети екранів у порядку виконання програми, а користувач може готувати дані для тестування всіх функцій програми та зможе

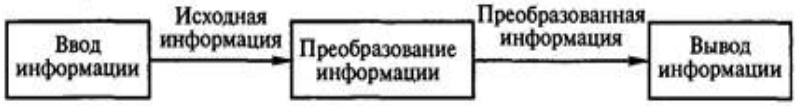
апробувати методику роботи з програмою. Зовнішні специфікації зазвичай фіксуються в ТЗ чи ЕП, але можуть бути уточнені у ТП.

До внутрішніх специфікацій відносяться описи складу внутрішніх частин програми, опис їх взаємозв'язку, а також внутрішні функціональні специфікації.

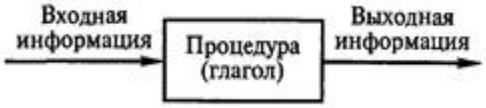
Внутрішні функціональні Специфікації включають описи алгоритмів як усієї програми, так і її частин з урахуванням специфікації внутрішніх даних програми (змінних, особливо структурованих).

Наведені далі абстракції процедури, даних та об'єктів лежать в основі багатьох методів розробки програмного забезпечення. Загалом будь-яку програму можна надати набором процедурних абстракцій (рис. 1.2). Аналізуючи рис. 1.2 можна отримати узагальнену абстракцію процедури, зображену на рис. 1.3. Абстракції процедур найповніше втілилися у технології структурного програмування.

Розділяючи в програмі тіло процедури та звернення до неї, мова високого рівня реалізує два важливі методи абстракції: абстракцію через параметризацію та абстракцію через специфікацію.



*Мал. 1.2.*Абстракція програми як набору процедур, що обробляють дані



*Мал. 1.3.*Анотація процедури

Абстракція через параметризацію дозволяє, використовуючи параметри, уявити фактично необмежений набір різних обчислень однією процедурою, що є абстракція всіх цих наборів. Наприклад, необхідна процедура сортування масиву цілих чисел А. При подальшій розробці програми можливе виникнення потреби у сортуванні іншого масиву, але з іншим ім'ям. Використання абстракції через параметризацію узагальнює процедуру сортування та робить її більш універсальною.

Абстракція через специфікацію дозволяє абстрагуватися від процесу обчислень, описаних у тілі процедури, рівня знання лише те, що дана процедура має у результаті реалізувати. Це досягається шляхом завдання для кожної процедури специфікації, що описує ефект її роботи, після чого зміст звернення до цієї процедури стає зрозумілим через аналіз цієї специфікації, а не самої процедури.

При аналізі специфікації для з'ясування сенсу звернення до процедури слід дотримуватися наступних двох правил.

Правило 1.Після виконання процедури вважатимуться, що виконано кінцеве умова.

Виконання кінцевої умови за дотримання початкових умов — це те, заради чого і побудована процедура. Якщо це процедура пошуку максимального значення масиві, то кінцева умова — факт, що максимальний елемент знайдено. Якщо це процедура обчислення квадратного кореня, то кінцева умова знаходження квадратного кореня.

Правило 2 Можна обмежитися лише тією інформацією, яку передбачає кінцева умова.

Ці правила демонструють дві переваги абстракції через специфікацію. Перше у тому, що програмісти, використовують цю процедуру, нічого не винні знайомитися з текстом її тіла. Отже, їм не потрібно усвідомлювати, наприклад, подробиці алгоритму відшукання

квадратного кореня, встановлюючи, чи справді повернутий результат — кількість, яку шукає.

Друге правило показує, що насправді маємо справу з абстракцією: абстрагуючись від тіла процедури можна не звертати уваги на несуттєву інформацію. Саме таке «ігнорування» інформації та відрізняє абстракцію від декомпозиції. Звичайно, аналізуючи тіло процедури, можна витягти деяку кількість інформації, яка не випливає з кінцевої умови (як, наприклад, те, що знайдений елемент перший або останній у розглянутому вище прикладі). У специфікації подібна інформація про результат, що повертається, відкидається.

Абстракції через параметризацію та через специфікацію є потужним засобом створення програм. Вони дозволяють визначити два види абстракцій: процедурну абстракцію та абстракцію даних. У загальному випадку кожна процедурна абстракція та абстракція даних використовують обидва способи.

Наприклад, абстракцію SQRT (витяг квадратного кореня) можна порівняти з операцією: вона абстрагує окрему подію або завдання. Ми будемо посилатися до таких абстракцій, як до процедурних абстракцій. Зазначимо, що абстракція SQRT включає як абстракцію через параметризацію, так і абстракцію через специфікацію.

На процедурній абстракції засновано розробку структури програми в технології структурного програмування. Базова компонента технології структурного програмування — модуль, якому зазвичай відповідає підпрограма (процедура чи функція мовами програмування високого рівня).

Розглядаючи програму не як набір процедур, а передусім деякі набори даних, кожен із яких має дозволену групу процедур, отримуємо абстрактне уявлення програми, представлене на рис. 1.4. Аналіз рис. 1.4 дозволяє отримати абстракцію даних, показану на рис. 1.5.

У технології абстрактних даних Дейкстри застосовується функціональна модель як набору діаграм потоків даних (далі — ДПД; DFD — Data Flow Diagram), які описують зміст операцій та обмежень. ДПД відображає функціональні залежності значень, що обчислюються в системі, включаючи вхідні значення, вихідні значення та внутрішні сховища даних. ДПД - це граф, на якому показано рух значень даних від їх джерел через процеси, що їх перетворюють, до їх споживачів в інших об'єктах. Фрагменти ДПД показано на рис. 1.6.

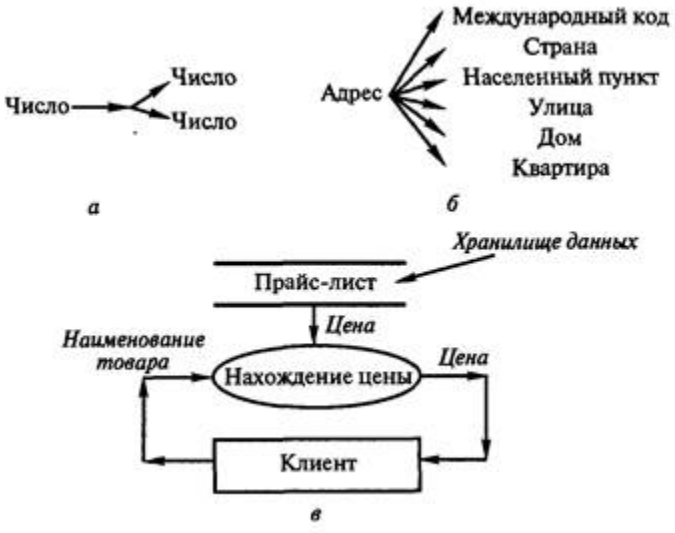
Мал. 1.4. Абстракція програми як набору даних, що обробляються процедурами

*Мал. 1.5.*Абстракція даних

Потік даних з'єднує вихід об'єкта (або процесу) із входом іншого об'єкта (або процесу). Він представляє проміжні дані обчислень.

Сховище даних- Це пасивний об'єкт у складі ДПД, в якому дані зберігаються для подальшого доступу. Сховище даних допускає доступ до даних у порядку, відмінному від того, в якому вони були туди поміщені. Агрегатні сховища даних, як, наприклад, списки та таблиці, забезпечують доступ до даних у порядку їх надходження або за ключами.

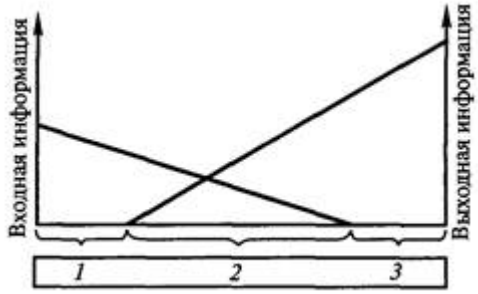
ДПД показує всі шляхи обчислення значень, але з показує, у порядку ці значення обчислюються. Рішення про порядок обчислень пов'язані з керуванням програмою, що відображається у динамічній моделі. Ці рішення, що виробляються спеціальними функціями, або предикатами, визначають, чи буде виконано той чи інший процес, але при цьому не передають процесу жодних даних, тому їх включення до функціональної моделі необов'язково. Тим не менш, іноді буває корисно включати зазначені предикати у функціональну модель, щоб в ній були відображені умови виконання відповідного процесу.



*Мал. 1.6.*Фрагменти діаграми потоків даних (ДПД):

а- Копіювання даних (числа); б – розщеплення даних; в – активний об'єкт «Клієнт» за допомогою операції «Знаходження ціни» працює зі сховищем «Прайс-лист»

Дейкбуд запропонована відносно технологія, що рідко застосовується, заснована на абстракції даних. Ця технологія є альтернативою структурному програмуванню. У чистому вигляді вона успішно застосовувалася розробки СУБД та інших виробів, орієнтованих перетворення інформації з однієї форми на іншу.
Якщо структурному програмуванні головними є функції та процедури (дії), то технології абстрактних даних Дейкстри на чолі ставляться дані.
За цією технологією спочатку дуже ретельно специфікуються вихід, вхід, проміжні дані; велика увага приділяється типізації даних з використанням структур для об'єднання близької за змістом інформації у єдині дані. Зазвичай розписується схема ієрархії даних. Тут зручно застосовувати моделі як діаграм потоків даних.
Нижче наводиться опис методики отримання програми з раціональною структурою даних, яка ґрунтується на абстракції даних Дейкстри. Ця методика ставить на перше місце дані, що з деякими труднощами забезпечується методикою розподілу програми на смислові модулі шляхом виділення смислових підфункцій у технології структурного програмування.
*Крок 1*Грунтуючись на потоці даних у задачі, виділіть 3-10 значеннєвих частин обробки даних.
*Крок 2*Визначте головний вхідний та вихідний потоки даних завдання.
*Крок 3*Простежте, як слід вхідний потік від частини до частини, від входу до кінця обробки знайдіть цю точку. Простежте від кінця до початку, отже вихідний потік; знайдіть абстрактну точку, де з'явився (рис. 1.7).
Знайдені точки поділяють завдання на дві чи три найбільш незалежні (за даними) частини.
*Крок 4.*Подайте незалежні частини підпрограмами та визначте їх функції. Ці підпрограми будуть підпорядкованими до модуля, розбиття якого виконується.
*Крок 5.*Визначте сполучення підпрограм за даними.



*Мал. 1.7.*Перетворення головного вхідного потоку інформації у вихідний потік:

1 - ділянка відповідає перетворення інформації вхідного потоку на проміжну інформацію; 2 — ділянка відповідає отриманню вихідної та проміжної інформації з вхідної та проміжної інформації; 3 - ділянка відповідає отриманню вихідної інформації з проміжної інформації

Сучасна об'єктно-орієнтована технологія, що витісняє технології структурного програмування та абстракції даних, поєднує в собі абстракції процедур і даних у новій абстракції — об'єкті (рис. 1.8). Поняття абстракції даних розширено доти, що як внутрішні дані, і код процедур розглядаються як новий тип даних — об'єкт.
Об'єктна модель базується на двох постулатах: є об'єкти; об'єкти взаємодіють передачею повідомлень.
Методи структурного проектування допомагають спростити процес розробки складних систем з допомогою використання алгоритмів як готових будівельних блоків. Об'єкт - більший будівельний блок. Він може включати як дані (поля), так і процедури (методи). Узбуйнення будівельних блоків стало необхідністю при створенні великих програм.
При процедурному програмуванні акцент робиться на обробці (алгоритмі), яка потрібна на виконання необхідних обчислень. Парадигма: вирішу, які потрібні процедури; використовуй найкращі доступні алгоритми.

*Мал. 1.8.*Абстракція об'єкту

Парадигма програмування на основі абстрактних даних Дейкстри: визнач організацію даних та вияви всі стани значень даних, вважай, що процедури – це щось, що змінює дані.
У структурному програмуванні основний структурної одиницею є модуль (The module). Парадигма: розбий програму на систему ієрархічно підлеглих модулів (процедур) так, щоб забезпечити максимальну якість тестування під час виконання розробленого плану реалізації програми. Важливим є те, в якому з модулів та на якому рівні ієрархії модулів описувати ті чи інші дані. p align="justify"> При структурному підході інформаційні потоки протікають в одному напрямку: від вихідних даних до результату.

Розміщений в окремому файлі набір пов'язаних процедур разом із даними, які вони обробляють, називають програмною одиницею (Unit). Часто слово "Unit" перекладають як модуль. Так виник термін «модульне програмування». У модульному програмуванні акцент змістився від проектування процедур у бік організації даних. Крім іншого, це було відображенням факту збільшення розмірів програм. Парадигма: розв'яжи, які потрібні модулі; розбий програму так, щоб приховати дані в модулях.
В об'єктно-орієнтованому програмуванні абстракція даних є основним аспектом якісного проектування. Парадигма: програма представляється набором об'єктів, які, взаємодіючи один з одним за допомогою повідомлень, змінюють себе та навколишні об'єкти. Принцип модульного програмування використовується в об'єктно-орієнтованому програмуванні. Зазвичай, у одному Unit описується або один клас, або кілька класів, успадкованих від одного класу. Механізми Unit реалізують приховування інформації. Об'єктно-орієнтований підхід характеризується різноспрямованістю та різнорідністю інформаційних потоків.

1.11. МНЕМОНІКА ІМЕН У ПРОГРАМАХ

Запропонована методика складання імен (ідентифікаторів) носить рекомендаційний характер. Для використання цієї методики у конкретному проекті необхідна її адаптація. Складені відповідно до методики імена можна використовувати в програмах для іменування констант, змінних, типів, процедур, об'єктів, файлів і т. д. Після адаптаційної переробки методика може стати складовою стандарту проекту. Імена, що використовуються у програмних продуктах, повинні:
- відповідати призначенню (з імені має однозначно слідувати його призначення та, навпаки, з призначення - його ім'я);
— мати впізнаваність (ця властивість імені дозволяє покращити читання вихідних текстів програм);
— забезпечувати запам'ятовуваність (ім'я необхідно легко запам'ятати для того, щоб щоразу не повертатися до документації або тексту програми, в якому це ім'я визначено);
- бути короткими (надто довгі імена не запам'ятовуються і, незважаючи на підвищену впізнаваність порівняно з короткими іменами, ускладнюють читання вихідного тексту програми);
— мати унікальність (як наслідок, «відповідати призначенню»: імена мають складатися таким чином, щоб у всій системі не було двох однакових глобальних імен).
Звичайно, хотілося б домогтися одночасного виконання всіх викладених вище вимог у сукупності, але оскільки вимога стислості суперечить вимогі «володіти впізнаваністю», а іноді й «забезпечувати запам'ятовуваність», необхідно знаходити оптимальний компроміс у дотриманні всіх вимог.
В ідеальному випадку імена не слід запам'ятовувати: їх потрібно складати таким чином, щоб кожен раз, знаючи, для якого об'єкта складаєте ім'я, ви приходили б до одного і того ж варіанта імені.
Текст даних рекомендацій не лімітує використання великих і малих літер, спосіб поділу слів і те, які слова використовувати в іменах, - додаткові обмеження накладає конкретну мову програмування. Також не розглядається відповідний для конкретної мови програмування символ роздільника слів.
Імена констант і змінних відносяться до даних, а імена даних утворюються від іменників. Імена процедур повинні бути активними, тобто базуватися на активному дієслові, за яким слід іменник. Імена об'єктів зазвичай утворюються від іменників, але в поодиноких випадках можуть включати дієслова та прикметники. Повне ім'я методу складається з імені об'єкта, якому належить метод, символ «.» роздільника та власне імені методу об'єкта. Для таких повних імен дуже важко досягти стислості. Метод є процедурною абстракцією, та її власне ім'я утворюється від дієслова.
Отже, ім'я складається із слів. Нехай довжина імені – це кількість слів, використаних у цьому імені. Ім'я А є батьківським стосовно імені Б, якщо довжина імені Б більша, ніж довжина

імені А, і перші (ліворуч) слова імені Б збігаються зі словами імені А в тому самому порядку. Ім'я А можна розглядати як загальний префікс для імен групи Б. Наприклад, ім'я debug є батьківським для імен debug_info, debug_mode, debug_log, debug_error_get, debug_error_set тощо. Ім'я debug_error, у свою чергу, є батьківським для debug_error_get.
Ім'я А є дочірнім по відношенню до імені Б, якщо ім'я Б є батьківським по відношенню до імені А.
Імена належать одній групі, якщо ці імена мають однакову довжину та одного спільного предка А. Довжина імені А на одиницю менша за довжину імен цієї групи. У такому разі А буде ім'ям цієї групи. Наприклад, імена debug_error_get, debug_error_set є іменами групи debug_error. Імена debug__info, debug_mode, debug_log та debug_error, у свою чергу, є іменами групи debug.
Нехай потужність групи А – загальна кількість імен у цій групі. Префікс імені - це слово, яке записується найпершим у імені та не враховується при визначенні довжини, спорідненості та приналежності до групи. Префікси використовують, наприклад, для вказівки типів змінних чи полів: i_count, b_valid, is_protected. i, b, is префікси.
Вимоги ієрархічної організації імен можуть частково порушуватись або взагалі не використовуватися при складанні локальних імен (імен для локальних змінних, імен полів таблиць, імен властивостей та методів об'єктів тощо). Однак у випадку, якщо локальних імен багато, є сенс застосовувати ці вимоги і до локальних імен.
Якщо ім'я А є дочірнім по відношенню до імені Б, ім'я Б є позначенням деякого об'єкта. Це означає, що всі слова імені, крім останнього імені, можуть бути утворені тільки іменниками. Тільки останнє слово в імені може бути іменником, дієсловом або прикметником. Це правило, однак, іноді може порушуватися. Наприклад, є певна дія та набір глобальних налаштувань (констант), які контролюють цю дію. У такому разі в іменах цих констант передостаннім словом буде дієслово.
У деяких випадках імена об'єктів можуть бути дієсловами. Наприклад, підсистему очищення бази даних було б логічно назвати слово «clear» (очистити). У такому разі дієслово «очистити» стоятиме в середині імені.
*приклад 1.*change_user_password – погане ім'я. Перше слово - дієслово і воно не може позначати ім'я об'єкта. Крім того, може виявитися, що для кожного користувача в системі зберігається кілька паролів, наприклад, пароль для доступу до свого облікового запису (account) та пароль для входу в чат. У такому випадку у двох різних місцях може знадобитися ввести дві функції (змінити пароль для доступу до account та змінити пароль для входу в чат) з однаковими іменами, що суперечить пункту «відповідати призначенню» загальних вимог до імен.
*приклад 2.*passport_password_change або passport_password_change — добрі імена. У системі є підсистема керування обліковими записами користувачів, звана passport. Частина цієї підсистеми, що займається керуванням паролями, називають passport_password. Одну з функцій цієї частини – зміна пароля – називають passport_password_change.
Довжина імені має бути мінімальною. Не використовуйте в зайвих іменах слів. Кожне слово, використане в імені, має означати конкретний об'єкт, якому належить це ім'я чи конкретну дію чи властивість, якому відповідає це ім'я. Імена об'єктів, дій та властивостей, у свою чергу, повинні складатися з імен, довжина яких дорівнює одному слову.
*приклад 1.*ConvertIntegerDateToSQLStrDate - погане ім'я. Як ви вважаєте, чи згадаєте ви його з точністю до символу через день?
Слова "Convert" і "To" найчастіше можна взагалі опустити, оскільки очевидно, що якщо є два формати дати, то, отже, відбувається перетворення з одного формату на інший.
Слово "Date" повторювати двічі не потрібно, оскільки і вихідне дане, і результат є датою.
Усі SQL-запити у програмі – це рядки. Тому слово "Str" - зайве.
*приклад 2.*IntegerDateSQL - прийнятне ім'я. Існує підсистема керування датами, і ця функція конвертує дату, представлену у вигляді цілого числа в рядок, який можна використовувати

SQL-запиті. Недоліком цього імені є відсутність активного дієслова. Порівняйте це ім'я із вихідним варіантом прикладу 1.
У системі може з'явитися група імен потужністю 1.
*приклад 1.*PASSPORT_DEAD_REMOVE_TIMEOUT - погане ім'я, якщо з померлими користувачами (так на сайті названі користувачі, які занадто довго не з'являються) не можна робити жодних інших операцій, крім видалення.
*приклад 2.*PASSPORT_DEAD_TIMEOUT – гарне ім'я.
*приклад 3.*passport_is_login_valid – погане ім'я. Слово "is" - зайве.
*приклад 4.*passport_login_valid – гарне ім'я. Є підсистема керування обліковими записами passport. У ній є частина, яка займається керуванням логінами користувачів passport_login. Функція passport_login_valid перевіряє, чи є логін правильним.
У деяких випадках, однак, можуть бути створені групи довжиною 1, наприклад, якщо передбачається, що до цієї групи в майбутньому будуть додані нові імена.
Скорочення у словах у випадку неприпустимі. Якщо використовуєте в іменах слова зі скороченнями, то може скластися ситуація, коли довго згадуватимете, яким саме способом скоротили це слово і чи скорочували його взагалі. Тим більше, що те саме слово можна скоротити різними способами.
Це стосується і використання множини іменників, дієслів у другій і третій формі і т. п. Скрізь, де можливо, потрібно використовувати початкову форму слова, для того щоб уникнути різночитання.
Крім того, якщо допускаєте різні скорочення (або будь-яку іншу плутанину з формами одного і того ж числа), то може виявитися, що з призначення імені не випливає однозначно саме ім'я через те, що не виконується пункт «відповідність до призначення» загальних вимог до імен.
Слова не в початковій формі можуть бути використані тільки в тому випадку, якщо вони використовуються багато разів і при цьому у всіх місцях однаково.
Наслідок: як останнє слово імені може бути використане тільки загальноприйняте скорочення (або початкова форма), яке СКРІЗІ (і багато разів) у програмі використовується саме в такому варіанті. Якщо скорочення використовується рідко, краще використовувати початкову форму слова.
*приклад 1.*StrToFloat – погане ім'я. У деяких мовах програмування є зарезервоване слово «String». У такому разі виходить, що в деяких випадках до рядків звертаємося на повне ім'я, а в деяких — на скорочення. Проте скорочення "Str" - загальноприйняте в системах програмування фірми "Borland". При використанні цих систем, але не в SQL запитах, таке скорочення резонно використовувати, і ім'я StrToFloat стає хорошим ім'ям.
*приклад 2.*StringToFloat - хороше ім'я (якщо не враховувати наявність зайвого слова "To", але "Te" добре показує, що це ім'я процедури конвертора типів).
*приклад 3.*mp_pagelist — хороше ім'я, якщо в групі «mp» багато імен або це скорочення використовується в такому ж написанні і такому ж сенсі багато разів в інших іменах (mp — скорочення від «main page»).
*приклад 4.*PASSPORT_PASSWORD_LENGTH_MIN - хороше ім'я, скорочення в кінці - загальноприйняте і скрізь у системі використовується саме в такому варіанті написання.
*Приклад 5.*TPassportPrivileges — погане ім'я для таблиці, де зберігається список привілеїв. Вочевидь, що у таблиці зберігається багато всяких привілеїв, і множина у разі зайве.
*Приклад 6.*TPassport_Privilege — хороше ім'я, проте якби не йшлося про бази даних, префікс «T» відповідав би типу, а не змінній.
Не допускається використання префіксів без особливої необхідності.
Випадки, у яких використання префіксів виправдане:
• якщо це імена змінних, а імена типів змінних, можна використовувати префікс «T»;
• якщо імена ваших сутностей перемішуватимуться зі сторонніми іменами;
• у разі локальних імен.

Імена, які використовуються в обмеженому контексті, можуть бути дуже короткими. Традиційно імена i та j використовуються для позначення лічильників, p і q – для покажчиків, s – для рядкових, а ch – для літерних змінних. Ці традиційні найкоротші імена можуть відповідати префіксам, що пояснюють тип змінних.
*приклад 1.*is_passport_privilege_valid – погане ім'я. Префікс у глобальному імені зайвий.
*приклад 2.*passport_privilege_valid – гарне ім'я.
*приклад 3.*i_order - хороше ім'я для поля в таблиці, що вказує на порядок чогось. Префікс «i» характеризує цілий тип.
Додаткові рекомендації щодо складання імен:
• не починайте і не закінчуйте імена символом підкреслення;
• не використовуйте імена, що складаються лише з малих літер (виключення становлять імена констант та макровизначень);
• не слід в одній і тій же програмі використовувати імена, що відрізняються лише написанням літер — малої або великої;
• залежно від можливостей мови програмування можна розділяти частини імен символом підкреслення або написанням великої літери чергової частини імені;
• використання малих літер на початку кожного слова імені ускладнює трансляцію тексту програми з однієї мови програмування на низку інших мов.
Ряд імен, що розуміються, важко ретельно коментувати. Такий коментар краще наводити праворуч від опису імені.
При описі логічної організації змінних, файлів чи класів слід застосовувати додаткові коментарі.
Рефакторинг(Від англ. Refactoring) - Оптимізація, поліпшення реалізації програми без зміни її функціональності.
Стосовно вже кимось або колись написаних програм може здійснюватися реакторинг імен, структури даних програми, структури програми та коду. Одночасно з рефакторингом коду може бути здійснений і рефакторинг опису алгоритму природною мовою.
Щодо імен під рефакторингом розуміється зміна імен таким чином, щоб вони відповідали новим вимогам. Імена – це дуже важлива частина програми. Багато програмістів схильні применшувати значущість імен.
Незрозумілі імена — це нечитана програма, а програму, що не читається, важко супроводжувати.
Розглянемо випадки, у яких може знадобитися рефакторинг імен:
Випадок 1- Зміна правил. Можна скласти різні правила освіти імен і слідувати спочатку одним правилам, потім, уточнивши ці правила, слідувати іншим. Незмінним має залишатися лише одне правило: всі імена у проекті мають бути побудовані за одними правилами.
*Випадок 2*частковий рефакторинг імен. Якщо з будь-яких причин змінили правила складання імен, слід оновити всі імена, які не підходять під нові правила.
*Випадок 3*неможливість однозначного передбачення майбутнього. Будь-який проект розвивається. На етапі створення може виявитися, що спроектована структура не повна або спочатку передбачалася одна структура, а потім стала очевидною інша, краща структура.
*Випадок 4*рефакторинг імен, на вашу думку, не потрібен: ви вже закінчуєте роботу над проектом, не плануєте в майбутньому його підтримувати і вам все одно, що про вас подумають люди, яким доведеться розбиратися у вашому коді.
Кваліфіковані програмісти витрачають деякий час на очищення свого коду для простоти подальшого використання. Ясність коду визначається ясністю імен даних, зрозумілістю призначення та послідовності дій, ясністю імен процедур та об'єктів.

1.12. ПРОБЛЕМА ТИПОВИХ ЕЛЕМЕНТІВ У ПРОГРАМУВАННІ

Під типовими елементами, чи «кубиками», розуміються якісь окремо виготовлені типові частини, у тому числі можна було збирати безліч програм. Проблема «кубиків» притаманна як програмуванню, й у різних галузях вона проявляється по-різному.

Область машинобудування поруч із такими «кубиками», як болти, гайки, оперує безліччю нетипових елементів. Наприклад, ліве крило і праве крило автомобіля хоч і дуже схожі один на одного, але не можуть бути взаємозамінними і можуть використовуватися лише в конкретній моделі автомобіля. У галузі машинобудування значні зусилля проектувальників витрачаються проектування елементів. Кількість елементів, з яких складаються конструкції, зазвичай не перевищує декількох сотень.

Найбільш повно вирішено проблему «кубиків» у галузі радіоелектроніки. Резистори, ємності, лампи, транзистори, мікросхеми, ряди функціональних блоків є стандартизованими та взаємозамінними. У цій галузі проектувальники вирішують завдання синтезу штучних систем із десятків і навіть сотень тисяч елементів.

Програми є найскладнішими штучними системами, у яких загальна кількість елементів (операторів) може сягати кількох мільйонів. Нові технології програмування використовують дедалі нові, зазвичай, більші, типові елементи побудови програм.

Першими укрупненими типовими елементами були підпрограми. До цього часу бібліотеки математичних методів зазвичай постачаються у вигляді набору підпрограм. Складність навіть найпростішого, дуже поширеного такого типового елемента, як редактор текстів, характеризується вже десятком підпрограм.

Для тиражування таких елементів програм, як редактор текстів, система ієрархічного меню, елементів діалогу типу заповнення бланків фірма Borland Inc. запропонувала застосовувати TPU - Turbo Pascal Unit (модуль OBJ у ряді мов). TPU-файл дозволив використовувати механізм приховування в секції Implementation нецікавих внутрішніх підпрограм та внутрішніх даних і, навпаки, поза файлом механізм приховування забезпечив відкритість виклику лише корисних для користувача процедур та використання внутрішніх глобальних змінних, описаних у секції Interface. Після цього нововведення програмісту для використання, наприклад редактора, написаного не ним, треба знати лише інформацію, описану в секції Interface. Механізм приховування інформації в межах файлу був введений ще в низку компіляторів різних мов. Реалізація механізму приховування спростила завдання використання "кубиків".

Об'єктно-орієнтовані мови програмування дали чотири нових механізми використання кубиків:

1) механізм класів, що породжують під час виконання будь-яку кількість однотипних об'єктів, наприклад, ряд однотипних кнопок;
2) можливість тиражування об'єктів від програми, що породила, у всі нові програми;
3) динамічно лінкувані бібліотеки з класами, що породжують об'єкти;
4) механізм складання програм із «кубиків» — об'єктів у процесі їх виконання.

Перший механізм полегшив розвиток систем візуального програмування, під час роботи у яких значну частину програми можна створити шляхом відбору мишкою стандартних «кубиків».

Другий механізм призвів до виникнення об'єктно-орієнтованих СУБД, які постачають програмам як дані, а й код, обробляє ці дані.

Третій і четвертий механізми дозволили спробувати будувати гнучкі програми, що мають властивість можливого розвитку при зміні умов їх експлуатації. Вперше можливість реалізації отримали ідеї еволюційного програмування. Ідеї минулих технологій еволюційного програмування були явно недостатніми для забезпечення гнучкості програм, що для тривалого розвитку у мінливому зовнішньому середовищі вимагало неможливого однозначного передбачення майбутнього. Згідно з третім механізмом виникли COM-технології. На основі четвертого механізму з бази готових «кубиків П-об'єктів» створюються нові «кубики» і з них нові програми.

Таким чином, теоретично програмування проблема «кубиків» залишається найважливішою проблемою, поетапне вирішення якої дозволило створювати найбільші за кількістю складових частин штучні системи — програми. Другий аспект проблеми «кубиків» — здешевлення програмних розробок за рахунок повторного використання у нових програмах

частин, створених у попередніх розробках. Якщо є «кубики», то технології програмування необхідно включити методи, що полегшують синтез цілісних систем з окремих «кубиків».

ВИСНОВКИ

• Проектування – високоінтелектуальний процес. Для поняття теорії проектування необхідно оперувати безліччю термінів та визначень, такими як проектна ситуація, технологія, оптимізація програмних розробок. Усе це говорить необхідність ретельно підходити до вивчення словникового апарату теорії проектування.
• Програми в основному складні системи з мільйонів машинних інструкцій. Складність визначається чотирма основними причинами: складністю завдання; складністю управління процесом розробки; складністю опису поведінки окремих підсистем; складністю забезпечення гнучкості кінцевого програмного продукту.
• Під час розробки програмного забезпечення слід використовувати такі загальні принципи: частотний; модульності; функціональної вибірковості; генерованості; функціональної надмірності; "за замовчуванням".
• Однією з найважливіших складових успішного проектування є системний підхід, який передбачає всебічне дослідження складного об'єкта.
• Під час створення та розвитку ПЗ рекомендується застосовувати такі загальносистемні принципи: включення; системної єдності; розвитку; комплексності; інформаційної єдності; сумісності; інваріантності.
• У програмуванні існують різні парадигми, що призводять до різних підходів під час написання програм: процедурно-орієнтований; об'єктно-орієнтований; логічно-орієнтований; орієнтований правила; орієнтований обмеження; паралельне програмування, і навіть багато інших.
• Необхідно пам'ятати, що проектування невід'ємне від різних стандартів (ДСТУ, ANSI, проекту) та їх слід дотримуватись як при оформленні документації, так і для уніфікації вашого проекту.
• Програми створюються, експлуатуються та розвиваються у часі, проходячи свій життєвий цикл. Характерна риса життєвого циклу ПЗ - відсутність етапу утилізації.
• У процесі виконання проекту передбачаються окремі моменти часу, що характеризуються закінченим оформленням результатів усіх робіт, виконаних розробниками до цього моменту. Відповідно до ДСТУ можливі такі стадії розробки: ТЗ; ЕП; ТП; РП; Використання. Можливі також і нестандартні етапи та стадії. Набір етапів та стадій відображає результати проектування самого процесу проектування.
• Моделі відіграють важливу роль у проектуванні програм. При побудові моделей використовується абстрагування та декомпозиція.
• Кожна стадія проекту завершується затвердженням програмних документів. Документи включають опис (специфікації). Специфікації є моделі. Специфікації поділяються на зовнішні та внутрішні.
• Раціональний вибір стандартних елементів («кубиків») має два аспекти: зручність при повторному використанні та можливість здійснення синтезу з малих елементів загальніших елементів.
• Імена, що використовуються в програмах, повинні відповідати призначенню, мати впізнаваність, забезпечувати запам'ятовуваність, бути короткими, мати унікальність.

Контрольні питання

1. Дайте визначення проектування.
2. Що таке евристика?
3. У чому схожість і відмінність алгоритму та евроритму?
4. Що розв'язує задачу оптимізації розробки програм?
5. Назвіть п'ять ознак складної системи.
6. На чому ґрунтується частотний принцип розробки програм?
7. Які види аналізу застосовуються при системному підході?

8. Що таке принцип сумісності?
9. Для чого необхідна стандартизація проектування та програмування?
10. Назвіть основні етапи життєвого циклу програмних виробів.
11. Назвіть основні стадії та етапи розробки програм за ДСТУами.
12. У чому сутність моделювання?
13. Які типи абстракцій ви знаєте?
14. Що таке первинна функціональна специфікація?
15. Які механізми використання «кубиків» надали об'єктно-орієнтовані мови програмування?
16. Що таке рефакторинг?
17. Навіщо потрібний рефакторинг імен?
18. Чому важко визначити ідеальні імена?

Тема 2 ОПТИМІЗАЦІЯ ПРОГРАМНИХ РОЗРОБОК

2.1. ВИБІР ОПТИМАЛЬНОГО ВАРІАНТУ ПРОЕКТНОГО РІШЕННЯ

На різних етапах проектування (особливо часто на початкових етапах) перед розробником постає завдання вибору найкращого варіанта з безлічі допустимих проектних рішень, які задовольняють вимогам.

Неминучою платою за спробу отримати рішення за умов неповної інформації про об'єкт проектування є можливість помилкових рішень. Тому в такій ситуації особа, яка приймає рішення (ЛПР), повинна виробляти таку стратегію щодо прийняття рішень, яка хоч і не виключає можливості прийняття неправильних рішень, але зводить до мінімуму пов'язані з цим небажані наслідки. Для зменшення невизначеності ЛВР може провести експеримент, але це дорого і вимагає великих витрат часу. Тому ЛПР має прийняти рішення про форму, час, рівень експерименту.

Саме собою ухвалення рішення є компроміс. Приймаючи рішення, необхідно зважувати судження про цінність, що включає розгляд багатьох чинників, зокрема економічних, технічних, наукових, ергономічних, соціальних тощо.

Прийняти «правильне» рішення означає вибір такої альтернативи з-поміж можливих, в якій з урахуванням усіх різноманітних факторів буде оптимізовано загальну цінність. Процес прийняття рішення при оптимальному проектуванні характеризують такі основні риси: наявність цілей (показників) оптимальності, альтернативних варіантів об'єкта, що проектується, і облік істотних факторів при проектуванні.

Поняття «оптимальне рішення» при проектуванні має цілком певне тлумачення — найкраще у тому чи іншому сенсі проектне рішення, яке допускається обставинами. У переважній більшості випадків одна і та ж проектна задача може бути вирішена декількома способами, що призводять не тільки до різних вихідних характеристик, а й до класів програм. Най універсальніші програми - це текстові (процесори) редактори, що допускають використання графіки. Вони дозволяють оформляти вихідні дані, здійснювати ручний набір обґрунтування рішення з результатами розрахунку, робити висновок на друк. Для деяких цілей кращі електронні таблиці. Багатьох користувачів цілком влаштовують інтегровані системи, що включають текстовий процесор, процесор електронних таблиць, графічні процесори (малюнки та ділову графіку), систему управління базою даних (СУБД), системи модемного та мережевого зв'язку користувачів. При цьому одне з рішень може поступатися за одними показниками і перевершувати інші за іншими. Може бути так, що різні рішення взагалі характеризуються різним набором показників. У умовах важко зазначити, яка програмна система як оптимальна, і навіть краще.

Найбільші збитки приносять помилки при виборі сукупності показників якості. Перепустка одного показника може виявитися трагічною. Щоб не робити таких помилок, треба накопичувати базу знань усіх сукупностей показників, які були використані для проектування конкретних систем. З іншого боку, використання традиційних сукупностей показників не дозволяє виходити нові вироби. Вихід із цього становища: певну частку у процесі проектування відводити під творчий пошук.

База знань сукупностей показників має складатися як із загальних (загально програмних), так і спеціальних (предметно-орієнтованих) показників. Нині використовують такі класифікації показників:
1) показники функціонування, що характеризують корисний ефект від використання програмної системи за призначенням, і область застосування (наприклад, бібліотечна інформаційно-пошукова система характеризується такими показниками функціонування: максимальним обсягом літературних джерел, що зберігаються; максимальною кількістю одночасно працюючих користувачів; списком оброблюваних запитів; часом реакції на кожний запит при максимальній кількості користувачів;
2) показники надійності, що характеризують властивості програмної системи зберігати свою працездатність у часі;
3) показники технологічності, що характеризують ефективність конструкторсько-технологічних рішень для забезпечення високої продуктивності праці при виготовленні та супроводі;
4) ергономічні показники, що характеризують систему людина-виріб-середовище та враховують комплекс гігієнічних, антропологічних, фізіологічних та психічних властивостей людини, що виявляються у виробничих та побутових умовах;
5) естетичні показники, що характеризують зовнішні властивості системи: виразність, оригінальність, гармонійність, цілісність, відповідність середовищу та стилю;
6) показники стандартизації та уніфікації, що характеризують ступінь використання у програмній системі стандартизованих виробів та рівень уніфікації його частин;
7) патентно-правові показники, що визначають кількість патентів, ступінь патентного захисту, патентну чистоту;
8) економічні показники, що характеризують витрати на розробку, виготовлення, експлуатацію програмної системи та економічну ефективність експлуатації.
Середовище проекту характеризується можливостями капіталовкладення, можливостями колективу-виробника, науково-технічними досягненнями, соціальним та природним середовищем.

2.2. ПРИКЛАД ВИБОРУ ОПТИМАЛЬНОГО ВАРІАНТУ ПРОГРАМНОГО РІШЕННЯ

Щоб зробити вибір оптимального варіанта, треба мати кілька варіантів реалізації виробу. А щоб їх порівнювати, треба сформулювати ряд характеристик чи ненормованих критеріїв. Після нормування характеристик відповідно до шкал виходять нормовані критерії, які зручні для аналізу. Один із найпростіших способів отримання нормованих значень критеріїв - це проставлення експертами оцінок за п'яти-або десятибальною шкалою.
Фірма Borland Inc., створивши свій компілятор, вирішила розробити демонстраційну програму, яка могла б показати найбільшу кількість можливостей компілятора. У табл. 2.1 наводяться найменування критеріїв, варіанти реалізації програм та оцінки за п'ятибальною шкалою. Цю таблицю склали учні одному з практичних занять. Ними ж було виставлено оцінки.
Постачання компілятора у вихідних текстах може призвести до його іноді некваліфікованих масових модифікацій, що, своєю чергою, може викликати недовіру до виробу і завдати шкоди репутації фірми.

Таблиця 2.1
Бальна оцінка варіантів реалізації програми за критеріями

Критерії	Варіанти реалізації				
	СУБД	ЕТ	ОС	Редактор текстів	Гра
Обсяг програми	4	4	4	2,5	3

Зрозумілість	2	4	1	5	2
Нові знання	2	4	2	3	2
Інтерес	4	3	3	3	5
Використання у власних розробках	2	5	5	4	1

Система управління базами даних (СУБД) може бути великою або невеликою програмою. Головне в СУБД мало зрозумілі алгоритми обробки даних. Інтерес для користувача представляє бібліотека обробки даних, а чи не готова програма.

Електронна таблиця (ЕТ) надає можливість демонстрації користувачеві збирання програми із низки програмних файлів. Електронна таблиця містить функції редактора та інтерпретатора арифметичних виразів. Програма може бути прикладом реалізації обчислювальних алгоритмів. Сама програма та її окремі частини можуть вставлятися у програми користувачів. Програма таблиці має середній розмір.
Операційна система може мати будь-який обсяг. Зрозумілість текстів ОС невисока.
Простий текстовий редактор є короткою програмою, що складається з декількох підпрограм.
Складний текстовий процесор виходить з простого редактора екстенсивним доповненням великої кількості сервісних функцій з алгоритмами, що важко сприймаються.
Таким чином, саме для поставленої мети розробки перемагає варіант електронної таблиці (ЕТ), яка включає: клітинний редактор з ідеології функціонування, близький до текстового редактора; алгоритми роботи із файлами складної структури; інтерпретатор мови формул із виконавцем математичних розрахунків.
Фірма «Borland Inc.» з ранніми розробками компілятора (Turbo Pascal 4.0) постачала демонстраційну програму найпростішої електронної таблиці MicroCalc.
У пізнішому дистрибутиві Turbo Pascal 6.00 з'явилася нова демонстраційна версія електронної таблиці TurboCalc, реалізована з використанням об'єктно-орієнтованої технології. Оскільки й інші варіанти реалізації програм викликають інтерес у користувачів, фірма з пізніми розробками компілятора почала постачати їх. Так постачалися: гра в шахи з незрозумілими алгоритмами; текстовий редактор як бібліотечна програма; бібліотека підтримки роботи з базами даних Сам компілятор у вихідному коді фірмою Borland Inc. ніколи не постачався.

2.3. МЕТОДИ СИНТЕЗУ ВАРІАНТІВ РЕАЛІЗАЦІЙ ПРОГРАМ

Щоб відібрати оптимальне рішення необхідно синтезувати безліч можливих рішень (варіантів), що включають оптимальне рішення.
Жодне завдання не вирішується саме собою. Щоб отримати рішення, провадяться різні розумові дії. Ці дії не хаотичні, а мають методичну спрямованість, хоча зазвичай людина про це не підозрює.
Існує безліч методів синтезу варіантів проекту. Ось лише деякі з найбільш прийнятних для програмування: метод проб і помилок; евристичних прийомів; мозкового штурму; метод аналогій та морфологічних таблиць.
Раніше й до нашого часу більшість нестандартних завдань вирішувалася людиною на інтуїтивному рівні, т. е. шляхом спроб і помилок.
Метод проб та помилок- Це послідовне висування та розгляд ідей. Людина, стикаючись із проблемою, багаторазово подумки шукає відповідь, перебирає варіанти і, нарешті, знаходить рішення. Десятки, сотні, тисячі спроб упродовж днів, тижнів, років. Зрештою, в більшості випадків рішення знаходиться. У програмуванні цей метод зазвичай застосовується для оптимізації архітектури систем та структури програм.
Головний недолік методу спроб і помилок - це, по-перше, повільне генерування нових ідей, а по-друге, відсутність захисту від психологічної інерції, тобто висування ідей тривіальних, звичайних, неоригінальних.

Наступним кроком у вдосконаленні технології став перехід до спрямованих методів пошуку рішень, які базуються на розкритті та описі процесу рішення, представленні його у вигляді деякого евристичного алгоритму. Спрямованість евристичних методів - "розгойдувати" мислення, допомогти по-новому побачити завдання, подолати стереотипи.

Якщо, вирішуючи конкретне завдання, проектувальник не обмежиться досягненням лише миттєвої мети, а зможе «зазирнути у майбутнє» і виділити інваріантні частини системи, ці частини, будучи хіба що будівельним матеріалом цієї системи, можуть бути основою й у систем, які ще будуть проектуватись. Майбутні системи можуть вирішувати зовсім інші завдання. У цьому корисно було б створювати та накопичувати бібліотеки інваріантних частин системи або навіть паралельно проектувати кілька систем (об'єктів), що мають як подібні, так і різні цілі. «Зазирнути в майбутнє» можна лише добре знаючи минуле та сьогодення, а також нові досягнення програмування.

Цілеспрямовані методи творчості цілком застосовні не лише до технічних систем, а й до програмних. Розглянемо найбільш відомі з них, а також їхнє можливе застосування як при колективному, так і індивідуальному використанні.

Метод евристичних прийомів дозволяє як з'єднувати по-новому відомі частини, а й винаходити нові. Він базується на виділенні базових прийомів, знайдених під час аналізу кращих програмних виробів.

При успішному розв'язанні будь-якої творчої задачі людина отримує два результати - саме розв'язання поставленої задачі та методичний досвід, тобто з'ясування процесу вирішення даної конкретної задачі. Але проблема полягає в тому, що вирішення одного завдання не можна просто перенести на рішення іншого. Тому лише після вирішення певної кількості завдань у людини з'являвся набір правил, вказівок чи прийомів розв'язання того чи іншого завдання. Такі методичні правила називають евристичними прийомами.

Евристичний прийом- Спосіб вирішення певного протиріччя. В евристичному прийомі міститься короткий розпорядження або вказівка, як перетворити вихідний прототип або в якому напрямку потрібно шукати, щоб отримати рішення. Евристичний прийом містить підказку, але гарантує перебування рішення.

Складність використання евристичних прийомів полягає в тому, що не будь-яка людина може бачити завдання в цілому, тобто не має системного підходу у вирішенні завдань. Причому необхідність у такому підході зростає із збільшенням складності завдання. Це призводить до того, що людина не може застосувати евристичний прийом до конкретного завдання і не розуміє, про що йдеться. Різним людям потрібно докласти різних зусиль, щоб здогадатися про те, як застосувати евристичний прийом та отримати розв'язання задачі. Але перед розв'язанням задачі повинні бути описані та з'ясовані критерії, за якими оцінюватиметься отримане рішення. Це допоможе відкинути непотрібні рішення при використанні евристичних прийомів і зрозуміти, в якому напрямку слід рухатись, щоб дійти до вирішення.

Отже, у кожної людини, яка займається створенням програм, згодом накопичується досвід і з'являються способи вирішення різноманітних завдань. Причому зі збільшенням досвіду у цих способах збільшується частка системного підходу, тобто згодом людина отримує такий спосіб, який стає застосовним для вирішення більшої кількості завдань, ніж було раніше. Поступово у фахівця накопичується фонд таких практичних прийомів, але цей фонд є індивідуальним і не завжди доступним іншим користувачам. Тому необхідно систематизувати такі фонди та зробити їх більш систематичними. Актуальним завданням є створення фонду евристичних прийомів, застосовуваного на вирішення завдань оптимізації програмних розробок.

Найбільше евристичних прийомів спрямовано подолання протиріч. Суперечність у завданні — ситуація, яка потребує одночасного поліпшення двох суперечливих показників якості та поєднання, начебто, несумісних вимог. Ряд прийомів легко сприяє активізації мислення.

Перше, що спадає на думку в ситуації з протиріччями, — знайти найкраще співвідношення між показниками. Якщо потенційні можливості структури об'єкта великі, то цьому шляху іноді вдається отримати прийнятні значення всіх показників. Однак це вдається не завжди. Наступний природний для людини крок у ситуації, коли потенційні можливості структури недостатні і компроміс виявляється неприйнятним, — перейти до нової структури з великими потенціями, які забезпечують досягнення прийнятних значень показників якості, що конкурують.

Однак кращим є інший варіант дій. Як показує досвід, у багатьох ситуаціях містяться приховані ресурси, використання яких дозволяє кардинально вирішити протиріччя. Такими прихованими ресурсами у багатьох випадках є три види ресурсів: тимчасові, просторові та «непрямі». До «непрямих» ми відносимо ресурси, наявні у системі (але, зазвичай, не вважаються ресурсом, по крайнього заходу, у межах розв'язуваного завдання), використання яких пов'язані з якимись додатковими затратами.

Найбільш продуктивні способи вирішення протиріч - рознесення їх у часі, просторі або використання «непридатних» ресурсів. Ці загальні рекомендації можуть мати різне трактування у конкретних ситуаціях.

Доцільність використання методу евристичних прийомів для постановки завдань розробки програм перевірено педагогічною практикою авторів. Найчастіше досить короткої підказки учням як евристичного прийому, щоб вони самостійно правильно сформулювали завдання. Використовуючи фонд евристичних прийомів, Б.С. Воїнів та В.В. Костерін успішно синтезували низку нових механізмів алгоритму пошуку глобального екстремуму функцій багатьох змінних на сітці коду Грея.

Приклад використання методу евристичних прийомів до створення алгоритмів описаний у книзі Д. Пойа. Укорочений фонд евристичних прийомів програмування описаний у додатку 3.

Метод мозкового штурму- Один із популярних методів колективної творчості. Його психологічна основа – взаємна стимуляція мислення у групі. Конкуренція для людей за кількість висунутих ідей робить цей метод більш ефективним порівняно з роботою кожної окремої людини поза групою. Метод мозкового штурму є по суті тим самим методом спроб і помилок, проте він створює умови для психологічної активізації творчого процесу, знижує інерцію мислення, чому особливо сприяє наявність групи людей з боку. Метод простий, доступний та ефективний. Реалізація методу виглядає так.

Рішення проводиться у два етапи – генерації та аналізу. На етапі генерації створюється творча група з 5-15 осіб (фахівці-суміжники і люди «з боку», які не мають жодного досвіду в галузі, до якої належить вирішуване завдання). Групі пояснюється суть завдання, що потребує вирішення, та проводиться етап генерації ідей. На цьому етапі не допускається критика пропонованих ідей. Заохочується висування навіть навіжених ідей. Потім група експертів аналізує висловлені ідеї та відбирає ті, які заслуговують на більш ретельне опрацювання.

Методом мозкового штурму працюють команди знавців у популярній телепередачі: «Що? Де? Коли? П'ятдесят секунд іде генерація ідей і лише десять секунд витрачається на обговорення висунутих ідей.

Стосовно програм методом мозкового штурму можна згенерувати ідеї щодо розподілу функцій обробки інформації для людей і машиною; набору основних прототипів та запозичених з них ідей; реалізації деяких евристичних алгоритмів обробки інформації; реклами та збуту програмної продукції; створення нових програм з урахуванням частин створюваних і раніше створених програм.

Методи аналогій є найпопулярнішими методами для програмістів. Подолати психологічну інерцію, знайти нове рішення допомагають несподівані порівняння, що дозволяють глянути ситуацію під незвичайним кутом.

Суть одного з методів полягає у наступному. Удосконалену систему тримають хіба що у фокусі уваги і переносять її у властивості інших програм із колекції, які мають до неї

жодного відношення. У цьому виникають незвичайні поєднання, які намагаються розвинути далі.
Згідно з проведеним опитуванням, більшість професійних програмістів саме цим методом генерували зовнішній вигляд своїх програмних систем та визначили способи реалізації багатьох функцій програм.
Метод морфологічних таблиць є простим та ефективним особливо там, де необхідно знайти велику кількість варіантів досягнення мети. Останнім часом він використовується досить широко як розвиток творчої уяви. Метод по суті аналогічний складання різних варіантів будинку з кубиків дерев'яного конструктора і полягає в тому, що для об'єкта, що цікавить нас, формується набір відмітних ознак: найбільш характерних підсистем, властивостей або функцій. Потім кожного з них визначаються альтернативні варіанти реалізації (деталі конструктора). Комбінуючи альтернативні варіанти, можна отримати багато різних рішень. Аналізуючи їх, виділяють кращі варіанти.
Приклад морфологічної таблиці є прайс-лист комп'ютерної фірми. У прайс-листі міститься інформація про декілька типів корпусів ЕОМ; материнських плат; процесорів і т. д. Кожна частина забезпечена технічними характеристиками та ціною. Не всі варіанти елементів можуть бути зістиковані між собою. Головне, що характеризує прайс-лист, - це відсутність критерію якості комп'ютера для конкретного користувача. Дивлячись на прайс-лист, треба синтезувати цей критерій і вибрати оптимальний склад елементів. Як результат синтезу можуть бути виявлені варіанти побудови комп'ютерів, орієнтованих різні категорії користувачів.
Проблема тут така сама, як і в ситуації підбору одягу для дівчини. Якщо ознайомитися з товаром у ряді магазинів, то неважко за рекомендацією «краще дороге» купити окремі елементи гардеробу. Швидше за все ці «найкращі» елементи не будуть загалом виглядати на дівчині через несумісність кольорів, фасону, та й самого вигляду та характеру дівчини, тобто відсутня оптимізація за критерієм цілого.
Покупка сама може перетворитися на болісне ходіння по магазинах з тисячами примірок. При цьому, швидше за все, буде помилково придбаний не той товар.
Для аналізу критерію цілого краще залучити досвідчених експертів (наприклад, піти магазинами з подругами), які можуть вказати на помилки вибору.
Досвідчений кутюр'є допоможе зробити вибір дорогого модного одягу. Можливо, щоб краще підходити під запропонований одяг, вам доведеться позайматися з психологом, косметологом та інструктором фізкультури для зміни іміджу. На щастя, багато дівчат здатні самі одягатися «дешево та сердито» і вміють самостійно адаптувати свій імідж.
Морфологічна таблиця, складена в компактній формі, допоможе уникнути багаторазового ходіння по одним і тим же магазинам, що заощадить ваш час та час експертів. Морфологічна таблиця дозволить вам та експертам переглянути значно більше варіантів і зробити більш оптимальний вибір.
У 1983 р. В.В. Костериним успішно застосовано морфологічна таблиця для синтезу ідей побудови алгоритму нелінійного програмування пошуку глобального екстремуму функцій багатьох змінних на сітці коду Грея. Алгоритми нелінійного програмування призначені для пошуку екстремумів функцій багатьох змінних. У методах прямого пошуку екстремум виявляється шляхом розрахунку множини точок функції при аргументах, що визначаються самим алгоритмом пошуку. У табл. 2.2 наведено дану морфологічну таблицю, яка містить класифікаційні ознаки окремих механізмів алгоритмів нелінійного програмування на рівні основних принципів. Наведені класифікаційні ознаки виділялися за основними функціональними ознаками окремих механізмів. Кожній класифікаційній ознакі відповідає безліч реалізацій механізмів у вигляді значень класифікаційних ознак.
Цікаво відзначити, що кількість можливих реалізацій алгоритмів нелінійного програмування за цією таблицею становить N = 5 * 6 * 8 * 5 * 7 * 7 * 6 = 352800, що значно перевищує число опублікованих методів (близько 2000)!
Таблиця 2.2

Морфологічна таблиця принципів функціонування алгоритмів нелінійного програмування

Класифікаційні ознаки	Значення класифікаційних ознак
Початкова точка пошуку	1.1 1.2 1.3 1.4 1.5
Зондування гіперповерхні	2.1 2.2 2.3 2.4 2.5 2.6
Стратегія кроків пошуку	3.1 3.2 3.3 3.4 3.5 3.6 3.7 3.8
Напрямок пошуку на кроці	4.1 4.2 4.3 4.4 4.5
Стратегія кроку пошуку	5.1 5.2 5.3 5.4 5.5 5.6
Механізм самонавчання	6.1 6.2 6.3 6.4 6.5 6.6 6.7
Механізм завершення пошуку	7.1 7.2 7.3 7.4 7.5 7.6

Значення класифікаційних ознак класифікаційної ознаки "Механізм початкової точки пошуку":
ознака 1.1 - з точки, вказаної користувачем;
ознака 1.2 - із середньої точки області визначення;
ознака 1.3 - з точки на межі області визначення;
ознака 1.4 - з випадкової початкової точки пошуку;
ознака 1.5 - початкова точка пошуку не задається.
Значення класифікаційних ознак класифікаційної ознаки «Первинне зондування гіперповерхні»:
ознака 2.1 - у вигляді великої кількості випадкових точок, зондуючих всю гіперповерхню;
ознака 2.2 - послідовні спуски з ряду випадкових початкових точок;
ознака 2.3 - конкуруючі спуски з випадкових точок, що додаються;
ознака 2.4 - зондування гіперповерхні випадковими точками з виявленням та більш ретельним дослідженням «підозрілих областей»;
ознака 2.5 - сканування всієї гіперповерхні з використанням різних розгорток, наприклад, Пеано;
ознака 2.6 - окремий механізм початку пошуку відсутня.
Значення класифікаційних ознак класифікаційної ознаки «Стратегія кроків пошуку»:
ознака 3.1 - один крок;
ознака 3.2 - Послідовні кроки до виявлення екстремуму;
ознака 3.3 - здійснювати всі кроки по тому самому механізму;
ознака 3.4 - перемикати механізми кроків від глобального методу до локального;
ознака 3.5 - перемикати механізми кроків від глобальних далі до усереднених і локальних;
ознака 3.6 - перемикати механізми кроків за евристичними правилами;
ознака 3.7 - мала кількість послідовних кроків з обмеженого ряду лідируючих конкуруючих початкових точок;
ознака 3.8 - кроки пошуку відсутні.
Значення класифікаційних ознак класифікаційної ознаки «Напрямок пошуку на кроці»:
ознака 4.1 - нова точка в напрямку апроксимації градієнта, побудованого на основі даних поточної та попередньої пробної точок;
ознака 4.2 - за результатами обробки невеликої кількості перспективних точок, отриманих на попередніх кроках;
ознака 4.3 - за результатами аналізу функції, що апроксимує випадкові точки в перспективному напрямку;
ознака 4.4 - зондування гіперповерхні великою кількістю випадкових точок та подальшою побудовою апроксимуючої функції;
ознака 4.5 - вздовж межі області визначення цільової функції;
ознака 4.6 – механізм відсутній.
Значення класифікаційних ознак класифікаційної ознаки «Механізм стратегії кроку пошуку»:
ознака 5.1 - пробні точки тільки на відстані передбачуваного екстремуму;

ознака 5.2 - пробні точки на більшій відстані, ніж передбачуваний екстремум;
ознака 5.3 - пробні точки на відстані, дещо меншій, ніж у передбачуваного екстремуму;
ознака 5.4 - об'єднання ознак 5.1 та 5.2;
ознака 5.5 - об'єднання ознак 5.1, 5.2 та 5.3;
ознака 5.6 - суміщення пошуку напряму та відстані до екстремуму;
ознака 5.7 - поділ пошуку напрямку та відстані до екстремуму.
Значення класифікаційних ознак класифікаційної ознаки «Механізм самонавчання»:
ознака 6.1 - звуження меж пошуку у міру просування до екстремуму;
ознака 6.2 - Поступове підвищення точності пошуку;
ознака 6.3 - виявлення форми гіперповерхні за результатами попередніх кроків та перехід на спеціальний механізм уточнення екстремуму;
ознака 6.4 - виявлення форми гіперповерхні за результатами попередніх кроків та перехід на спеціальний механізм просування вздовж ярів;
ознака 6.5 - виявлення форми гіперповерхні за результатами попередніх кроків та відмова від поточного знайденого екстремуму;
ознака 6.6 - Зміна щільності ймовірності випадкових точок для різних зон пошуку;
ознака 6.7 – механізм відсутній.
Значення класифікаційних ознак класифікаційної ознаки «Механізм завершення пошуку»:
ознака 7.1 - не виявляється напрямок поліпшення функції на наступному кроці;
ознака 7.2 - витрачений ресурс часу;
ознака 7.3 - досягнуто заздалегідь задане значення цільової функції;
ознака 7.4 - вичерпані можливості алгоритму пошуку екстремуму;
ознака 7.5 - виконано заздалегідь задану кількість кроків пошуку;
ознака 7.6 — немає поліпшень у «далекому» та «близькому» околицях.
Черговий принцип побудови методу нелінійного програмування виходить шляхом відбору за одним із значень класифікаційних ознак у кожному окремому рядку табл. 2.2.
Оболонки візуального програмування, наприклад Delphi, реалізують метод морфологічного синтезу під час побудови форм діалогу програм з урахуванням візуальних компонент.

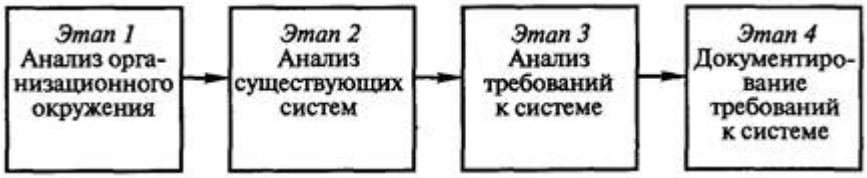
2.4. АНАЛІЗ ВИМОГ ДО СИСТЕМИ (СИСТЕМНИЙ АНАЛІЗ) І ФОРМУЛЮВАННЯ ЦІЛІВ

Завдання оптимізації розробки програм полягає у досягненні цілей за мінімально можливої витрати ресурсів.
Системний аналіз на відміну попереднього системного дослідження — це поглиблене вивчення інформаційних потреб користувачів, яке буде покладено основою детального проектування нової автоматизованої системи (АС).
Кінцевий продукт цього етапу — набір функцій, що виконуються, або функціональні вимоги, тобто документована постановка системних вимог до нової АС. Коли йдеться про створення великої системи, цей документ є звітом про системний аналіз, який здійснюється за етапами, показаними на рис. 2.1.
Перший етап системного аналізу (аналіз організаційного оточення) пов'язаний з тим, що неможливо створити працездатну інформаційну систему, якщо дослідники нічого не знають про особливості функціонування організації, функції якої має обслуговувати автоматизована система (АС) та елементом якої вона є. Слід розуміти особливості та тип діяльності, управлінську структуру, методи управління, зв'язки підрозділів, персонал, динаміку інформаційного обміну між окремими працівниками та робочими групами (форми документів та звітів, терміни, кількість екземплярів тощо).
Другий етап системного аналізу (аналіз існуючих систем) обумовлений тим, що в організаціях можуть існувати якісь АС з певними ресурсами (інформаційними, програмними, технічними, а також персоналом). Навіть тоді, коли проводиться повне оновлення технічної та програмної баз, існуюче інформаційне забезпечення відображає ядро головних потреб, які

не можна не ігнорувати в новій системі, а навпаки, стартуючи від нього, слід розвинути і розширити.

Слід ретельно вивчати, які завдання вирішують «старі» системи, яке обладнання та програмне забезпечення мають, який персонал працює в інформаційному відділі; чи існують бази даних, якою є їх структура і якими методами формуються звіти про результати — усе це — найважливіші питання.

Важливо з'ясувати: чи застосовується кодування інформації та які рівні кодів у своїй використовуються (місцеві, державні, міжнародні); існуючий регламент обробки даних, кого і чому він не влаштовує, чи бувають практичні затримки даних та звітів, причини затримки; чи є документація на стару систему. Такий перелік цілей слід пам'ятати в особистому щоденнику дослідника системи.



*Мал. 2.1.*Етапи проведення системного аналізу інформаційних систем

Другий етап системного аналізу — це найскладніший і найвідповідальніший крок у масу метаінформації, т. е. «даних даних». Головними орієнтирами в організації метаінформації є об'єкти (документи, діаграми, аналітичні тексти та записки, економічні показники, сукупності), а також процеси створення об'єктів, процеси їх передачі, обробки, зберігання. На виконання третього етапу системного аналізу (аналіз вимог системи) дослідник повинен мати певні знання типових методів вирішення основних управлінських завдань (облікових, аналітичних, планових, оперативних). Все це є складовою спеціальних знань системного аналітика. Тому системний аналітик повинен простежити всі галузі комплексу вимог у розмовах з кінцевими користувачами, а потім зробити аналітичні системні висновки. Ці вимоги є та підтримуються існуючою інформаційною системою, а це має стати функціональною вимогою до нової покращеної системи.

Третій етап системного аналізу – визначення того, що має бути у новій АС. Це методологічний етап синтезу вимог до нової системи, які випливають із перших двох кроків аналізу, а також із спеціальних знань та засобів системних аналітиків. Фахівці з системного аналізу знають, що однією з підступних пасток у їхній роботі є можливість сплутати аналіз існуючої системи з тим, що має бути. Тому другий та третій етапи системного аналізу чітко розділені. Про це важливо пам'ятати, щоб не помилитися в оцінках основи, на якій будується нова система. У системотехнічної методології аналіз існуючого відокремлюється від аналізу характеристик майбутнього, ніж сприйняти бажане за дійсне.

Четвертий, підсумковий етап системного аналізу (документування вимог до нової системи) має узагальнити наявні аналітичні матеріали та створити документоване відображення функціональних вимог до нової АС. Документ «Вимоги до системи», або «Функціональні вимоги», є основою подальшої роботи спеціалістів інформаційного відділу для створення детального проекту нової системи, тобто створення специфікацій всіх її елементів, програм, інструкцій.

Отже, етап системного аналізу відповідає питанням, що повинна мати інформаційна система задоволення вимог користувачів, а етап системного проектування відповідає питанням, як конкретно здійснити таку АС.

У багатьох аспектах системний аналіз є найважчою частиною процесу створення системи. Проблеми, з якими стикається системний аналітик, пов'язані між собою, що є однією з головних причин складності їх вирішення:

1) аналітику складно отримати вичерпну інформацію з метою оцінки вимог до системи з погляду замовника;
2) замовник, у свою чергу, не має достатньої інформації про проблему обробки даних, щоб судити, що є здійсненним, а що ні;
3) аналітик стикається з надмірною кількістю докладних відомостей про предметну область і про нову систему;
4) специфікація системи через обсяг та технічні терміни незрозуміла для замовника;
5) у разі зрозумілості специфікації для замовника, вона буде недостатньою для розробників, які створюють систему.

Отже, на даному етапі еволюційного розвитку ситуація в галузі проектування АС має такий вигляд:

• є суб'єкт — потенційний замовник, який зазнає дискомфорту, для подолання якого необхідно вирішити низку проблем, і тому цей суб'єкт є джерелом діяльності;
• є перелік потреб, які потрібно задовольнити;
• відомі прототипи програмних засобів з механізмами, які в сукупності могли б задовольнити наявні потреби, але ці механізми не пов'язані в єдине ціле так, щоб задовольнити всі потреби.

Тепер необхідно сформулювати цілі та визначити обмеження на реалізацію програмного продукту.

Формулювання цілей – перший та найважливіший етап процесу проектування. Саме на цьому етапі закладаються основи успіху у вирішенні всього завдання. Помилки у виборі та формулюванні мети не можуть бути компенсовані на наступних етапах. Причина проста — все, що робиться на наступних етапах розробки, йде від поставленої мети. Отже, такі помилки жодними методами на наступних етапах неможливо компенсувати.

Зазвичай, всі помилки початкових етапів, виявлені наступних етапах, мають такі причини:
1) постановка недосяжної мети;
2) прагнення розробника та постановника завдання спростити завдання;
3) невміння розробника виділити із формулювання постановника окремо опис проблеми та постановку завдання;
4) не виявлені обмеження.

Ситуації, коли можлива конкуренція цілей, повинні виявлятися; необхідно узгодити цілі так, щоб найкращим чином у рамках можливостей, що визначаються обмеженнями, досягалися цілі кожного з можливих суб'єктів (замовників). Хороші цілі завжди є компромісом між бажанням якнайкраще задовольнити потреби та обмеженнями, що накладаються реалізацією та засобами.

Можна сформулювати послідовність рекомендацій (контрольних питань):

*Рекомендація 1.*Не довіряйте наявним формулюванням завдання; Рішення починайте з нуля, з виділення суб'єкта, виявлення причин його дискомфорту та потреб. Справа в тому, що найчастіше формулювання, пропоноване замовником, невдала або зовсім неприйнятна, оскільки описує насправді незадоволену потребу, видаючи її за завдання.

*Рекомендація 2.*Уточніть вимоги до кінцевого результату:
1) які функції та з якими показниками якості повинен виконувати функції об'єкт?
2) який ефект буде отримано у разі успішного розв'язання задачі?
3) які допустимі витрати, як вони пов'язані з показниками якості?

Може виявитися, що витрати істотно перевищать ефект, тому слід відмовитися від рішення, або шукати більш прийнятне.

*Рекомендація 3.*Уточніть умови, за яких передбачається реалізація знайденого рішення:
1) ретельно досліджуйте пов'язані з цим обмеження та переконайтеся, що всі вони дійсно мають місце;
2) виявите особливості реалізації, зокрема, допустимий ступінь складності, передбачувані масштаби застосування.

*Рекомендація 4.*Вивчіть завдання, використовуючи таку інформацію:
1) як вирішуються завдання, близькі до аналізованої?

2) як вирішуються завдання, обернені до розглядуваної? (Особливу увагу слід звернути при цьому на галузі застосування, для яких подібні завдання є найбільш актуальними.)
*Рекомендація 5.*Подумки зміните умови завдання та досліджуйте її розв'язання в нових умовах: змінюйте від нуля до нескінченності складність об'єкта, час процесу, витрати, умови середовища.
*Рекомендація 6.*Ретельно відпрацюйте формулювання завдання, бажано з використанням найзагальніших понять та термінів.
*Рекомендація 7.*Сформулюйте ідеальний кінцевий результат і у процесі рішення прагнете його отримати.
Аналіз вимог зосереджений на інтерфейсі системи людина-програма-машина та інформаційних потоках між ними. Тут вирішується, що робить людина, а що робить машина та як вона це робить. У результаті аналізу вирішується питання доцільності застосування ЕОМ.
У процесі аналізу розглядаються:
1) робота без ЕОМ та з ЕОМ з різним ступенем автоматизації;
2) варіанти використання існуючих програм як без модифікацій, так і з їхньою модифікаціями;
3) варіанти із спеціально створеними програмами;
4) час обробки інформації;
5) вартість обробки інформації;
6) ймовірність помилок, їх наслідки та якість обробки інформації.
Аналіз вимог сприяє кращому розумінню системи та досягненню найкращого задоволення потреби.
У ході проведення системного аналізу аналізуються надсистема, система та підсистема у вигляді складових проектованої системи.
Під час проектування необхідно враховувати такі ефекти.
Ефект заміни цілей. Процес діяльності з досягненню мети, зазвичай, пов'язані з подоланням деяких бар'єрів, часто непередбачених проблем. Намагаючись подолати їх, людина змушена змінювати початковий план дій, пристосовуючи його до конкретних умов. Природним для людини є і прагнення спростити своє завдання, діяти звичними, добре знайомими способами та засобами. Це може призвести найчастіше на завершальних етапах реалізації або в процесі експлуатації до заміни вихідної мети. У результаті досягнуто зовсім іншого результату, ніж передбачалося спочатку.
Тому чітке формулювання завдання, його відстеження на всіх етапах є необхідною умовою успішної діяльності.
Цілі та засоби. Наступний аспект, який не завжди оцінюється належним чином, — облік обмежень, що накладаються умовами реалізації, зокрема, властивостями реалізуючої системи та обмеженнями ресурсів. Здавалося б, забезпечення найбільшої ефективності об'єкта проектування з позицій над системи - головне завдання, але парадокс у тому, що не можна формулювати цілі, не маючи уявлення про те, як, за допомогою яких засобів, якої системи вони можуть бути досягнуті. Не будь-яка мета досяжна, і це або неможливістю чи незнанням, як реалізувати систему, що забезпечує досягнення мети, або є обмеження, які можуть зірвати її досягнення, тощо.
Повторимо раніше висловлену думку: «хороші цілі завжди є компромісом між бажанням якнайкраще задовольнити потреби та обмеженнями, що накладаються реалізацією та засобами». Практика показує, що цілі найкраще формулюються людьми, які знають можливості їх досягнення (або групою фахівців, які володіють різними аспектами проблеми).
Узгодження цілей. Насправді під час проектування цілей нерідко доводиться зіштовхуватися із ситуацією, коли функціонування одного об'єкта має задовольнити потреби низки суб'єктів (над систем). У цьому випадку стосовно кожної над системи можуть бути сформульовані свої цілі, які найчастіше виявляються суперечливими: більш повне задоволення потреб одного із суб'єктів може бути забезпечене лише за рахунок іншого. Якщо

етапі проектування мети не узгоджені, то, зазвичай, погано задовольняються потреби як однієї, і інший над системи. Тому ситуації можливості конкуренції цілей повинні обов'язково виявлятися: необхідно узгодити цілі те щоб найкраще, у межах можливостей, визначених обмеженнями, досягалися мети кожної з над систем.

Формулювання та формалізація цілей. Цікавою видається інтерпретація цілей через потреби та обмеження ресурсів. При цьому можна виділити три варіанти формулювання цілей.

1. Обмеження відсутні. Досягнення кожної зі сформульованих цілей певною мірою задовольняє потребу, як правило, не знімаючи її повністю. Тому говорять про залишкову потребу: що менша залишкова потреба після досягнення мети, то вище оцінюється і сама мета. Отже, найкращою буде та мета, після досягнення якої залишкова потреба виявиться мінімальною.

2. Наведене вище формулювання привабливе, але мало реальне. Досягнення будь-якої мети потребує наявності ресурсів, причому величина їх (ресурсів) істотно залежить від формулювання мети. Тому найбільш прийнятною є формулювання, коли потрібно найкраще задовольнити потребу при заданих обмеженнях на ресурси.

3. Можлива ситуація, коли можуть бути визначені обмеження щодо ступеня задоволення потреб та потрібно при цьому мінімізувати витрати ресурсів. Тоді мета формулюється так: необхідно забезпечити заданий рівень задоволення потреби при мінімальному витраті ресурсів. Облік ресурсів та інших обмежень є обов'язковим у більшості завдань проектування, отже, на етапі формулювання цілей обов'язковий аналіз, що дозволяє зіставити ступінь досягнення цілей і ресурси.

Незалежно від формулювання ціль як результат діяльності може бути задана формально через показники якості, що характеризують ступінь її досягнення. Вимоги до показників якості можуть бути задані у трьох видах: • прирівняти; обмежити; домогтися екстремуму. У загальне формулювання цілей можуть входити складові, що задаються у різний спосіб.

Рівні опису цілей.У процесі проектування цілі можуть бути описані на рівні (виражені мовою) суб'єкта, зовнішнього та внутрішнього опису об'єкта. Суб'єкт при цьому характеризується своїми потребами, об'єкт при зовнішньому описі - показниками якості, а при внутрішньому, структурованому - через параметри та змінні стани. Тому нерідко говорять про завдання цілей у просторі потреб, показників якості та стану. Для опису цілей кожному рівні використовуються відповідні поняття та величини.

З урахуванням вищесказаного, загальна методика проектування цілей виглядає так. Будується опис над системи та визначаються показники, що характеризують ефективність її функціонування. Потім визначаються функції, які мають виконуватися проектованим об'єктом, і конкретні вимоги щодо них через показники якості над системи — цим визначаються цілі у просторі потреб. При подальшій конкретизації об'єкта рівня його показників якості досліджується вплив останніх на показники над системи. Це дозволяє виразити цілі через показники якості об'єкта - задати їх у просторі показників якості. Подальша конкретизація об'єкта дозволяє визначити зв'язки між показниками якості та значеннями параметрів та змінних стану, а також виразити цілі через вимоги до них – описати їх у просторі станів.

Незважаючи на прозорість методики, на практиці при проектуванні цілей доводиться стикатися з серйозними труднощами, пов'язаними в основному зі складністю опису над системи та взаємозв'язку її показників з показниками об'єкта, а також оцінки взаємозв'язку між значеннями показників та необхідними для досягнення цілей ресурсами.

Етап формулювання цілей може призвести до різних ситуацій.

*Ситуація 1.*Вихід на добре знайомі цілі, відомі розробнику. У цьому випадку знадобиться лише пошук та коригування відомих рішень, тобто в результаті аналізу потреб користувача проектувальник приходить до висновку, що задовольнити ці потреби може вже існуючий програмний продукт з невеликими змінами.

*Ситуація 2.*Діаметрально протилежний варіант – нові цілі. У цьому випадку ми маємо справу із завданням, яке має явно видиму мету і немає коштів для її безпосереднього досягнення.

2.5. ПРОЕКТНА ПРОЦЕДУРА ПОСТАНОВКИ ЗАВДАННЯ РОЗРОБКИ ПРОГРАМИ

Проектна процедура базується на володінні системним підходом стосовно аналізу програмних систем. Спочатку розглядається система - людина (люди), ЕОМ, програма, інші об'єкти, наприклад технічні, як ланка, що виконує весь комплекс обробки інформації. Далі ці елементи розглядаються окремо для уточнення вимог. Програма має лише інформаційний вхід та інформаційний вихід.

Полегшити первинну генерацію цілей часто дозволяє модель «чорної скриньки», зображена на рис. 2.2. У процесі проектних ітерацій дана модель «сіріє» і стає «білою», якщо на ній фіксувати всі знайдені факти.

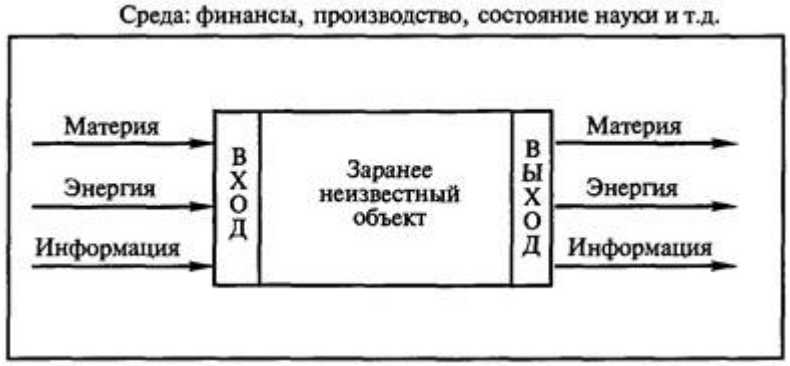
Суть проектної процедури.

*Крок 1*Проаналізуйте вихід системи, визначте склад та форму вихідних даних.

*Крок 2.*Проаналізуйте вхід системи, визначте склад та форму вхідних даних.

*Крок 3*Визначте критерії якості перетворення вхідної інформації на вихідну інформацію.

*Крок 4.*Визначте обмеження.



*Мал. 2.2.*Модель «чорної скриньки» об'єкта, що проектується

*Крок 5.*Визначте основні алгоритми обробки інформації та розрахуйте час їх виконання.

*Крок 6*Визначте обмеження.

*Крок*7. До знаходження прийнятної постановки генеруйте варіанти постановок. Використовуйте класифікацію алгоритмів, критеріїв, методів прийняття рішень, евристичні прийоми, практичні прийоми.

При виконанні даної проектної процедури бажано розширити низку досліджуваних варіантів з використанням методу проб і помилок, методу аналогій, методу морфологічного синтезу рішень.

По-перше, з усього різноманіття можливих програмних рішень виділіть клас допустимих та перспективних.

По-друге, вивчіть варіанти, які є найкращими за окремими показниками. Ці варіанти можуть бути прийняті для подальшого аналізу.

Сукупність функцій системи, умов та обмежень їх існування називається безліччю споживчих властивостей системи. Під поняттям «споживач» розуміється система вищого ієрархічного рівня, що «експлуатує» споживчі властивості вихідної системи.

Системне опис об'єкта через комплексне опис споживчих властивостей дозволяє у межах єдиного методичного апарату вирішити питання вибору досконалішого варіанта рішення і підготувати об'єктивні умови включення людської фантазії. Поняття досконалості дуже складне, тому що до раціональних критеріїв досконалості дуже часто підмішуються такі суб'єктивні уподобання, як зиск, смак, мода тощо.

Загальна тенденція розвитку, т. е. еволюція об'єкта шляху до досконалості, має внутрішні закономірності і можна зрозуміти з урахуванням цих закономірностей. Які ж ці закономірності?

Закономірність 1.З точки зору споживчих функцій: «Чим ширший склад споживчих функцій, чим інтенсивніша кількісна сторона їхнього прояву, тим досконаліша система».

приклад. З двох дівчат одного віку, однакової освіти, які проживали з дитинства до теперішнього моменту пліч-о-пліч в одному місті, за умови, що вони однаково улюблені, неможливо вибрати найкращу за дружину. Якщо ввести додатковий критерій, завдання спрощується. Наприклад, наявність посагу. Однак багатокритеріальне завдання вибору може стати важким і його легко вирішувати лише з використанням спеціальних методик.

Спробуйте самостійно розв'язати завдання: «Яка з відомих вам програм редактора текстів досконаліша?»

Закономірність 2.З погляду впливу чинників довкілля комп'ютера: «Чим ширше інтервал умов довкілля, всередині якого здатні реалізовуватися споживчі властивості конкретної системи, тим система досконаліше».

приклад. З двох дівчат — кандидаток за дружину — набагато краще, ніж невибаглива — це вже зовсім банальна істина.

Чи згодні ви, що з двох програм редактора текстів краща та, яка може виконуватися на ЕОМ з меншою пам'яттю.

Закономірність 3.З погляду інтервалу обмежень штучної довкілля: «Чим вже інтервал обмежень реалізації споживчих функцій цієї системи, тим система досконаліше».

приклад. При виборі майбутньої дружини, очевидно, що за інших рівних умов велику привабливість має наречена, здатна виконувати домашні (та інші) функції без допомоги асистентів у вигляді матінки, різних підозрілих друзів-приятелів і т.д.

Який редактор текстів краще: 1) із вбудованим орфографічним контролем; 2) із зовнішньою програмою орфографічного контролю, розробленою іншим виробником?

*Крок 1*Сформулювати завдання розвитку по кожному із споживчих властивостей.

*Крок 2.*Дослідити та спрогнозувати тенденції розвитку споживчого попиту по кожному із споживчих властивостей.

*Крок 3*На основі прогнозу про тенденції розвитку споживчого попиту скласти повний пріоритетний список усіх можливих завдань удосконалення об'єкта.

приклад. Проведемо системний аналіз електронного архіву (ЕА), що забезпечує доступ до документів та їх зберігання в електронному вигляді. Мета створення ЕА полягає у забезпеченні оперативного та повноцінного доступу до всіх документів, що зберігаються і надходять. Для цього потрібно вирішити два основні завдання: запровадити масив наявних в архіві документів та забезпечити можливість оперативного повнотекстового доступу до електронних документів.

*Крок 1*Перелічимо основні функції ЕА:

• сканування;
• розпізнавання та коригування помилок;
• створення та міграція електронних документів та образів;
• індексування документів;
• оперативний пошук та відображення документів.

Для реалізації даних функцій в ЕА повинні бути підсистеми введення, зберігання, індексування, пошуку та відображення інформації, аналізу, управління потоками, адміністрування та науково-технічного супроводу.

У системі можна виявити наступний ряд обмежень на реалізованість споживчих функцій:

- неможливість зберігання образу документів з використанням магнітних дискових носіїв внаслідок їх високої вартості та невисокої надійності без багаторазового резервування;
— непридатність використовуваних нині офісних сканерів (не дозволяють вводити документи на паперових носіях низької якості: рукописні, зліплі, вицвілі, порвані, різних розмірів та щільності, погано про друковані, забруднені тощо);

— СУБД, особливо реляційного типу, спочатку не орієнтовані на інтенсивну обробку надвеликого обсягу інформації.
*Крок 2.*Завдання проектування:
1) розгортання високопродуктивної мережі, що включає графічні робочі станції та потужні сервери введення та обробки інформації;
2) використання сканерів та відповідні русифіковані програмні засоби для введення документів з паперових носіїв низької якості;
3) забезпечення ефективного індексування та повнотекстового пошуку неструктурованої інформації великого обсягу.
*Крок 3*Можливість технічної реалізації аналізованої системи:
- З'явилися дешеві носії - компактні диски; різко знизився показник вартість/продуктивність для високошвидкісних обчислювальних систем, мереж та пристроїв;
— набули розвитку апаратно-програмні системи, що реалізують паралельну обробку запитів; підвищився рівень інтерфейсу роботи із СУБД;
- З'явилися нові інформаційні технології індексування надвеликих масивів даних;
— розроблено та розвиваються вітчизняні технології та програмні продукти розпізнавання та аналізу російськомовних текстів;
— намітився напрямок впровадження засобів штучного інтелекту, що дозволяють моделювати та аналізувати великі масиви інформації.
*Крок 4.*Як пріоритетні завдання вдосконалення системи можна виділити такі:
1) використання комбінації різних технологій індексування та пошуку. Намітилося кілька напрямів побудови електронних архівів залежно від методів пошуку (використання атрибутного пошуку структурованих даних і повнотекстового індексування неструктурованих даних);
2) використання спеціалізованих промислових сканерів, орієнтованих потокове введення архівних документів. Відмінною рисою таких сканерів є ротаційний механізм переміщення документів, що дозволяє вводити дані з паперових носіїв поганої якості;
3) через високі вимоги до швидкості доступу до пошукового образу документа та його цілісності, здійснення його зберігання у високошвидкісних відмовостійких системах зберігання, наприклад RAID-масивах. Найбільш підходящими носіями можуть бути магнітооптичні, фазоінверсні (PD/CD), компакт-диски (CD-R) і WORM-диски. Для автоматизації пошуку інформації, розміщеної на цих дисках, її вилучення та роботи з дисками використовуються автоматичні бібліотеки, або оптичні дискові автомати (JukeBox);
4) використання тільки потужних RISC-платформ, що масштабуються, орієнтованих на паралельні обчислення.
Поданий спосіб опису та завдання споживчих властивостей систем дозволяє деталізувати результати тенденцій розвитку споживчого попиту, перекласти їх на мову розробників, поставити орієнтири превентивного вдосконалення систем.
Взагалі прагнення враховувати у будь-якій діяльності вимоги до кінцевого результату є проявом дії механізму зворотний зв'язок, воно підвищує керованість і спрямованість діяльності, отже, і якість результату. Образ ідеального рішення таки служить як порівняння між собою конкретних типів пошукової діяльності, а й утримання процесу пошуку у певних рамках, направляючи його до необхідному результату. Межі цих рамок можуть бути задані наступними ознаками ідеальності (вони ж критерії порівняння та вибору).
*Ознака 1.*Порівняння за рівнем розвитку споживчих властивостей, що максимально досягається.
*Ознака 2.*Порівняння систем та пошук рішення на основі максимального резерву розвитку.

2.6. ПСИХОФІЗІОЛОГІЧНІ ОСОБЛИВОСТІ ВЗАЄМОДІЇ ЛЮДИНИ ТА ЕОМ

Психофізичні особливості взаємодії людини та ЕОМ - науково-дослідний напрямок, що вивчає процеси, що відбуваються в людино-машинній інформаційній системі.

ЕОМ доповнює людину, але не замінює її, тому розгляд основних особливостей їхньої співпраці необхідний.

Логічний метод міркування. У людини він ґрунтується на інтуїції, використанні накопиченого досвіду та уяві. Метод ЕОМ - суворий та систематичний. Найбільш вдалим є поєднання реалізованих на ЕОМ окремих розрахункових процедур із визначенням людиною їхньої логічної послідовності.

Здатність до навчання. Людина навчається поступово, ступінь «освіченості» ЕОМ визначається її програмним забезпеченням. Бажано, щоб кількість інформації, яка отримується на запит користувача, була змінною і могла змінюватися на вимогу користувача.

Поводження з інформацією. Місткість мозку людини для збереження деталізованої інформації невелика, але мозок має інтуїтивну, неформальну можливість організації інформації. Ефективність вторинного звернення до пам'яті залежить від часу.

У ЕОМ ємність пам'яті велика, організація формальна і деталізована, вторинне звернення залежить від часу. Тому доцільно накопичувати та організовувати інформацію автоматичним шляхом та здійснювати її швидкий виклик за зручними для людини ознаками.

Оцінка інформації. Людина вміє добре розділяти значну та несуттєву інформацію. ЕОМ такою властивістю не має. Тому повинна існувати можливість перегляду макроінформації великого обсягу, що дозволяє людині вибрати частину, що його цікавить, не вивчаючи всю накопичену інформацію.

Ставлення до помилок. Людина часто припускається суттєвих помилок, виправляючи їх інтуїтивно, при цьому метод виявлення помилок найчастіше також інтуїтивний. ЕОМ, навпаки, не виявляє жодної терпимості до помилок і метод виявлення помилок суворо систематичний. Проте у сфері формальних помилок можливості ЕОМ значно більше, ніж за виявленні неформальних. Тому потрібно забезпечити можливість користувачеві вводити в ЕОМ вихідну інформацію у вільній формі, написану за правилами, близькими до звичайних математичних виразів та розмовної мови. ЕОМ виконує контроль та наводить інформацію до стандартного вигляду, зручного в процедурах обробки та формального усунення помилок. Потім бажано зворотне перетворення цієї інформації для показу користувачеві в наочній, наприклад, графічній формі, для виявлення смислових помилок.

Поводження зі складними описами. Людині важко сприйняти велику кількість інформації. Тому слід доручати ЕОМ автоматичне розбиття складних змін щодо незалежні частини, охоплювані одним поглядом. Природно, що зміни, зроблені в одній із цих частин, повинні автоматично проводитися у всіх інших.

Розподіл уваги кілька завдань. Виконати цю умову людині переважно не вдається. При вирішенні під завдання доводиться відволікатися від основного завдання. Тому в ЕОМ повинна бути організована система переривань, що відновлює стан основного завдання на момент, необхідний користувача. Аналогічним чином ЕОМ обслуговує процедуру аналізу кількох варіантів рішення.

Пам'ять щодо проведеної роботи. Людина може забути і те, що вже зроблено, і те, що йому заплановано зробити ще. Цей недолік компенсується ЕОМ, яка чітко фіксує та інформує користувача про виконані процедури та майбутню роботу.

Здатність зосереджуватись. Ця здатність у людини залежить від багатьох факторів, наприклад, від тривалості та напруги уваги, впливу середовища, загального стану. Втомою зумовлюються розсіяність, подовження реакцій, недоцільні дії. У зв'язку з цим інтерактивна система з розподілом часу має адаптуватися до часу реакції окремого користувача.

Терпіння. При багаторазовому повторенні тих самих дій людина може відчувати прикрість. Тому передбачається, наприклад, введення вихідних даних одним масивом при багаторазовому аналізі цих даних. До цього відноситься включення в систему макрокоманд або гнучких сценаріїв.

Самопочуття. ЕОМ має берегти самопочуття користувача, його почуття власної гідності та показувати йому, що саме машина його обслуговує, а не навпаки. Питання, відповіді та

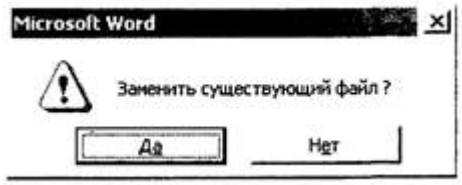
зауваження повинні відповідати розмові між підлеглим та його керівником, що визначає хід та напрямок роботи.

Емоційність. Це почуття властиве людині і далеке від ЕОМ. Програми повинні збуджувати у користувача позитивні емоції та не допускати негативних емоцій.

2.7. КЛАСИФІКАЦІЯ ТИПІВ ДІАЛОГУ ПРОГРАМ

В даний час поширені такі діалоги типу: питання-відповідь; вибір із меню; заповнення бланків; на основі команд; роботи у вікнах; за принципом електронної таблиці; гіпертексту; наближення до природної мови; віртуальної реальності.

Діалог типу питання-відповідь, поданий на рис. 2.3 є одним з універсальних. За запитом можливе введення значень різних типів. У найпростішому випадку діалог може бути реалізований із використанням трьох кнопок: так, ні, почати діалог спочатку. Проблема при реалізації діалогу полягає у скруті забезпечення перегляду користувачем усієї передісторії питань програми та відповідей на них.



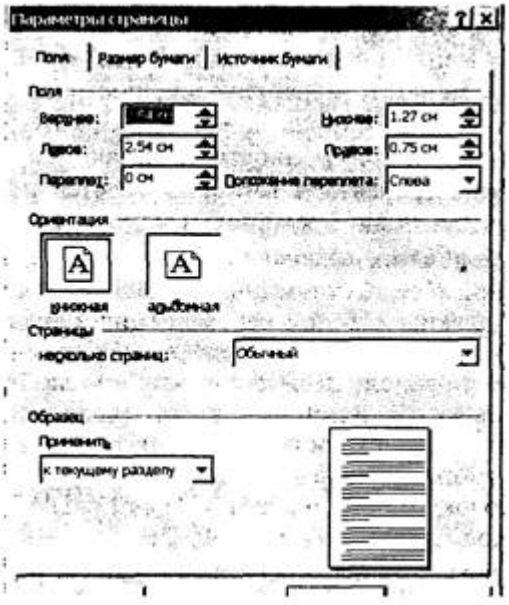
*Мал. 2.3.*Діалог типу питання-відповідь

Діалог типу меню не є універсальним. Досить складно реалізувати з використанням діалогу даного типу введення значень у широкому діапазоні. Вертикальні меню краще горизонтальних. Небажано в окремих меню пропонувати більше семи тем. Меню з великою кількістю тем бажано уявити системою ієрархічної системи підменю. Часто використовується в програмах меню в стилі фірми «Lotus» з одного головного горизонтального меню та ієрархічно побудованих інших вертикальних підменю. Рядок єдиного головного горизонтального меню надає впевненість користувачу-початківцю шляхом повідомлення йому факту про те, що є підменю в програмі. Діалог типу меню представлений на рис. 2.4.

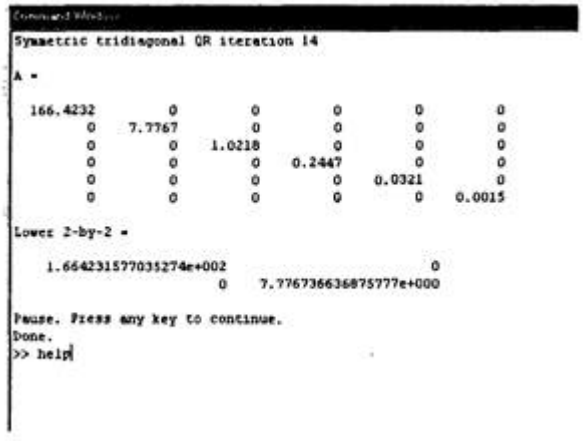
Діалог типу заповнення бланків відображає на окремому екрані якийсь бланк. За допомогою кліку миші або натискання клавіш <Tab>, а також клавіш-стрілок переміщення курсору забезпечується підведення курсору на будь-яке з виділених полів введення інформації. Зазвичай під час заповнення будь-якого поля бланка здійснюється контроль кодів натиснутих клавіш (введення по масці). Проблема реалізації діалогу полягає у прийнятті рішення за фактом введення неприпустимої за значенням інформації в полі або введення неприпустимої сукупності значень до ряду полів. Діалог типу заповнення бланків подано на рис. 2.5.

Діалог на основі команд, представлений на рис. 2.6, використовується в системах з апріорно незаданою послідовністю роботи. Складність використання діалогу цього типу полягає у необхідності попереднього вивчення користувачем мови команд та можливих послідовностей команд. Якщо команд мало у списку можливих команд, то користувачу на запит «?» може бути виданий перелік можливих команд. Після набору імені команди можливе видання списку полів команди та їх призначення.

*Мал. 2.4.*Діалог типу меню



*Мал. 2.5.*Діалог типу заповнення бланків



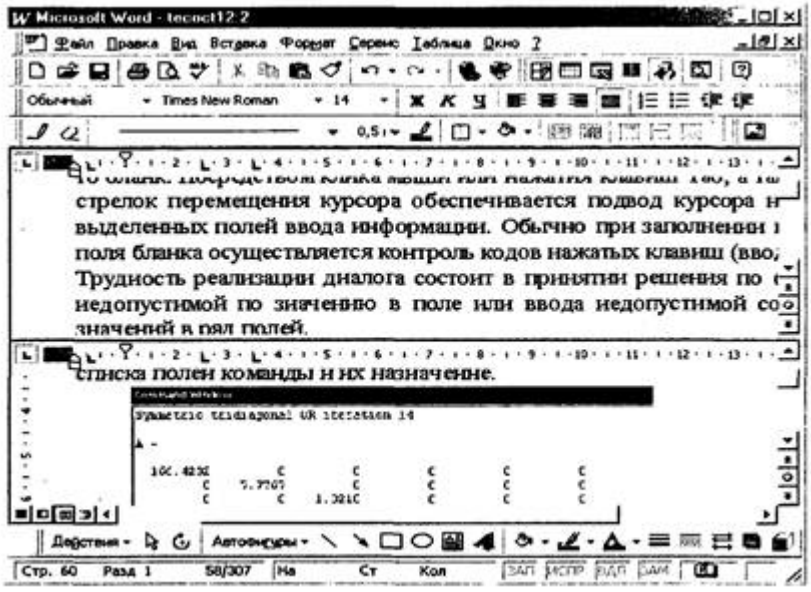
*Мал. 2.6.*Діалог на основі команд

Діалог типу роботи у вікнах (Мал. 2.7) дозволяє на екрані монітора організувати сукупність вікон як окремих програм, так і окремих документів і здійснювати роботу в кожному з окремих вікон. Даний діалог необхідний фахівцям, «у яких стіл завалений паперами», тобто фахівцям, яким для отримання нового документа потрібна інформація одразу з кількох документів.

Діалог типу за принципом електронної таблиці відображає інформацію у двовимірній системі координат за принципом гри «у морський бій» і в режимі «що бачу, те і отримаю в роздруківці» дозволяє надати табличній інформації форму, необхідну користувачеві (рис. 2.8).

Діалог типу гіпертексту(Мал. 2.9) забезпечує користувачеві перегляд на екрані монітора текстової (графічної, відео, аудіо та ін.) інформації з можливістю отримання додаткової інформації при виборі користувачем виділених на екрані ключових слів (посилань).

Діалог наближення до природної мови зазвичай забезпечує видачу користувачем запитів обмеженою природною мовою. Найбільшу складність при реалізації діалогу даного типу є складання обмеженого тезаурусу слів запиту та вивчення даного тезаурусу користувачем.



*Мал. 2.7.*Діалог типу роботи у вікнах

	E	F	G	H
1	Цена	Тип товара	Ссылка	
2	40 руб.	книга	http://www.books.ru/books/6549	
3	40 руб.	книга	http://www.books.ru/books/8341	
4	167 руб.	книга	http://www.books.ru/books/12404	
5	65 руб	книга	http://www.books.ru/books/15714	
6	40 руб	книга	http://www.books.ru/books/2289	
7	48 руб.	книга	http://www.books.ru/books/19190	
8	56 руб.	книга	http://www.books.ru/books/15389	
9	40 руб.	книга	http://www.books.ru/books/7732	
10	48 руб	книга	http://www.books.ru/books/8560	
11	40 руб.	книга	http://www.books.ru/books/701	
12	40 руб	книга	http://www.books.ru/books/9280	
13	76 руб.	компакт-диск	http://www.books.ru/books/13062	
14	77 руб	компакт-диск	http://www.books.ru/books/18668	
15	800 руб.	компакт-диск	http://www.books.ru/books/16654	

Мал. 2.8. Діалог типу за принципом електронної таблиці

*Мал. 2.9.*Діалог типу гіпертексту

Діалог типу віртуальна реальність використовується в різних тренажерах і може ґрунтуватись на використанні особливого обладнання типу кібершолом, тактильні рукавички, система запахів тощо.
Фонд різних діалогів полегшує вибір оптимального варіанта побудови зовнішніх специфікацій програм. Список дрібніших «будівельних елементів» діалогу можна отримати на панелі компонентів систем візуального програмування, наприклад Delphi.

ВИСНОВКИ

• На різних етапах проектування (особливо часто на початкових етапах) перед розробником постає завдання вибору найкращого варіанта з безлічі допустимих проектних рішень, які задовольняють вимогам. Прийняття «правильного» рішення означає вибір такої альтернативи з-поміж можливих, в якій з урахуванням усіх різноманітних факторів буде оптимізовано загальну цінність.
• Завдання оптимізації розробки програм полягає у досягненні цілей за мінімально можливої витрати ресурсів. Системний аналіз на відміну попереднього системного дослідження — це поглиблене вивчення інформаційних потреб користувачів, яке буде покладено основою детального проектування нової інформаційно-програмної системи (АС). Формулювання цілей розробки програмного продукту — перший і найважливіший етап процесу проектування. Помилки у виборі та формулюванні мети не можуть бути компенсовані на наступних етапах. Аналіз вимог сприяє кращому розумінню системи, і навіть досягненню найкращого задоволення потреби.
• Сукупність функцій системи, умов та обмежень їх існування називається безліччю споживчих властивостей системи. Функції системи — це властивості, що зумовлюють корисність (доцільність системи споживача).
• Будь-які класифікації програм підвищують можливість синтезу варіантів. Класифікації можна використовувати як під час роботи методом морфологічного синтезу, і методом аналогії.
• Перед тим, як шукати шляхи оптимізації розробки програм, необхідно виділити деякі ключові положення.
*Положення 1.*Проблема, яка має бути вирішена програмою, що розробляється, раніше якимось чином вирішувалася. Отже, необхідно вивчити методи, які раніше застосовувалися, по можливості, їх формалізувати і застосувати.
*Положення 2.*Для переважної кількості завдань у час існують програми, виконують схожі чи аналогічні функції. Тому необхідною умовою якісної розробки є ознайомлення з існуючими аналогами.

Контрольні питання

1. Перерахуйте основні види показників якості програмних систем.
2. Дайте визначення поняття "евристичний прийом".
3. Опишіть основні кроки, якими здійснюється системний аналіз.
4. Опишіть сутність проектної процедури розкриття проектної ситуації.
5. Наведіть приклади внутрішніх закономірностей під час опису споживчих властивостей системи.
6. Назвіть основні класи програм.
7. Назвіть основні особливості взаємодії людини та ЕОМ.
8. Дайте коротку характеристику основних типів діалогу програм.
9. У чому основне завдання оптимізації на етапі розробки програм?

Тема 3 ОСНОВНІ ІНЖЕНЕРНІ ПІДХОДИ ДО СТВОРЕННЯ ПРОГРАМ

3.1. ОСНОВНІ ВІДОМОСТІ

Традиційно інженери прагнули, деякі з них, не знижуючи якості проектів, домагалися значного скорочення термінів проектування. На початку Великої Великої Вітчизняної війни начальник Центрального артилерійського конструкторського бюро В.Г. Грабін розробив та застосував методи швидкісного комплексного проектування артилерійських систем з одночасним проектуванням технологічного процесу. Впровадження цього дозволило скоротити терміни проектування, виробництва та випробувань артилерійських знарядь з 30 міс (1939) до 2 — 2,5 міс (1943), збільшити їх випуск, зменшити вартість, спростити експлуатацію.

Інженерний технологічний підхід[20] визначається специфікою комбінації стадій розробки, етапів та видів робіт, орієнтованої на різні класи програмного забезпечення та особливості колективу розробників.

Основні групи інженерних технологічних підходів та підходи для кожної з них такі:

Підходи зі слабкою формалізацією не використовують явних технологій і їх можна застосовувати лише для дуже маленьких проектів, які, як правило, завершуються створенням демонстраційного прототипу. До таких підходів відносять звані ранні технологічні підходи, наприклад підхід «кодування і виправлення».

Суворі (класичні, жорсткі, передбачувані) підходи рекомендується застосовувати для середніх, великомасштабних та гігантських проектів із фіксованим обсягом робіт. Однією з основних вимог до таких проектів є передбачуваність.

Гнучкі (адаптивні, легкі) підходи рекомендується застосовувати для невеликих або середніх проектів у разі неясних або змінних вимог до системи. При цьому команда розробників має бути відповідальною та кваліфікованою, а замовники повинні брати участь у розробці.

Класифікація технологічних підходів до створення програм:

Підходи зі слабкою формалізацією

Підхід «кодування та виправлення»

Суворі підходи

Каскадні технологічні підходи:

-класичний каскадний;
- Каскадно-поворотний;
- Каскадно-ітераційний;
— каскадний підхід з видами робіт, що перекриваються;
- Каскадний підхід з підвидами робіт;
- Спіральна модель.

Каркасні технологічні підходи:

-раціональний уніфікований підхід до видів робіт.

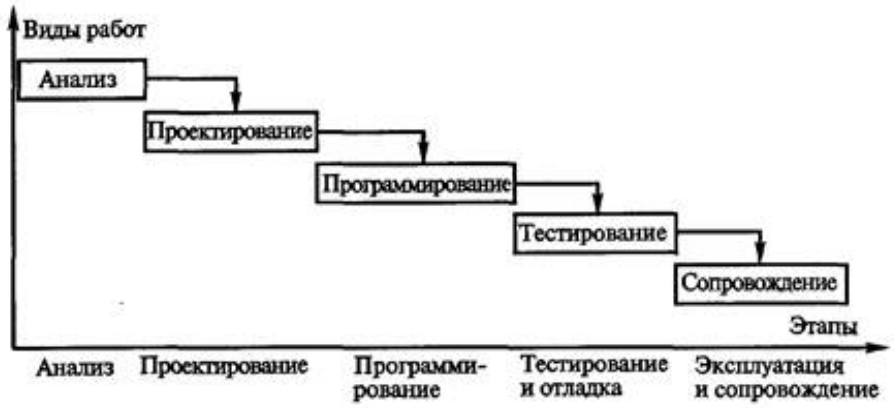
Генетичні технологічні підходи:
-синтезуюче програмування;
— складальне (розширюване) програмування;
- Конкретизуюче програмування.
Підходи на основі формальних перетворень:
-технологія стерильного цеху;
- Формальні генетичні підходи.
Гнучкі підходи
Ранні підходи швидкої розробки:
-еволюційне прототипування;
- Ітеративна розробка;
- Постадійна розробка.
Адаптивні технологічні підходи:
-екстремальне програмування;
- Адаптивна розробка;
Підходи дослідницького програмування:
-комп'ютерний дарвінізм.

3.2. РАННІ ТЕХНОЛОГІЧНІ ПІДХОДИ

Ранні технологічні підходи не використовують явних технологій, тому їх застосовують лише для дуже маленьких проектів, які зазвичай завершуються створенням демонстраційного прототипу. Як приклад підходу, не використовує формалізації, у розділі розглянуто підхід «кодування і виправлення».
Підхід «кодування та виправлення» (code and fix) спрощено може бути описаний в такий спосіб. Розробник починає кодування системи з самого першого дня, не займаючись серйозним проектуванням. Усі помилки виявляються зазвичай до кінця кодування і вимагають виправлення через повторне кодування.
Фактично кожен із програмістів так чи інакше застосовував цей підхід. При використанні цього підходу витрачається час лише на кодування та замовнику легко демонструвати прогрес у розробці рядків коду.
Цей підхід може бути рекомендований для використання у двох випадках:
• для дуже маленьких проектів, які мають завершитись розробкою демонстраційного прототипу;
• для підтвердження деякої програмної концепції.

3.3. КАСКАДНІ ТЕХНОЛОГІЧНІ ПІДХОДИ

Каскадні технологічні підходи задають деяку послідовність виконання видів робіт, що зазвичай зображується у вигляді каскаду. Іноді їх називають підходами з урахуванням моделі водоспаду.
Класичний каскадний підхід(від англ. pure waterfall - чистий водоспад) вважається "дідусем" технологічних підходів до ведення життєвого циклу. Його можна як відправну точку для величезної кількості інших підходів. Сформувався класичний каскадний підхід у період з 1970 по 1985 р. Специфіка «чистого» каскадного підходу така, що перехід до наступного виду робіт здійснюється лише після завершення роботи з поточним видом роботи (рис. 3.1). Повернення до вже пройдених видів робіт не передбачено.



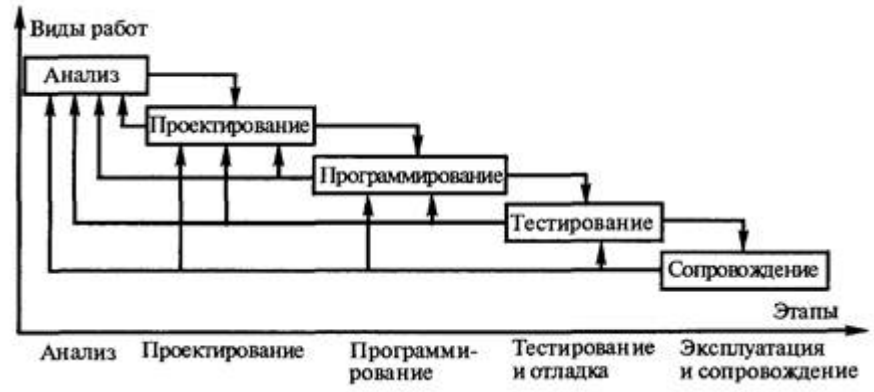
*Мал. 3.1.*Класичний каскадний підхід

Даний підхід може бути рекомендований до застосування в тих проектах, де на початку всі вимоги можуть бути сформульовані точно і повно, наприклад, в завданнях обчислювального характеру. Досить легко за такого технологічного підходу вести планування робіт і формування бюджету. Основним недоліком класичного каскадного підходу є відсутність гнучкості.

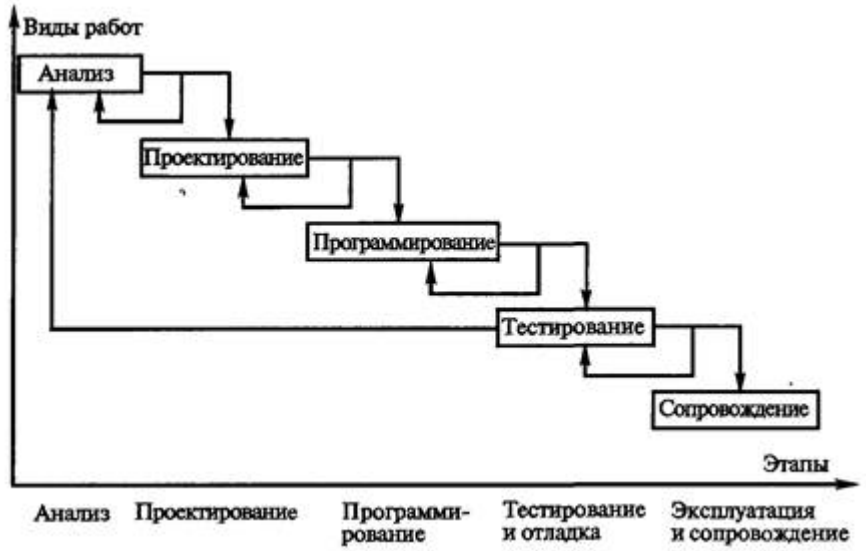
Каскадно-поворотний підхід долає нестачу класичного підходу завдяки можливості повернення до попередніх стадій та перегляду або уточнення раніше прийнятих рішень (рис. 3.2). Каскадно-поворотний підхід відбиває ітераційний характер розробки програмного забезпечення й у значною мірою реальний процес створення програмного забезпечення, зокрема і суттєве запізнення з досягненням результату. На затримку істотно впливають коригування при поверненнях.

Каскадно-ітераційний підхід передбачає послідовні ітерації кожного виду робіт доти, доки не буде досягнуто бажаного результату (рис. 3.3). Кожна ітерація є завершеним етапом і її результатом буде деякий конкретний результат. Можливо, цей результат буде проміжним, який не реалізує всю очікувану функціональність.

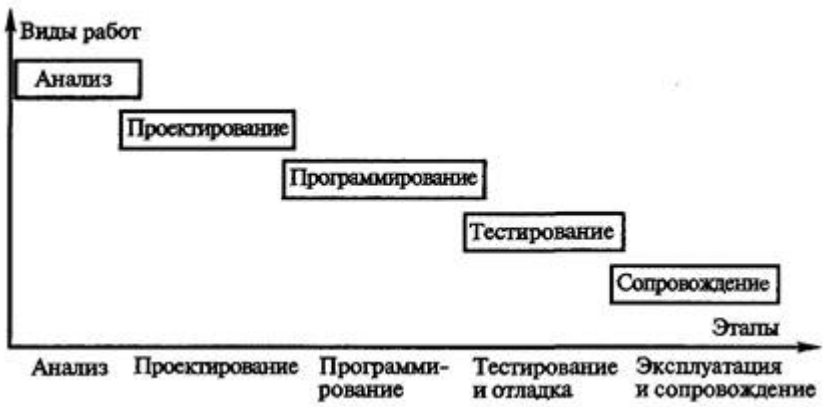
Каскадний підхід з видами робіт, що перекриваються.(англ. waterfall with overlapping), так само як і класичний каскадний підхід передбачає проведення робіт окремими групами розробників, але ці групи не змінюють спеціалізацію від розробки до розробки, що дозволяє розпаралелити роботи і певною мірою скоротити обсяг документації, що передається (рис. 3.4). .



*Мал. 3.2.*Каскадно-поворотний технологічний підхід

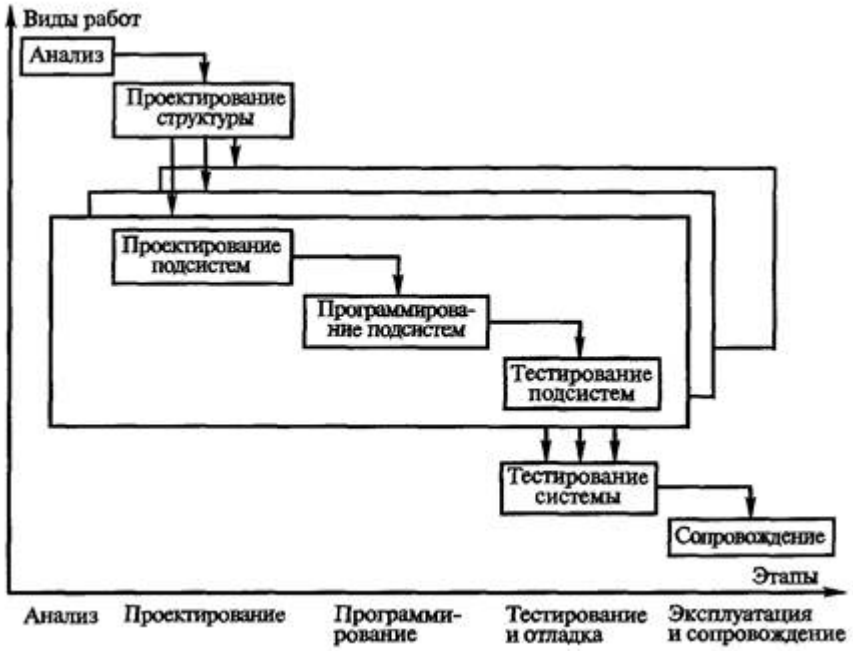


*Мал. 3.3.*Каскадно-ітераційний технологічний підхід



*Мал. 3.4.*Каскадний підхід з видами робіт, що перекриваються.

Каскадний підхід із підвидами робіт(англ. waterfall with subprocesses) дуже близький підходу з видами робіт, що перекриваються. Особливість його в тому, що з точки зору структури проект досить часто може бути розділений на підпроекти, які можуть розроблятися індивідуально (рис. 3.5). У цьому підході потрібна додаткова фаза тестування підсистем до об'єднання в єдину систему. Слід особливу увагу звертати на грамотне розподілення проекту на підпроекти, яке має врахувати всі можливі залежності між підсистемами.



*Мал. 3.5.*Каскадний підхід із підвидами робіт

*Мал. 3.6.*Спіральна модель

Спіральна модель*(Spiral model)*була запропонована Баррі Боемом (Barry Boehm) у середині 80-х років XX ст. з метою скоротити можливий ризик розробки. Фактично це була перша реакція на старіння каскадної моделі. Спіральна модель використовує поняття прототипу - програми, що реалізує часткову функціональність програмного продукту, що створюється. Створення прототипів здійснюється у кілька витків спіралі, кожен із яких складається з «аналізу ризику», «деякого виду робіт» і «верифікації» (рис. 3.6). Звернення до кожного виду роботи випереджає «аналіз ризику», причому якщо ризик перевищення термінів та вартості проекту виявляється суттєвим, то розробка закінчується. Це дозволяє запобігти більшим грошовим втратам у майбутньому.
Особливість спіральної моделі – у розробці ітераціями. Причому кожен наступний ітераційний прототип матиме більшу функціональність.

3.4. КАРКАСНІ ТЕХНОЛОГІЧНІ ПІДХОДИ

Каркасні підходи є каркасом для видів робіт і включають їх величезну кількість.

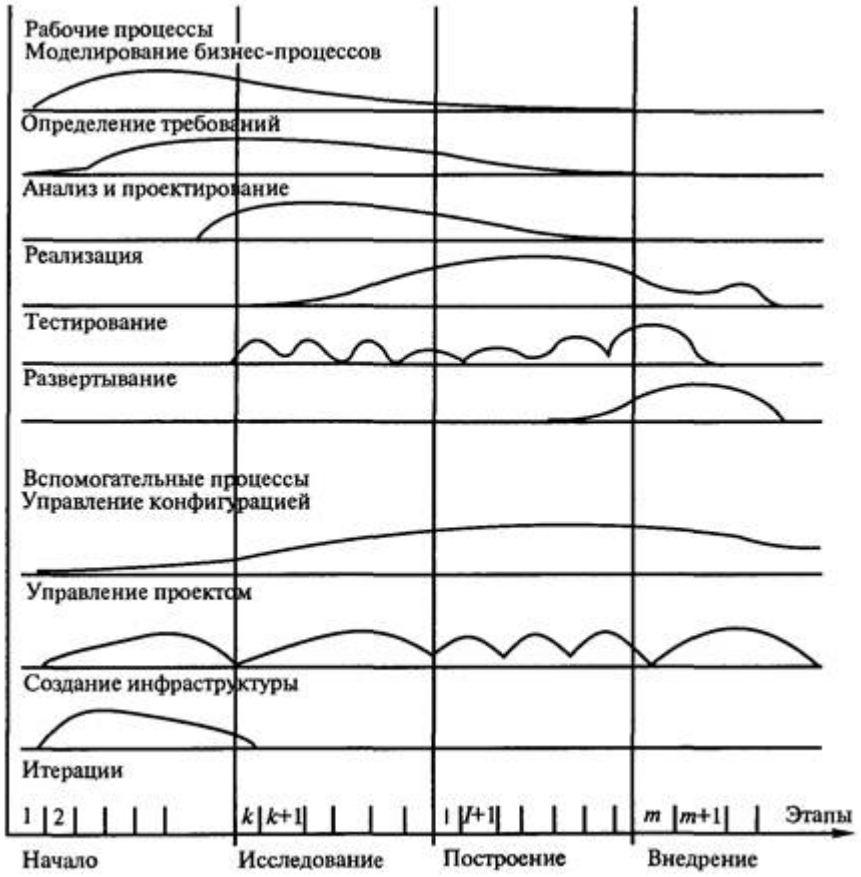
Раціональний уніфікований підхід до виконання робіт

(Rational unified process-RUP), викладений докладно в десятому розділі даного підручника, увібрав у себе найкраще з технологічних підходів каскадної групи. Вагомою перевагою цього підходу є створення інструментарію його автоматизованої підтримки — програмного продукту Rational Rose фірми «Rational Software Corpation».

З використанням підходу виділяють чотири етапи: початок, дослідження, побудова, використання. У період проходження цих етапів виконуються види робіт (наприклад, аналіз та проектування), які до того ж передбачають ітеративність їх виконання (рис. 3.7).

Основні особливості цього підходу:

• ітеративність із властивою їй гнучкістю;
• контроль якості з можливістю виявлення та усунення ризиків на ранніх етапах;
• перевага надається моделям, а не паперовим документам;
• основна увага приділяється ранньому визначенню архітектури;
• можливість конфігурування, налаштування та масштабування.



*Мал. 3.7.*Раціональний уніфікований підхід до видів робіт

3.5. ГЕНЕТИЧНІ ТЕХНОЛОГІЧНІ ПІДХОДИ

Термін «генетичний» у назві цієї групи підходів пов'язаний із походженням програми та дисципліною її створення.

Синтезуюче програмування передбачає синтез програми з її специфікації. На відміну від програми, написаної алгоритмічною мовою і призначеної для виконання на обчислювальній машині після трансляції у код, що використовується, документ мовою специфікацій є лише

базисом для подальшої реалізації. Для отримання цієї реалізації необхідно вирішити такі основні завдання:
-визначити деталі, які не можна виразити за допомогою мови специфікації, але які необхідні для отримання коду, що виконується;
- Вибрати мову реалізації та апаратно-програмну платформу для реалізації;
- зафіксувати відображення понять мови специфікацій на мову реалізації та апаратно-програмну платформу;
- Здійснити трансформацію подання (зі специфікації у виконувану програму мовою реалізації);
— налагодити та протестувати програму, що виконується.
Автоматична генерація програм специфікацій можлива для багатьох мов специфікацій, особливо для SDL, ASN.1, LOTOS, Estelle, UML (Rational Rose).
Складальне (розширюване) програмування передбачає, що програма збирається шляхом повторного використання відомих фрагментів (рис. 3.8).
Складання може здійснюватися вручну або може бути задана деякою мовою складання, або витягнута напівавтоматичним чином зі специфікації завдання. Цейтін у 1990 р. виклав основні напрями для створення техніки складального програмування:
- Вироблення стилю програмування, що відповідає прийнятим принципам модульності;
- Підвищення ефективності між модульних інтерфейсів; важливість апаратної підтримки модульності;
- Ведення великої бази програмних модулів; вирішення проблеми ідентифікації модулів та перевірки придатності щодо опису інтерфейсу. (Модулі мають стати «програмними цеглинами», з яких будується програма.)

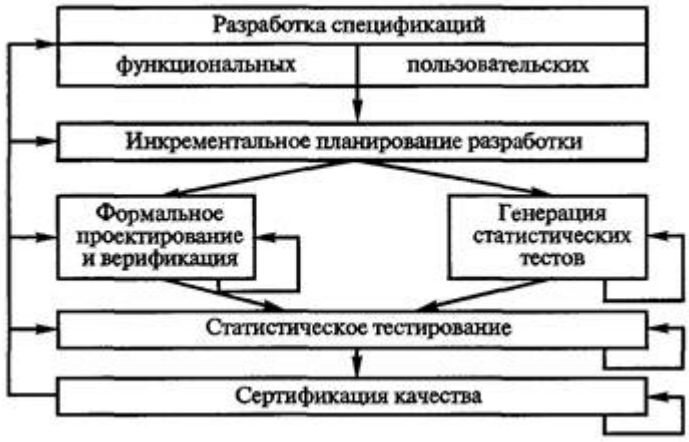
*Мал. 3.8.*Складальне програмування

Складальне програмування тісно пов'язане з методом повторного використання коду, причому як вихідного, так і бінарного. Виділяють кілька різновидів технологічних підходів складального програмування, які значною мірою визначаються базисною методологією.
1. Модульне складальне програмування - історично перший підхід, що базувався на процедурах та функціях методології структурного програмування.
2. Об'єктно-орієнтоване складальне програмування базується на методології об'єктно-орієнтованого програмування та передбачає поширення бібліотек класів у вигляді вихідного коду або упаковку класів у бібліотеку, що динамічно компонується.
3. Компонентне складальне програмування передбачає поширення класів у бінарному вигляді та надання доступу до методів класу через суворо певні інтерфейси, що дозволяє зняти проблему несумісності компіляторів і забезпечує зміну версій класів без перекомпіляції додатків, що їх використовують. Існують конкретні технологічні підходи, що підтримують компонентне програмування - COM (DCOM, COM+), CORBA, .Net. (Див. 6.6).
4. Аспектно-орієнтоване складальне програмування, у якому концепція компонента доповнюється концепцією аспекту — варіанти реалізації критичних щодо ефективності процедур. Аспектно-орієнтоване складальне програмування полягає у складанні повнофункціональних додатків з багатоаспектних компонентів, що інкапсулюють різні варіанти реалізації.

Конкретизуюче програмування передбачає, що окремі, спеціальні програми витягуються з універсальної програми.
Найбільш відома технологія конкретизуючого програмування - це підхід із застосуванням патернів проектування (див. 8.6).
Додатково до патернів існують каркаси (framework) — набори взаємодіючих класів, що становлять повторний дизайн для конкретного класу програм. Каркас диктує певну архітектуру програми, у ньому акумульовані проектні рішення, загальні для проектної галузі. Наприклад, є каркаси, які використовуються для розробки компіляторів.

3.6. ПІДХОДИ НА ОСНОВІ ФОРМАЛЬНИХ ПЕРЕТВОРЕНЬ

Ця група підходів містить максимально формальні вимоги щодо виду робіт створення програмного забезпечення.
Технологія стерильного цеху. Основні ідеї технології стерильного цеху (cleanroom process model) було запропоновано Харланом Міллзом у середині 80-х XX в. Технологія складається з наступних частин (рис. 3.9):
• розробка функціональних та користувальницьких специфікацій;
• інкрементальне планування розробки;
• формальна верифікація;
• статистичне тестування.
Процес проектування пов'язаний з поданням програми як функції у вигляді так званих «ящиків»:
• чорної скриньки з фіксованими аргументами (стимулами) та результатами (відповідями);
• ящика із станом, у якому виділяється внутрішній стан;
• прозорої (білої) скриньки, що представляє реалізацію у вигляді сукупності функцій при покроковому уточненні.
Використання ящиків визначають такі три принципи:
- всі визначені при проектуванні дані приховані (інкапсульовані) у ящиках;
— всі види робіт визначені як ящики, що використовують, послідовно або паралельно;
— кожну скриньку посідає певне місце у системній ієрархії.



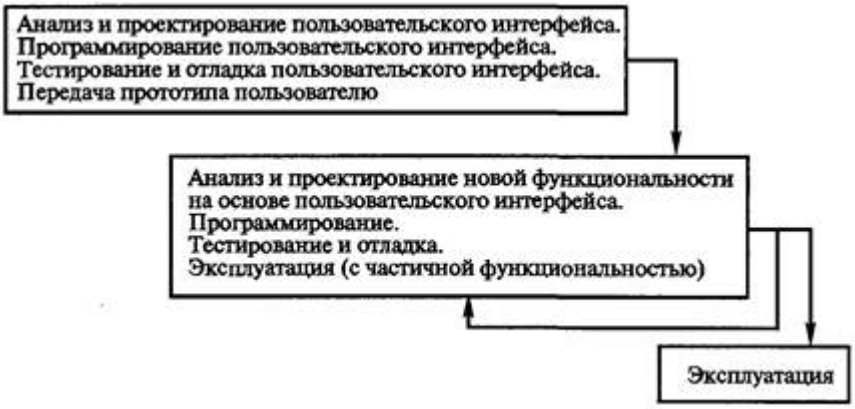
*Мал. 3.9.*Технологія стерильного цеху

Чорний ящик являє собою точну специфікацію зовнішнього, видимого з погляду поведінки користувача. Скринька отримує стимули від користувача і видає відповідь.
Прозорий ящик отримуємо із ящика зі станами, визначаючи процедуру, яка виконує необхідне перетворення. Таким чином, прозора скринька - це просто програма, що реалізує відповідну скриньку зі станом.

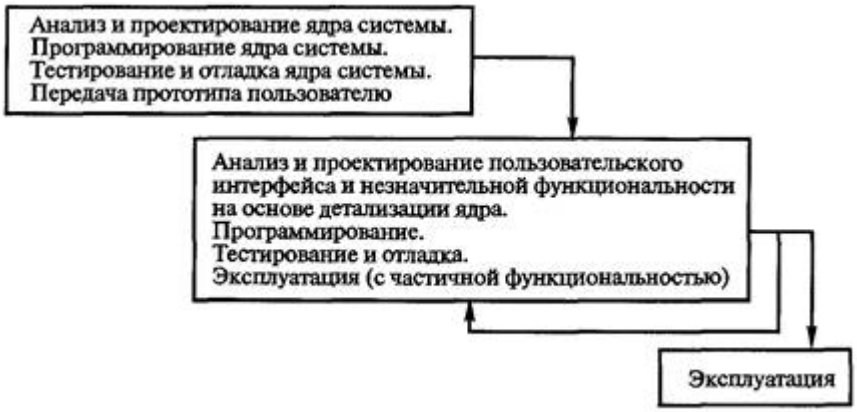
Однак у цій технології відсутня такий вид робіт, як налагодження. Його замінює процес формальної верифікації. Для кожної структури, що управляє, перевіряється відповідна умова коректності.
Технологія стерильного цеху передбачає бригадну роботу, т. е. проектування, уточнення, інспекцію та підготовку текстів ведуть різні люди.
Формальні генетичні підходи Склалися методи програмування, які мають властивість доказовості. Три такі методи відповідають вже дослідженим генетичним підходам, але з урахуванням формальних математичних специфікацій.
Формальне синтезуюче програмування використовує математичну специфікацію - сукупність логічних формул. Існують два різновиди синтезуючого програмування: логічне, в якому програма витягується як конструктивний доказ зі специфікації, що розуміється як теореми, та трансформаційний, в якому специфікація розглядається як рівняння щодо програми та символічними перетвореннями перетворюється на програму.
Формальне складальне програмування використовує специфікацію як композицію відомих фрагментів.
Формальне конкретизуюче програмування використовує такі підходи, як змішані обчислення та конкретизацію щодо анотацій.

3.7. РАННІ ПІДХОДИ ШВИДКОЇ РОЗРОБКИ

Розвитком та одночасно альтернативою каскадних підходів є група підходів швидкої розробки. Всі ці підходи поєднують такі основні риси:
• ітераційну розробку прототипу;
• тісна взаємодія із замовником.
Еволюційне прототипування. Перший прототип при еволюційному прототипуванні (evolutionary prototyping) зазвичай включає створення розвиненого інтерфейсу користувача, який може бути відразу ж продемонстрований замовнику для отримання від нього відгуків і можливих коректив. Основна початкова увага приділяється стороні системи, зверненої до користувача. Далі, доки користувач не визнає програмний продукт закінченим, до нього вноситься необхідна функціональність (рис. 3.10).
Еволюційне прототипування розумно застосовувати в тих випадках, коли замовник не може чітко сформулювати свої вимоги до програмного продукту на початкових етапах розробки, або замовник знає, що вимоги можуть кардинально змінитись.
Істотним недоліком цього підходу є неможливість визначити тривалість та вартість проекту. Неочевидною є кількість ітерацій, після яких користувач визнає програмний продукт закінченим.
Ітеративна технологія. Перший прототип ітеративної розробки (iterative delivery) вже повинен містити завершене ядро системи. Таким чином, у ньому вже зосереджена більшість функціональності. Чергові ітерації повинні допомогти користувачеві визначитися з доведенням інтерфейсу користувача та генерованих систем звітів та інших вихідних даних (рис. 3.11).
Допускається додавання незначної функціональності, що зазвичай не торкається ядра системи. Відмінність між двома дослідженими методами швидкої розробки полягає в тому, що ітеративна розробка починається зі створення ядра системи і далі деталізує його, а еволюційне прототипування орієнтоване на початкову розробку інтерфейсу користувача і додавання функціональності на його основі.



*Мал. 3.10.*Еволюційне прототипування



*Мал. 3.11.*Ітеративна розробка

Постадійна технологія (staged delivery) призначена усунути нестачу двох попередніх підходів - неможливість визначення термінів завершення проекту. Починаючи розробку, розробник досить добре знає, що буде являти собою створюваний програмний продукт. Основне завдання постадійної розробки – надати замовнику працюючу систему якомога раніше. Далі замовник зможе додавати нову функціональність та отримувати чергову версію системи. Однак кожна з версій, що отримуються після завершення стадій, є чинною.
Цей підхід вимагає ретельного та серйозного тестування чергової системи наприкінці кожної стадії перед передачею її користувачеві.

3.8. АДАПТИВНІ ТЕХНОЛОГІЧНІ ПІДХОДИ

Адаптивні технологічні підходи були задумані як підходи, які підтримують зміни. Вони лише виграють від змін, навіть коли зміни відбуваються у них самих. Дані підходи спрямовані на людини, а чи не на процес. Під час роботи в них необхідно враховувати природні якості людської натури, а не діяти їм наперекір (http://www.martinfowler.com).
Екстремальне програмування. Найбільш концентровано ідеї швидкої розробки програм виявились у підході екстремального програмування (extreme programming) (XP) (http://www.extremeprogramming.org). Дві основні риси, властиві швидким розробкам, є основними і в цьому підході. Методи, об'єднані у цьому підході, є принципово новими. Проте саме їхнє раціональне об'єднання та сукупне використання дають суттєві результати та успішно виконані проекти. Найбільшу користь підхід екстремального програмування

може принести розробці невеликих систем, вимоги до яких чітко не визначені і можуть змінитися.

Як відбувається традиційний процес розробки програмного забезпечення? Організується група аналітиків, яка прикріплюється до проекту. Група аналітиків протягом кількох годин на тиждень зустрічається з ймовірними користувачами, після чого вони випускають документацію на проект і приступають до його обговорення.

Використовуючи надану їм специфікацію, програмісти після декількох місяців випускають програмний продукт, який більш-менш відповідає тому, що від нього очікують. Найчастіше до завершення проекту ситуація може змінитися і користувачі захочуть внести зміни чи додавання, які здійснитися на даний момент не можуть. Тому програмісти, провівши тестування, здають програмний проект замовнику у вигляді, як він був замовлений. Замовник змушений розпочати фінансування розробки нової версії програми.

Екстремальне програмування дозволяє залучити кінцевих користувачів для тестування вже на ранніх етапах проектування та розробки системи. При цьому замовник звертається до розробників із проханням виготовити програмну систему. Протягом усієї роботи над проектом потрібна присутність представника замовника. Проект поділяється на три етапи:

— етап планування реалізації: замовники пишуть сценарії роботи системи на основі списку історій — можливих застосувань системи, програмісти адаптують їх до розробки, після чого замовники обирають першочергову з написаних сценаріїв;

— ітераційний етап: замовники пишуть тести та відповідають на запитання розробників, доки останні програмують;

- Етап випуску версії: розробники встановлюють систему, замовники приймають роботу.

Замовник в екстремальному програмуванні має багато можливостей вплинути на напрям роботи команди, оскільки ітерації проекту видають кожні два тижні програмне забезпечення, яке можна використовувати та тестувати. Беручи таку активну участь у розробці, замовник може бути впевненим у актуальності інформації, яка використовується в роботі системи.

В екстремальному програмуванні існує фундаментальний поділ ролей замовників та розробників, які працюють в одній команді, але мають різні права у прийнятті рішень. Замовник вирішує «що потрібно отримати», тоді як розробник вирішує «скільки це коштуватиме» і «скільки часу це займе».

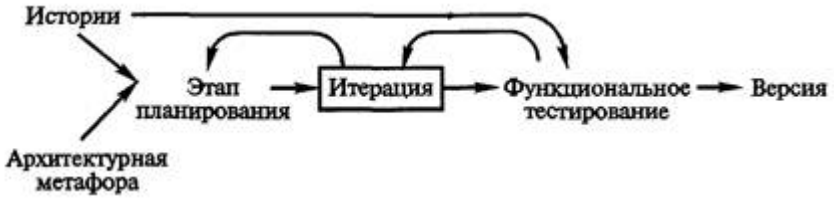
Практика екстремального програмування дозволяє розділити відповідальність між замовником та розробником. Поділ робочої сили дозволяє команді виконувати роботу точно вчасно, не втративши при цьому актуальність системи. Наприклад, якщо замовник хоче, щоб програма генерувала нові звіти на цьому тижні, розробники готові це надати. Але вони повинні повідомляти про можливі технічні ризики (якщо такі є) і вартість робіт з внесення змін.

Як вирішуються конфлікти? Що відбувається, коли замовник хоче отримати продукт до певної дати, але розробнику потрібно на його виготовлення трохи більше часу?

Екстремальне програмування пропонує кілька варіантів рішення: замовник приймає систему з меншими функціональними можливостями, замовник приймає пізнішу дату, замовник приймає рішення витратити гроші або час на розробку альтернативного варіанту або замовник може знайти іншу команду розробників.

Підхід починається з аналізу призначення системи та визначення першочергової функціональності. У результаті складається список історій – можливих застосувань системи. Кожна історія має бути орієнтована на певні завдання бізнесу, які можна оцінити за допомогою кількісних показників. Зрештою, цінність історії визначається матеріальними та тимчасовими витратами на її розробку командою розробників.

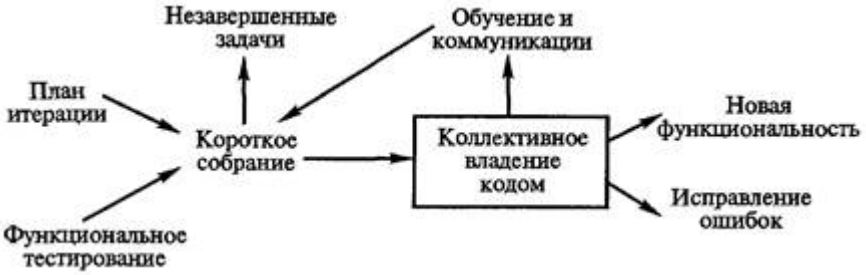
Замовник вибирає історії для чергової ітерації, ґрунтуючись на їх значущості для проекту та цінності. Для першої версії системи замовник визначає невелику кількість логічно пов'язаних найважливіших історій. Для кожної наступної версії вибираються найважливіші історії з числа історій, що залишилися (рис. 3.12).



*Мал. 3.12.*Робота над проектом на основі екстремального програмування

Одним із суттєвих методів даного підходу є функціональне тестування. Існують дві особливості процесу тестування:
• програмісти пишуть тести для тестування програми;
• ці випробування пишуться до початку кодування.
Для будь-якого автономного модуля (класу, процедури, Unit-модуля) програмісти пишуть окремий модульний тест, який має тестувати всі основні варіанти використання цього модуля та зберігатися разом із ним. Результати прогонів тестів мають бути лаконічними, наприклад, «OK! (10 tests)». Головне – тест повинен писатися до написання самого модуля! Таке тестування називають випереджальним. Внесення змін до кожної ітерації проекту (рефакторинг) завжди супроводжується прогоном всіх тестів, щоб гарантувати стабільну роботу системи. Впевненість у нормальній роботі як кожного окремого тесту, так і всіх тестів комплексного тесту надає розробникам впевненості у нормальній роботі чергової версії системи на кожній ітерації проекту.
Мета кожної ітерації (рис. 3.13) – включити до версії кілька нових історій. На зборах із планування ітерації визначається, які саме історії та яким чином вони будуть реалізовані командою розробників.
Колективне володіння кодом у процесі розробки (рис. 3.14) означає можливість кожному програміста у час удосконалити будь-яку частину коду у системі, якщо він вважатиме це за необхідне. Програміст перебирає відповідальність за реалізацію певних завдань. У разі виникнення питань щодо розроблюваного завдання можуть бути проведені короткі (15-хвилинні) збори в присутності замовника.

Мал. 3.13. Робота на ітерації екстремального програмування



*Мал. 3.14.*Колективне володіння кодом при екстремальному програмуванні

Щоб виконати завдання, відповідальний за неї програміст повинен знайти партнера (рис. 3.15). Остаточний код завжди пишеться двома програмістами на одній робочій станції. Екстремальне програмування приділяє значну увагу організації офісного простору, наголошуючи на суттєвому впливі навколишніх умов на роботу програмістів.

Адаптивна технологія. В основу підходу адаптивної розробки (Adaptive Software Development — ASD) покладено три нелінійні етапи, що перекривають один одного — обмірковування, співробітництво та навчання. Автор цього підходу Джим Хайсміт (Jim Highsmith) звертає особливу увагу на використання ідей в галузі складних адаптивних систем.

Результати планування (яке саме тут є парадоксальним) у адаптивній розробці завжди будуть непередбачуваними. При звичайному плануванні, відхилення від плану є помилкою, яку виправляють. При цьому відхилення ведуть до правильних рішень.

*Мал. 3.15.*Робота над кодом парою програмістів при екстремальному програмуванні

Програмісти повинні активно співпрацювати між собою для подолання невизначеності у підході адаптивної розробки. Керівники проектів повинні забезпечити хороші комунікації між програмістами, завдяки чому програмісти самі знаходять пояснення на питання, що виникають, а не чекають їх від керівників.

Навчання є постійною та важливою характеристикою підходу. І програмісти, і замовники у процесі роботи мають переглядати власні зобов'язання та плани. Підсумки кожного циклу розробки використовуються під час підготовки наступного.

3.9. ПІДХОДИ ДОСЛІДНОГО ПРОГРАМУВАННЯ

Дослідницьке програмування має такі особливості (http://www.osp.ru/pcworld/2001/01/062.htm):

— розробник ясно уявляє напрямок пошуку, але не знає заздалегідь, як далеко він зможе просунутися до мети;

- немає можливості передбачати обсяг ресурсів для досягнення того чи іншого результату;
- розробка не піддається детальному плануванню, вона ведеться методом спроб та помилок;

— робота пов'язана з конкретними виконавцями та відображає їх особисті якості.

В основі дослідницького програмування більшою мірою, ніж в інших підходах, лежить мистецтво.

Комп'ютерний дарвінізмНазва цього підходу було запропоновано Кеном Томпсоном (Ken Thompson). Підхід заснований на принципі висхідної розробки, коли система будується навколо ключових компонентів та програм, що створюються на ранніх стадіях проекту, а потім постійно модифікуються. Все більші блоки збираються з раніше створених дрібних блоків.

Комп'ютерний дарвінізм є методом спроб і помилок, заснований на інтенсивному тестуванні, причому на будь-якому етапі система повинна працювати, навіть якщо це мінімальна версія того, чого прагнуть розробники. Природний відбір залишить лише життєздатне.

Підхід складається з трьох основних видів робіт:

• макетування (прототипування);

• тестування;
• налагодження.
Однією з цікавих особливостей підходу є максимально можливе розпаралелювання процесів тестування та налагодження.

ВИСНОВКИ

• Традиційно інженери прагнули, а деякі з них домагалися, не знижуючи якості проектів, значного скорочення термінів проектування.
• Інженерний технологічний підхід визначається специфікою комбінації стадій та видів робіт, орієнтованої на різні класи програмного забезпечення та на особливості колективу розробників.
• Основні групи інженерних технологічних підходів: підходи зі слабкою формалізацією, суворі та гнучкі (адаптивні) підходи.
• Ранні технологічні підходи не використовують явні технології і їх зазвичай застосовують тільки для дуже маленьких проектів.
• Каскадні технологічні підходи задають деяку послідовність виконання видів робіт, що зазвичай зображується у вигляді каскаду (водоспаду).
• Каркасні підходи є каркасом для видів робіт. Одним із яскравих представників каркасного підходу є раціональний уніфікований підхід до виконання робіт, підтримуваний САПР на основі програмного продукту Rational Rose фірми Rational Software Corporation.
• Термін «генетичний» у назві групи генетичних технологічних підходів пов'язується з походженням програми та дисципліною її створення.
• Адаптивні технологічні підходи були задумані як підходи, що підтримують зміни. Вони лише виграють від змін, навіть коли зміни відбуваються у них самих. Найбільш підходом цієї групи, що бурхливо розвивається, є екстремальне програмування.
• Вибір оптимального інженерного технологічного підходу забезпечує скорочення термінів виконання проекту без зниження якості проекту.

Контрольні питання

1. За рахунок чого інженери вимагають різкого скорочення термінів виконання проектів?
2. Які відомі основні групи інженерних технологічних підходів?
3. Яка класифікаційна ознака покладена в основу поділу інженерних технологічних підходів на основні групи?
4. Які інженерні технологічні підходи розвинулися останнім часом?
5. У чому полягають переваги та недоліки каскадно-поворотного підходу порівняно з класичним каскадним підходом?
6. Який вид робіт відсутній у технології стерильного цеху?
7. У яких випадках розумно застосовувати еволюційне прототипування?
8. Які дві основні риси, властиві швидким розробкам, є базовими під час екстремального програмування?
9. У чому полягає суть адаптивної розробки?

Тема 4 СТРУКТУРА ДАНИХ ПРОГРАМ

4.1. ПОНЯТТЯ СТРУКТУРИ ДАНИХ ПРОГРАМ

Під структурою даних програм у випадку розуміють безліч елементів даних, безліч зв'язків з-поміж них, і навіть характер їх організованості.

Під організованістю даних розуміється продуманий пристрій із метою раціонального використання за призначенням. Приклади організованості даних: стек, організований масивом; структура даних для зберігання інформації про студентів; файл, що має організацію текстового файлу, байтна організація фізичної пам'яті машини.

М. Вірт визначив поняття програми так:

```
Алгоритми + структури даних = програми
```

Найпростіші структури даних, реалізовані мовою програмування, називають стандартними типами даних. Багато мов програмування дозволяють з урахуванням стандартних типів будувати типи даних, визначені програмістом (користувачем).

Що ж характеризує дані змістовніше, ніж значення? У 1973 р. М. Віртом було опубліковано статтю «Типи даних — це значення». З його погляду тип даних - це безліч значень. У статті йшлося також, що дані насамперед характеризуються набором операцій, які можна виконувати над цими даними, безліччю значень. Цей погляд і дав світові згодом деякі дуже корисні ідеї. Головна формула, якої стали дотримуватися:

```
ТИП ДАНИХ = БАГАТО ЗНАЧЕНЬ + НАБІР ОПЕРАЦІЙ
```

Важливо зрозуміти, що поняття даних та операцій дуже взаємопов'язані. Нехай є деяка структура даних, для якої існує операція Length, яка повертає довжину цієї структури деяких одиницях. Виникає питання: чи є десь дані, які називаються довжиною, чи ні. З змістовної точки зору це не має значення. Якщо ця операція застосовується до рядків, ознака кінця яких нуль (null terminated string), то обчислення довжини це дійсно операція, що вимагає обчислень. Якщо ця операція застосовується до рядків, перший байт яких означає довжину рядка, а далі йде сам рядок (як Turbo Pascal), то тут відбувається просто взяття даних з пам'яті, тобто довжина може бути операцією, а може бути даними, хоча це й не має значення для програміста.

Структури даних та алгоритми є основою побудови програм. Вбудовані в апаратуру комп'ютера структури даних представлені тими регістрами та словами пам'яті, де зберігаються двійкові величини. Закладені в конструкцію апаратури алгоритми - це втілені в електронних логічних ланцюгах жорсткі правила, за якими дані внесені в пам'ять інтерпретуються як команди, що підлягають виконанню центральним процесором.

Дані, що розглядаються у вигляді послідовності бітів або байтів, мають дуже просту організацію або, іншими словами, слабо структуровані. Для людини описувати і досліджувати складні дані в термінах послідовностей бітів або байтів дуже незручно. Завдання, які вирішуються за допомогою комп'ютера, рідко виражаються мовою бітів та байтів. Як правило, дані мають форму чисел, літер, текстів, символів та більш складних структур типу послідовностей, списків та дерев.

Мови програмування високого рівня підтримують системи формальних позначень однозначного опису як абстрактних структур даних, і алгоритмів програм. Використання

мнемоніки імен констант чи змінних полегшує роботу програмісту. Для комп'ютера всі типи даних зводяться зрештою до послідовності бітів (байтів) і мнемоніка імен йому байдужа. Компілятор пов'язує кожен ідентифікатор з певною адресою пам'яті, причому він враховує інформацію про тип кожної іменованої величини з метою перевірки сумісності типів. Людина має інтуїтивну здатність розумітися на типах даних і тих операціях, які для кожного типу справедливі. Так, наприклад, не можна витягти квадратний корінь зі слова або написати число з малої літери.

Стандартні типи даних, прийняті в мовах програмування, зазвичай включають натуральні та цілі числа, дійсні (дійсні) числа, літери, рядки тощо. Склад типів даних може різнитися в різних мовах. При виконанні програми значення змінної може змінюватися багаторазово, але її тип не змінюється ніколи. Завдяки типам компілятор може перевірити коректність операцій, що виконуються над тією чи іншою змінною. Таким чином, типи змінних багато в чому визначають структуру даних.

Програмістові, який хоче, щоб його програма мала реальне застосування в деякій прикладній галузі, не слід забувати про те, що програмування – це обробка даних. Реальний програмний продукт завжди має Замовник. Замовник має вхідні дані, і він хоче, щоб за ними були отримані вихідні дані, а якими засобами це забезпечується — його зазвичай не цікавить. Таким чином, завданням створення будь-якого програмного продукту є перетворення вхідних даних у вихідні через послідовні стани проміжних даних.

Структура даних програми багато в чому визначає алгоритми. Одне й те завдання може часто вирішуватися з допомогою різних структур даних. Для вирішення однієї і тієї ж задачі, але з структурами даних, що розрізняються, зазвичай потрібні різні алгоритми. Без попередньої специфікації структури даних неможливо розпочинати складання алгоритмів. Структура даних належить сутнісно «просторовим» поняттям: її можна звести до схеми організації інформації у пам'яті комп'ютера. Алгоритм є відповідним процедурним елементом у структурі програми — він служить рецептом розрахунку.

Перш ніж приступати до вивчення конкретних структур даних, дамо їхню загальну класифікацію за декількома ознаками.

Поняття «фізична структура даних» відбиває спосіб фізичного представлення даних у пам'яті машини і називається ще структурою зберігання, внутрішньої структурою, структурою пам'яті чи дампом.

Розгляд структури даних без урахування її представлення у машинній пам'яті називають абстрактною, чи логічною, структурою даних. У випадку між логічної і відповідної їй фізичної структурами є відмінність, внаслідок якого існують правила відображення логічної структури на фізичну структуру.

Структури даних, які застосовуються в алгоритмах, можуть бути надзвичайно складними. У результаті вибір правильного представлення даних часто служить ключем до успішного програмування і може більшою мірою позначатися на продуктивності програми, ніж деталі алгоритму.

Більшість авторів публікацій, присвячених структурам та організації даних, наголошують на тому, що знання структур даних дозволяє організувати їх зберігання та обробку максимально ефективним чином — з точки зору мінімізації витрат як пам'яті, так і процесорного часу. Іншим не менше, а можливо, і більш важливою перевагою, що забезпечується структурним підходом до даних, є можливість структурування складної програми для досягнення її зрозумілості людині, що скорочує кількість помилок при початковому кодуванні і необхідно при подальшому супроводі.

Іншим надзвичайно продуктивним технологічним прийомом, пов'язаним із структуризацією даних, є інкапсуляція, сенс якої полягає в тому, що сконструйований новий тип даних оформляється таким чином, що його внутрішня структура стає недоступною для програміста користувача цього типу даних. Програміст, який використовує такий тип даних у своїй програмі, може оперувати даними лише через дзвінки процедур.

Навряд чи з'явиться загальна теорія вибору структур даних. Найкраще, що можна зробити, це розібратися у всіх базових «цеглинках» і зібраних із них структурах. Здатність докласти ці знання до конструювання великих систем - це справа інженерної майстерності та практики.

4.2. ОПЕРАЦІЇ НАД СТРУКТУРАМИ ДАНИХ

Над усіма структурами даних може виконуватися п'ять операцій: створення, знищення, вибір (доступ), оновлення, копіювання.

Операція створення полягає у виділенні пам'яті для структури даних. Пам'ять може виділятися у процесі виконання програми за першої появі імені змінної у вихідній програмі чи етапі компіляції. У ряді мов (наприклад, С) для структурованих даних, що конструюються програмістом, операція створення включає в себе також установку початкових значень параметрів, створюваної структури.

Операція знищення структур даних протилежна за своєю дією операції створення. Не всі мови дозволяють програмісту знищувати створені структури даних. Операція знищення допомагає ефективно використати оперативну пам'ять.

Операція вибору використовується програмістами для доступу до даних усередині самої структури. Форма операції доступу залежить від типу структури даних, до якої здійснюється звернення. Під час операцій вибору використовуються покажчики. У широкому значенні слова покажчик - це змінна, що визначає місце конкретної інформації в структурі даних, наприклад, змінна, що містить значення індексу статичного масиву. У вузькому значенні слова покажчик вказує на фізичну адресу чогось. В останньому випадку покажчик — це змінна, яка є носієм адреси іншої простої чи структурованої змінної, а також процедури. Ідея, що лежить в основі концепції покажчиків, полягає в тому, щоб для досягнення контролю правильності використання покажчиків пов'язати певний тип даних із конкретним покажчиком.

Операція оновлення дозволяє змінити значення даних у структурі даних. Прикладом операції оновлення є операція присвоєння чи складніша форма — передача параметрів.

Операція копіювання створює копію даних у новому місці пам'яті.

Вищезазначені п'ять операцій є обов'язковими для всіх структур і типів даних. Крім цих загальних операцій кожної структури даних можуть бути визначені операції специфічні, що працюють лише з даними зазначеного типу (даної структури).

4.3. ЗАГАЛЬНА КЛАСИФІКАЦІЯ ЛОГІЧНИХ СТРУКТУР ДАНИХ

Упорядкованість елементів структури даних є важливим її ознакою.

Програмісти можуть на власний розсуд упорядкувати дані різних програм безліччю способів. Навіть в одній і тій же структурі даних програміст може по-різному розмістити ту саму інформацію. Так, у списку студентів прізвище може передувати імені та по батькові та, навпаки, ім'я та по батькові можуть передувати прізвищу. Максимальний елемент у відсортованому масиві може бути як першим, так і останнім. Тому характер упорядкованості елементів структури, визначений програмістом, необхідно коментувати з тією чи іншою ретельністю, що визначається здоровим глуздом та мнемонікою імен.

Існує безліч способів упорядкування інформації, але серед них є і загальні, найбільш часто зустрічаються і відомі більшості програмістів.

Приклад широко відомих структур даних із різною впорядкованістю наведено на рис. 4.1.

Структури за ознакою характеру упорядкованості їх елементів можна поділяти на лінійні та нелінійні. Приклади лінійних структур - масиви, рядки, стеки, черги, лінійні одно- та двозв'язкові списки. Приклади нелінійних структур багатозв'язкові списки, дерева, графи.

Прості та інтегровані структури даних. Прості – це вбудовані, стандартні, базові, примітивні структури даних, інтегровані – структуровані, похідні, композитні, складні структури даних. Інтегровані структури даних зазвичай відносять до типів даних, що визначаються програмістом.

Прості структури не можуть бути розчленовані на складові, більші, ніж біти і байти. З погляду фізичної структури, важливою є та обставина, що у цій машинної архітектурі, у цій

системі програмування завжди можна заздалегідь сказати, який буде розмір даного простого типу і яка структура його розміщення у пам'яті. З логічного погляду прості дані є неподільними одиницями. У мовах програмування прості структури описуються простими (базовими) типами. Прості структури даних є основою для побудови складніших інтегрованих структур.

*Інтегрованими*називають такі структури даних, складовими частинами яких є інші структури даних — прості чи, своєю чергою, інтегровані. Інтегровані структури даних конструюються програмістом з допомогою засобів інтеграції даних, наданих мовами програмування.

Мінливість структур даних також є дуже важливою ознакою. Мінливість - зміна числа елементів та (або) зв'язків між елементами структури. У визначенні мінливості структури не відображено факт зміни значень елементів даних, оскільки у разі всі структури даних мали властивість мінливості. За ознакою мінливості розрізняють структури статичні та динамічні.

*Мал. 4.1.*Приклади широко відомих структур даних

Оскільки за визначенням статичні структури відрізняються відсутністю мінливості, пам'ять для них виділяється автоматично, як правило, на етапі компіляції або при виконанні в момент активізації того програмного блоку, в якому вони описані. Виділення пам'яті на етапі компіляції є настільки зручним властивістю статичних структур, що у низці завдань програмісти використовують їх навіть уявлення об'єктів, які мають мінливістю. Наприклад, коли розмір масиву невідомий заздалегідь, йому резервується максимально можливий розмір.

У ряді мов програмування поряд із статичними змінними можуть використовуватися динамічні змінні. Динамічна змінна - це як би статична змінна, але розміщується в особливій області пам'яті поза кодом програми. У будь-який момент часу пам'ять для розміщення динамічних змінних може виділятися, так і звільнятися. Слід зазначити, що пам'ять для розміщення динамічної змінної виділяється за командою програми відразу в заздалегідь зазначеному обсязі і далі не може бути змінена, тобто структури даних, побудовані на використанні динамічних змінних, мають ту ж логічну структуру і мають таку ж мінливість, як і статичні структури даних. Тому далі динамічні змінні відноситимемо до статичних структур даних.

Фізичне уявлення динамічних змінних у пам'яті — зазвичай послідовне, як і в статичних структур, розміщення значень елементів у пам'яті.

Динамічні змінні розміщуються в області пам'яті (ДРП), що динамічно розподіляється. Область ДРП знаходиться поза областю коду програми. У зарубіжних джерелах ДРП позначається терміном heap — купа. Зазвичай заповнення області ДРП здійснюється з допомогою стандартних процедур диспетчування ДРП.

Зв'язкові динамічні структури даних. *Зв'язок*- особливий продуманий логічний пристрій збереження цілісності структури даних, елементи якої можуть перебувати в довільних, несуміжних, неконтрольованих за адресацією ділянках ДРП.

Звичайно, динамічні структури даних створюються з використанням динамічних змінних, але їхній логічний пристрій такий, що до виконання процедур доступу в програмі немає змінних, значення яких відповідають значенням елементів динамічної структури.

Динамічні зв'язкові структури, чи динамічні структури, за визначенням характеризуються відсутністю фізичної суміжності елементів структури у пам'яті, мінливістю та непередбачуваністю розміру (числа елементів) структури у процесі її обробки.

Оскільки елементи зв'язкової динамічної структури розташовуються за непередбачуваними адресами пам'яті, адреса елемента такої структури може бути обчислений з адреси початкового чи попереднього елемента. Зв'язкові структури даних пов'язані у єдину сутність системою покажчиків, які у як елементах, і статичних структурах, які забезпечують доступом до особливим елементам. Такі статичні структури називають дескрипторами. Саме таке представлення даних у пам'яті називають зв'язковим. Елемент зв'язкової динамічної структури і двох полів:

- — інформаційного поля, чи поля даних, у якому містяться ті дані (зокрема інтегровані), заради яких і створюється;
- — поля зв'язок, у кожному полі якого міститься один або кілька покажчиків, кожен із яких пов'язує даний елемент з іншими елементами структури.

Коли зв'язне уявлення даних використовується на вирішення прикладної завдання, для кінцевого користувача «видимим» робиться лише вміст інформаційного поля, а поле зв'язок використовується лише програмістом-розробником.
Переваги зв'язкового представлення даних полягають у можливості забезпечення значної мінливості структур:

- • розмір структури обмежується лише доступним об'ємом машинної пам'яті;
- • при зміні логічної послідовності елементів структури потрібно не переміщення даних у пам'яті, лише корекція покажчиків;

Недоліки зв'язного уявлення:

- • робота з вказівниками вимагає, як правило, вищої кваліфікації від програміста;
- • на поля зв'язок витрачається додаткова пам'ять;
- • доступ до елементів зв'язкової структури може бути менш ефективним за часом.

Останній недолік є найбільш серйозним і саме ним обмежується застосування зв'язкового представлення даних. Якщо суміжному поданні даних для обчислення адреси будь-якого елемента у всіх випадках достатньо було номера елемента та інформації, що міститься в дескрипторі структури, то для зв'язкового представлення адреси елемента не може бути обчислений з вихідних даних. Дескриптор зв'язкової структури містить один або кілька покажчиків, що дозволяють увійти до структури, далі пошук необхідного елемента виконується слідуванням по ланцюжку покажчиків від елемента до елемента. Тому зв'язне уявлення практично ніколи не застосовується в задачах, де логічна структура даних має вигляд вектора або масиву — з доступом за номером елемента, але часто застосовується в задачах, де логічна структура вимагає іншої вихідної інформації доступу (таблиці, списки, дерева тощо) .).
За ознакою фізичного розміщення структури даних розрізняють оперативні та файлові структури. Структури даних, які у оперативної пам'яті, називають оперативними структурами. Файлові структури відповідають структури даних зовнішньої пам'яті. Оперативна пам'ять є швидкою, а зовнішня – повільною.

4.4. КЛАСИФІКАЦІЯ ВИДІВ ОПЕРАТИВНИХ СТРУКТУР ДАНИХ ЗА ЇХ ЛОГІЧНИМ ПРИСТРІЙ

Часто, говорячи про ту чи іншу структуру даних, мають на увазі її логічне уявлення. Фізичне уявлення може відповідати логічному і, крім того, може істотно різнитися у різних програмних системах. Нерідко фізична структура, крім даних, містить прихований від програміста дескриптор або заголовок, що містить загальні відомості про фізичну структуру.

Наявність спеціальних даних дескриптора дозволяє здійснювати контрольований щодо помилок доступом до необхідних порцій даних.
*Статичний масив*така структура даних, що характеризується:
1) фіксованим набором елементів одного й того самого типу;
2) кожен елемент має унікальний набір значень індексів;
3) кількість індексів визначає мірність масиву, наприклад, два індекси - двомірний масив, або матриця, три індекси - тривимірний масив, один індекс - одномірний масив, або вектор;
4) звернення до елементу масиву виконується на ім'я масиву та значенням індексів для даного елемента.
Статичний вектор (одномірний масив)- Структура даних з фіксованим числом елементів одного і того ж типу (частковий випадок одновимірного статичного масиву). Кожен елемент вектора має унікальний номер заданого вектора. Звертання до елемента вектора виконується на ім'я вектора та номер необхідного елемента. З використанням статичного вектора можна реалізувати стеки, черги, деки тощо.
*Статична матриця (двовимірний масив)*структура даних з фіксованим числом елементів одного й того самого типу, рівним добутку кількості стовпців та кількості рядків (частковий випадок двовимірного статичного масиву). Кожен конкретний елемент матриці характеризується одночасно значеннями двох номерів номером рядка і номером стовпця. Матриця у фізичної пам'яті — вектор. Звертання до елемента вектора виконується на ім'я матриці, номер стовпця і номер рядка, які відповідають цьому елементу.
Статичний запис -кінцеве впорядковане безліч полів, що характеризуються різним типом даних. Записи є надзвичайно зручним засобом для представлення програмних моделей реальних об'єктів предметної області, бо, як правило, кожен такий об'єкт має набір властивостей, що характеризуються даними різних типів.
*Полем запису*може бути, у свою чергу, інтегрована структура даних вектор, масив або інший запис.
Найважливішою операцією для запису є операція доступу до обраного поля запису - кваліфікаційна операція. Практично у всіх мовах програмування позначення цієї операції має вигляд

```
<ім'я змінної - записи>.<ім'я поля>
```

У ряді прикладних завдань програміст може зіткнутися з групами об'єктів, набори властивостей яких перекриваються лише частково. Для завдань подібного роду розвинені мови програмування надають у розпорядження програміста записи з варіантами (union C, case Turbo Pascal).
Рядок— це лінійно впорядкована послідовність символів, що належать кінцевій множині символів, яка називається алфавітом. Говорячи про рядки, зазвичай мають на увазі текстові рядки — рядки, що складаються із символів, що входять до алфавіту
будь-якої обраної мови, цифр, розділових знаків та інших службових символів.
Базовими операціями над рядками є:
• визначення довжини рядка;
• привласнення рядків;
• конкатенація (зчеплення) рядків;
• виділення підрядка;
• пошук входження.
Операція визначення довжини рядка має вигляд функції, що повертається значення якої є цілим числом, що дорівнює поточному числу символів у рядку.
Операція присвоювання має той самий сенс, як і інших типів даних.
Порівняння рядків провадиться за такими правилами: порівнюються перші символи двох рядків. Якщо символи не рівні, то рядок, що містить символ, місце якого в алфавіті ближче до початку, вважається меншим. Якщо символи дорівнюють, порівнюються другі, треті символи і т. д. При досягненні кінця одного з рядків рядок меншої довжини вважається

меншим. При рівності довжин рядків, а головне за одночасної рівності всіх символів у рядках, рядки вважаються рівними.
Результатом операції зчеплення двох рядків є рядок, довжина якого дорівнює сумарній довжині рядків-операндів, а значення відповідає значенню першого операнда, за яким безпосередньо слідує значення другого. Операція зчеплення дає результат, довжина якого у випадку більше довжин операндов. Як і у всіх операціях над рядками, які можуть збільшувати довжину рядка (привласнення, зчеплення, складні операції), можливий випадок, коли довжина результату виявиться більшою, ніж відведений йому обсяг пам'яті. Ця проблема виникає лише у тих мовах, де довжина рядка обмежується.
Операція пошуку входження знаходить місце першого входження підрядка-еталона у вихідний рядок. Результатом операції може бути номер позиції у вихідному рядку, з якого починається входження еталона або покажчик початку входження. У разі відсутності входження результатом операції має бути деяке спеціальне значення, наприклад нульовий номер позиції або порожній покажчик.
На основі базових операцій можуть бути реалізовані будь-які інші, навіть складні операції над рядками. Наприклад, операція видалення рядка символів з номерами від n1 до n2 включно.
Статичний рядок(Тип String) в мові Pascal являє собою одномірний масив, в нульовому елементі якого знаходиться дескриптор з кількістю символів у рядку, а в наступних елементах - коди символів рядка.
Головний недолік статичного рядка - незмінність фізичної довжини, що призводить до неефективних витрат пам'яті.
Статична таблиця з фізичної точки зору є вектор, елементами якого є записи. Раніше було зазначено, що полями запису може бути інтегровані структури даних — вектори, масиви, інші записи. Аналогічно елементами векторів і масивів можуть бути інтегровані структури. Одна з таких складних структур – таблиця. Частою, характерною логічною особливістю таблиць і те, що доступом до елементам таблиці виробляється за номером (індексу), а, по ключу — за значенням однієї з властивостей об'єкта, описуваного записом-елементом таблиці. Ключ - це властивість, що ідентифікує цей запис у безлічі однотипних записів. Як правило, до ключа пред'являється вимога унікальності даної таблиці. Ключ може включатися до складу запису і бути одним з її полів, але може і не включатися до запису, а обчислюватись за положенням запису. Таблиця може мати один чи кілька ключів. Наприклад, при інтеграції до таблиці записів про студентів вибірка може проводитись як за особистим номером студента, так і за прізвищем.
Отже, основний операцією під час роботи з таблицями є операція доступу до запису по ключу — конкретного значення поля записи. Вона реалізується процедурою пошуку. Оскільки пошук може бути значно ефективнішим у таблицях, упорядкованих за значеннями ключів, часто над таблицями необхідно виконувати операції сортування.
Найпростішим методом пошуку елемента, що знаходиться в неупорядкованому наборі даних, за значенням його ключа є послідовний перегляд кожного елемента набору, який триває доти, доки не буде знайдено бажаний елемент. Якщо циклічно переглянуто весь набір, але елемент не знайдено, значить, ключ не міститься в наборі. Даний алгоритм може виявитися ефективним лише у випадку, якщо набір елементів не надто великий. За двох-трьох елементів цикл взагалі не потрібен!
Для досягнення високої за швидкістю ефективності використовують розрізняються алгоритми сортування та пошуку для роботи з оперативними та файловими структурами. Огляд різних алгоритмів сортування та пошуку наведено в [17].
*Списком*називають упорядковане безліч, що складається з змінного числа елементів, яких застосовні операції включення, винятки. Список, що відбиває відносини сусідства між елементами, називають лінійним. Логічні списки (та їх приватні види: стеки, черги, деки) можна реалізувати статичним вектором чи вектором як динамічної змінної, але у випадках на розмір списку накладаються обмеження. Якщо обмеження на довжину списку не

допускаються, то список подається у пам'яті як зв'язковий структури. Для зняття обмежень лінійні зв'язкові списки доцільно реалізовувати динамічними структурами даних. Такі списки називатимемо динамічними.

Стек- це лінійний список з однією точкою доступу до його елементів, що називається вершиною стека. Додати або видалити елементи можна лише через його вершину. Принцип роботи стека: LIFO (Last In-First Out – останнім прийшов – першим виключається).

Основні операції над стеком:

• включення нового елемента (англ. push - заштовхувати);

• виключення елемента зі скла (англ. pop - вискакувати).

Допоміжні операції:

• визначення поточного числа елементів у стеку;

• перегляд елементів стеку (наприклад, для друку);

• очищення стеку;

• неруйнівне читання елемента з вершини стека (може бути реалізоване як комбінація основних операцій: pop та push).

Черга- Це лінійна структура даних, в один кінець якої додаються елементи, а з іншого кінця вилучаються. Принцип роботи черги: FIFO (First In - First Out - першим прийшов - першим вийшов).

Грудень(Від англ. Deq - double ended queue) - особливий вид черги у вигляді послідовного списку, в якому як включення, так і виключення елементів може здійснюватися з будь-якого з двох кінців списку. Частковий випадок дека - дек з обмеженим входом та дек з обмеженим виходом.

Розгалужений список, або дерево, це список, елементами якого можуть бути теж списки. Нехай є покажчик однією елемент даних (вузол), званий коренем даного дерева. Корінь містить покажчики на ряд вузлів, кожен із вузлів ряду може містити покажчики на підпорядковані ним вузли і т. д. Вузли, які більше не посилаються на нові вузли, називають листям. Таким чином, дерево росте від вузла-кореня до вузлів-листя, розгалужуючись у вузлах. Вузли крім службової інформації про покажчики, що зв'язують дерево, містять корисну інформацію.

*Біранрне дерево*дерево, у кожному вузлі якого відбувається розгалуження тільки на два піддерева (гілки): ліве та праве.

*Лісом*називають кінцеве безліч дерев, що не перетинаються.

Граф— складна нелінійна багатозв'язкова динамічна структура, що відображає властивості та зв'язки складного об'єкта, має наступні властивості:

• на кожен елемент (вузол, вершину) може бути довільна кількість посилань;

• кожен елемент може мати зв'язок із будь-якою кількістю інших елементів;

• кожна зв'язка (ребро, дуга) може мати спрямування та вагу.

У вузлах графа міститься інформація про елементи об'єкта. Зв'язки між вузлами задаються ребрами графа, які можуть мати спрямованість, що показується стрілками. І тут їх називають орієнтованими, а ребра без стрілок — неориентированными.

Граф, всі зв'язки якого орієнтовані, називають орієнтованим графом чи орграфом; з усіма неорієнтованими зв'язками - неорієнтованим графом; зі зв'язками обох типів – змішаним графом.

Конкретні організації структур даних та алгоритми реалізації операцій з ними розглянуті у [21, 23, 25].

4.5. ПРОЕКТУВАННЯ ТА ДОКУМЕНТУВАННЯ ОПЕРАТИВНИХ СТРУКТУР ДАНИХ

Ряд розглянутих структур даних можна реалізувати з використанням статичних структур даних, динамічних змінних та динамічних структур даних. Багатоваріантність реалізації структур вимагає ухвалення конкретного проектного рішення про спосіб їх реалізації. При

прийнятті проектного рішення застосовують такі критерії, як обсяг пам'яті, можливий набір операцій, швидкість виконання операцій.
Однак тривале збереження інформації можливе лише у зовнішній пам'яті, тому проектування оперативних структур даних програми має вестися у невідривному зв'язку (паралельно) із проектуванням структури файлів програми. Дані багатьох оперативних структур повинні зберігатися у файлах та відновлюватися за інформацією, записаною раніше у файл.
Нехай потрібно спроектувати програму електронної таблиці. Такий проект виконала фірма Borland Inc, коли їй знадобилася демонстраційна програма. Обґрунтування потреби та мети розробки цього проекту було розглянуто в гол. 2.
Що бачить користувач під час роботи з електронною таблицею? - Величезний двомірний масив клітин.
Що користувач може записати у клітини? — Числові значення, рядки текстів та формули. Кожна клітина також повинна зберігати інформацію про формат виведення числових значень (формати: цілий, грошовий, науковий тощо). Отже, кожна клітина повинна містити атрибут того, що знаходиться в клітині: порожня клітина, числове значення у клітині, рядок тексту, коректна формула, некоректна формула. Нехай інформація про значення числа має тип розширений, речовий (10 байт); рядки тексту містять до 79 символів; інформація формули складається з поля зі значенням, розрахованого за формулою (10 байт), і навіть поля тексту формули (79 байт). Найдовша інформація у клітини з формулою: інформація формату (2 байти), значення, розраховане за формулою (10 байт), поле тексту формули (79 байт). Разом довжина інформації клітини становить 91 байт.
Нехай програма працюватиме з електронною таблицею розміром 100×100 клітин. Тоді інформація електронної таблиці у разі використання структури даних у вигляді статичної матриці займає 91×100×100 байт = 910 000 байт ≈ 889 кбайт.
Необхідний обсяг розміщення структури перевищує стандартну пам'ять комп'ютера класу IBM PC XT — 640 кбайт, тому треба відмовитися від використання структури даних як статичної матриці.
Провівши додатковий аналіз, з'ясовуємо, що з роботі з електронною таблицею більшість клітин залишається порожніми, т. е. електронна таблиця близька до розрідженої матриці. Що відомо про розріджені матриці?
Насправді (наприклад, під час роботи з кінцевими графами) зустрічаються масиви, які з певних причин можуть записуватися на згадку не повністю, а частково. Це особливо актуально для масивів настільки великих розмірів, що їх зберігання в повному обсязі пам'яті може бути недостатньо. До таких масивів відносять симетричні та розріджені масиви.
Наприклад, квадратна матриця, яка має елементи, розташовані симетрично щодо головної діагоналі, попарно рівні один одному, називають симетричною. Якщо матриця порядку n симетрична, то її фізичної структурі досить відобразити не n2, лише n(n + 1)/2 її елементів.
Доступ до трикутного масиву організується таким чином, щоб можна було звертатися до будь-якого елемента вихідної логічної структури, в тому числі і до елементів, значення яких, хоч і не представлені в пам'яті, можуть бути визначені на основі значень симетричних елементів. Насправді до роботи з симетричною матрицею розробляються такі процедури:

- • формування вектора;
- • перетворення індексів матриці на індекс вектора;
- • записи у вектор елементів верхнього трикутника елементів вихідної матриці;
- • отримання значення елементів матриці з упакованого її представлення.

У цьому проектному випадку немає особливої симетрії значень елементів.
Розріджений масив - масив, більшість елементів якого рівні між собою, так що зберігати в пам'яті досить невелику кількість значень, відмінних від основного (фонового) значення інших елементів. Розрізняють два їх види:

- • масиви, в яких розташування елементів зі значеннями, відмінними від фонового значення, можуть бути описані математичними закономірностями;
- • масиви з випадковим розташуванням елементів.

У разі роботи з розрідженими масивами питання розміщення їх у пам'яті реалізуються на логічному рівні з урахуванням їхнього виду.
Пам'ятаючи про це, класифікуємо випадок електронної таблиці як структуру даних як двомірного масиву з випадковим розташуванням рідкісних елементів і натомість порожніх значень.
Звідси рішення. Скористаємося гібридною динаміко-статичною структурою зберігання клітинної інформації з використанням статичної матриці. Застосуємо статичну матрицю записів розміром кількість рядків, помножену кількість стовпців. Кожен елемент матриці складається із запису з двома полями: поля формату виведення числових значень (2 байти) та поля покажчика на інформацію клітини (4 байти).
Структура даних порожньої електронної таблиці як статичної матриці тепер займає (2 + 4) * 100 * 100 = 60 000 байт ≈ 59 кбайт. Об'єм менше 64 кбайт для єдиної статичної структури відповідає можливостям Turbo Pascal.
Процедура ініціалізації порожньої таблиці полягатиме у присвоєнні кожному полю формату значення стандартного формату та покажчика значення Nil. Обсяг пам'яті, який займає статичним масивом, під час роботи програми будь-коли змінюється.
Після закінчення введення інформації в обрану клітину, якщо клітина не порожня (значення покажчика на структуру клітини *Nil), звільняється пам'ять, виділена раніше під колишню інформацію клітини. Нова інформація клітини записується в ділянку ДРП, що дорівнює за обсягом тільки корисної інформації клітини. У відповідне поле вказівника обраної клітини записується значення вказівника виділеної ділянки ДРП. Для запису тільки корисної інформації в клітини застосовуємо записи з варіантами (union C, case Turbo Pascal). Корисна інформація клітини включає постійне поле атрибута вмісту клітини, а також варіантні поля іншої інформації.
Нехай електронна таблиця заповнена 300 числовими значеннями, 200 текстовими рядками завдовжки 40 символів і 400 формулами з текстом формул по 30 символів. У цьому випадку для розміщення електронної таблиці в оперативній пам'яті потрібно всього
300 * (2 + 10) + 200 * (2 + 41) + 400 * (2 + 10 + 31) = 29400 байт ≈ 28,8 кбайт.
Як видно, при роботі з електронною таблицею обсяг інформації, яку займає динамічна структура клітин, зростає повільно. Остаточно приймаємо даний варіант реалізації, виділивши з атрибуту випадок помилки при розрахунку формули в окремий атрибут Error.
Нижче наведено приклад реалізації мовою Turbo Pascal структури даних електронної таблиці. Почнемо опис структури з глобальних описів:

```
Type
Real = Extended; {Потрібен співпроцесор}
Const
{Структура даних електронної таблиці}
MAXCOLS = 100; {Розмір таблиці}
MAXROWS = 100;
MAXINPUN = 79; {Довжина рядка, що вводиться}
{Значення атрибуту виду клітини}
TXT = 0; {У клітці текст}
VALUE = 1; {У клітці значення}
FORMULA = 2; {У клітці формула}
{Тип варіантної інформації клітин}
Type
TString = String [MAXINPUT]; {Тип рядків, що вводяться}
TCellRec = record {Тип інформації клітини}
Error: Boolean; {Поле помилки формули}
case Attrib: Byte of {Attrib – це поле}
TXT: (TextStr: TString); {У клітці текст}
VALUE: (Value: Real); {У клітці значення}
```

```
FORMULA: (Fvalue: Real; {У клітині формула}
Formula: TString);
end;
end;
{Тип покажчика на тип клітини}
TCellPtr = ^TCellRec;
{Тип елемента таблиці}
TCellTableElement = record
CellFormat: Word: {Формат клітини}
CellPtr: TCellPtr; {Покажчик на клітину у ДРП}
end:
{Тип масиву інформації клітин таблиці}
TCellsTable = array [1..MAXCOLS, 1..MAXROWS] of TCellPtr;
Var {Глобальні змінні}
Cells: TCellsTable; {Статична матриця всіх клітин}
CurCell: TCellPtr; {Покажчик на поточну клітинку}
CurCol, {Колонка поточної клітини}
CurRow: Word; {Рядок поточної клітини}
```

Як видно, з метою стислості викликів більшості процедур програми було ухвалено рішення про використання дуже невеликого набору глобальних змінних. При назві констант використані лише малі літери. Імена типів мають префікс "T". Імена, які часто використовуються в парі, вирівняні за довжиною, наприклад: MAXCOLS, MAXROWS, CurCol, CurRow. Два останніх імені, що використовуються парно, були вирівняні по довжині. При вирівнюванні скорочено слово column - колонка. Глобальні імена, що використовуються в багатьох процедурах, зроблені короткими.

Крім описаного у темі 1 рефакторингу імен можна проводити рефакторинг структури даних програми. При рефакторинг структури даних замість декількох самостійних масивів можливе використання таблиці і т. д. Особливу увагу при рефакторинг слід приділяти коментування логічної структури даних.

4.6. ФАЙЛОВІ СТРУКТУРИ

4.6.1. Фізична організація файлів

Файл — впорядкований набір інформації на зовнішньому носії (найчастіше дисковому носії). Фізична інформація файлу зовнішньому носії співвідноситься з логічною структурою даних оперативної пам'яті методами доступу операційних систем.

Зазвичай файлова система операційної системи комп'ютера містить такі засоби:

• керування файлами: зберігання файлів, звернення до них, їх колективне використання та захист;

• забезпечення цілісності файлів – гарантування того, що файл містить лише те, що потрібно;

• засоби керування зовнішньою пам'яттю (розподіляють зовнішню пам'ять для розміщення файлів).

У разі диска великого обсягу на ньому може бути багато тисяч файлів. Якщо групувати всю інформацію про місцезнаходження файлів та дескриптори файлів в одному місці, пошук конкретного файлу буде займати занадто багато часу. У цьому випадку вигідно використовувати багаторівневі каталоги файлів і системне ім'я файлу формувати з ім'ям шляху від кореневої папки (кореневої директорії) до файлу (як у UNIX, MS DOS, MS Windows) або від поточної папки (поточної директорії), в якому знаходиться файл виконуваної програми.

Дескриптор файлу або блок керування файлом може містити таку інформацію:

1) рядкове ім'я файлу;

2) тип файлу (розширення імені) - інформація для користувача про передбачувану інформацію у файлі;

3) розміщення файлу у зовнішній пам'яті;

4) тип організації файлу (прямий, послідовний, індексно-послідовний тощо);

5) тип пристрою (незнімний, знімний, що допускає лише читання тощо);
6) дані (атрибути) для контролю доступу (власник, груповий користувач, допущений та загальнодоступний користувачі);
7) диспозицію (файл постійний чи тимчасовий);
8) дату та час створення;
9) дату та час останньої модифікації.
Елементи перерахування 1, 2 та 3 визначають повне ім'я файлу.
При традиційній незв'язної фізичної організації файл може займати кілька рознесених ділянок зовнішньої фізичної пам'яті. У разі розподілу за допомогою списків секторів (блоків) сектори, що належать до одного файлу, містять посилання-покажчики один на одного. Усі вільні сектори диска містяться у списку вільного простору. Подовження або скорочення файлу змінює лише список вільних секторів. Однак логічна вибірка суміжних значень може вимагати тривалих підведення головок дисковода. Зберігання посилань зменшує обсяг пам'яті.
Найбільш загальними операціями роботи з файлами є такі операції:
• пов'язування повного імені файлу із файловими змінними;
• відкриття файлу (наприклад, для запису, лише читання, зміни довжини);
• закриття файлів;
• Встановлення атрибутів файлу.
Закриття файлу є важливою операцією. При її виконанні відбувається фізичне виштовхування інформації із файлового буфера операційної системи на зовнішній носій, а також звільняються ресурси операційної системи.
Операція встановлення атрибутів файлу дозволяє змінювати атрибути файлу, наприклад, встановлювати, що файл може використовуватись тільки для читання тощо.

4.6.2. Логічна організація файлів

Розглянемо можливості логічної організації файлів Turbo Pascal.
Оператори мови Read, ReadLn, Write, WriteLn (при файловій змінній типу Text) забезпечують роботу з файлами єдиного типізованого в мові Pascal виду - текстовими файлами, що являють собою логічно послідовність текстових рядків. Самі текстові файли логічно мають послідовну організацію. Наприклад, щоб прочитати сотий рядок, необхідно до цього прочитати всі попередні рядки. Для текстового файлу в Turbo Pascal є процедура «Append» додавання текстової інформації в кінець текстового файлу. Процедура Append повністю характеризує можливість мінливості текстових файлів (у текстових файлах навіть не можна замінити вміст одного рядка на інший рядок).
Операторами мови Read, Write (файлова змінна має тип File of тип_запису) також можна послідовно записувати у файл або зчитувати з файлу в тій же послідовності один або кілька записів певного типу (фіксованої довжини). Такі файли називають типізованими або файлами у вигляді блокованих записів фіксованої довжини. Якщо записів у типізованих файлах кілька, то за допомогою операції «Seek» можна задати будь-який номер наступного запису, що змінюється або зчитується. Таким чином, реалізовані методи як послідовного, і прямого доступу до інформації файлу, що одночасно утворює комбінований доступ.
*Файлам із довільною організацією*мовою Turbo Pascal відповідають нетипізовані файли, або бінарні. З будь-яким типізованим файлом можна працювати як із нетипізованим файлом. Нетипізовані файли в Turbo Pascal описуються за допомогою зарезервованого слова «File». Зазвичай роботу з такими файлами здійснюють за допомогою підпрограм BlockRead, BlockWritte, Seek. Також до нетипізованих файлів можуть бути використані всі стандартні засоби роботи з файлами, крім Read, Write, Flush. При використанні процедури Seek кожен блок нетипізованого файлу розглядається як фізичний запис довжиною 128 байт.
Текстові файли Turbo Pascal (як у кодуванні MS DOS, так і в Windows) зазвичай мають розширення (тип) txt і в бінарному (фізичному) поданні є одним записом довільної довжини,

що містить послідовність всіх символів рядків, що закінчуються символами «0D16», « 0A16». Останнім символом файлу (необов'язково) може бути символ 1A16, що є ознакою кінця текстового файлу. Символ 0D16 (CR) — повернення каретки без просування паперу. Символ «0A16» (LF) — рух паперу на один рядок донизу.
Таким чином, можна розглядати типізований текстовий файл як нетипізований (бінарний), що складається з одного запису у вигляді масиву символів.
Turbo Pascal практично (за винятком додавання до кінця текстового файлу) не підтримує мінливість структури файлів фізично. Щоб досягти мінливості структури файлів не тільки шляхом повільного копіювання інформації в новий файл з новою структурою, програміст змушений вдаватися до розробки процедур зміни структури існуючих файлів з використанням низькорівневого програмування на рівні блоків операційної системи. При цьому потрібний високий професіоналізм програміста для забезпечення цілісності файлів, наприклад, при відключенні комп'ютера під час виконання таких процедур.
Уникнути програмування на низькому рівні дозволяє прийом запису змін до тимчасового файлу правок. На логічному рівні старий незмінений файл і короткий файл редагування (або файл додавання в кінець старого файлу) розглядаються як єдиний файл. При виході з програми, а також у певні моменти автоматичного збереження відбувається копіювання з об'єднанням інформації старого файлу та правок файлу в тимчасовий файл. Далі старий файл перейменовується на файл з розширенням імені BAK. Нарешті, тимчасовий файл перейменовується на робочий файл. Тепер нескладно реалізувати процедури відновлення файлової інформації у разі збоїв апаратури.

4.6.3. Документування файлів

Структура файлів створюється одночасно з виявленням оперативних структур даних та з розробки процедур запису інформації у файл та зчитування інформації з файлу. Опис файлів зазвичай починається із вказівки призначення, повного імені файлу, атрибутів, диспозиції, організації та виду доступу. У документальному описі організації файлів стандартної організації досить згадати тип файлу. Наприклад, файл типу текстовий у кодуванні MS DOS. За потреби можна додатково описати порядок смислових рядків тесту.
Документування порядку проходження інформації у файлах, що складаються зі зблокованих записів фіксованої довжини і з великою кількістю полів, а також документування складних нетипізованих файлів зазвичай виконують трьома способами.
Згідно з першим способом порядок інформації у файлі визначається послідовним розглядом ланцюжків байтів файлу з використанням таблиць.
За другим способом, порядок розміщення інформації у файлі визначається коментованими описами оперативної структури даних мовами програмування, з яких здійснюється запис інформації у файл і які передбачається зчитування інформації з файлу.
Згідно з третім способом опис виконаний за другим способом, доповнюється текстами процедур «читання/запису» файлу.
Практика показала, що використання документації файлів, складеної сторонніми фірмами за другим і особливо за третім способом не викликало труднощів.
"Читання/запис" файлів зі складною довільною організацією, як правило, проводиться послідовними порціями. Перша порція зчитується у статичну запис оперативної пам'яті. Цей запис називають заголовним (header). Вона містить один або кілька байтів ідентифікації, які необхідні для перевірки автентичності файлу (приналежності до конкретних програм). У заголовній інформації може бути вказана версія файлу. Зчитування наступних порцій здійснюється як у статичні, так і динамічні зв'язкові змінні, причому їх довжина може визначатися інформацією, отриманою як з заголовної порції, так і з попередніх порцій.
Розглянемо приклад документування файлу представленої раніше електронної таблиці з допомогою таблиць структури файла. При цьому алгоритми процедур запису інформації до

файлу та зчитування інформації з файлу проектувалися одночасно з оперативними структурами електронної таблиці.

При розробці структури файлу було додано такі глобальні описи:

```
Const
{Характеристики файлу}
FILEIDENT = 'My Spreadsheet'; {Ідентифікатор}
FILESEXTENSION = 'MSS'; {Стандартний тип файлу}
Var
FeleName: String; {Ім'я файлу таблиці}
{Видима ширина колонок таблиці}
ColWidth: array [1..MAXCOLS] of Byte;
{Інформація про заповнення таблиці}
LastCol, {Остання заповнена колонка таблиці}
LastRow: Word; {Останній заповнений рядок таблиці}
```

Локальні описи:

```
Var
EndOfFile; Char; {Ознака кінця текстового файлу}
Col; Word; {Номер колонки клітини}
Row; Word; {Номер стовпця клітини}
Count; Word; {Кількість заповнених клітин таблиці}
Size; Word; {Довжина інформації клітини}
CPtr; TCellPtr; {Покажчик на клітку}
F; File; {Файлова змінна}
Blocks; Word; {Прочитано або записано байт}
```

Файл зберігання електронної таблиці є файлом постійного зберігання, бінарним довільною довжиною; має ім'я, визначене користувачем, але з розширенням імені MSS.

Організація заголовної частини файлу електронної таблиці представлена табл. 4.1.

Таблиця 4.1

Заголовна частина файлу електронної таблиці

Оперативна інформація	Довжина оперативної інформації, байт	Коментар
FILEIDENT	Length (FILEIDENT)	Константа рядка ідентифікації
EndofFile	SizeOf (EndOfFile)	Ознака кінця текстового файлу
LastCol	SizeOf (LastCol)	Остання заповнена колонка таблиці
LastRow	Sizeof (LastRow)	Останній заповнений рядок таблиці
Count	Sizeof (Count)	Число заповнених клітин таблиці на ділянці таблиці (1, 1, LastCol, LastRow)
ColWidth	Sizeof (ColWidth[1] * MAXCOLS)	Вектор ширин колонок таблиці від 1 до MAXCOLS

Запис у файл EndOfFile зі значенням 2610 = 1A16 (Ctrl + Z) забезпечує виведення на екран лише рядка ідентифікації під час перегляду файлу за допомогою більшості програм перегляду текстових файлів.

Під час читання файлу електронної таблиці зчитана інформація першого текстового рядка файлу перевіряється на збіг з FILEIDENT.

Інформація про заповнення таблиці характеризує ділянку таблиці (1, 1, LastCol, LastRow), у якого користувач вніс зміни інформації таблиці.

Значення Count під час запису розраховується з використанням двох вкладених циклів, що задають номери всіх клітин на ділянці таблиці (1, 1, LastCol, LastRow). У циклах значення Count збільшується на одиницю, якщо значення покажчика інформацію клітини ≠ Nil.

У таблиці 4.2 наведено організацію інформації чергової непустої клітини файлу електронної таблиці.

Таблиця 4.2

Інформація чергової непустої клітки файлу електронної таблиці

Оперативна інформація	Довжина оперативної інформації,	Коментар

	байт	
Col	SizeOf (Col)	Номер стовпчика клітини
Row	SizeOf (Row)	Номер рядка клітки
Cells [Col, Row].	Sizeof (Word)	Формат клітини
Size	Sizeof (Size)	Довжина інформації клітини
Фактична інформація клітини	Size	Інформація клітини

Значення Col, Row визначають збережені чи збережені у файлі координати кожної непустої клітини. Фрагмент коду програми збереження інформації непустої клітини таблиці наведено нижче:

```
if Cells [Col, Row].CellPtr <> nil then
begin
CPtr := Cells [Col, Row].CellPtr;
case CPtr^.Attrib of
TXT : Size := Length (CPtr^.TextStr) + 3;
VALUE : Size := Sizeof (Real) + 2;
FORMULA : Size : = Length (CPtr^.Formula) + Sizeof (Real) + 3;
end; {case}
BlockWrite (F, Col, SizeOf (Col), Blocks);
BlockWrite (F, Row, SizeOf (Row), Blocks);
BlockWrite (F, Cells [Col, Row]. CellFormat, Sizeof (Word), Blocks);
BlockWrite (F, Size, SizeOf (Size), Blocks);
BlockWrite (F, CPtr^, Size, Blocks);
end;
```

ВИСНОВКИ

• Під структурою даних програми розуміють безліч елементів даних, зв'язків з-поміж них, і навіть характер їх організованості. Структури даних та алгоритми є основою побудови програм.
• Структура даних може бути фізичною та логічною. У загальному випадку між логічною та відповідною їй фізичною структурами є відмінність, внаслідок якої існують правила відображення логічної структури на фізичну структуру.
• Над усіма структурами даних може виконуватися п'ять операцій: створення, знищення, вибір (доступ), оновлення, копіювання.
• Важлива ознака структури даних – характер упорядкованості її елементів. Існує безліч способів упорядкування інформації, серед яких є і загальні, найбільш часто зустрічаються і відомі більшості програмістів.
• Фізичне уявлення може не відповідати логічному уявленню та, крім того, суттєво різнитися у різних програмних системах.
• Багато з розглянутих структур даних можна реалізувати з використанням статичних структур даних, динамічних змінних і динамічних структур даних.
• Файл — впорядкований набір інформації на зовнішньому носії (найчастіше дисковому носії).

Контрольні питання

1. Що таке структура даних програми?
2. Що розуміють під організованістю даних?
3. У якій формі можуть надаватися дані?
4. Що відбиває фізична структура даних?
5. У чому різниця між фізичною та логічною структурами даних?
6. Які операції можуть виконуватись під структурами даних?
7. Наведіть приклади широко відомих структур даних.
8. Чим характеризується статичний масив?

9. Що таке рядок? Які бувають види рядків?
10. Назвіть найпростіший спосіб пошуку елемента.
11. Назвіть основні операції над стеком.
12. Назвіть процедури для роботи із симетричною матрицею.
13. Наведіть приклад реалізації мовою Turbo Pascal структури даних електронної таблиці.
14. Які засоби містить файлова система?
15. Яку інформацію містить дескриптор файлу чи блок керування файлом?
16. З чого зазвичай починається опис файлів?
17. Якими методами зазвичай виконують документування складних нетипізованих файлів?
18. Що таке рефакторинг?
19. У яких випадках може знадобитися рефакторинг імен?

Тема 5 ПРОЕКТНА ПРОЦЕДУРА РОЗРОБКИ ФУНКЦІОНАЛЬНИХ ОПИСІВ

5.1. ЗАГАЛЬНІ ВІДОМОСТІ ПРО ПРОЕКТНИЙ ПРОЦЕДУР

Розвиток окремих напрямів програмування, філології, психології, теорії проектування та штучного інтелекту наблизився до точки, коли відчувається нагальна необхідність інтеграції накопичених результатів. Спроба такої інтеграції втілилася у проектній процедурі (методиці), яка може бути застосована для складання функціональних описів (структурованих описів процесів). Інструкція використання будь-яким пристроєм, інструкція взагалі або алгоритм програми є описами функціонування. Описи функціонування можуть бути як структурованими відповідно до підходу до структурування, що викладається, так і неструктурованими. Таким чином, у цьому розділі розглядається саме проектна процедура (методика) розробки функціональних описів функціонування систем, що відрізняється використанням особливого структурування.

Програмісти можуть застосовувати проектну процедуру під час написання:

- Документальних описів обчислювальних та інших алгоритмів, призначених для тиражування;
- Інструкцій роботи користувача в системі організації (установи) з використанням ЕОМ та програми, що розробляється;
- описів зовнішнього функціонування програми у формі сценарію роботи програми (такі описи передують внутрішньому проектуванню програми);
- Внутрішніх специфікацій функціонування програми (методу вирішення задачі, алгоритму програми в цілому);
- Вихідних текстів модулів програми при використанні технології структурного програмування;
- код методів об'єктів при використанні технології об'єктно-орієнтованого програмування;
- Текстів допомоги з використання програми.

Вміння застосовувати проектну процедуру є корисним і непрограмістам. За допомогою цієї проектної процедури можна складати:

• короткі та зрозумілі описи будь-яких процесів;
• накази та розпорядження на виконання робіт;
• інструкції щодо користування виробами;
• опис принципів функціонування виробів;
• опис бізнес-процесів;

• бухгалтерські інструкції щодо проведення розрахунків;
• тексти посадових вказівок організаційного забезпечення.
Цей список не є вичерпним і можливі нові застосування.
Докладніше актуальність розробки функціональних описів поза сферою програмування характеризується такими прикладами.
«Копати траншею від паркану до обіду» — невдале розпорядження (недостатньо повно виявлено та вказано вхідну та вихідну інформацію, а також відсутня мета).
Припустимо, що наприкінці червня якийсь вітчизняний бухгалтер звернувся із проханням про відпустку. Головний бухгалтер, найімовірніше, дасть приблизно таку відповідь: "Не дам жодної відпустки до здачі піврічного звіту". Уявімо, що є набір щоденних детально вичерпних інструкцій по роботі даного бухгалтера. Набір таких інструкцій називається описом бізнес-процесів. І тут і у разі хвороби робота бухгалтера передається резервному спеціалісту. Припустимо, що зверху спущений якийсь документ, згідно з яким треба подати якісь дані, які раніше не розраховувалися. Швидше за все, цей документ є заплутаною інструкцією, що допускає різночитання або навіть містить помилки для застосування в деяких особливих випадках. Джерелом таких помилок може бути не злий намір укладача інструкції, а нездатність її упорядника охопити всі особливі випадки з величезного (астрономічного!) кількості шляхів виконання інструкції, розробленої за традиційною операційною методикою. Тепер головний бухгалтер починає обдзвонювати колег та інстанції, сподіваючись отримати роз'яснення. Нехай він розібрався в інструкції. Тепер перед ним постає завдання по віддачі розпоряджень працівникам бухгалтерії як працювати за умов, що змінилися. Багато бухгалтерів, не знаючи основ сучасної алгоритмізації, швидше за все, звернуться до програмістів, які добре оплачуються, для уникнення труднощів у віддачі розпоряджень.
На відміну від бухгалтера, який документально фіксує всі свої дії, виконроб на будівництві свої розпорядження з виробництва робіт готує та віддає усно. Це пояснюється традиційним ухиленням від відповідальності у разі травматизму робітників. Усна підготовка розпоряджень призводить до таких помилок, як щось не вдається розмістити у вже збудованій частині будівлі. Таким чином, будівництво ведеться циклічним процесом: будувати – ламати – будувати тощо, що веде до подорожчання будівництва.
Продаж і навіть перепродаж готових технічних виробів за чинним законодавством передбачає наявність інструкції користування виробом російською мовою. Однак часто виявляється, що інженери складають такі довгі та заплутані інструкції користування об'єктами техніки, що споживачі починають експлуатацію виробу з неприпустимих дій, які можуть призвести до виробу навіть до неремонтно-придатного стану.
Крім апробації у сфері програмування автори підручника провели апробацію викладених у ньому методик під час навчання непрограмуючих спеціальностей. Виявилося, що навчання методики розробки функціональних описів (на прикладах складання інструкцій взагалі, описів бізнес-процесів) цілком доступне студентам другого курсу спеціальності бухгалтерський облік, навіть якщо вони не вивчали цю методику в курсі програмування. Понад те, половина учнів дев'ятого класу звичайної школи цілком здатна повністю освоїти цей матеріал. Слід зазначити такий факт: до навчання лише кілька учнів класу реально могли написати план, та був твір. Після навчання майже три чверті учнів могли написати план, та був його розвинути у твір, т. е. школярі реально освоїли елементи дедуктивного мислення! Витрати освоєння матеріалу склали 8 год лекційних і 16 год практичних занять. Таким чином, у навчальних всього за 24 години навчальних занять вдається розвинути первинні навички дедуктивного мислення і володіння початковими методами системного підходу.
Написання технічної документації – особливий жанр письменницького мистецтва. Нині у розвинених країнах з'являється нова спеціальність Technical Writer – технічний письменник. Ймовірно, одна із сфер застосування проектної процедури полягає у її використанні такими фахівцями.

Хорошим функціональним описом є безпомилковий опис, однозначний для читача, короткий, суть якого розуміється швидко. Згідно з проектною процедурою, хороший функціональний опис складається від загального до приватного з використанням особливих конструкцій пропозицій — типових елементів (типових структур або просто структур), що становлять семантичний скелет майбутніх інструкцій.

Зазвичай людина мислить пропозиціями природної мови. Якщо навчитися впорядковувати думки у процесі мислення, можна навчитися отримувати алгоритми та інші функціональні описи зі швидкістю як не меншою, ніж до навчання, і навіть більшої. Досвідчений програміст пише текст мовою програмування зі швидкістю, з якою він думає, а разі простих алгоритмів — зі швидкістю набору тексту на клавіатурі. Інструкції, призначені для виконання людьми, можуть містити алгоритми і евроритми.

Одна з переваг застосування проектної процедури полягає в зниженні розумової втоми програміста або укладача інструкцій за рахунок виключення необхідності неодноразового повторення розумового процесу для отримання однієї і тієї ж ідеї, що забувається.

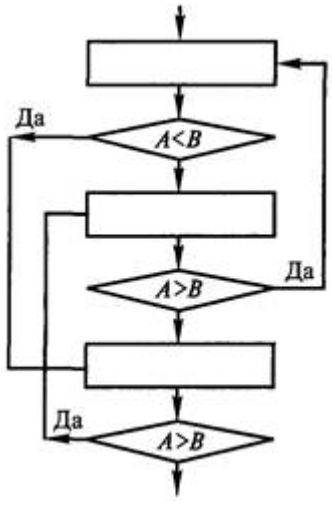
Головна перевага полягає в однозначності відповідності функціонального опису задуму, що досягається вичерпним тестуванням. При операційному підході до складання описів функціонування вичерпне тестування принципово неможливе через складність розв'язуваної задачі (потрібно відразу відтестувати велику і складну структуру - всю програму чи інструкцію).

Ще одна перевага полягає у отриманні само документованих текстів програм. Само документовані програми виходять шляхом застосування спеціального стандартизованого способу оформлення текстів програм з використання коментарів та стандартних типових структур кодування.

Використання стандартних типових структур передбачає особливу декомпозицію алгоритму програми чи евроритму інструкції за принципом від загального до приватного, що вимагає від розробника володіння дедуктивним мисленням.

5.2. ІСТОРІЯ ВИНИКНЕННЯ ПРОЕКТНОЇ ПРОЦЕДУРИ

З появою ЕОМ актуальним став пошук методів опису обчислювальних алгоритмів. У 60-х роках вже застосовувалися два способи опису алгоритмів: словесний покроковий та графічний у вигляді схем алгоритмів програм (жаргонно: блок-схем алгоритмів).



*Мал. 5.1.*Зображення алгоритму у формі графічної схеми алгоритму

При словесно-покроковому способі алгоритми описувалися за наведеним нижче принципом.

*Крок 1*Виконується така дія для того-то. Якщо виходить, що A <, то переходимо до кроку 4.

*Крок 2.*Виконується така дія для того-то.

*Крок 3*Якщо A > B, переходимо до виконання кроку 1.
*Крок 4.*Виконується така дія для того-то.
*Крок 5.*Якщо A > B, переходимо до виконання кроку 2.
Зображення цього алгоритму у вигляді схеми алгоритму наведено на рис. 5.1.
Недоліки кожного із способів наведені у табл. 5.1, з якої видно, що графічний спосіб у вигляді схем алгоритмів програм полегшив лише відстеження передач управління та одночасно ускладнив опис суті процесів (їх коментування).
Таблиця 5.1
Недоліки словесно-покрокового та графічного способів у вигляді схем опису алгоритмів програм

Спосіб опису	Недоліки
Словесно-покроковий	Незрозуміло, що є головним, а що другорядним (щось можна зрозуміти лише після індукції основного задуму). Важко відстежуються передачі керування Неможливість ефективного тестування.
Графічний у вигляді схем алгоритмів програм	Незрозуміло, що є головним, а що другорядним (щось можна зрозуміти лише після індукції основного задуму). Важко записувати коментарі. Для розуміння схеми алгоритмів необхідно доповнювати досить довгими текстовими описами, які мають містити велику кількість тестових даних різних маршрутів обчислень. Неможливість ефективного тестування.

До кінця 70-х років проектна процедура отримання алгоритмів базувалася на операційному (маршрутному) мисленні, яке закладається ще у школі математичними та фізичними навчальними дисциплінами. Операційний підхід не вимагає вільного володіння дедуктивним мисленням і ґрунтується на більш простому і вже освоєному індуктивному мисленні «від приватного до спільного». При операційному мисленні спочатку записуються послідовні дії головним основним маршрутом. Потім ці дії доповнюються операціями розгалуження (if), операціями безумовного переходу (go to) та додатковими діями інших маршрутів.
В результаті використання операційного підходу виходили програми (підпрограми) зі структурними елементами у вигляді операторів мови програмування та єдиною надскладною структурою "вся програма" типу "спагетті" - безліччю невпорядкованих передач управлінь вперед і назад (див. рис. 5.1). Все це можна зарахувати і до більшості текстових інструкцій. Програми, алгоритми яких виходять операційним підходом, були практично несупроводжуваними. Навіть під час використання графічних схем алгоритмів їх потрібно було доповнювати досить довгими текстовими описами. Ці текстові описи мали містити величезну кількість окремих тестових даних для відстеження всіх маршрутів обчислень з метою розуміння алгоритму кожного окремого маршруту. Складність таких описів призводила до невідповідності коментарів обчислювальним процесам, а також невиправдано тривалого аналізу алгоритму з прорахунком траси рахунку по всіх маршрутах в процесі налагодження. При використанні схем алгоритмів програм поза документацією залишається перебіг думок проектувальника щодо отримання алгоритмів. Саме креслення схем алгоритмів програм займає досить багато часу. Величезне, астрономічне число обчислювальних маршрутів вимагало підготовки астрономічної кількості тестів. Програми були ненадійними, порівняно з нинішніми програмами. Налагодження програми у вигляді однієї нероздільної структури «спагетті» і довжиною всього 600 рядків займало час до півроку. Написання та налагодження такої програми за сучасних технологій проводиться програмістом за два-три робочі дні.
Відповідно до сучасних технологій програмування, описи алгоритмів у словесно-покроковій та графічній формах у вигляді схем алгоритмів програм практично не використовуються. Їх замінили самодокументовані тексти, що складаються із стандартних структур кодування. Складання опису проектної процедури передували праці Дейкстри (з концепцією програмування без go to), однією з перших робіт з структурного кодування програм [9] і досвід програмування і викладання авторів.

5.3. ЗАГАЛЬНИЙ ОПИС ПРОЕКТНОЇ ПРОЦЕДУРИ

Принципи опису послідовності дій для алгоритмів та єврроритмів практично не відрізняються. В описах алгоритмів зазвичай є лише більша формалізація. При написанні програм програміст зазвичай використовує комп'ютер. Як правило, алгоритми більшості процедур програм є найпростішими. Зазвичай код таких процедур досвідчені програмісти пишуть одразу сидячи за монітором. Кодування навіть алгоритмічно складних процедур можна здійснювати з використанням комп'ютера, адже сучасний комп'ютер відмінно замінює папір та олівець! Просто треба паралельно у двох вікнах монітора розробляти документальний опис процедури та кодувати текст коду самої процедури. Навіть якщо код програми є само документованим, окремий документ міститиме детальний опис структури даних усієї програми, а також наочні тести, що детально характеризують весь алгоритм. Наявність такої документації спростить супровід програми.

Виконання проектної процедури починається з першого кроку, який полягає у повному з'ясуванні завдання зовнішньому рівні. Цьому допомагають набори тестових прикладів та модель «чорної скриньки». Набори тестових прикладів сприяють обліку всіх можливих випадків роботи алгоритму чи евроритму. Модель «чорного ящика» сприяє виявленню цільового призначення алгоритму, евроритму (інструкції) або іншої штучної системи, що розробляється. Модель «чорної скриньки» також допомагає виявляти вхідну та вихідну інформацію програм та інструкцій або визначити матеріальні, енергетичні та інформаційні входи та виходи інших штучних систем, що розробляються. Існують алгоритми та єврроритми, складання яких можна починати або з аналізу «чорної скриньки», або з підготовки первинних тестів. Також відомі алгоритми, розробки яких до якісного завершення робіт доводилося багаторазово перемикатися як у роботу з «чорним ящиком», і на складання первинних текстів.

При розробці алгоритмів програм вхідна, проміжна та вихідна інформація характеризується структурою даних, розміщених в оперативній та зовнішній пам'яті, та інформацією, що відображається на екрані монітора.

При розробці єврроритмів інструкцій вхідна, проміжна та вихідна інформація характеризується набором предметів (фізичних та документів) та їх станом, наприклад: порожній чайник; чайник, заповнений наполовину обсяг холодною водою; включений вимикач «мережа» у правому верхньому куті панелі; документ за формою № 5 із заповненою першою графою. Тут доречно доповнювати опис предметів їх малюнками. Враховуйте, що предмети змінюються у часі та просторі.

Первинні тестові приклади повинні включати як звичайні, і стресові набори тестових вхідних даних. Кожен стресовий набір тестових даних призначений виявлення реакції у випадках, наприклад неправильних дій користувача, поділу на нуль, виходу значення допустимі кордону, невідповідності інструкції стану справ тощо. буд. Будь-який набір тестових даних має містити опис результату.

Другий крок складання єврроритму або алгоритму починається з розробки узагальнюючих тестів або узагальнюючого тесту, що включають як найменший набір тестів всі випадки з первинних тестів.

На основі узагальнюючих тестів, а також виявлених входів та виходів системи готується наочний узагальнюючий тест або кілька наочних узагальнюючих тестів. Наочний тест повинен відображати ланцюжок перетворення інформації від вихідної інформації через проміжну інформацію до результуючої інформації. Найголовніша вимога до узагальнюючого тесту – його наочність у поданні порядку зміни інформації. Відомо, що з вивченні геометрії успіх розв'язання завдання визначається наочністю малюнків. Наочні геометричні малюнки зазвичай виходять багатоваріантним їх побудовою, і потрібно придбання деяких навичок у розвиток мистецтва їх виконання. Усе це стосується і складання наочних узагальнюючих тестів. Наочність узагальнюючих тестів досягається правильним вибором способу відображення інформації, що забезпечує сприйняття порядку перетворення інформації.

Наочні уявлення зазвичай грунтуються моделі типу абстракції даних, викладеної в гол. 1. Однією її форм може бути функціональна модель як набору діаграм потоків даних (ДПД). Інша форма може відповідати малюнку з раціональним способом розміщення його площі масивів з їх іменами, значеннями елементів, значеннями індексів елементів, значеннями і іменами простих змінних, з'єднаними стрілками у напрямку передачі інформації. Ще однією формою є форма, близька до траси рахунку, що показує зміни значень змінних на етапах процесу. Можливі інші форми, наприклад таблиці правил.

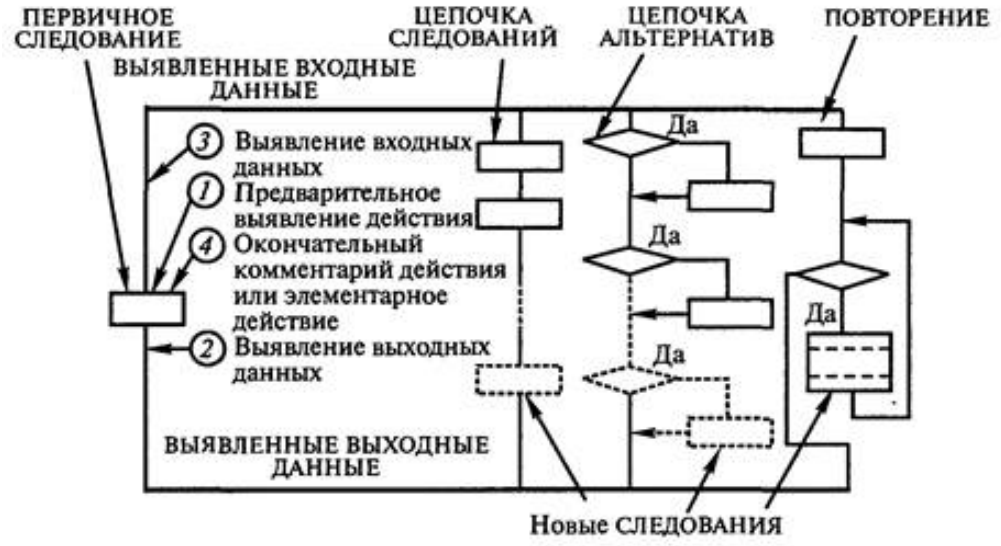
Наголосимо на важливості роботи з даними. Дані багато в чому визначають алгоритми. При вирішенні однієї і тієї ж задачі, вибір різних структур даних призводить до алгоритмів або єврроритмів, що абсолютно розрізняються.

Подальше виконання проектної процедури полягає у отриманні на підставі моделі абстракції даних моделі процедурної абстракції, також викладеної в гол. 1.

Будь-які алгоритми або єврроритми повинні складатися тільки зі стандартних структур, кожна з яких має суворо один інформаційний вхід і один інформаційний вихід. Використання інших (нестандартних) структур призводить або до подовження опису, або до неможливості тестування (через неможливо великого обсягу необхідних тестів), або до втрати зрозумілості. Втрата зрозумілості відбувається через те, що в неструктурованому алгоритмі чи єврроритмі одні й ті самі частини алгоритму чи єврроритму за одних даних виконують одне, а за інших — інше. Тому частини неструктурованих алгоритмів чи евроритмів неможливо однозначно характеризувати засобами природної мови.

Структурі СЛІД в інструкціях і програмах відповідає суворо одна дія.

Далі проектна процедура виконується ітеративними кроками: до досягнення елементарних дій (елементарних операторів мови програмування або елементарних операцій) окремі структури СЛІД, з яких складається опис будь-якого алгоритму або евроритму, декомпозуються з дотриманням принципу від загального до одного з трьох стандартних структур (рис. 5.2).): ланцюжок Слідкувань; Ланцюжок АЛЬТЕРНАТИВ; ПОВТОРЕННЯ. У разі довгого алгоритму вичерпання інформації узагальнюючого тесту готуються нові узагальнюючі тести під нові завдання структури.



*Мал. 5.2.*Виявлення виду чергової структури під час виконання проектної процедури розробки функціональних описів

З трьох виявлених структур будь-яка структура містить у собі одну або кілька нових структур виду СЛІД з більш приватними діями. Ці нові СЛІДКИ можуть піддаватися декомпозиції на наступній ітерації виконання проектної процедури.

Отже, описи послідовності процесів для алгоритмів і евроритмов мало розрізняються, тому розробку розглянемо разом. Процес кодування програм, суміщений з документуванням, передбачає різноманітні роботи і потребує особливих знань, тому кодування програм розглянемо окремо. Зазначимо, що алгоритм ряду програм (наприклад, математично не складних) та код програм народжуються одночасно, причому народжуються на основі думок програміста. Звідси випливає, що ознаки алгоритмічних типових структур та порядок їх деталізації, описані в даному підрозділі, є одними й тими самими для складання алгоритмів, евроритмів та програм.

На кожній ітерації проектної процедури доводиться вирішувати завдання: «А яка саме із трьох структур буде виявлена?» При вирішенні цього завдання необхідну інформацію можна отримати лише з аналізу узагальнюючих тестів. Аналіз тестів щодо пошуку найголовнішої на даний момент структури є для початківців дуже непростою справою. «Розкріпляти мислення» допомагає набір ознак структур, викладений у табл. 5.2 а також набір евристичних прийомів, викладений далі.

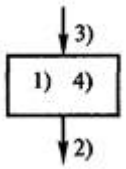
Таблиця 5.2.

Зведена таблиця характеристик структур та ознак структур

Структура	Характеристики	Ознака
СЛЕДЖЕННЯ	Описується або простими поширеними реченнями природної мови, або реченнями без присудка (наприклад, «Навантаження меблів», «Рішення квадратного рівняння»)	Відповідає строго одній дії
Ланцюжок слідів	Являє собою ланцюжок із послідовно виконуваних дій	Послідовно виконувані різнорідні дії
Ланцюжок альтернатив:		Одна або кілька дій, кожна з яких виконується за певної умови або не виконується взагалі
проста АЛЬТЕРНАТИВА	Описується конструкцією: «Якщо виконується якась умова, то виконується СЛІД 1»	
АЛЬТЕРНАТИВА із двох дій	Описується конструкцією: «Якщо виконується якась умова, то виконується СЛІД 1, в іншому випадку виконується СЛІД 2»	
ВИБІР	Є ланцюжком з більш ніж двох найпростіших альтернатив з однією дією	
ПОВТОРЕННЯ:		Багаторазово виконувана дія (але обов'язково кінцеве число разів). Повторенням відповідають думки: «Ця дія має бути виконана п'ять разів»; «Ця дія виконується багаторазово до настання такої події». Ознаками ПОВТОРЕНЬ також є змінна кількість АЛЬТЕРНАТИВ, будь-яка думка про повернення «назад», щоб повторити якісь дії. Часто головний загальний
ПОВТОРЕННЯ «ДО»	Описується конструкцією: «До виконання якоїсь умови багаторазово виконувати СЛІД»	
ПОВТОРІННЯ «ПОКИ»	Описується конструкцією: «Поки виконується якась умова, багаторазово виконувати СЛІД»	
НЕУНІВЕРСАЛЬНЕ	Забезпечує задану кількість	

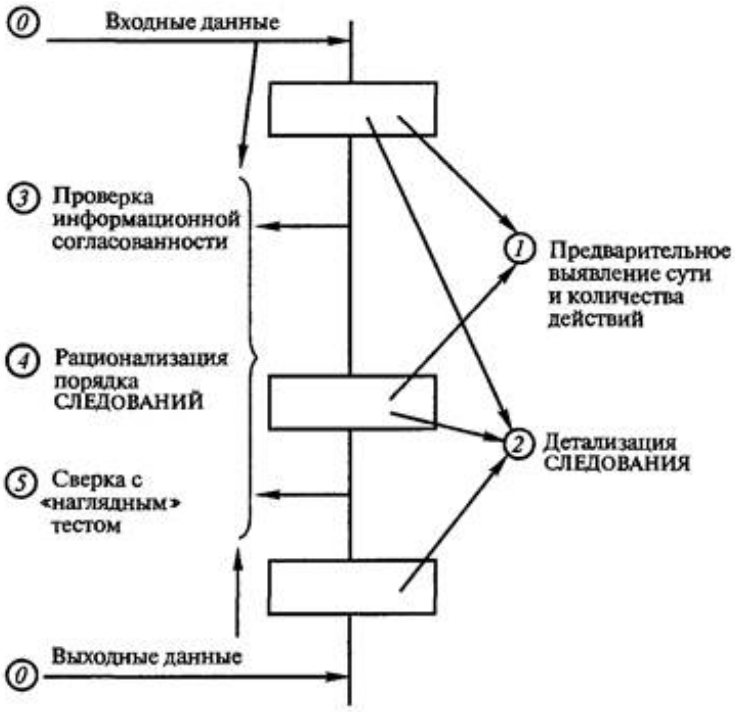
ПОВТОРЕННЯ	повторень	процес виду повторення прихований у контексті «і т. д.» або «і т. п.», «це зовсім просто», або навіть у крапках «...»

Набір евристичних прийомів:
1. «Хороші наочні ілюстрації – запорука успіху!».
2. "Думай від спільного до приватного!".
3. "Загальний процес визначає роботу приватних!".
4. «Це не головний процес, ви загрузли в частковості!».
5. «Не забувай запроваджувати нові терміни (імена змінних)!».
6. «Виділивши головну дію, ви вже вирішуєте простішу задачу!».
7. «Якщо закінчилася інформація в узагальнюючих тестах, то готуйте нові узагальнюючі тести для вирішення нових приватних завдань!».
8. «Якщо в процесі декомпозиції потрібно описати процес виходу з якоїсь точки опису в якусь іншу, це означає, що були неправильно виконані попередні деталізації через неправильне виявлення найбільш загальної дії і потрібно коректно переробити попередню роботу!».
9. «Іноді чергова деталізація не виходить через неусвідомлену потребу щодо введення допоміжної змінної (прапора події), що характеризує, чи відбулася раніше якась подія!».
Кожне СЛІДЖЕННЯ відповідає строго одній дії. Головне у діях – дієслово. Основне, що необхідно виконати при описі окремої структури СЛІД: в описі повинна міститися лише одна думка. Будь-яка структура виду СТЕЖЕННЯ може бути описана простою поширеною пропозицією природної (російської) мови. Наприклад, "Наступна дія полягає в завантаженні меблів в автомобіль". Однак враховуючи те, що у разі складання алгоритмів програм суть типових структур пояснюється самими операторами мови програмування, застосовується більш короткий опис у вигляді неповної речення без присудка. У разі підлягає утворюють від дієслів. Наприклад, "Навантаження меблів", "Рішення квадратного рівняння", "Введення даних".
Порядок деталізації одиночного СЛІДЖЕННЯ з використанням моделі «чорної скриньки» показано на рис. 5.2; 5.3:
1) попереднє виявлення суті дії;
2) виявлення вихідний інформації, бо вихід визначає вхід, а чи не навпаки;
3) виявлення вхідної інформації;



*Мал. 5.3.*Модель «чорної скриньки»

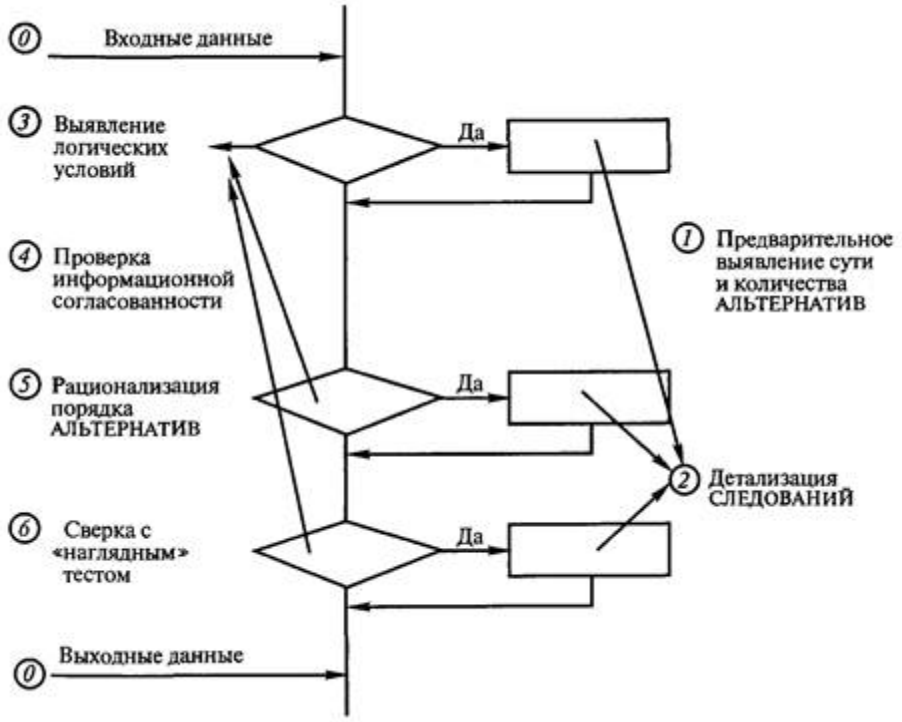
4) запис уточненого коментаря суті дії чи одного заключного елементарного оператора.
При описі фізичних та технічних систем необхідно використовувати повнішу модель «чорної скриньки» (див. рис. 2.2).
Ланцюжок слідів відповідає однозначному опису послідовності дій.
Ознакою ланцюжка Слідування є ніколи не порушується послідовність дій (послідовність Слідування). Ланцюжок слідів відповідає послідовність простих пропозицій і складносурядні пропозиції, які краще перетворити на прості пропозиції.
Порядок деталізації ланцюжка слідів показано на рис. 5.4:
1) попереднє виявлення суті дій кожного з СЛІД, визначення кількості СЛІД;
2) деталізація кожного зі СЛІДІВ як одиночного СЛІДЖЕННЯ;



*Мал. 5.4.*Порядок дій при деталізації структури Ланцюжок слідів

3) перевірка інформаційної узгодженості всіх СЛІД, а також вхідної та вихідної інформації всього ланцюжка СЛІД;
4) раціоналізація порядку СЛІД (зосередження СЛІД, що співпрацюють в інформаційному обміні);
5) звіряння з узагальнюючими тестами.
Окремі Слідування структури Ланцюжок Слідкувань надалі можуть бути декомпозовані Ланцюжком Слідкувань (більш приватних). Однак зустрічаються окремі структури СЛІДЖЕННЯ, які не можуть бути декомпозовані ланцюжком СЛІДЖЕНЬ. Такі СЛЕДЖЕННЯ можуть бути описані або структурою виду ланцюжка альтернатив (розгалуження), або структурою виду повторення (ЦИКЛ).
Ознакою ланцюжка альтернатив або вибору є одна або кілька дій, кожна з яких виконується за певної умови або не виконується взагалі (обов'язково кінцеве число різних дій за різних умов).
Ланцюжок Альтернатив, в окремому випадку, може складатися з однієї або декількох простих альтернатив з однією дією. Пропозиція найпростіша АЛЬТЕРНАТИВА з однією дією має таку конструкцію: «Якщо виконується якась умова, то виконується такий процес (СТЕЖЕННЯ)». (Інакше СЛІДКА пропускається.) Пропозиція Альтернативу з двох дій має наступну конструкцію: «Якщо виконується якась умова, то виконується СЛІД 1, в іншому випадку виконується СЛІД 2». В принципі структура Альтернативу з двох дій еквівалентна ланцюжку з двох найпростіших альтернатив з однією дією. При деталізації процесів, що включають більше двох альтернатив, може бути отримана єдина структура-ВИБІР у вигляді ланцюжка послідовно записаних структур АЛЬТЕРНАТИВУ з однією дією. Тут слід зазначити, що від зовні схожого ланцюжка слідів, кожне слідування якої є найпростішою альтернативою з однією дією, структура ВИБІР відрізняється такою властивістю, що при виконанні всього ланцюжка альтернатив може виконуватися лише одне з альтернативних слідств або жодного з.
*прикла*думов альтернатив у разі структури ВИБІР:
Якщо А < В, то виконати СЛІД 1;
Якщо А = В, то виконати СЛЕДЖЕННЯ2;

Якщо А > В, то виконати СЛІД.
Альтернатива з однією дією можна здійснити дострокове припинення процесу виконання алгоритму в тому випадку, який відповідає виявленню умов неможливості правильного подальшого виконання алгоритму або евроритму.
Деталізація всіх наступних структур передує нульову дію - запис на початку і в кінці вхідних і вихідних даних, виявлених в процесі деталізації попередніх їм СЛІД.
Порядок деталізації Ланцюжки Альтернатива показано на рис. 5.5:
1) попереднє виявлення суті дії кожного зі СЛІДІВ альтернативних дій, визначення кількості таких СЛІДЖЕНЬ;



*Мал. 5.5.*Порядок дій при деталізації ланцюжка АЛЬТЕРНАТИВ, згідно з проектною процедурою розробки функціональних описів

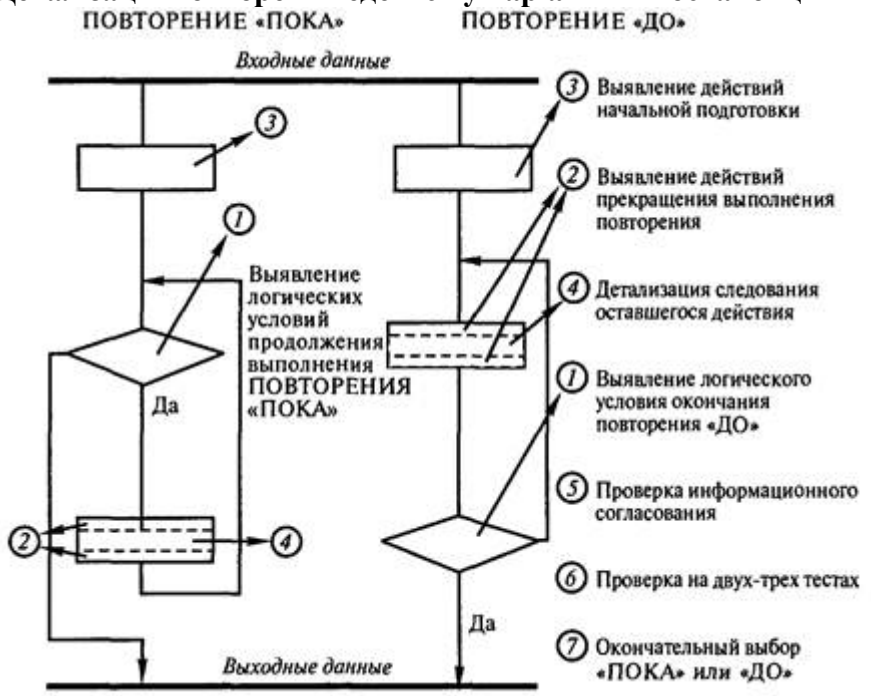
2) деталізація кожного зі СЛІДІВ як одиночного СЛІДЖЕННЯ;
3) виявлення та запис логічної умови виконання кожного з альтернативних СЛІДІВ;
4) перевірка інформаційної узгодженості всіх СЛЕДЖЕНЬ та логічних умов у ланцюжку, а також вхідної та вихідної інформації всього ланцюжка альтернатив;
5) раціоналізація порядку альтернатив;
6) перевірка виконання всіх маршрутів на тестах, отриманих із узагальнюючого тесту.
Ознакою ПОВТОРЕННЯ є багаторазово виконувана дія (але обов'язково кінцеве число разів). ПОВТОРЕННЯМ відповідають думки: «Ця дія має бути виконана п'ять разів»; «Ця дія виконується багаторазово до настання такої події». Ознаки ПОВТОРЕНЬ — змінна кількість АЛЬТЕРНАТИВ, будь-яка думка про повернення «назад», щоб повторити якісь дії. Часто головний більш загальний процес виду повторення ховається в контексті «і т. д.» або «і т. п.», «це зовсім просто», або навіть у крапках «...».
Пропозиція виду ПОВТОРЕННЯ може бути записана або у формі ПОВТОРЕННЯ «ДО» (ЦИКЛ «ДО»), або у формі ПОВТОРЕННЯ «ПОКИ» (ЦИКЛ «ПОКИ»).
Пропозиція ПОВТОРЕННЯ «ДО» має таку конструкцію: «До виконання якоїсь умови багаторазово виконувати СЛІД».
Пропозиція ПОВТОРЕННЯ «ПОКИ» має таку конструкцію: «Поки виконується якась умова, багаторазово виконувати СЛІД».

Різниця між пропозиціями повторення «ДО» та повторення «ПОКИ» полягає в тому, що, згідно з першою пропозицією, дія СТЕЖЕННЯ має бути виконана хоча б один раз, а згідно з другим, — СЛІДЖЕННЯ може не виконуватися жодного разу.

Структура НЕУНІВЕРСАЛЬНЕ ПОВТОРЕННЯ або просто забезпечує задану кількість повторень якогось процесу або виконання якогось процесу при значенні змінної циклу, значення якої змінюється за правилами арифметичної прогресії.

Порядок деталізації ПОВТОРЕНЬ показано на рис. 5.6.

Деталізація повторень ведеться у варіантній постановці



*Мал. 5.6.*Порядок дій при деталізації ПОВТОРЕНЬ

1) виявлення та запис логічної умови завершення ПОВТОРЕННЯ «ДО» або умови продовження виконання ПОВТОРЕННЯ «ПОКИ»;
2) виявлення процесів припинення виконання повторення;
3) виявлення дій СЛІДІ підготовки підготовки ПОВТОРЕННЯ;
4) деталізація СЛЕДЖЕННЯ дії, що залишилася, як одиночного СЛІДЖЕННЯ;
5) перевірка інформаційної узгодженості всіх СЛІД, логічних умов, а також вхідної та вихідної інформації всього ПОВТОРЕННЯ;
6) перевірка на 2-3 тестах, отриманих з узагальнюючого тесту;
7) остаточний вибір варіанта реалізації ПОВТОРЕННЯ у вигляді структури ПОВТОРЕННЯ «ПОКИ» або у вигляді структури ПОВТОРЕННЯ «ДО».

5.4. РЕКОМЕНДАЦІЇ ПОЧИНАЮЧИМ З СКЛАДАННЯ ОПИСІВ АЛГОРИТМІВ І ЕВРОРИТМІВ

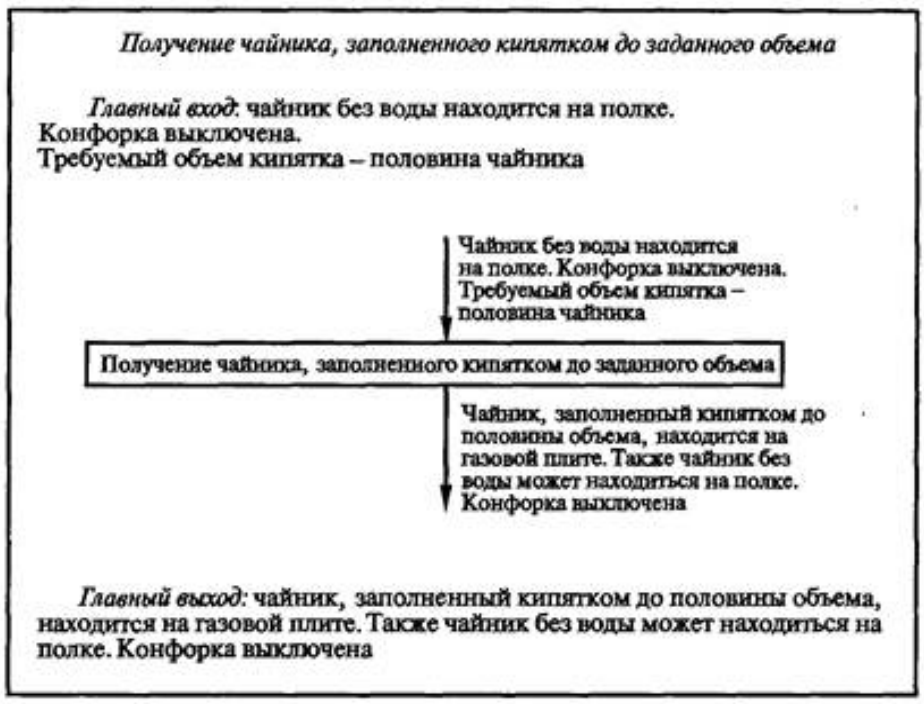
Перші спроби роботи з проектною процедурою вимагають величезної кількості аркушів паперу, але це необхідно, тому що дозволяє уважно розглянути окремий аркуш і безболісно переробити невдалі деталізації. Після досягнення деякого досвіду можна все більшу частину роботи виконувати без її запису на окремих аркушах. Проте перша робота виконується суворо окремих листах. Особливо це важливо у разі наявності досвідченого фахівця, який по аркушах документа зможе виявити дефекти навичок у учня, а також видати серію індивідуальних підказок, що навчається.

Виконання проектної процедури починається з підготовки та розгляду тестових прикладів для деталізації всього описуваного процесу у вигляді одного СЛІДЖЕННЯ.
На окремому аркуші складається безліч тестових прикладів, що охоплюють усі випадки обчислень чи дій з інструкції.
Паралельно з підготовкою тестів здійснюємо аналіз моделі «чорної скриньки». Беремо ще один чистий аркуш паперу, внизу аркуша записуємо терміни вихідних об'єктів та їх стану. Щодо алгоритмів визначаємо структуру даних: вводимо позначення змінних, послідовностей, векторів, матриць та визначаємо порядок розміщення в них інформації. Стосовно фізичних систем визначаємо розташування та стани об'єктів. Наприклад, чайник без води знаходиться на полиці. Результат дій - чайник, заповнений окропом до половини обсягу, знаходиться на плиті. Вихід визначає вхід, а чи не навпаки! Тому спочатку записуються результати дій і потім стану об'єктів до дій.
Далі у верхній частині листа записуються терміни вхідних об'єктів та стану вхідних об'єктів. У середині листа записується одна проста поширена пропозиція, що характеризує процес перетворення вхідної інформації у вихідну інформацію. Багаторазовим аналізом входу, виходу та суті пропозиції уточнюється вся інформація листа.
На основі сукупності тестів на окремому аркуші складаємо один або кілька узагальнюючих тестів. При цьому потрібно спроектувати раціональну форму ілюстрування: наприклад, малюнки проміжних станів даних процесу чи таблиці станів даних або фізичних об'єктів. Узагальнюючий тест необхідний розуміння суті функціонування процесу. На узагальнюючому тесті наводяться позначення всіх вхідних, вихідних та внутрішніх змінних чи станів фізичних об'єктів. З тесту мають бути видно всі процеси перетворення вхідної інформації на результат.
Навчальним, які перебувають у початковій стадії вивчення, рекомендується попередньо описати перебіг процесу за принципом: «Як можеш!» Якість опису має відповідати спробі пояснення процесу якійсь пересічній людині чи навіть дитині. Цей текст допоможе (з використанням ознак) відрізнити спільні дії від приватних.
Нарешті, приступаємо до складання семантичного скелета структурованого опису функціонування.
Під час навчання деталізації черговий структури слід здійснювати окремому аркуші. При деталізації ланцюжків слідів рекомендується орієнтувати лист вузькою стороною вгору, а при деталізації ланцюжків альтернатив і повторень лист краще орієнтувати широкою стороною вгору (це дозволить розміщувати на аркуші траси прорахунку тестів). Попередньо у верхній частині аркуша записується речення, яке було отримано раніше з деталізації кожного з цих процесів у вигляді одного СЛІДЖЕННЯ. Під ним записується вхідна інформація СЛІД, а внизу аркуша - вихідна інформація. Далі здійснюється сама деталізація, результати якої після перевірки на тестових прикладах та літературної обробки переносяться до чистовика опису алгоритму в цілому.
При деталізації Ланцюжки слідств рівномірно по вільній частині листа записуються пропозиції суті послідовно виконуваних дій (конкретний смисловий коментар до процесу). Далі здійснюється робота над кожним із них як над окремим СЛІДЖЕННЯМ. Далі здійснюється перевірка інформаційної узгодженості слідування та раціоналізується їх порядок, уточнюється суть СЛІД. При перевірці необхідно переконатися, що для наступних СЛІДІВ дані вже були визначені попередніми СЛІДЖЕННЯМИ.
При деталізації ланцюжка альтернатив рівномірно по вільній частині листа записуються (у кількості альтернативних дій) конструкції у вигляді потрібної кількості наступних послідовних речень: слово «Якщо», кілька чистих рядків для поля умови, слова «то виконується дія», далі залишається кілька чистих рядків для пропозиції СЛІД (конкретний смисловий коментар процесу).
Після деталізація всіх записаних СЛЕДЖЕНЬ як одного СЛІДЖЕННЯ здійснюється запис умов виконання альтернативних процесів. Далі здійснюється перевірка інформаційної узгодженості входу та виходу кожного зі СЛІД, їх умов виконання, а також вхідної та

вихідної інформації всього ланцюжка альтернатив. Ланцюжок лише з двох альтернатив може бути описаний пропозицією виду: «Якщо виконується умова (конкретна умова виконання дії по ТО), то виконується процес (конкретний змістовий коментар процесу), інакше виконується інший процес (конкретний змістовий коментар процесу).
При деталізації ПОВТОРЕНЬ на вільній частині листа записуються слова обох заготовок для ПОВТОРЕННЯ «ДО» та ПОВТОРЕННЯ «ПОКИ». Кожна заготівля починається з чистих рядків для майбутнього СЛІДУ визначення підготовки повторюваних дій. Далі для ПОВТОРЕННЯ «ДО»: записується рядок «До виконання умови закінчення процесу, що повторюється»; залишається кілька чистих рядків для запису умови закінчення дії, що повторюється; записується рядок «багаторазово виконується така дія:...». Для ПОВТОРЕННЯ «ПОКИ» записується рядок «Поки виконується умова»; залишається кілька чистих рядків для запису умови продовження виконання дії, що повторюється; записується рядок «багаторазово виконується така дія:...». Під цими рядками на обох заготовках залишаються чисті рядки для СЛІД, що визначає закінчення повторення процесу і СЛІД, що багаторазово виконується дії. Заповнення заготовок здійснюється в наступному порядку: спочатку умова закінчення (продовження для повторення «ПОКИ») виконання дії, що повторюється; потім записується СЛІД, що визначає закінчення повторення процесу; Потім — СЛІД визначення визначення повторюваних дій і в останню чергу — СЛІД багаторазово виконуваної дії. Далі здійснюється перевірка інформаційної узгодженості входу та виходу кожного зі СЛІД, умов виконання дії, а також вхідної та вихідної інформації всієї структури. Якщо на початок деталізації був ясно, який із варіантів повторень раціональніше деталізувати насамперед, то послідовно деталізуються обидві заготовки. Остаточний варіант відбирається шляхом їх порівняння за критеріями стислості та зрозумілості.
Після закінчення деталізації структурних окремих частин документа необхідно зробити його збирання. При складанні зазвичай потрібне літературне редагування документа як єдиного цілого. Після смислової та літературної обробки результат досліджень процесу переноситься до чистовика. Головний принцип — передай іншим хід своїх думок так, щоб вони змогли зрозуміти, але при цьому не слід перестаратися в деталях. Зайві коментарі можуть викликати роздратування читача та посилити нерозуміння.

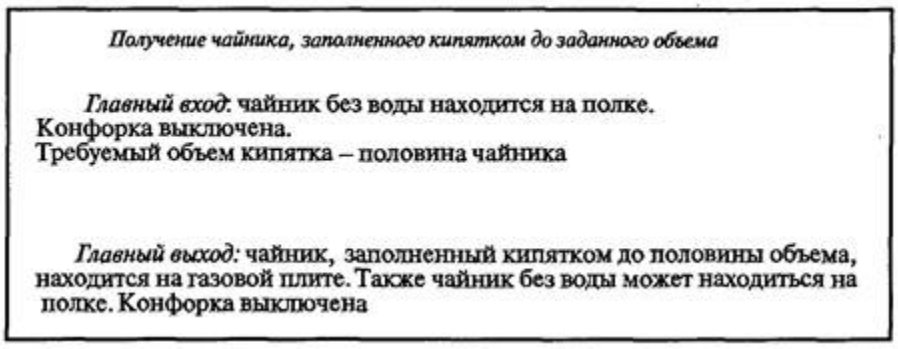
5.5. ПРИКЛАД РОЗРОБКИ ОПИСУ ПРОЦЕСУ «КИП'ЯЧЕННЯ ВОДИ У ЧАЙНИКУ»

Нижче показано покрокове виконання проектної процедури на прикладі розробки опису процесу кипіння води в чайнику. Доповніть цей опис наочними малюнками на аркуші 1 самостійно.
Аркуш 2.Аналіз процесу як одного СЛІДЖЕННЯ.
Первинний опис суті дії: "Кип'ятіння води в чайнику".
*Вихід:*Чайник, наповнений окропом до половини обсягу, знаходиться на газовій плиті. Конфорку вимкнено.
*Вхід:*Чайник без води знаходиться на полиці. Конфорку вимкнено. Необхідний об'єм окропу – половина чайника.
Остаточний опис суті дії: "Отримання чайника, заповненого окропом до заданого обсягу".
Працюємо із тестами. З тестів з'ясовуємо, що одержати окріп неможливо без води, сірників та газу. Сірники можуть закінчитися у процесі запалення газу. Може закінчитись вода в процесі заповнення чайника. Також може закінчитись газ у магістралі. Приймаємо, що виявлення даних фактів буде здійснено у процесі виконання інструкції, тому інформацію про наявність сірників, води та газу виключаємо зі складу вхідної інформації.
Нами отримано СЛІД (рис. 5.7).
Аркуш 3- Містить зображення процесів інструкції в наочній формі. Розробте його самостійно.
Аркуш 4- може містити опис процесу «Отримання чайника, заповненого окропом до заданого об'єму», виконаний у будь-який доступний спосіб.

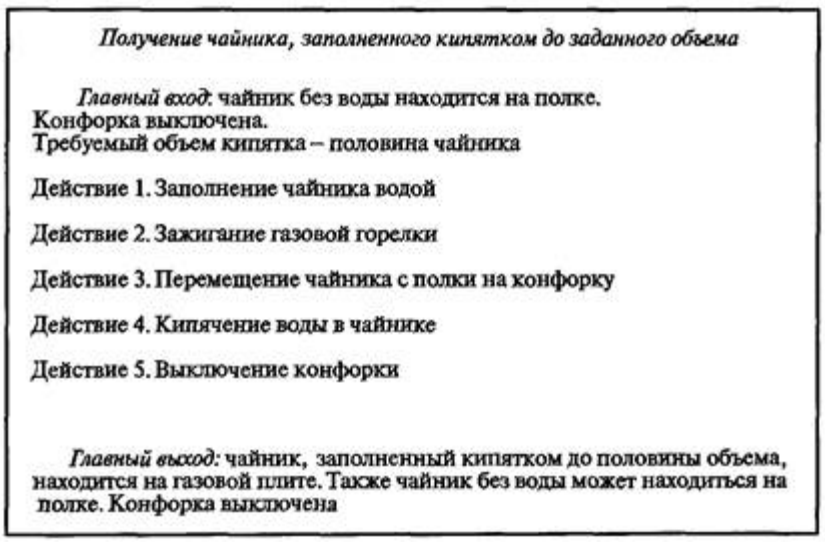


*Мал. 5.*7. Деталізація із застосуванням графічного зображення «чорної скриньки»

Аркуш 5.Декомпозиція процесу "Отримання чайника, заповненого окропом до заданого обсягу".
Спочатку на лист переносимо інформацію попередньої структури СЛІД, одержуємо макет листа, представлений на рис. 5.8.
Далі, виходячи з міркувань, що для цього процесу необхідно виконати ряд послідовних дій, отримуємо макет аркуша з ланцюжком стежень, представлений на рис. 5.9.
Після деталізації кожного з прямування, перевірки інформаційної узгодженості та уточнення суті прямування в ланцюжку отримуємо макет аркуша, зображений на рис. 5.10.
Перевірка інформаційного узгодження виявила, що дія 2 може бути раніше дії 1 з міркувань економії газу. Чайник не можна ставити на конфорку до запалювання газу через небезпеку закопчення денця кіптявою сірника. Аналогічно перевірялася послідовність інших процесів.
Дія 1 може бути декомпозірована ще однією ланцюжком СЛІД, зображеної на рис. 5.11.

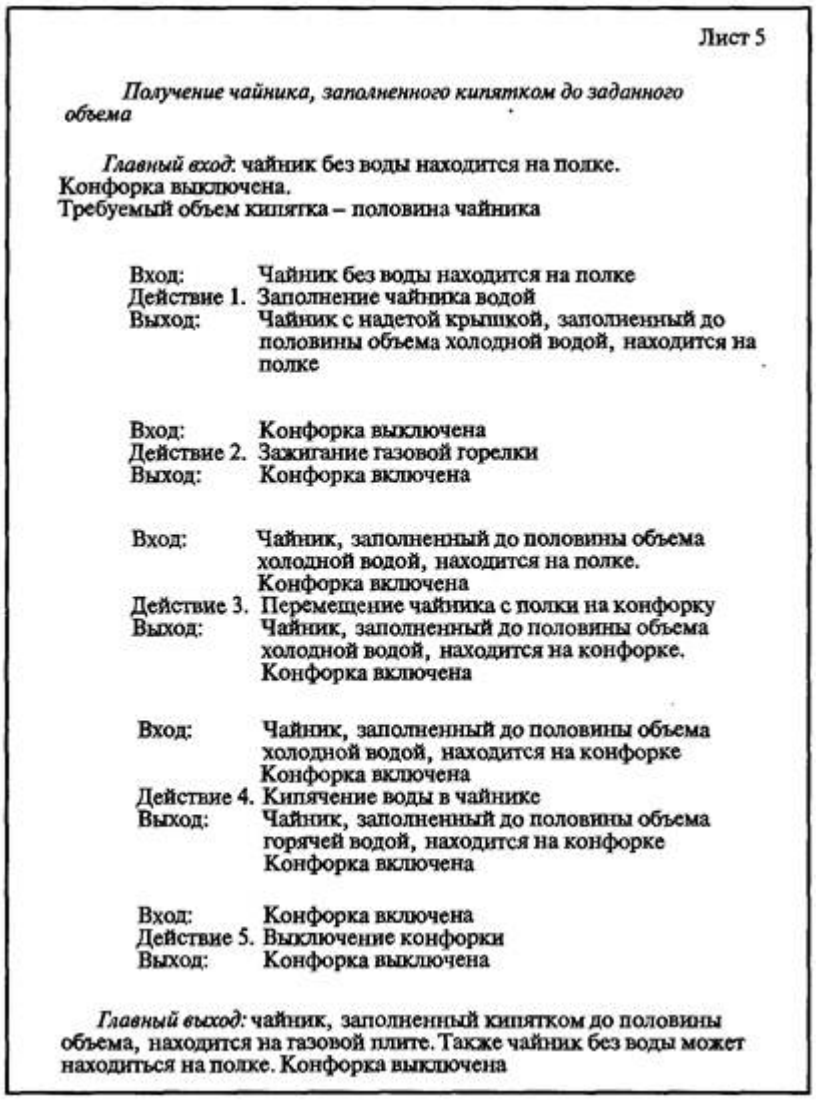


*Мал. 5.8.*Початковий вид аркуша 5



*Мал. 5.9.*Вид аркуша 5 після попереднього виявлення суті дій

Дія 1 «Заповнення чайника водою» може бути декомпозіроване ще однією ланцюжком слідок, зображеної на рис. 5.11.
Дія 4 "Наповнення чайника водою", показане на рис. 5.11 декомпозується ПОВТОРЕННЯМ, представленим на рис. 5.12.



Лист 5

Получение чайника, заполненного кипятком до заданного объема

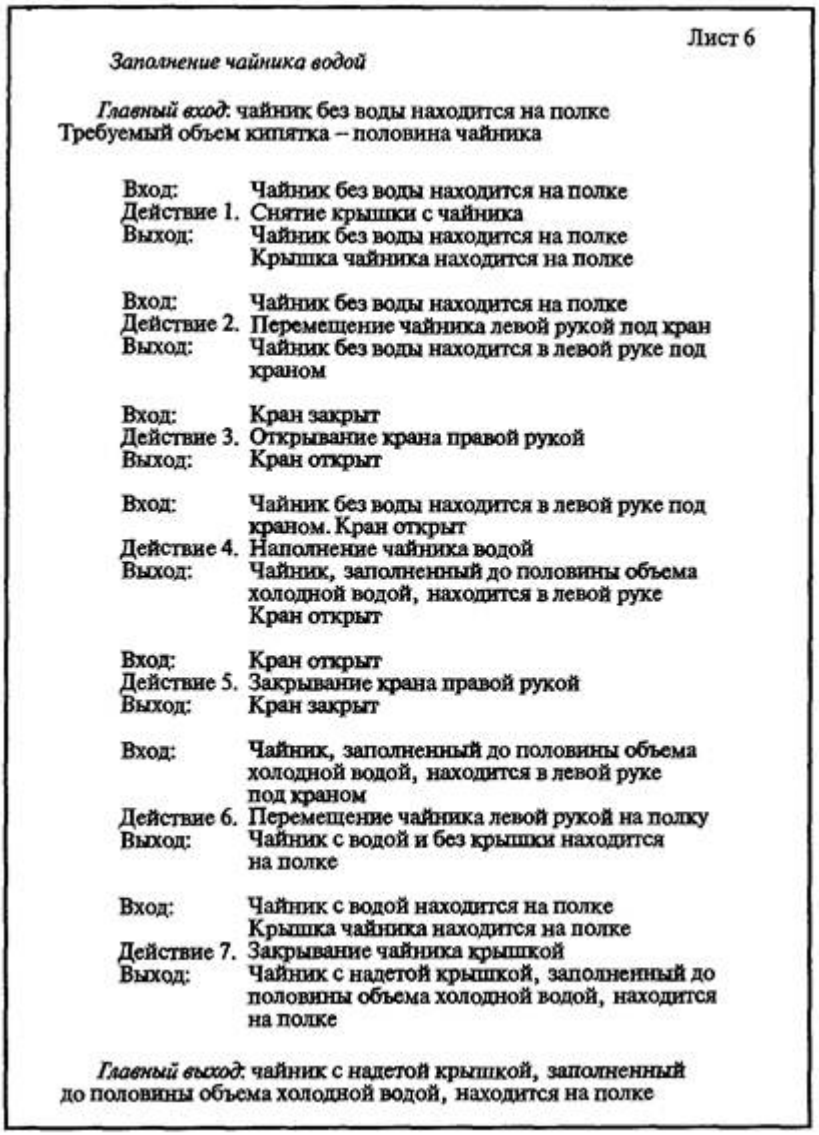
Главный вход: чайник без воды находится на полке. Конфорка выключена. Требуемый объем кипятка – половина чайника

Вход:	Чайник без воды находится на полке
Действие 1.	Заполнение чайника водой
Выход:	Чайник с надетой крышкой, заполненный до половины объема холодной водой, находится на полке
Вход:	Конфорка выключена
Действие 2.	Зажигание газовой горелки
Выход:	Конфорка включена
Вход:	Чайник, заполненный до половины объема холодной водой, находится на полке. Конфорка включена
Действие 3.	Перемещение чайника с полки на конфорку
Выход:	Чайник, заполненный до половины объема холодной водой, находится на конфорке. Конфорка включена
Вход:	Чайник, заполненный до половины объема холодной водой, находится на конфорке Конфорка включена
Действие 4.	Кипячение воды в чайнике
Выход:	Чайник, заполненный до половины объема горячей водой, находится на конфорке Конфорка включена
Вход:	Конфорка включена
Действие 5.	Выключение конфорки
Выход:	Конфорка выключена

Главный выход: чайник, заполненный кипятком до половины объема, находится на газовой плите. Также чайник без воды может находиться на полке. Конфорка выключена

*Мал. 5.10.*Остаточний вигляд листа

Декомпозиція ланцюжком СЛІД дії «Аналіз процесу заповнення чайника на форс-мажорні обставини» представлена на рис. 5.13.



Лист 6

Заполнение чайника водой

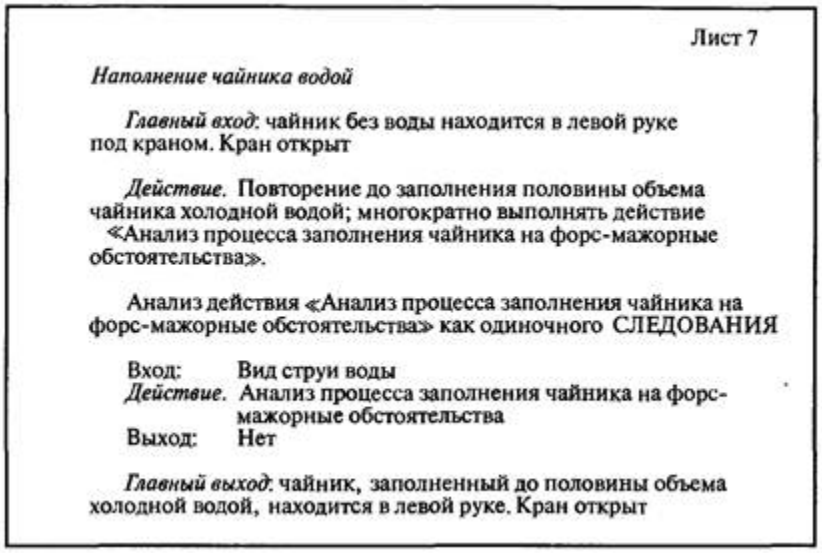
Главный вход: чайник без воды находится на полке
Требуемый объем кипятка – половина чайника

Вход:	Чайник без воды находится на полке
Действие 1.	Снятие крышки с чайника
Выход:	Чайник без воды находится на полке Крышка чайника находится на полке
Вход:	Чайник без воды находится на полке
Действие 2.	Перемещение чайника левой рукой под кран
Выход:	Чайник без воды находится в левой руке под краном
Вход:	Кран закрыт
Действие 3.	Открывание крана правой рукой
Выход:	Кран открыт
Вход:	Чайник без воды находится в левой руке под краном. Кран открыт
Действие 4.	Наполнение чайника водой
Выход:	Чайник, заполненный до половины объема холодной водой, находится в левой руке Кран открыт
Вход:	Кран открыт
Действие 5.	Закрывание крана правой рукой
Выход:	Кран закрыт
Вход:	Чайник, заполненный до половины объема холодной водой, находится в левой руке под краном
Действие 6.	Перемещение чайника левой рукой на полку
Выход:	Чайник с водой и без крышки находится на полке
Вход:	Чайник с водой находится на полке Крышка чайника находится на полке
Действие 7.	Закрывание чайника крышкой
Выход:	Чайник с надетой крышкой, заполненный до половины объема холодной водой, находится на полке

Главный выход: чайник с надетой крышкой, заполненный до половины объема холодной водой, находится на полке

*Мал. 5.11.*Деталізація СЛІД «Заповнення чайника водою», виявленого на рис. 5.10

Дія «Форс-мажорне завершення інструкції» представляє ланцюжок СЛІД, показаному на рис. 5.14. Зазначимо, що вихідна інформація аварійного завершення інструкції визначається вхідною інформацією всієї інструкції (див. рис. 5.7).



Лист 7

Наполнение чайника водой

Главный вход: чайник без воды находится в левой руке под краном. Кран открыт

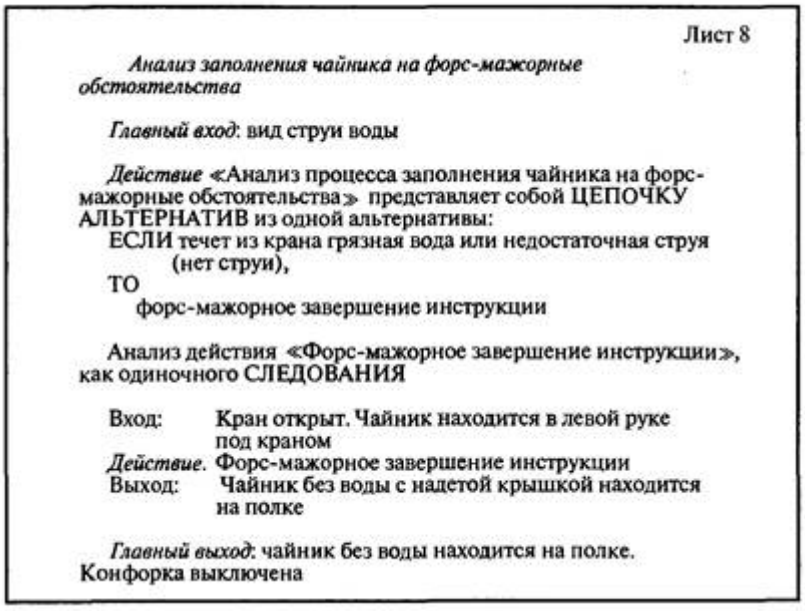
Действие. Повторение до заполнения половины объема чайника холодной водой; многократно выполнять действие «Анализ процесса заполнения чайника на форс-мажорные обстоятельства».

Анализ действия «Анализ процесса заполнения чайника на форс-мажорные обстоятельства» как одиночного СЛЕДОВАНИЯ

Вход: Вид струи воды
Действие. Анализ процесса заполнения чайника на форс-мажорные обстоятельства
Выход: Нет

Главный выход: чайник, заполненный до половины объема холодной водой, находится в левой руке. Кран открыт

*Мал. 5.12.*Декомпозія повторенням дії 4 «Наповнення чайника водою» (див. рис. 5.11)



Лист 8

Анализ заполнения чайника на форс-мажорные обстоятельства

Главный вход: вид струи воды

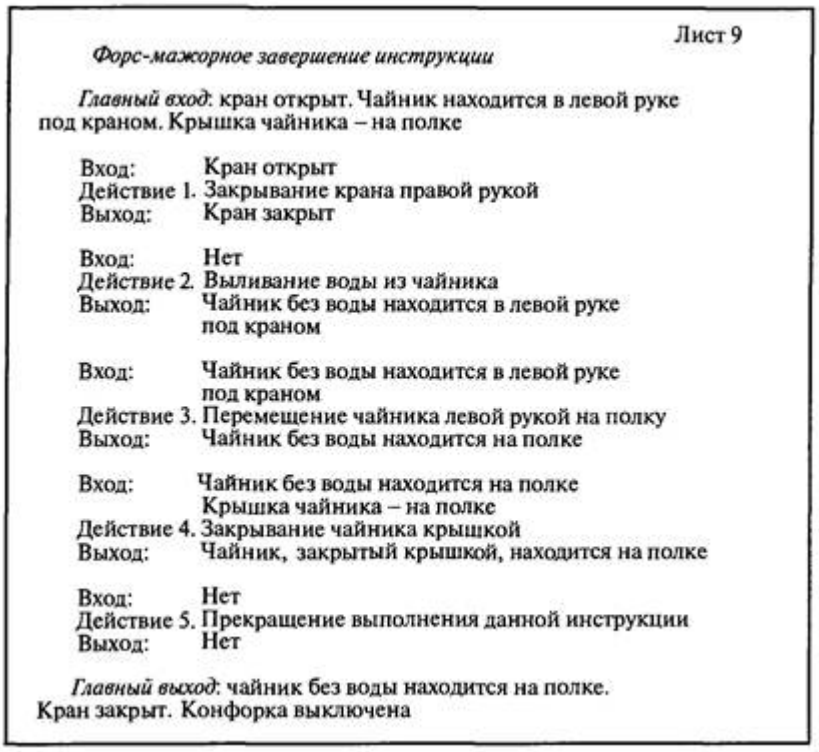
Действие «Анализ процесса заполнения чайника на форс-мажорные обстоятельства» представляет собой ЦЕПОЧКУ АЛЬТЕРНАТИВ из одной альтернативы:
ЕСЛИ течет из крана грязная вода или недостаточная струя (нет струи),
ТО
форс-мажорное завершение инструкции

Анализ действия «Форс-мажорное завершение инструкции», как одиночного СЛЕДОВАНИЯ

Вход: Кран открыт. Чайник находится в левой руке под краном
Действие. Форс-мажорное завершение инструкции
Выход: Чайник без воды с надетой крышкой находится на полке

Главный выход: чайник без воды находится на полке. Конфорка выключена

*Мал. 5.13.*Декомпозиція ланцюжкового слідства дії «Аналізу процесу заповнення чайника на форс-мажорні обставини» (див. рис. 5.12)



Лист 9

Форс-мажорное завершение инструкции

Главный вход: кран открыт. Чайник находится в левой руке под краном. Крышка чайника – на полке

Вход: Кран открыт
Действие 1. Закрывание крана правой рукой
Выход: Кран закрыт

Вход: Нет
Действие 2. Выливание воды из чайника
Выход: Чайник без воды находится в левой руке под краном

Вход: Чайник без воды находится в левой руке под краном
Действие 3. Перемещение чайника левой рукой на полку
Выход: Чайник без воды находится на полке

Вход: Чайник без воды находится на полке
Крышка чайника – на полке
Действие 4. Закрывание чайника крышкой
Выход: Чайник, закрытый крышкой, находится на полке

Вход: Нет
Действие 5. Прекращение выполнения данной инструкции
Выход: Нет

Главный выход: чайник без воды находится на полке. Кран закрыт. Конфорка выключена

*Мал. 5.14.*Декомпозиція дії «Форс-мажорне завершення інструкції» представляє ланцюжок СЛІД.

Дія 2 (див. рис. 5.10) «запалювання газового пальника» декомпозується циклом: доки не запалилася конфорка або не виявлено форс-мажорні обставини, багаторазово виконувати дію «Запалювання конфорки». Подальший розвиток алгоритму дозволив виявити структури, викладені далі.
Дія (СТЕЖЕННЯ) «Запалювання конфорки» є ланцюжком СЛІД: «Запалювання сірника», «Включення газу», «Підпалювання газу», «Відключення газу при невдачі», «гасіння сірника», «викидання сірника». Форс-мажорними обставинами є відсутність газу або закінчення сірників. Їм відповідає логічна змінна.
Дія (СТЕЖЕННЯ) «Відключення газу при невдачі» є альтернативою: якщо газ не запалився, а сірник прогорів, то необхідно закрити газ.
Тут виявлено попередню недоробку з вихідними даними дії 2. Адже після закінчення дії 2 конфорка може бути як включена, так і не включена. Виправити ситуацію можна двома способами. За першим способом за дією 2 можна вставити найпростішу альтернативу з однією дією: якщо виявлено форс-мажорні обставини, аварійно завершити інструкцію. Відповідно до другого способу, дії 3, 4, 5 можуть бути представлені одним СЛІДОМ, всередині якого повинна знаходитися Ланцюжок Альтернатив послідовного виконання кожного з колишніх дій 3, 4, 5 за умови відсутності виявлення форс-мажорних обставин.

5.6. ПРИКЛАД ОПИСУ ПРОГРАМИ «РЕДАКТОР ТЕКСТІВ»

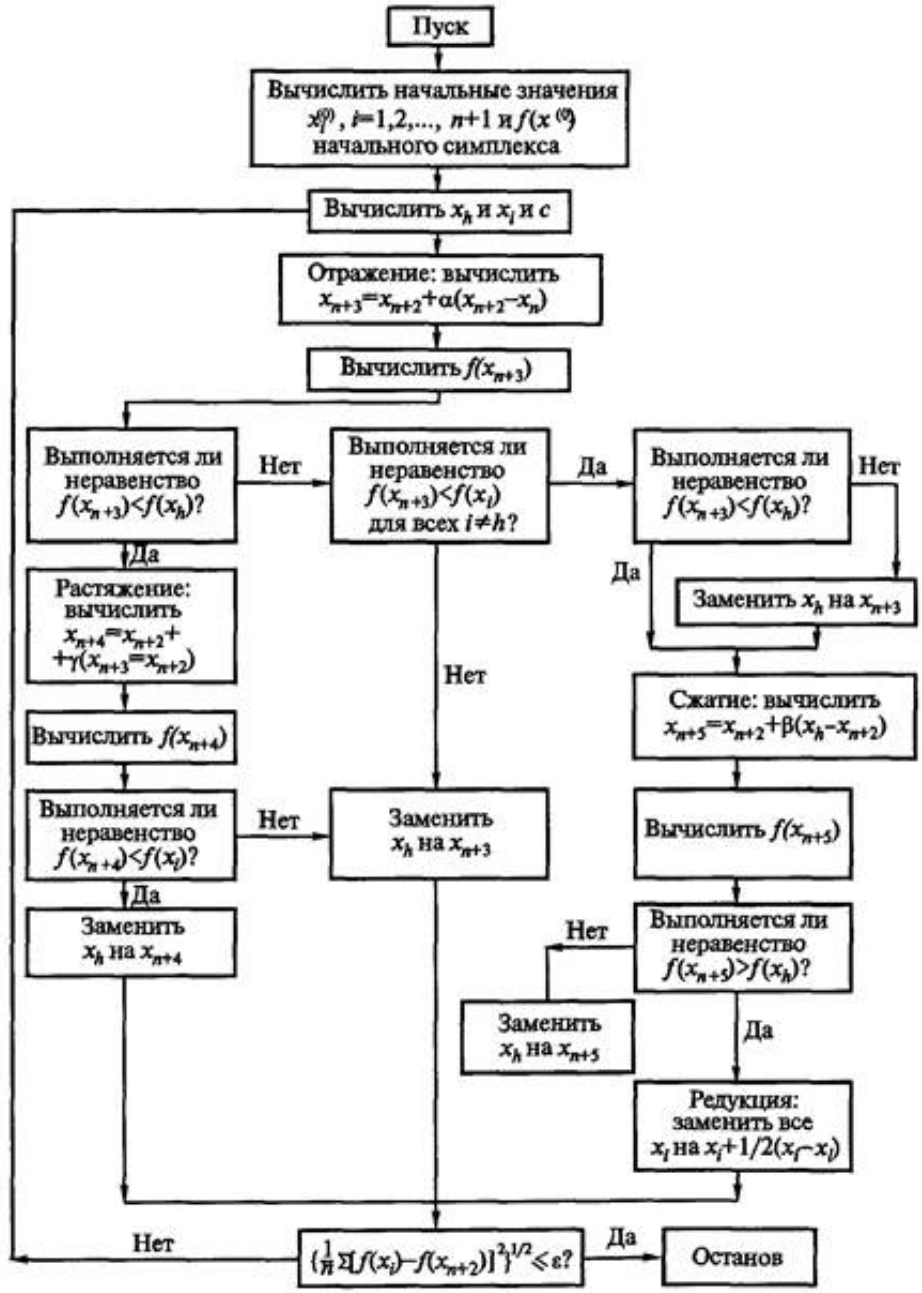
Нижче наведено приклад опису програми «Редактор текстів», складений одним із учнів. У прикладі наводиться спочатку зовнішня функціональна специфікація, потім внутрішня специфікація.
Програма «Редактор текстів» призначена для створення нових та коригування існуючих текстових файлів MS DOS у діалоговому (користувач-ЕОМ) режимі роботи. ЕОМ формує екран із вікном, у якому відображено ділянку тексту з текстового файлу (макет екрану відповідає внутрішньому редактору програми Norton Commander). Користувачеві

забезпечується можливість вставки в текст у вікні екрана будь-якого символу клавіатури за символом, позначеним на екрані курсором. Виняток становить ряд символів, які є ознаками команд керування або незадіяними символами (наведено список символів). Після подачі користувачем команди запису, всі зміни тексту, здійснені користувачем, записуються у файл.
Основний принцип роботи редактора текстів полягає у перенесенні рядків тексту з необхідних ділянок файлу спочатку в буферний масив пам'яті довжиною 65535 байт (символів) з подальшим копіюванням необхідних рядків з буферного масиву у вікно екрану.
Запуск програми здійснюється командою із зазначенням імені файлу, що редагується. Далі, доки не буде вказано коректне ім'я файлу, може почати багаторазово виконуватися алгоритм «Запит користувача на введення або коригування імені файлу».
Потім задаються початкові значення структурованої змінної "Система координат", в якій є поля: "Положення курсору щодо файлу"; «Положення курсору щодо буферного вікна редактора»; "Положення буферного вікна редактора щодо файлу".
Після цього здійснюється очищення буферного масиву редактора рядкових змінних з 5 * 23 = 115 рядків довжиною по 225 символів.
Далі при параметрі "Перший рядок файлу" виконується алгоритм "Завантаження рядків файлу, починаючи із зазначеного рядка в буферний масив редактора". Потім до подачі користувачем однієї з команд завершення редагування із збереженням інформації (або збереження) виконується головний цикл програми. Нарешті, якщо було дано команда завершення зі збереженням інформації, то інформація з буферного масиву переписується файл. Виконання програми завершується очищенням екрана.
Контроль імені файлу, що редагується, полягає в наступному. Якщо файл із вказаним ім'ям відсутній на диску, виводиться попереджувальне повідомлення про створення нового "порожнього" файлу. Якщо користувач не вказав ім'я файлу, що редагується або відмовився працювати зі створеним «порожнім» файлом, то відбувається аварійне завершення програми з поясненням причини завершення.
Усередині головного циклу програми виконується ряд із трьох послідовних дій. "Алгоритм відображення" відображає на екрані 23 рядки тексту з буферного масиву, починаючи із заданого рядка. Далі встановлюється курсор дисплея на позицію екрана. Здійснюється введення коду натиснутої кнопки. Якщо код натиснутої клавіші відповідає керуючій клавіші, то виконується одна з альтернативних дій щодо виконання команди, яка відповідає цій клавіші. В іншому випадку вставляється символ у текст.

5.7. РЕФАКТОРИНГ АЛГОРИТМІВ І ЕВРОРИТМІВ

Алгоритм Нелдера-Міда є широко відомим і застосовується як алгоритм прямого пошуку локального екстремуму речових функцій від 2 до 6 речових змінних.
Наступний абзац містить фрагмент тексту з книги Д. Хіммельблау [26], в якому міститься частина опису алгоритму Нелдера - Міда (методу багатогранника, що деформується).
У методі Нелдера і Міда мінімізується функція n незалежних змінних з використанням n + 1 вершин багатогранника, що деформується, в En. Кожна вершина може бути ідентифікована x вектором. Вершина (точка) в En, в якій значення f(x) максимально, проектується через центр ваги (центроїд) вершин, що залишилися. Покращені (нижчі) значення цільової функції знаходяться послідовною заміною точки з максимальним значенням f(x) більш «хороші» точки, доки знайдено мінімум f(x).
Початковий багатогранник зазвичай вибирається у вигляді регулярного симплексу (але це не обов'язково) з точкою на початку координат. Процедура відшукання вершини в En, у якій f(x) має найкраще значення, складається з наступних операцій:
Відображення — проектування x(k)h через центр тяжіння відповідно до співвідношення x(k)n+3 = x(k)n+2 + α(x(k)n+2 — x(k)h), де α > 0 є коефіцієнтом відбиття; x(k)n+2 — центр тяжіння, x(k)h — вершина, у якій функція f(x) приймає найбільше з n + 1 її значень на k етапі.

Розтягування виконується, якщо f(x(k)n+3) ≤ f(x(k)i), то вектор (x(k)n+3 - x(k)n+2) розтягується відповідно до співвідношення x(k)n+4 = x(k)n+2 + γ(x(k)n+3 - x(k)n+2), де γ - коефіцієнт розтягування. Якщо f(x(k)n+4) < f(x(k)i), то x(k)h замінюється на x(k)n+4 і процедура продовжується з кроку 1 (k = k + 1), інакше x(k)n замінюється на x(k)n+3 процедура триває з кроку 1 (k = k + 1).
Стиснення — якщо f(x(k)n+3) > f(x(k)i) для всіх i ≠ h, то вектор (x(k)h — x(k)n+2) стискається відповідно до формули x(k)n+5 = x(k)n+2 + β(x(k)h — x(k)n+2), де 0 < β < 1 являє собою коефіцієнт стиснення. Потім x(k)h замінюється на x(k)n+5 і перетворюється на крок 1 (k = k + 1).
Редукція — якщо f(x(k)n+3) > f(x(k)h), всі вектори (x(k)i — x(k)1), i = 1...n + 1 зменшуються 2 рази з відліком від x(k)1 відповідно до формули x(k)i = x(k)i + 0,5(x(k)i — x(k)1), i = 1...n + 1. Потім повертаємось до кроку 1 для продовження пошуку на k + 1 кроці.
Критерій закінчення пошуку, використаний Нелдером і Мідом, був перевіркою умови середнього квадратичного відхилення функцій f(x(k)i) від довільного малого числа ε.
Хіммельблау скористався традиційним математичним стилем викладу опису алгоритму. Алгоритм має структуру виду "спагетті", що видно зі схеми алгоритму, розробленої автором (рис. 5.15). Схема як «спагетті» — це приховані go to, які ми виявляємо у тексті програми, розробленої автором. Він же наводить графічні малюнки принципу розрахунку точок та графічний малюнок послідовності просування кращих точок симплексу по кроках методу при вирішенні тестового завдання. Витративши 11 сторінок тексту, навівши текст неструктурованої програми на 12 сторінках, автор книги [26] не зумів зрозуміло описати свій алгоритм.



*Мал. 5.15.*Схема алгоритму, розроблена Д. Хіммельблау

Порівняйте спосіб подання опису алгоритму, складений автором [22], та функціональний опис алгоритму після рефакторингу. Функціональний опис алгоритму наведено нижче. Особливістю алгоритму є просування до екстремуму цілою хмарою пробних точок, яка умовно названа симплексом (деформованим багатогранником). Загальна кількість точок у симплексі дорівнює збільшеному на одиницю числу змінних пошуку. Просування до екстремуму не по лінії від однієї точки до іншої, а всередині якогось безлічі пробних точок забезпечило нереагування на дрібні дефекти цільової функції та проходження щодо «широких» ярів.

Інформація, що вводиться:

n - число змінних функції, що мінімізується;

X0 = (x01, x02, ..., x0n) - точка першого початкового наближення;

h - крок регуляризації симплексу;

εx - похибка знаходження екстремуму за параметрами.

Робота алгоритму починається з початкової підготовки симплексу точок. Після виконання процедур регуляризація симплексу та розрахунку значень цільової функції у всіх точках симплексу має вигляд

$$\begin{array}{c} 1 \\ 2 \\ 3 \\ \dots \\ n+1 \end{array} \begin{pmatrix} x_1^0 & x_2^0 & x_3^0 & \dots & x_n^0 \\ x_1^0+h & x_2^0 & x_3^0 & \dots & x_n^0 \\ x_1^0 & x_2^0+h & x_3^0 & \dots & x_n^0 \\ \dots & \dots & \dots & \dots & \dots \\ x_1^0 & x_2^0 & x_3^0 & \dots & x_n^0+h \end{pmatrix} \begin{pmatrix} Q_1 \\ Q_2 \\ Q_3 \\ \dots \\ Q_{n+1} \end{pmatrix}$$

Процедура регуляризації симплексу полягає в копіюванні у всі поля значень аргументів симплексу значення вектора X0, і далі проводяться зміни значень діагональних компонентів векторів з номерами від 2 до n + 1 шляхом їх збільшення на крок регуляризації симплексу h.

У симплексі Q1, Q2, Q3, ..., Qn+1 — це розраховані значення функції, що мінімізується, при відповідних за рядком значеннях аргументів мінімізованої функції.
Далі до виконання умови закінчення пошуку здійснюються ітерації (кроки) пошуку. Умовою закінчення пошуку цього є невіддаленість від кращої точки симплекса інших точок симплекса більш як на e2x.
Кожна ітерація починається з знаходження номерів l і k відповідно кращої і гіршої за значенням функції точок симплекса. Далі здійснюється розрахунок точки Xц - положення центру мас усіх точок симплексу за винятком найгіршої точки 1. Це пізнє поліпшення алгоритму. Д. Хіммельблау не виключав найгіршої точки.

$x_{iц} = \frac{i}{n}\sum_{j=1}^{n+1}(x_{ji}),$ де n - кількість точок у симплексі; i – номер компоненти вектора X(i = 1, 2, ..., n); j – номер точки в симплексі.
Вважаються нульовими значення xkj доданків сум, де k — номер найгіршої точки.
Працюючи методу кожної з ітерації може обчислюватися одне з спеціальних пробних точок: a — точка відбиття, p — точка розтягування, y — точка стискування. Крапки обчислюються за формулами:
Xα = Xц + α(Xц - Xk), Xβ = Xц + β(Xц - Xk), Xγ = Xц + γ(Xц - Xk).
Доцільно ці схожі формули реалізувати однією функцією.
Значення коефіцієнтів α = 1, β = 0,5 та γ = 2 підібрані експериментальним шляхом (в оригіналі опису алгоритму ці значення можна виявити лише з тексту програм). Розрахунок точок α, β, γ доцільно здійснювати однією процедурою з параметром у вигляді значень коефіцієнтів α, β, γ.
Після розрахунку точки Xц обчислюється пробна точка. Далі виконується одна з альтернативних дій.
Якщо Qα ≤ Q1, виконуються дії досягнення сильного успіху. Якщо Q1 ≤ Qα < Qk, є слабкий успіх, а точка α записується на місце k-точки. Якщо Qα ≥ Qk, виконуються дії відсутності успіху. Розраховується Xβ та Qβ. Далі, якщо Qβ ≤ Qk, то точка β записується на місце k-точки, інакше, якщо точка β гірша за точку k, виконується процедура редукції симплексу та процедура розрахунку значення функції в точках симплексу.
Дії досягнення сильного успіху. Розраховується Xγ і Qγ. Найкраща точок α або β записується на місце найгіршої k-точки симплексу.
Події відсутності успіху. Розраховується Xβ та Qβ. Далі виконується дія зі зміни симплексу за відсутності успіху.
Дія зі зміни симплексу за відсутності успіху є альтернативою: якщо Qβ ≤ Qk, то точка β записується на місце k-точки, інакше, якщо точка β гірша за точку k, виконується процедура редукції симплексу.

При виконанні процедури редукції симплексу всі точки симплексу стягуються до кращої точки симплексу на половину свого попереднього видалення і далі виконується процедура розрахунку значень цільової функції у всіх точках симплексу.

5.8. Кодування типових структур на мовах програмування

Зазвичай розробку алгоритмів програм поєднують із кодуванням тексту програми. Окреме від програмування написання алгоритмів практично не відрізняється від написання інструкцій.

Кодування програми має здійснюватись лише з використанням стандартних структур!

Заборонено використання міток, операторів безумовного переходу на мітку (go to), операторів дострокового виходу із структури break!

При кодуванні мовою С оператор break може використовуватися лише при кодуванні структури switch.

При використанні іншої процедурно-орієнтованої мови програмування (не Pascal) необхідно попередньо закодувати використовуваною мовою програмування всі описані в цьому підрозділі стандартні структури без зміни їх логіки!

Так, при програмуванні мовою С структура УНІВЕРСАЛЬНИЙ ЦИКЛ — «ДО» включатиме операцію «!» (НЕ):

```
/* підготовка циклу */
do
{
/* Тіло циклу */
...
}
while (! (L));
```

У наведеній вище структурі ненульове значення змінної L відповідає закінченню виконання циклу, а чи не його продовженню виконання, як у оператора мови програмування!

Використання «лінією» операції (!) не подовжить програму. Сучасні компілятори автоматично інвертують логічну умову завершення циклу.

Структурі СЛІД В програмах можуть відповідати: виконання всієї програми; виклик процедури.

Згідно зі стандартом проекту, АЛЬТЕРНАТИВА має чотири конструкції. Розглянемо їх запис мовою програмування Pascal.

Конструкція для однієї альтернативи:

```
if L then begin
{Дія при L=True}
...
end;
```

Конструкція для двох альтернатив:

```
if L then begin
{Дія при L=True}
...
end
else
begin
{Дія при L=False}
...
End;
```

Перший варіант конструкції для кількох альтернатив (ВИБОРУ):

```
if L1 then Begin
{Дія при L1=True}
end;
...
if L2 then
```

```
begin
{Дія при L2=True}
...
end;
if L3 then
begin
{Дія при L3=True}
...
end;
```

Другий варіант конструкції для кількох альтернатив (ВИБОРУ):

```
Switch: = 0;
L1: = ...;
L2: = ...;
L3: = ...;
...
if L1 then Switch := 1;
if L2 then Switch := 2;
if L3 then Switch := 3;
...
case Switch of
1:begin
{Дія при L1=True}
...
end;
2:begin
{Дія при L2=True}
...
end;
3:begin
{Дія при L3=True}
...
end;
else
begin
{Виведення повідомлення про помилкове кодування модуля}
...
end;
end; {End of Case}
```

Розглянемо запис варіантів кодування структури АЛЬТЕРНАТИВУ мовою програмування З.

Конструкція для однієї альтернативи:

```
if (L)
{
/*Дія при L ≠ 0*/
...
}
```

Конструкція для двох альтернатив:

```
if (L)
{
/*Дія при L ≠ 0*/
...
}
else
{
/*Дія при L = 0*/
...
}
```

Перший варіант конструкції для кількох альтернатив (ВИБОРУ)

```
if (L1)
{
/*Дія при L1 ≠ 0*/
...
}
else if (L2)
{
/*Дія при L2 ≠ 0*/
...
}
else if(L3)
{
/*Дія при L3 ≠ 0*/
...
}
...
}
```

Другий варіант конструкції для кількох альтернатив (ВИБОРУ):

```
Selector = 0;
L1 = ...;
L2 = ...;
L3 = ...;
...
if (L1) Selector = 1;
else if (L2) Selector = 2;
else if (L3) Selector = 3;
...
switch (Selector)
case 1:
/*Дія при L1≠ 0*/
...
break;
case 2:
/*Дія при L2≠ 0*/
...
break;
case 3:
/*Дія при L3 ≠ 0*/
...
break;
default:
/*Виведення повідомлення про помилкове кодування модуля*/
exit (-1);
} /*Кінець switch*/
```

Права конструкція відповідає дуже складній логіці умов. У найпростіших випадках допускається спрощене кодування (перший приклад Pascal, другий Q:

```
if a > b then x:=y+3 else x:=y+6; {Мова Pascal}
if (a > b) x=y+3; else x = y +6; /*Мова C*/
```

ВИБІР із двох і більше АЛЬТЕРНАТИВ не можна кодувати за допомогою вкладення інших структур найпростіших АЛЬТЕРНАТИВ через велику ймовірність помилок.
Порядок деталізації структур АЛЬТЕРНАТИВУ:
1) залежно кількості альтернативних дій записуються всі оператори структури;
2) визначаються самі альтернативні дії як СЛІДКИ;
3) записуються логічні умови альтернативних действий;
4) перевіряється інформаційна узгодженість логічних умов та дій;
5) на кількох текстових прикладах здійснюється перевірка. ПОВТОРЕННЯ в програмуванні називаються циклами.

Зазвичай стандартом проекту передбачено низку конструкцій циклів. Неуніверсальний ЦИКЛ-ДО має дві конструкції та використовується для завдання заданого числа повторень. Розглянемо їх запис мовою програмування Pascal.

Конструкція по зростанню:
```
for i:=3 to 5 do begin
{тіло циклу i=3,4,5}
...
end;
```

Конструкція зі спадання:
```
for i:=5 downto 3 do begin
{тіло циклу i=5,4,3}
...
end;
```

Розглянемо запис варіантів кодування структури неуніверсального ЦИКЛ-ДО мовою програмування З.

Конструкція по зростанню:
```
for (i=3; i<=5; i++)
{
/ * Тіло циклу i = 3,4,5 * /
...
}
```

Конструкція зі спадання:
```
for (i=5; i>=3; i--)
{
/ * Тіло циклу i = 5,4,3 * /
...
}
```

Тут і – змінна циклу. Зазвичай, ці цикли не вимагають після кодування додаткового тестування.
Універсальні цикли мають конструкції ЦИКЛ-ДО та ЦИКЛ-ПОКИ. Їх запис на мові Pascal наведено нижче:

Універсальний ЦИКЛ-ПОКИ:
```
{Підготовка}
while L do
begin
{Тіло циклу}
...
end;
```

Універсальний цикл ЦИКЛ-ДО:
```
{Підготовка}
repeat
{Тіло циклу}
...
until L;
```

Нижче наведено запис тих самих структур мовою С:

Універсальний ЦИКЛ-ПОКИ:
```
/*Підготовка*/
while (L)
```

```
{
/*Тіло циклу*/
...
}
```

Універсальний ЦИКЛ-ДО:

```
/*Підготовка*/
do
{
/* Тіло циклу */
...
}
while (! (L))
```

Тут L логічний вираз. Його значення True є умовою продовження виконання ЦИКЛ-ПОКИ або умовою закінчення виконання ЦИКЛ-ДО. Підготовка та тіло циклу є СЛЕДЖЕННЯМИ. Тіло циклу виконується стільки разів, як і весь цикл. Ознакою ЦИКЛ-ПОКИ є можливість не виконання тіла циклу жодного разу. При рівноцінності з двох конструкцій ЦИКЛ-ДО і ЦИКЛ-ПОКИ вибирають ту, запис якої коротше.
Взагалі ЦИКЛ-ДО можна закодувати структурою ЦИКЛ-ПОКИ, якщо у підготовці записати деякі дії із тіла циклу. З ЦИКЛУ-ДО виходить ЦИКЛ-ПОКИ за його охоплення структурою АЛЬТЕРНАТИВУ.
Порядок декомпозиції циклів:
1) набирається "порожній" текст оператора циклу;
2) записується логічне умова продовження ЦИКЛ-ПОКИ чи завершення ЦИКЛ-ДО (у своїй виявляється змінна циклу);
3) декомпозується та дія тіла циклу, яка змінює логічну умову до невиконання умови ЦИКЛ-ПОКИ або до виконання умови ЦИКЛ-ДО;
4) деталізується СЛІД «Підготовка циклу»;
5) деталізується решта дії тіла циклу як СЛІД;
6) проводиться перевірка інформаційної узгодженості всіх елементів циклу;
7) проводиться перевірка правильності роботи циклу за допомогою безмашинного розрахунку траси виконання тестів із трикратним (або більше), одноразовим виконанням циклу та взагалі без виконання.
У текстах програм може використовуватися ще одна обчислювальна структура – РЕКУРСІЯ. Ознакою цієї структури є наявність рекурсивних формул та обчислень. Все, що робить рекурсія, можна продати за допомогою циклів і масивів. Однак якщо мова програмування допускає рекурсію, її використання може скоротити код програми. Рекурсія завжди дуже ретельно коментується.

5.9. МЕТОДИКА РОЗРОБКИ АЛГОРИТМІВ ПРОГРАМ

Розглянемо порядок роботи з методики розробки структурованих алгоритмів на прикладі. Нехай потрібно розробити програму розв'язання квадратного рівняння, яке має вигляд ax2 + bx + c = 0.
Робота за методикою починається з повного з'ясування задачі. Цьому допомагає застосування моделі «чорного ящика», розробка форм програми, що вводиться і виводиться (наприклад, у вигляді макетів екрану), підготовка первинних тестових прикладів. Немає для всього різноманіття завдань точний порядок виконання цих дій. Дані дії часто доводиться виконувати паралельно, перемикаючись з дії на дію з вичерпанням можливостей розвитку поточної дії та відкриття можливостей розвитку чергової дії.
Спочатку алгоритм повинен представляти одну типову структуру СЛІД (одна дія зі змістом виконати всі дії програми, наприклад, програма нарахування зарobленої плати, але не програма нарахування зарobленої плати та/або рішення квадратного рівняння).

Дивлячись на тести та зображення моделі «чорної скриньки» (див. рис. 5.3), деталізуємо весь алгоритм як одну СЛІД (послідовно виконувану дію) у порядку: а) попередній запис сенсу дії «чорної скриньки»; б) вихідна та/або виведена інформація; в) вхідна та/або інформація, що вводиться; г) визначається дія в «чорній скриньці» (одна пропозиція).

При розробці алгоритмів програм вхідна, проміжна та вихідна інформації характеризуються структурою даних. Важливим є порядок розміщення значень у масивах, імена та значення констант опису розмірностей масиву, імена та значення змінних, що характеризують поточні значення використовуваного розміру масиву, ім'я та порядок зміни змінної індексу поточного елемента масиву. Форма введеної та виведеної на екран або друк інформації може бути показана макетами екранів або документів.

Первинні тестові приклади повинні включати як звичайні, і стресові набори тестових вхідних даних. Кожен стресовий набір тестових даних призначений виявлення реакції у випадках. Наприклад: невірних дій користувача, поділу на нуль, виходу значення за допустимі межі тощо. буд. Будь-який набір тестових даних має містити опис результату.

Досліджуючи «чорну скриньку» стосовно розв'язання квадратного рівняння, можемо записати попередній коментар суті всіх дій програми: «Програма розв'язання квадратного рівняння виду a*x*x + b*x + c = 0».

Далі з'ясовується, що ще не виявлено вихідну інформацію «чорної скриньки», тому необхідно перейти до підготовки тестів, що допоможе продовжити роботу з «чорною скринькою». У даному випадку підготувати тести допоможе аналіз завдання.

Отже, нехай відома "шкільна" формула розв'язання квадратного рівняння виду ax2 + bx + c = 0.

Відомо також, що спочатку треба обчислити дискримінант рівняння D:

D = b2 - 4ac.

Навіть якщо забули про випадок негативності дискримінанта нічого страшного немає. Записуємо формулу рішення:

$$x1 = (-b - \sqrt{D})/(2a);$$

$$x2 = (-b + \sqrt{D})/(2a).$$

Нам відомо, якщо D < 0, то з негативного числа не можна витягувати квадратний корінь. Тому згадуємо, що за негативного дискримінанта немає коренів. Ще виявляємо факт особливого випадку, якому відповідає факт при D = 0 наявності двох рівних коренів. Ще відомо, що ділити на нуль не можна, а за a = 0 маємо саме цей випадок. У цьому випадку вихідне квадратне рівняння перетворюється на лінійне рівняння:

bx+c=0.

Рішення рівняння, що вийшло, буде наступним:

x = (-c) / b.

Це рішення можливе лише у разі a = 0 і (одночасно) b ≠ 0. У разі a = 0 і (одночасно) b = 0 і (одночасно) c ≠ 0 лінійне рівняння не має рішення.

Аналізуючи вихідне рівняння, з'ясовуємо, що у випадку a = 0 і (одночасно) b = 0 і (одночасно) c = 0 рівняння має безліч рішень (коріння x1 і x2 — будь-які числа).

Складемо наочну таблицю правил розв'язання квадратного рівняння (табл. 5.3).

Таблиця 5.3

Наочна таблиця правил розв'язання квадратного рівняння

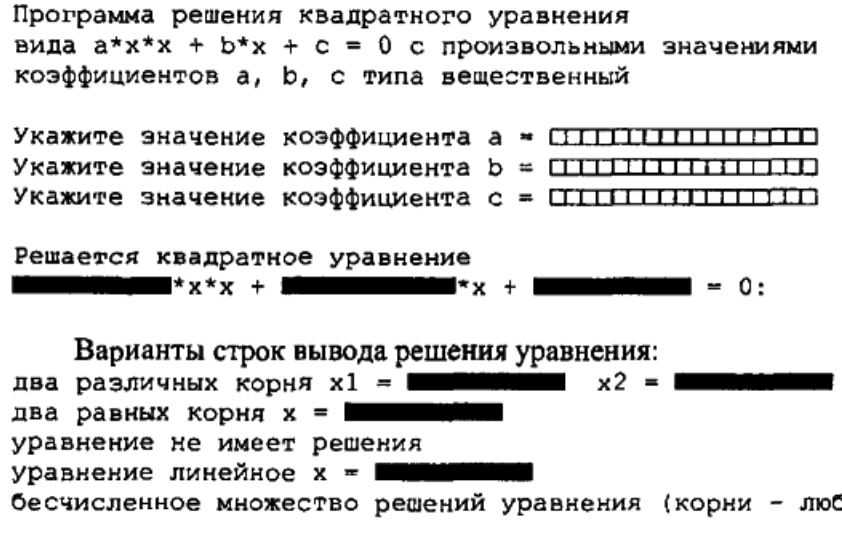
№ п/п	a	b	з	d	Варіант розв'язання
1	a ≠ 0	Будь-яке	Будь-яке	d > 0	Два різні корені
2	a ≠ 0	Будь-яке	Будь-яке	d = 0	Два рівні корені
3	a ≠ 0	Будь-яке	Будь-яке	d < 0	Немає рішення
4	a = 0	b ≠ 0	Будь-яке	Ні	Є корінь лінійного рівняння
5	a = 0	b = 0	c ≠ 0	Ні	Немає рішення
6	a = 0	b = 0	з = 0	Ні	Безліч рішень

У табл. 5.3 немає поєднань значень, які ще виявлено. Тепер можна визначити вихідну інформацію «чорної скриньки», яка видається у п'яти варіантах:
1) рівняння має безліч рішень (коріння x1 і x2 — будь-які числа);
2) значення двох різних коренів x1 та x2;
3) значення двох рівних коренів у вигляді x1 і доповнює написи про два рівні корені;
4) напис не має рішення;
5) значення одного кореня x1 із написом, що рівняння є лінійним.
Тип змінних, у яких розміщуються вихідні значення коренів x1 і x2, - речовий (Real). Тепер визначаємо вхідну інформацію. З вихідного рівняння випливає, що вхідною інформацією є значення трьох коефіцієнтів a, b, c речовий типу (Real). У ході аналізу формул було встановлено, що значення трьох коефіцієнтів a, b, c можуть набувати будь-яких значень, що було не очевидно до аналізу формул розв'язання рівняння (наприклад, випадок a = 0).
Імена змінних будуть досить мнемонічними, якщо дотримуватися прийнятих у математиці позначень.
Остаточний коментар суті дій усієї програми: «Програма розв'язання квадратного рівняння a*x*x + b*x + c = 0 з довільними значеннями коефіцієнтів a, b, c типу речовий». Факт довільності значень коефіцієнтів a, b, c на етапі попереднього виявлення суті дії «чорної скриньки» ще виявлено.
Зрештою, готуємо тестові приклади.
Сукупність тестів для всіх виявлених випадків розв'язання квадратного рівняння:
1) при a = 0, b = 0, c = 0 безліч рішень (коріння x1 і x2 — будь-які числа);
2) при a = 2, b = 3, c = -2 значення двох різних коренів x1 = -2 і x2 = 0,5;
3) при a = 1, b = 4, c = 4 значення двох рівних коренів у вигляді x1 = x2 = -2 і виведення доповнюючого напису про два рівні корені;
4) при a = 2, b = 5, c = 4 виведення напису «немає рішення»;
5) при a = 0, b = 2, c = -8 значення одного кореня x1 = 4 з написом, що рівняння є лінійним;
6) при a = 0, b = 0, c = 2 виведення напису "немає рішення". Тепер можна відразу написати фрагмент програми, що відповідає виконаній роботі:

```
Program Kvadrat;
{ Програма розв'язання квадратного рівняння
виду a * x * x + b * x + c = 0 з довільними
значеннями коефіцієнтів a, b, c типу
речовий }
Uses
Crt, Dos;
Var
a, b, c: Real; {Коефіцієнти квадратного рівняння}
xl, x2: Real; {Корні квадратного рівняння}
begin
end.
```

Шляхом компіляції фрагмента програми можна перевірити коректність синтаксису. Тепер підготуємо макет зображення на екрані монітора (рис. 5.16). На макеті зображення екрана монітора символами □ позначені поля введення інформації, а символами ■ поля виведення інформації.
Зазвичай, що виводиться на екран інформація не містить імен змінних, але в даному випадку прийняті в математиці імена доцільно відобразити на екрані.
Макет зображення монітора також є тестом. На основі первинних тестових прикладів та організації вхідної та вихідної інформації готується або один тест, або кілька узагальнюючих тестів. Усі складені тести на вирішення квадратного рівняння увійшли в узагальнюючий тест.
Розробка наочних тестів. Для найпростішого алгоритму розв'язання квадратного рівняння первинні тести разом з математичними формулами вже мають наочність. Продовжимо складання програми розв'язання квадратного рівняння.

Спочатку потрібно виконати дії, визначені макетом зображення на екрані монітора, так як для виконання алгоритму розв'язання квадратного рівняння необхідні вихідні дані як коефіцієнтів a, b, c. Однак введення має передувати виведення інформації, що пояснює призначення програми і суть чергової порції даних, що вводиться. Це визначає первинну деталізацію одиночне СЛІДЖЕННЯ в ланцюжок СЛІД. Первинне одиночне СТЕЖЕННЯ не може містити вхідний та вихідний інформації. Вхідна інформація повинна бути введена, або присвоєна. Вихідна інформація після завершення програми нікого не цікавить. Отже, нижче представлено ланцюжок послідовних дій.



*Мал. 5.16.*Макет зображення екрану монітора

```
ClrScr; {Очищення екрана}
{Виведення інформації про призначення програми}
WriteLn ('Програма розв'язання квадратного рівняння');
WriteLn ("виду a * x * x + b * x + c = 0 з довільні", "ми значеннями");
WriteLn ('коефіцієнтів a, b, c типу речовий');
WriteLn;
{Введення значень коефіцієнтів a, b, c}
Write ('Вкажіть значення коефіцієнта a =');
ReadLn (a); {Введення a}
Write ('Вкажіть значення коефіцієнта b=');
ReadLn (b); {Введення b}
Write ('Вкажіть значення коефіцієнта з =');
ReadLn (c); {Введення з}
{ Виведення перевірочно-протокольної інформації про введені значення
коефіцієнтів a, b, c}
WriteLn;
WriteLn ('Вирішується квадратне рівняння');
WriteLn (a:10:4, '*x*x + ', b:10:4, '* x + ', з:10:4, '= 0:'); { Саме рішення
квадратного рівняння }
WriteLn;
Write ('Для завершення програми натисніть');
WriteLn ('будь-яку клавішу...');
Repeat until KeyPressed; { Цикл очікування натискання будь-якої клавіші }
```

Дія «Очищення екрану» — артефакт реалізації, а не завдання, але наскільки приємніше оглядати екран без зайвої інформації. Також артефактом реалізації з'явилися два останні оператори, які забезпечують зручність перегляду результатів роботи програми. За відсутності цих операторів та знаходження в оболонці Turbo Pascal для перегляду результатів роботи програми довелося б вручну перейти у вікно результатів.

Написання операторів WriteLn, Write, ReadLn не викликало труднощів завдяки заздалегідь підготовленому макету зображення екрана монітора.

Дія «Само рішення квадратного рівняння» представлена коментарем та порожнім рядком, що наголошує на факті додавання дій у майбутньому. Це пояснюється тим, що рішення та виведення результатів рішення багатоваріантні, а, отже, дію «Само рішення квадратного рівняння» не можна уявити ланцюжком Слідування. Багатоваріантність передбачає керуючі структури.

Уточнюємо коментарі, оператори виводу Write та WriteLn.

Проводимо перевірку інформаційної узгодженості СЛІД в ланцюжку. Переконуємося, що всі наступні дії забезпечені інформацією, визначеною попередніми діями, у конкретному випадку операторами ReadLn. Особливу увагу звертаємо на дії, які використовують певну інформацію. Це оператор

```
WriteLn (a:10:4, '*x*x + ', b:10:4, '*x + ', c: 10:4, '= 0: ');
```

та майбутня дія

```
{ Саме рішення квадратного рівняння}.
```

Переконуємося, що на момент виконання значення коефіцієнтів a, b, c вже визначено. Тепер можна зібрати реалізовану частину програми та шляхом її виконання на тестах переконатися, що дії введення та виведення введеної інформації програми виконуються коректно.

```
Program Kvadrat;
{ Програма розв'язання квадратного рівняння
виду а * х * х + Ь * х + с = 0 з довільними значеннями
коефіцієнтів а, Ь, з типу речовий }
Uses
Crt, Dos;
Var
a, b, c: Real; {Коефіцієнти квадратного рівняння}
xl, x2: Real; {Корні квадратного рівняння}
begin
ClrScr; {Очищення екрана}
{Виведення інформації про значення програми}
WriteLn ('Програма розв'язання квадратного рівняння');
Write ('виду а * х * х + b * х + с = 0 з довільними');
Write ('значеннями');
WriteLn ('коефіцієнтів а, b, з типу речовий');
WriteLn
{Введення значень коефіцієнтів а, b, с}
Write ('Вкажіть значення коефіцієнта а =');
ReadLn (a); {Введення а}
Write ('Вкажіть значення коефіцієнта b=');
ReadLn (b); {Введення b}
Write ('Вкажіть значення коефіцієнта з =');
ReadLn(c); {Введення з}
{ Висновок перевірочно-протокольної інформації про введені значення коефіцієнтів
а, Ь, с}
WriteLn;
WriteLn ('Вирішується квадратне рівняння');
Write (a:10:4, '*x*x + ', b:10:4, '*x + ');
WriteLn (c:10:4, ' = 0 : ' );
{ Саме рішення квадратного рівняння }
WriteLn;
WriteLn ('Для завершення програми натисніть ', 'будь-яку клавішу...');
repeat until KeyPressed; { Цикл очікування натискання будь-якої клавіші}
end.
```

Під час складання програми довелося здійснити перенесення частини оператора WriteLn на новий рядок.

Тепер здійснюємо декомпозицію дії «Само розв'язання квадратного рівняння».

Багатоваріантність обчислень передбачає ланцюжок альтернатив. Аналізуючи математичні формули узагальнюючого тесту, табл. 5.3 та склад наборів вихідної інформації, виявляємо, що ланцюжок альтернатив містить чотири альтернативні дії. Рядкам з 1-ї по 3-ю табл. 5.3 відповідає одна дія, оскільки для їх виконання потрібно обчислене значення дискримінанта d. Записуємо коментар попереднього СЛІДЖЕННЯ всього ланцюжка альтернатив, набір

вхідної інформації (вихідної інформації - ні) і оформляємо заготівлю операторів ланцюжка альтернатив разом з підлеглими Слідуваннями:

Вхідна інформація: a, b, c

```
{ Саме рішення квадратного рівняння }
if
then
begin
{Продовження рішення з обчисленням дискримінанта}
end;
if
then
begin
{ Рішення лінійного рівняння }
end;
if
then
begin
{ Введення повідомлення: лінійне рівняння не має рішення}
WriteLn ("Немає рішення")
end;
if
then
begin
{ Висновок повідомлення: безліч рішень рівняння }
Write ('незліченна безліч рішень на рівні');
WriteLn ('ня (коріння - будь-які числа)');
end;
```

В останній альтернативі один рядок виводиться одним оператором.

Далі відповідно до дій запишемо логічні умови виконання дій. При цьому простим порівнянням перевіряти на рівність значення двох речових змінних не можна. Наприклад, при порівнянні f = g, які вважаються рівними 5, навіть якщо g = 5,00000, в силу округлень при обчисленнях значення f може бути рівним або 4,99999, або 5,00000, або 5,00001. Згідно з цим прикладом шляхом простої перевірки на рівність факт рівності буде встановлений в одному випадку з трьох.

Для надійного порівняння двох дійсних чисел використовують прийом використання нерівності $| f - g | \le \varepsilon$, де ε — свідомо мале число. Мовою програмування ця нерівність має вигляд

```
Abs (f - g) <= 1e - 6
```

Продовжуємо кодування структури. Дивлячись на дії, записуємо логічні умови виконання дій. Вхідна інформація: a, b, c.

```
{ Саме рішення квадратного рівняння }
if (Abs(a) > 1e - 6)
then
begin
{Продовження рішення з обчисленням дискримінанта}
end;
if ((Abs (a) <= 1e - 6) and (Abs (b) > 1e - 6))
then
begin
{ Рішення лінійного рівняння }
end;
if ((Abs(a) <= 1e - 6) and (Abs(b) <= 1e - 6 and (Abs(c) > 1e - 6))
then
begin
{ Висновок повідомлення: лінійне рівняння не має рішення}
WriteLn ('Немає рішення');
end;
if ((Abs(a) <= 1e - 6) and (Abs(b) <= 1e - 6 and (Abs(c) <= 1e - 6))
then
begin
{ Висновок повідомлення: безліч рішень рівняння }
```

```
Write ('незліченна безліч рішень на рівні');
WriteLn ('ня (коріння - будь-які числа)');
end;
```

Здійснимо складання програми, що вийшла. При складанні видалимо надлишкові коментарі та надмірні операторні дужки begin - end, що охоплюють лише один оператор. Випробуємо отриману програму на тестах a = 0, b = 0, c = 0 a = 0, b = 0, c = 2. Зібраний варіант програми:

```
Program Kvadrat;
{ Програма розв'язання квадратного рівняння
виду a * x * x + b * x + c = 0 з довільними значеннями
коефіцієнтів a, b, c типу речовий }
Uses
Crt, Dos;
Var
a, b, c: Real; {Коефіцієнти квадратного рівняння}
x1, x2: Real; {Корні квадратного рівняння}
begin
ClrScr; { Очищення екрана }
{Виведення інформації про призначення програми}
WriteLn ('Програма розв'язання квадратного рівняння');
Write ("виду a * x * x + b * x + c = 0 з довільними");
Write ('значеннями');
WriteLn ('коефіцієнтів a, b, c типу речовий');
WriteLn;
{Введення значень коефіцієнтів a, b, c};
Write ('Вкажіть значення коефіцієнта a =');
ReadLn(a); {Введення a}
Write ('Вкажіть значення коефіцієнта b=');
ReadLn(b); {Введення b}
Write ('Вкажіть значення коефіцієнта з =');
ReadLn(c); {Введення з}
{ Висновок перевірочно-протокольної інформації
про введені значення коефіцієнтів a, b, c}
WriteLn;
WriteLn ('Вирішується квадратне рівняння');
Write (a:10:4, '*x*x + ', b:10:4, '*x + ');
WriteLn(з:10:4, '= 0:');
{ Саме рішення квадратного рівняння }
if (Abs (a) > 1e - 6)
then
begin
{Продовження рішення з обчисленням дискримінанта}
end;
if ((Abs(a) <= 1e - 6) and (Abs(b) > 1e - 6))
then
begin
{ Рішення лінійного рівняння }
end;
if ((Abs(a) <= 1e - 6) and (Abs(b) <= 1e - 6) and (Abs(c) > 1e - 6))
then
WriteLn ('Немає рішення');
if ((Abs(a) <= 1e - 6) and (Abs(b) <= 1e - 6 and (Abs(c) <= 1e - 6))
then
begin
Write ('незліченна безліч рішень на рівні');
WriteLn ('ня (коріння - будь-які числа)');
end;
WriteLn;
Write ('Для завершення програми натисніть');
WriteLn ("будь-яку клавішу ...");
repeat until KeyPressed; { Цикл очікування натискання будь-якої клавіші }
end.
```

Декомпозуємо дію «Рішення лінійного рівняння». Ця дія представляє ланцюжок із двох елементарних операторів. Виконаємо перевірку інформаційної узгодженості дій:

Вхідна інформація: b, c.

```
{ Рішення лінійного рівняння }
x1 := -c/b;
WriteLn ('рівняння лінійне x = ', x1: 10: 4);
```

Декомпозуємо дію "Продовження рішення з обчисленням дискримінанта". Дана дія представляє ланцюжок Слідкувань з двох Слідкувань.

Вхідна інформація: a, Ь, c

```
{Продовження рішення з обчисленням дискримінанта}
{Обчислення дискримінанта квадратного рівняння}
d := Sqr (b) - 4.0 * a * c; { Рішення рівняння }
```

Змінна d у нас не описана, тому до секції Var необхідно додати рядок опису:

```
d: Real; { Значення дискримінанта }
```

Декомпозуємо дію «Рішення рівняння». Відповідно до табл. 5.3 ця дія представляє ланцюжок альтернатив з трьох альтернатив в ланцюжку. Здійснивши деталізацію цих альтернатив у встановленому порядку, отримаємо:

Вхідна інформація: a, b, c, d.

```
{ Рішення рівняння }
if d > 1e-6
then
begin
{Розрахунок двох різних коренів}
end;
if ((d >= -1e-6) and (d <= 1e-6))
then
begin
{Розрахунок двох рівних коренів}
WriteLn ('два рівні корені x = ', (-b)/(2.0 * a) :10:4);
end;
if d < -1e-6 then begin
{Виведення напису: рівняння не має рішення}
WriteLn ('рівняння немає рішення');
end;
```

Декомпозуємо дію «Розрахунок двох різних коренів». Ця дія представляє ланцюжок із трьох елементарних операторів. Виконаємо перевірку інформаційної узгодженості дій:

Вхідна інформація: a, b, c, d

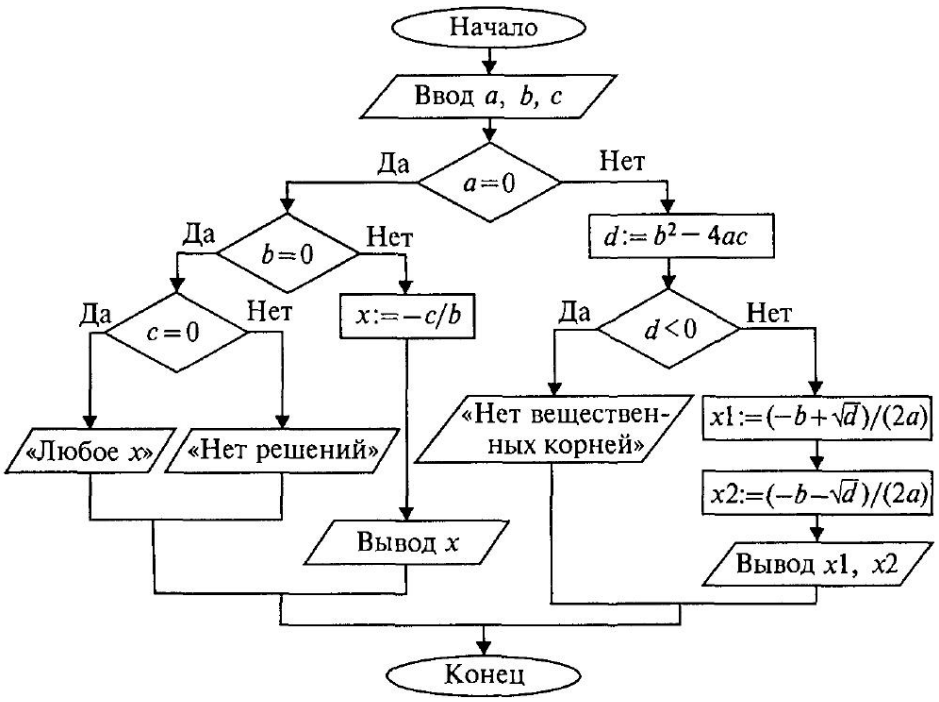
```
{Розрахунок двох різних коренів}
x1 := ((-b) - Sqrt (d))/(2.0*a);
x2 := ((-b) + Sqrt (d))/(2.0*a);
Write ('два різні корені x1 = ', x1:10:4);
WriteLn ('x2 =', x2:10:4);
```

Тут виведення одного рядка здійснено двома операторами. Здійснимо складання всієї програми, вилучивши надлишкові коментарі та надлишкові операторні дужки begin - end, що охоплюють лише один оператор. Випробуємо отриману програму на всіх заздалегідь підготовлених тестах. Зібраний варіант програми:

```
Program Kvadrat;
{ Програма розв'язання квадратного рівняння
виду a * x * x + b * x + c = 0 з довільними значеннями
коефіцієнтів a, b, c типу речовий }
Uses
Crt, Dos;
Var
a, b, c: Real; {Коефіцієнти квадратного рівняння}
xl, x2: Real; {Корні квадратного рівняння}
dl: Real; {Значення дискримінанта}
begin
ClrScr; { Очищення екрана }
{Виведення інформації про призначення програми}
WriteLn ('Програма розв'язання квадратного рівняння');
WriteLn ("виду a * x * x + b * x + c = 0 з довільними", "значеннями");
WriteLn ('коефіцієнтів a, b, c типу', 'речовий');
```

```
WriteLn
{Введення значень коефіцієнтів a, b, c}
Write ('Вкажіть значення коефіцієнта a =');
ReadLn(a); { Введення a }
Write ('Вкажіть значення коефіцієнта b=');
ReadLn(b); {Введення b}
Write ('Вкажіть значення коефіцієнта з =');
ReadLn(c); { Введення з }
{ Висновок перевірочно-протокольної інформації
про введені значення коефіцієнтів a, b, c}
WriteLn;
WriteLn ('Вирішується квадратне рівняння');
WriteLn (a:10:4, '*x*x + ', b:10:4, '*x + ', з:10:4, '= 0:');
{ Саме рішення квадратного рівняння }
if (Abs (a) > 1e-6)
then
begin
{Продовження рішення з обчисленням дискримінанта}
{Обчислення дискримінанта квадратного рівняння}
d := Sqr(b) - 4.0 * a * c;
{ Рішення рівняння }
if d > 1e-6
then
begin
{Розрахунок двох різних коренів}
x1 := (-b) - Sqrt(d)/(2.0*a);
x2: = (-b) + Sqrt (d) / (2.0 * a);
Write ('два різні корені x1 = ', x1:10:4);
WriteLn ( 'x2 = ', x2: 10:4);
end;
if ((d >= -1e-6) and (d <= 1e-6))
then
WriteLn ('два рівні корені x = ', (-b)/(2.0*a):10:4);
if d <-1e-6 then
WriteLn ('рівняння немає рішення');
end;
if ((Abs(a) <= 1e-6) and (Abs(b) > 1e-6))
then
begin
{ Рішення лінійного рівняння }
x1 := -c/b;
WriteLn ('рівняння лінійне x = ', x1:10:4);
end;
if ((Abs(a) <= 1e-6) and (Abs(b) <= 1e-6 and (Abs(c) > 1e-6))
then
WriteLn ('Немає рішення');
if ((Abs(a) <= 1e-6 and (Abs(b) <= 1e-6 and (Abs(c) <= 1e-6)))
then
begin
Write ("Безліч рішень",
'рівні');
WriteLn ('ня (коріння - будь-які числа)');
end;
WriteLn;
Write ('Для завершення програми натисніть');
WriteLn ('будь-яку клавішу ...');
repeat until KeyPressed; { Цикл очікування натискання будь-якої клавіші }
end.
```

Для інтересу можна навести блок-схему:



5.10. ПРИКЛАД ВИКОНАННЯ НАВЧАЛЬНОЇ РОБОТИ «РОЗРОБКА АЛГОРИТМА ПРИМНОЖЕННЯ»

Як приклад наводиться навчальна робота, виконана одним із учнів. Роботу було оформлено на окремих аркушах формату А4. Курсивом виділено пояснення авторів підручника, які були додатково ними внесені до тексту роботи.

Сторінка 1(без нумерації) є титульний лист з найменуванням: «ЗАВАННЯ НА СКЛАДАННЯ СТРУКТУРОВАНОГО АЛГОРИТМУ».

Сторінка 2містить постановку завдання та набір тестів, складених до розробки алгоритму процесу.

КРОК 1. ПОСТАНОВКА ЗАВДАННЯ

Скласти алгоритм множення двох позитивних чисел із довільною (до ста) кількістю цифр. Цифри співмножників та результату повинні знаходитися в одновимірних масивах. Розрядність результату має перевищувати 100 цифр.

Крок 2. НАБІР ТЕСТІВ, СКЛАДЕНИХ ДО РОЗРОБКИ АЛГОРИТМУ ПРОЦЕСУ

Нехай гранична розрядність співмножників дорівнює трьом цифрам, а результату чотирьом. Аналогічно наведеному зразку множення чисел 391*56 = 21896 (переповнення) було складено тести: 23*132 = 3036; 111 * 11 = 1221; 999 * 99 = 98901 (переповнення); 00 * 000 = 0; 1 * 0 = 0.

```
   3 9 0
 x
     5 6
 ---------
 0 2 2 3 5 4 0 0
+
 0 1 1 9 4 5 0 0
 ---------
 0 2 1 1 0 8 0 4 0 0
```

Алгоритм множення зазвичай вивчався у молодших класах школи з маршрутного опису процесу рахунки. Через теоретичну кількість маршрутів більшість зі шкільної лави не знає процесу множення при нульових співмножниках!

Сторінка 3містить результати аналізу вихідної та вхідної інформації обчислювального процесу зі структурами даних. Раціональність вибраної структури даних значною мірою визначає раціональність алгоритму.

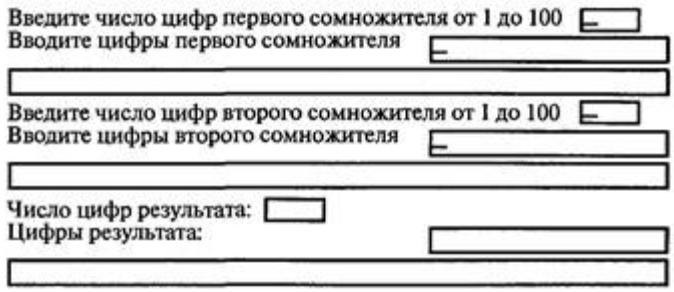
Крок 3. АНАЛІЗ ВИХІДНОЇ ТА ВХІДНОЇ ІНФОРМАЦІЇ ВИЧИСЛЮВАЛЬНОГО ПРОЦЕСУ

Аналіз вихідний та вхідний інформації починається з розгляду моделі «чорного» ящика, показаної на рис. 5.3.

```
Program MultNumbers;
{Розрахунок добутку двох чисел}
uses
Crt;
const
Digits = 100; {Число цифр у числах}
type
TNumber = record
D: array[1..Digits] of Byte;
{BD[1] знаходиться молодший розряд числа}
N: word; {Число розрядів у числі від 1 до Digits}
end;
var
C1: TNumber; {Перший співмножник}
C2: TNumber; {Другий співмножник}
R: TNumber; {Результат множення}
Error: boolean; {True - помилка переповнення}
```

Макет екрану з рядками діалогу програми наведено на рис. 5.17. Замість трьох останніх рядків можливий висновок: "Помилка переповнення".

Сторінка 4 містить наочне зображення процесу перетворення вхідних даних узагальнюючого тесту або тестів у вихідні дані з усіма внутрішніми даними та/або трасу виконання узагальнюючого тесту або тестів. Узагальнюючий тест чи тести складаються з урахуванням тестів сторінки 2 і за мінімальної кількості тестів охоплює всі маршрути процесу обчислень. Наочність зображення змін усіх даних сприяє спрощенню процесу розробки алгоритму. Раціональність вибраної структури даних значною мірою визначає раціональність алгоритму.



*Мал. 5.17.*Макет екрану з рядками діалогу програми

Крок 4. НАГЛЯДНЕ ЗОБРАЖЕННЯ ПРОЦЕСУ ПЕРЕТВОРЕННЯ ВХІДНИХ ДАНИХ УЗАГАЛЬНОГО ТЕСТА У ВИХІДНІ

```
      C1.N = [3] [2] [1] C1.D    i
               3   9   0
        x C2.N = [2] [1] C2.D    j
                 5   6
    ---------------------------
  +        2  ²3  ⁵4  00
       1  ¹9  ⁴5  ⁰0
    ---------------------------
           2  ¹1  ⁰8  ⁰4  ⁰0
R.N = [5] [4] [3] [2] [1] R.D
```

Вводимо опис нових внутрішніх змінних:

```
var {Робочі змінні}
i, j: word;
```

На перший погляд, здається, що необхідно ввести змінну кількість проміжних масивів (залежно кількості цифр другого співмножника) для запам'ятовування продукту множення першого співмножника на чергову цифру другого співмножника. Однак уважний аналіз структур даних дозволяє знайти інший спосіб виконання розрахунків. При цьому способі спочатку обнулюється результат. Далі результат послідовно збільшується на зрушений на розряд ліворуч продукт множення першого співмножника на чергову цифру другого співмножника. Переоформляємо наочне зображення процесу перетворення вхідних даних узагальнюючого тесту.

```
R.D =              0      R.N = 0
        [1] +             Итерация 1.
    390 x 6  =  2340      P
R.D =           2340      R.N = 4
R.D =           2340      R.N = 4
        [2] +             Итерация 1.
    390 x 5  =  1950      P
R.D =          21840      R.N = 5
```

Новые переменные:

```
var
  p     : word;  {Значение числа переноса при
                  умножении C1.D на очередную
                  цифру C2.D}
```

На сторінці 4 або наступній іноді корисно помістити опис алгоритму у звичайному неструктурованому розумінні.

Кожна наступна сторінка *містить результати процесу деталізації чергової виділеної від загального до приватного структури.*

Крок 5. НАСЛІДНІСТЬ ДЕТАЛІЗАЦІЇ АЛГОРИТМУ

Крок 5.1. Результати деталізації СЛІД «Вся програма»

Слідування «Вся програма» деталізується ланцюжком Слідування:

```
begin
ClrScr; {Очищення екрана}
{Введення коректного значення числа цифр першого співмножника}
C1.N
Write('Вводите цифри першого співмножника');
{Введення цифр першого співмножника у порядку від C1.D[C1.N] до C1.D[1]}
C1.D
WriteLn;
{Введення коректного значення числа цифр другого співмножника}
C2.
Write('Вводьте цифри другого співмножника');
{Введення цифр другого співмножника у порядку від C2.D[C2.N] до C2.D[1]}
WriteLn;
{Розрахунок твору співмножників}
RD RN Error
{Усунення провідних нулів}
WriteLn;
{Виведення результату твору}
WriteLn;
end.
```

Без відступів показана вхідна та вихідна інформація структур, яка використовувалась при перевірці інформаційної узгодженості СЛІД в ланцюжку СЛІД.

СЛІД «Усунення провідних нулів» необхідно при використанні співмножника, що складається з декількох нулів.

Крок 5.2. Деталізація СЛІД «Введення коректного значення числа цифр першого співмножника»

СЛІД «Введення коректного значення числа цифр першого співмножника» декомпозується циклом:

```
{Введення коректного значення числа цифр першого співмножника}
repeat
Write('Введіть число цифр першого співмножника');
Write(' від 1 до ', Digits, '');
ReadLn(C1.N);
until ((C1.N >= 1) and (C1.N <= Digits));
```

Цикл тестований трьома тестами: C1.N=1; C1.N=3; C1.N=Digits.

Аналогічно декомпозується процес "Введення коректного значення числа цифр другого співмножника".

Крок 5.3. Деталізація СЛІД «Введення цифр першого співмножника в порядку від C1.D[C1.N] до C1.D[1]

СЛІД «Введення цифр першого співмножника в порядку від C1.D[C1.N] до C1.D[1] декомпозується циклом:

```
{Введення цифр першого співмножника у порядку від C1.D[C1.N] до C1.D[1]}
for i := C1.N downto 1 do begin
{До введення символу цифри}
repeat
ch: = ReadKey; {Читання символу клавіатури}
Val(ch, C1.D[i], InCode); {Перетворення на значення}
until(InCode = 0);
Write(ch);
end;
```

Опис нових змінних:

```
var {Робочі змінні}
InCode: слово;
ch: Char;
```

Незважаючи на те, що тут, порушуючи правила, деталізовано відразу два цикли, у тестуванні немає необхідності.

Аналогічно декомпозується процес «Введення цифр другого співмножника у порядку від C2.D[C2.N] до C2.D[1].

Крок 5.4. Деталізація СЛІД «Висновок результату твору»

СЛІД «Висновок результату твору» декомпозується АЛЬТЕРНАТИВОЮ - РОЗВИТОК З ДВОМИ ДІЯМИ:

```
{Виведення результату твору}
if ERROR
then
WriteLn(Помилка переповнення)
else
begin
{Виведення продукту множення}
end;
```

Тести: ERROR = True; ERROR = False.

Крок 5.5. Деталізація СЛІД «Виведення продукту множення»

СЛІД «Виведення продукту множення» декомпозується циклом:

```
{Виведення продукту множення}
for i := RN downto 1 do
Write(RD[i]);
```

У тестуванні немає потреби.

Крок 5.6. Деталізація СЛІДЖЕННЯ «Усунення провідних нулів»

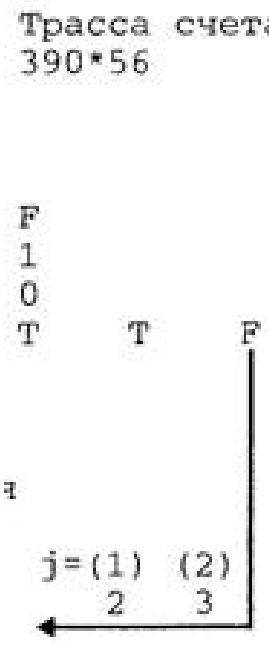
СЛІД «Усунення провідних нулів» декомпозується циклом:

```
{Усунення провідних нулів}
while ((RN > 1) and (RD[i] = 0)) do
Dec(RN); {RN := RN - 1}
```

У тестуванні немає потреби.

Крок 5.7. Деталізація СЛІД «Розрахунок твору співмножників»
СЛІД «Розрахунок твору співмножників» декомпозується циклом:
Вхід: C1, C2.

```
{Розрахунок твору співмножників}
{Цикл визначає номер j чергової цифри другого співмножника}
ERROR: = False;
j: = 1;
RD [1]: = 0;
while ((j <= C2.N) and
(not(ERROR))) do
begin
{Збільшення результату на зрушений продукт множення першого співмножника на j-у
цифру другого співмножника}
Inc(j); {j:=j+1}
end;
```



Вихід: RD, RN, ERROR

Структура тестувалась на тестах: 390*56; 390 * 56, але при Digits = 5; 0 * 0 при C1.N = 0; 1*0 при C1.N = 1 та інших тестах.
Крок 5.8. Деталізація СЛІДЖЕННЯ «Збільшення результату на зрушений продукт множення першого співмножника на j-у цифру другого співмножника
СЛЕДЖЕННЯ деталізується циклом:
Вхід: C1, C2.
{Збільшення результату на зрушений продукт множення першого співмножника на j-у цифру другого співмножника}

```
p: = 0;
i := 0; {Номер цифри першого співмножника}
while(((i < C1.N) or (p <> 0)) and (not(ERROR))) do
begin
Inc(i);
{Розрахунок чергової цифри результату та цифри перенесення}
end;
```

Вихід: RD, RN, ERROR
Крок 5.9. Деталізація СЛІД «Розрахунок чергової цифри результату та цифри перенесення»
СЛЕДЖЕННЯ деталізується альтернативою:
Вхід: i, j, C1.D[i], C2.D[j], p, RD, Digits.

```
{Розрахунок чергової цифри результату та цифри перенесення}
```

```
{Контрольований розрахунок ir — номери чергової цифри результату}
ir := i + j - 1;
if (ir > Digits) then
ERROR := True else
begin
{Зміна довжини результату RN}
if (RN < ir)
then
begin
RN := ir;
RD [ir]: = 0; {Обнулення нової цифри результату}
end;
{Отримання чергової цифри C1D першого співмножника}
if (i <= C1.N) then C1D := C1.D[i] else C1D := 0;
{Зміна чергової цифри результату та p}
RD: = p + RD [ir] + C1D * C2.D [j];
RD[ir]: = RD mod 10;
p := RD div 10;
end;
```

Вихід: RD, RN, ERROR, p.

Опис нових змінних:

```
var
ir, C1D, RD: слово; {Робочі змінні}
```

Крок 6. РЕЗУЛЬТАТИ ЗБІРАННЯ ПРОГРАМИ

```
Program MultNumbers;
{Розрахунок добутку двох чисел}
uses
Crt;
const
Digits = 100; {Число цифр у числах}
type
TNumber = record
D: array[1..Digits] of Byte;
{BD[1] знаходиться молодший розряд числа}
N: word; {Число розрядів у числі від 1 до Digits}
end;
var
C1: TNumber; {Перший співмножник}
C2: TNumber; {Другий співмножник}
R: TNumber; {Результат множення}
Error: boolean; {True - помилка переповнення}
var
p: word; {Значення числа перенесення при множенні C1.D на чергову цифру C2.D}
var {Робочі змінні}
i, j, ir, C1D, RD, InCode: word;
ch: char; begin
ClrScr; {Очищення екрана}
{Введення коректного значення числа цифр першого співмножника}
repeat
Write('Введіть число цифр першого співмножника)
Write('1 до', Digits,'');
ReadLn(C1.N);
until ((C1.N >= 1) and (C1.N <= Digits));
Write('Вводите цифри першого співмножника');
{Введення цифр першого співмножника у порядку від C1.D[C1.N] до C1.D[1]}
for i := C1.N downto 1 do
begin
{До введення символу цифри}
repeat
ch: = ReadKey; {Читання символу клавіатури}
Val(ch, C1.D[i], InCode); {Перетворення на значення}
until(InCode = 0);
Write(ch);
```

```
end;
WriteLn;
{Введення коректного значення числа цифр другого співмножника}
repeat
Write('Введіть число цифр другого співмножника');
Write(' від 1 до ', Digits,' ');
ReadLn(C2.N);
until ((C2.N >= 1) and (C2.N <= Digits));
Write('Вводьте цифри другого співмножника');
{Введення цифр другого співмножника у порядку від C2.D[C2.N] до C2.D[1]}
for i := C2.N downto 1 do
begin
{До введення символу цифри}
repeat
ch: = ReadKey; {Читання символу клавіатури}
Val(ch, C2.D[i], InCode); {Перетворення на значення}
until(InCode = 0);
Write(ch);
end;
WriteLn;
{Розрахунок твору співмножників}
{Цикл визначає номер j чергової цифри другого співмножника}
ERROR: = False;
j: = 1;
RD [1]: = 0;
while ((j <= C2.N) and (not(ERROR))) do
begin
{Збільшення результату на зрушений продукт множення першого співмножника на j-у
цифру другого співмножника}
P: = 0;
i := 0; {Номер цифри першого співмножника}
while(((i < C1.N) or (p <> 0)) and (not(ERROR))) do
begin
Inc(i);
{Розрахунок чергової цифри результату та цифри перенесення}
{Контрольований розрахунок ir — номери чергової цифри результату}
ir: = i + j - 1;
if (ir > Digits) then
ERROR := True
else
begin
{Зміна довжини результату RN}
if (RN < ir)
then
begin
RN := ir;
RD [ir]: = 0; {Обнулення нової цифри результату}
end;
{Отримання чергової цифри C1D першого співмножника}
if (i <= C1.N)
then
C1D := C1.D[i]
else
C1D: = 0;
{Зміна чергової цифри результату та p}
RD: = p + RD [ir] + C1D * C2.D [j];
RD[ir]: = RD mod 10;
p := RD div 10;
end;
end;
Inc(j); {j:=j+1}
end;
{Усунення провідних нулів}
while ((RN > 1) and (RD[i] = 0)) do
```

```
Dec(RN); {RN := RN - 1}
WriteLn;
{Виведення результату твору}
if ERROR
then
WriteLn('Помилка переповнення')
else
begin
{Виведення продукту множення}
for i := RN downto 1 do
Write(RD[i]);
end;
WriteLn;
end.
```

Після закінчення складання програми має сенс ще раз відредагувати коментарі із вилученням «зайвих» коментарів.

5.11. ПРИКЛАД ЗАСТОСУВАННЯ ПРОЕКТНОЇ ПРОЦЕДУРИ ДЛЯ КОДИРУВАННЯ ПРОГРАМИ ДРУКУ КАЛЕНДАРЯ НА ПРИНТЕРІ

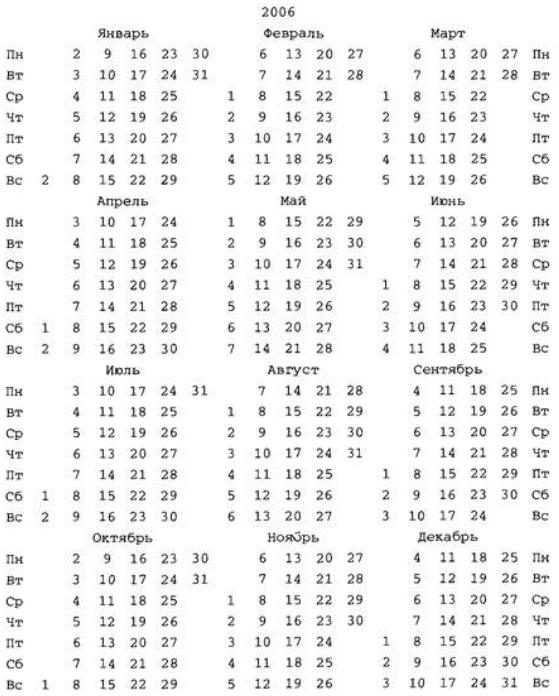
Нехай потрібно розробити програму друку календаря заданого року на матричному принтері. Обмежимося роками після 1917 р. Матричний принтер може друкувати інформацію послідовно один рядок за другим.

Нижче наведено розроблений макет друку вихідної інформації. З аналізу макета стало очевидно, що вхідною інформацією програми є лише рік календаря, що виводиться на друк. Розробляємо макет екрану діалогу програми.

Приступаємо до розробки оперативної структури даних програми. Що потрібно для друку календаря року? Звичайно, можна створити статичну матрицю, яка містить усі символи календаря, або статичний вектор усіх рядків макету календаря. Спробуємо обійтися без великих масивів, формуючи послідовно черговий рядок інформації.

Що бачимо? Інформація дат місяця представлена на макеті сімома рядками та шістьма колонками дат місяців. Що потрібне для друку дат конкретного місяця? Кількість порожніх клітин дат на початку місяця та кількість днів на місяці. Кількість днів у всіх місяцях року фіксована, за винятком лютого. Кількість днів у лютому залежить від того, чи є рік високосним чи ні. Отже, кількість днів у всіх місяцях року маємо у статичному масиві. Кількість порожніх клітин дат на початку наступного місяця визначається інформацією попереднього місяця. Цю інформацію також маємо в статичному масиві. Для заповнення статичного масиву кількості порожніх клітин дат на початку місяців року необхідно знати інформацію про кількість порожніх клітин дат на початку січня року друку календаря. Кількість порожніх клітин дат на початку січня року друку календаря можна визначити виходячи з кількості порожніх клітин дат на початку січня 1917 з урахуванням виявленого з макета наступного факту, який полягає в тому, що кількість порожніх клітин дат на початку січня наступного року більше на 1 у невисокосний рік та на 2 у високосний рік.

```
Програма друку календаря заданого року
Вкажіть рік роздрукованого календаря після 1917 року
■■■■
(Не можу скласти календар ■■■■ року
Щоб завершити програму, натисніть будь-яку клавішу ...)
Чекайте, йде друк...
Щоб завершити програму, натисніть будь-яку клавішу ...
```



```
                                2006
          Январь                Февраль               Март
Пн      2  9 16 23 30      6 13 20 27      6 13 20 27 Пн
Вт      3 10 17 24 31      7 14 21 28      7 14 21 28 Вт
Ср      4 11 18 25      1  8 15 22      1  8 15 22    Ср
Чт      5 12 19 26      2  9 16 23      2  9 16 23    Чт
Пт      6 13 20 27      3 10 17 24      3 10 17 24    Пт
Сб      7 14 21 28      4 11 18 25      4 11 18 25    Сб
Вс   2  8 15 22 29      5 12 19 26      5 12 19 26    Вс
          Апрель                Май                   Июнь
Пн      3 10 17 24      1  8 15 22 29      5 12 19 26 Пн
Вт      4 11 18 25      2  9 16 23 30      6 13 20 27 Вт
Ср      5 12 19 26      3 10 17 24 31      7 14 21 28 Ср
Чт      6 13 20 27      4 11 18 25      1  8 15 22 29 Чт
Пт      7 14 21 28      5 12 19 26      2  9 16 23 30 Пт
Сб   1  8 15 22 29      6 13 20 27      3 10 17 24    Сб
Вс   2  9 16 23 30      7 14 21 28      4 11 18 25    Вс
          Июль                  Август                Сентябрь
Пн      3 10 17 24 31      7 14 21 28      4 11 18 25 Пн
Вт      4 11 18 25      1  8 15 22 29      5 12 19 26 Вт
Ср      5 12 19 26      2  9 16 23 30      6 13 20 27 Ср
Чт      6 13 20 27      3 10 17 24 31      7 14 21 28 Чт
Пт      7 14 21 28      4 11 18 25      1  8 15 22 29 Пт
Сб   1  8 15 22 29      5 12 19 26      2  9 16 23 30 Сб
Вс   2  9 16 23 30      6 13 20 27      3 10 17 24    Вс
          Октябрь               Ноябрь                Декабрь
Пн      2  9 16 23 30      6 13 20 27      4 11 18 25 Пн
Вт      3 10 17 24 31      7 14 21 28      5 12 19 26 Вт
Ср      4 11 18 25      1  8 15 22 29      6 13 20 27 Ср
Чт      5 12 19 26      2  9 16 23 30      7 14 21 28 Чт
Пт      6 13 20 27      3 10 17 24      1  8 15 22 29 Пт
Сб      7 14 21 28      4 11 18 25      2  9 16 23 30 Сб
Вс   1  8 15 22 29      5 12 19 26      3 10 17 24 31 Вс
```

Нам потрібно чотири рядки найменувань місяців кварталу та сім рядків найменувань днів тижня. Ця інформація залишається незмінною у процесі виконання програми, тому розміщуємо її у константних масивах Turbo Pascal.

Отже, нам потрібна дія, що багаторазово застосовується: визначення, чи є досліджуваний рік високосним чи ні. Оформляємо цю дію як функцію з результатом логічного типу, що повертається. Алгоритм цієї функції візьмемо зі шкільного підручника з астрономії.
Текст програми друку календаря на матричному принтері наведено нижче. Звіряючись із цим текстом, самостійно закодуйте програму.

```
Program Calendar;
{Програма друку календаря заданого року на матричному принтері.
Рік має бути починаючи з YEARBASE року}
Uses
Crt, Dos;
Const
YEARBASE = 1917; {Початковий базовий рік}
BLANKS1917 = 0; {Кількість порожніх клітин на початку січня базового року}
KVARTALNAME: array [1...4] of String [53] =
( {Константи вкорочені при макетуванні! }
'Січень Лютий Березень',
'Квітень Травень Червень',
' Липень Серпень Вересень',
' Жовтень Листопад Грудень'
);
WEEKDAYNAME: Array [1...7] of String [2] = ('Пн', 'Вт', 'Ср', 'Чт', 'Пт', 'Сб',
'Вс');
Var
F: Text; {Файлова змінна}
Year: Word; {Рік календаря, що роздруковується}
Kvartal: Word; {Квартал року}
{Кількість днів у кожному місяці}
MonthsDays: Array [1...12] of Word;
{Кількість порожніх клітин на початку січня року Year}
Balnks: Word;
{Кількість порожніх клітин на початку кожного місяця}
BlanksDaus: Array [1...12] of Word;
idW: Word; {Номер дня тижня}
iMonth: Word; {Номер місяця на рік}
```

```
iKvartalMonth: Word; {Номер місяця у кварталі}
iCol: Word; {Номер колонки у місяці кварталу}
iCell: Word; {Номер клітини у місяці кварталу}
i: Integer;
Function Vys (Year: Word): Boolean;
{Функція повертає True у разі високосності року Year}
begin
Vys: = False;
if (((Year mod 4 = 0) and (Year mod 100<>0))
or (Year mod 1000 = 0))
then
Vys: = True;
end; {Vys}
begin { Основна програма }
ClrScr; {Очищення екрана}
Write ('Програма друку календаря');
WriteLn ('заданого року');
Write ('Вкажіть рік роздруковуваного');
WriteLn ('календаря після', YEARBASE: 5, 'року');
ReadLn (Year);
{Контроль запровадженого року}
if Year < YEARBASE then
begin
{Аварійне завершення програми}
Write ('Не можу скласти календар');
WriteLn (Year: 5, 'року');
Write ('Для завершення програми');
WriteLn ('натисніть будь-яку клавішу...');
repeat until KeyPressed;
Halt (1);
end;
WriteLn ('Чекаєте, йде друк...');
Assign (F, 'PRN');
Rewrite (F);
{Друк календаря на принтері}
{Частина прогалин у наступному рядку була вилучена!}
WriteLn (F, '', Year);
{Підготовка інформації}
{Визначення кількості порожніх клітин у січні року Year}
Blanks: = BLANKS1917;
i := YEARBASE;
while (I Year) do begin
{Збільшення Blanks}
Inc (Blanks); {У будь-який рік плюс 1}
if Vys (i)
then
Inc (Blanks); {Минулий рік високосний, +2}
{Коригування Blanks}
if (Blanks >= 7) then Blanks := Blanks - 7;
Inc (i); {Поточний рік}
end;
{Визначення кількості днів у кожному місяці}
for i := 1 to 12 do
MonthsDays [i]: = 31;
MonthsDays [4]: := 30;
MonthsDays [6]: = 30;
MonthsDays [9]: = 30;
MonthsDays [11]: = 30;
MonthsDays [2]: = 28;
if Vys (Year) then MonthsDays [2] : = 29;
{Визначення кількості порожніх клітин на початку кожного місяця}
BlanksDays [1] := Blanks;
for i : = 2 to 12 do
if BlanksDays [i - 1] + MonthsDays [i - 1] < 35
```

```
then
BlanksDays [i] := BlanksDays [i - 1] + MonthsDays [i - 1] - 28
else
BlanksDays [i]: = BlanksDays [i - 1] + MonthsDays [i - 1] - 35;
{Завдання номерів кварталів}
{Друк тіла календаря}
for Kvartal := 1 to 4 do begin
{Друк найменування кварталу}
WriteLn (F, KVARTALNAME [Kvartal]);
{Друк дат кварталу}
{Завдання номера дня тижня}
for iDW := 1 to 7 do
begin
{Друк найменування днів тижнів}
Write (f, WEEKDAYNAME [iDW];
{Друк трьох місяців дат кварталу}
for iKvartalMonth : = 1 to 3 do begin
{Розрахунок номер місяця у кварталі}
iMonth: = (Kvartal - 1) * 3 + iKvartalMonth;
{Друк шести колонок дат дня тижня кварталу}
for iCol := 1 to 6 do begin
iCell := (iCol - 1)*7 + iDW;
if ((iCell > BlanksDays [iMonth]) and (iCell <= BlanksDays [iMonth] + MonthsDays
[iMonth])))
then
Write (F, iCell - BlanksDays [iMonth] : 3)
else
Write (F, '');
end;
end;
{Друк назви днів тижнів}
WriteLn (F, WEEKDAYNAME [iDW];
end;
end;
Close (F);
Write ('Для завершення програми');
WriteLn ('натисніть будь-яку клавішу...');
Repeat until KeyPressed;
end.
```

ВИСНОВКИ

• З появою ЕОМ актуальним став пошук способів опису обчислювальних алгоритмів. У 60-х роках вже застосовувалися два способи опису алгоритмів: словесний покроковий та графічний у вигляді схем алгоритмів (жаргонно: блок-схем алгоритмів).

• Згідно з сучасними технологіями програмування, описи алгоритмів у словесно покроковій та графічній формах у вигляді схем алгоритмів практично не використовуються. Їх замінили самодокументовані тексти, що складаються із стандартних структур кодування.

• Хорошим функціональним описом є безпомилковий опис, однозначний для читача, короткий, суть якого розуміється швидко. Згідно з проектною процедурою, хороший функціональний опис складається від загального до приватного з використанням особливих конструкцій пропозицій типових елементів (типових структур або просто структур).

• Будь-які алгоритми або євроритми мають складатися лише зі стандартних структур. Кожна стандартна структура має один інформаційний вхід і один інформаційний вихід. Використання інших (нестандартних) структур призводить або до подовження опису, або до неможливості тестування (через неможливо великого обсягу необхідних тестів), або до втрати зрозумілості.

• При розробці євроритмів вхідна, проміжна та вихідна інформації зазвичай характеризуються найменуваннями предметів, їх станом, місцем розташування та часом.

• При розробці алгоритмів програм вхідна, проміжна та вихідна інформації характеризуються іменами та набором значень як простих, так і структурованих змінних (записів, масивів).
• Спочатку алгоритм або евроритм повинен представляти одну типову структуру СЛІД (одна дія зі змістом «виконати всі дії програми»). Далі до досягнення елементарних дій (елементарних операторів мови програмування або елементарних операцій) окремі структури Слідування декомпозуються з дотриманням принципу від загального до приватного однієї з трьох стандартних структур: Ланцюжок Слідкувань; Ланцюжок АЛЬТЕРНАТИВ; ПОВТОРЕННЯ.
• Перехід на нову мову програмування рекомендується розпочинати з розробки стандарту кодування типових обчислювальних структур.
• Алгоритм багато в чому визначається структурою даних.
• Тести – необхідний атрибут розробки алгоритму.
• Узагальнюючий тест або тести – мінімальний набір тестових даних, що охоплюють усі можливі випадки обчислень.
• Алгоритми зі старих книг краще розуміються після їхнього рефакторингу.

КОНТРОЛЬНІ ПИТАННЯ

1. Перерахуйте основні недоліки кожного із способів опису алгоритмів.
2. Що таке функціональний опис?
3. Навіщо призначена проектна процедура розробки функціональних описів?
4. Викладіть вимоги щодо способу мислення користувача проектною процедурою розробки функціональних описів.
5. У яких випадках програмісти можуть використовувати проектну процедуру?
6. У яких випадках програмісти не можуть застосовувати проектну процедуру?
7. Які основні характеристики структур СЛІД, АЛЬТЕРНАТИВ і ПОВТОРЕННЯ?
8. Яким є порядок основних кроків (етапів) виконання проектної процедури?
9. Наведіть приклад опису зовнішніх та внутрішніх даних програми.
10. Навіщо потрібна модель «чорної скриньки»?
11. Назвіть тести до програмування. Призначення. Зміст. Форми.
12. Назвіть ознаки структури Ланцюжок СЛІД.
13. Назвіть порядок деталізації одиночного СЛІДЖЕННЯ.
14. Назвіть порядок деталізації ланцюжка СЛІД.
15. Назвіть ознаки структур типу АЛЬТЕРНАТИВУ.
16. Назвіть порядок деталізації ланцюжка альтернатив.
17. Запишіть приклад кодування альтернатив з однією та двома діями.
18. Запишіть приклад кодування альтернатив з трьома та більше діями.
19. Назвіть ознаки структури ПОВТОРЕННЯ.
20. Назвіть порядок деталізації структур універсальних циклів ДО та ПОКИ.
21. Запишіть приклад кодування структури НЕУНІВЕРСАЛЬНИЙ ЦИКЛ-ДО.
22. Запишіть приклад кодування структур УНІВЕРСАЛЬНІ ЦИКЛИ ДО І ПОКИ.
23. Як робиться доказ коректності алгоритмів під час виконання проектної процедури.
24. Доповніть опис процесу кипіння води в чайнику наочними малюнками.

Тема 6 АРХІТЕКТУРА ПРОГРАМНИХ СИСТЕМ

6.1. ПОНЯТТЯ АРХІТЕКТУРИ ПРОГРАМНОЇ СИСТЕМИ

Розробка архітектури системи — це процес розбиття великої системи більш дрібні частини. Для позначення цих частин вигадано безліч назв: програми, компоненти, підсистеми...
Процес розробки архітектури — етап, необхідний під час проектування систем чи комплексів, але необов'язковий під час створення програми. Якщо зовнішні специфікації (екранні форми, організація файлів тощо. буд.) описують програмну систему з погляду користувача, наступний крок проектування полягає у розробці архітектури, а й слід проектування структури кожної програми.
Незважаючи на те, що немає точного визначення програмної системи, можна сказати, що вона є набором рішень безлічі різних, але пов'язаних між собою завдань, і далі покластися на інтуїцію у випадках, коли треба відрізнити програму від системи.
Приклади систем: ОС, СУБД, система продажу авіаквитків та ін.
Приклади програм: редактор текстів, компілятор, програми надсилання запитів від касира та ін.
Поняття архітектури програмної системи можна проілюструвати на прикладі. Нехай є на якомусь підприємстві якась САПР. Припустимо, що підприємство досить велике, і САПР буде цілим комплексом різних програмних продуктів, причому найчастіше різних виробників. Архітектурою цієї системи буде опис зв'язків цих програмних засобів в одне ціле. Очима програміста: САПР - комплекс комплексів програм.

6.2. СИСТЕМИ З ОКРЕМИХ ПРОГРАМ

Програмна система може складатися з окремих розроблених різними організаціями програм, що виконуються. Об'єднання функцій цих програм у цілу єдину програму може призвести до нестачі оперативної пам'яті машини, а сама технологія може бути економічно невиправданою.
Близький аналог цієї системи – система, керована командним файлом. Найпростіша архітектура такої системи реалізується послідовним викликом кожної програми. Програми обмінюються даними через файли, записані на диску, або через елементи даних, що знаходяться в оперативній пам'яті ЕОМ за відомими абсолютними адресами.
Вже наприкінці 1970-х років можна було швидко реалізувати дуже зручний введення даних у програму. Стосовно пізнішої операційної системи MS DOS достатньо було написати текстовий файл maket.txt з текстами пояснень суті даних та символом «?» позначити поля даних, що вводяться. Далі готувався командний файл із послідовністю команд:
1) видалення файлу work.txt з диска;
2) копіювання файлу maket.txt у файл work.txt;
3) запуск готової програми текстового редактора із параметром work.txt;

4) запуск програми користувача обробки даних (вхідна інформація програми – файл work.txt, вихідна інформація програми – файл result.txt);
5) запуск готової програми перегляду текстових файлів із параметром result.txt.
Після старту командного файлу користувач у вікні текстового редактора міг читати пояснення щодо введення інформації, знаходити поля введення даних пошуком підрядка із символом «?», вводити значення поля введення з коригуванням. Після введення та коригування даних користувач виходить із програми текстового редактора, що автоматично запускає програму обробки даних, а після завершення її роботи автоматично запускається програма перегляду текстових файлів, яка забезпечує користувачеві можливість перегляду результатів роботи програми обробки даних.
Якщо потрібно реалізувати меню вибору окремих програм, неможливо обійтися послідовним викликом команд командного файла. Для цього випадку можна написати програму, яка візуалізує меню на екрані та повертає номер обраного користувачем меню операційної системи. Повернути номер вибраної теми меню можна викликом підпрограми Halt (номер_мен) модуля DOS Turbo Pascal, де номер_мен — значення вибраного номера теми меню.
Далі, використовуючи команди DOS, організуйте виклик потрібних програм командним файлом:

```
IF ERRORLEVEL 2 GOTO ITEM2
IF ERRORLEVEL 1 GOTO ITEM1
. . . . . .

:ITEM1
. . . . . .
:ITEM2
. . . . . .
```

Використовуючи процедуру Halt для присвоєння системної змінної ERRORLEVEL необхідного значення в кожній із програм та командний файл з вибором потрібної програми, можна створити найпростіший механізм управління порядком виконання програм, коли кожна програма, що завершує роботу, визначає, яка програма повинна виконуватися наступною.
Замість стандартного монітора командних файлів для виклику у довільному порядку вже готових програм можна написати програму свого монітора на основі підпрограми виклику готових програм.
Хоча проектування систем з окремих програм виконується, принаймні, останні 30 років, але не вироблено жодної методики, крім особливо корисної поради: зобразіть функціональну схему процесу, а потім розбийте процес на програми.

6.3. СИСТЕМИ З ОКРЕМИХ РЕЗИДЕНТНИХ ПРОГРАМ

Резидентна програма – програма, яка постійно перебуває в оперативній пам'яті машини та не перешкоджає запуску нових програм. Після запуску резидентна програма стає частиною операційної системи MS DOS шляхом зміни значення межі пам'яті операційної системи, далі вона налаштовує якесь переривання на передачу управління в свою точку входу, а потім завершує роботу. Можна запустити ще кілька резидентних програм та звичайну програму. Після виконання заданих переривань MS DOS запускаються резидентні програми.
Кожна резидентна програма може бути завантажена в будь-якій послідовності. Резидентні програми можуть містити вектор переривань, які вказують на блоки даних кожної з програм. Ці блоки можуть містити ідентифікатор програми контролю наявності програми та дані міжпрограмного обміну. Непотрібні програми можуть бути видалені за допомогою спеціальних програм, так і за допомогою універсальної програми Release.

6.4. СИСТЕМИ З ПРОГРАМ, ОБМІНЮЮЧИХ ДАНИМИ ЧЕРЕЗ ПОРТИ

Такий обмін зазвичай реалізується при багатопроцесорній (багатомашинної) обробці. Порт кожної із програм представляє програму накопичення та верифікації як вхідних, так і

вихідних даних у відповідних чергах. У міру виконання поточної роботи з вхідного порту береться чергова порція інформації, обробляється, результати записуються у вихідний порт і далі програма приступає до обробки наступної роботи. Інші програми надсилають інформацію у вхідний порт і забирають результати роботи з вихідного порту.

6.5. ПІДХІД ДО ПРОЕКТУВАННЯ АРХІТЕКТУРИ СИСТЕМИ НА ОСНОВІ АБСТРАКТНИХ МАШИН ДЕЙКСТРИ

Найнижчий рівень абстракції – це рівень апаратури. Кожен рівень реалізує абстрактну машину з дедалі більшими можливостями.

1. Принцип 1. На кожному рівні абсолютно нічого не відомо про властивості вищих рівнів. Цим досягається скорочення зв'язків між рівнями.

Принцип 2 На кожному рівні нічого не відомо про внутрішню будову інших рівнів. Зв'язок рівнів здійснюється лише через певні заздалегідь сполучення.

Принцип 3 Кожен рівень є окремо відкомпільованими програмами. Деякі з цих модулів є внутрішніми для рівня, тобто недоступними іншим рівням. Імена інших модулів відомі на вищому рівні і являють собою сполучення з цим рівнем.

Принцип 4 Кожен рівень має певні ресурси і або приховує їх з інших рівнів, або надає іншим рівням деякі їх абстракції. Наприклад, у системі управління файлами один із рівнів може містити фізичні файли, приховуючи їх організацію від решти системи. Інші рівні можуть володіти ресурсами: у каталозі, у словнику даних та ін.

Принцип 5 Кожен рівень може забезпечувати деяку абстракцію даних у системі. Наприклад, файли послідовного та прямого доступу на одному рівні однаково реалізуються на іншому рівні.

Принцип 6 Припущення, які на кожному рівні робляться щодо інших рівнів, повинні бути мінімальними, ці припущення можуть набувати вигляду угод, які повинні дотримуватися перед виконанням функцій, або належати до подання даних або факторів зовнішнього середовища.

*Принцип*7. Зв'язки між рівнями обмежені явними аргументами, що передаються з одного рівня на інший. Неприпустиме спільне використання глобальних даних кількома рівнями. Більше того, бажано повністю виключити використання глобальних даних (навіть усередині рівня) у системі.

Принцип 8 Будь-яка функція, що виконується рівнем абстракції, має бути представлена єдиним входом. Аргументи, що пересилаються між рівнями, мають бути окремими елементами даних, а чи не складними структурами.

Підхід до проектування архітектури системи з урахуванням абстрактних машин Дейкстри можна пояснити на прикладі.

Процесор фірми «Intel» може виконувати операції арифметики і, здійснюючи порівняння двох величин, може виконувати команди переходу на команди в заданих адресах пам'яті. Програмувати таку ЕОМ можна як прямий запис двійкових команд.

ЕОМ IBM PC має спеціальний постійний пристрій з програмами BIOS. Після встановлення BIOS виходить машина з додатковими командами завантаження програми з дисків, читання інформації з будь-якого сектора дисків, читання символу з клавіатури, виведення інформації на екран і т.д. та відсутності співпроцесора.

Після встановлення операційної системи MS DOS на машину IBM PC виходить машина з новою підтримкою даних як файлів і з новими командами роботи з файлами і директоріями (копіювання, видалення тощо. буд.). Нова машина може виконувати операції над речовими числами з плаваючою точкою. З'являються команди запуску файлів та інші нові команди.

Після встановлення операційної системи MS Windows 3.1 (при встановленні операційної системи MS Windows 95 одночасно встановлюється і MS DOS) з'являються нові команди керування вікнами для роботи фахівців зі столами, завареними паперами. З'являється можливість одночасного запуску різних програм у різних вікнах із можливістю між віконного обміну інформацією.

Після інсталяції програмного комплексу Microsoft Office з'являються середовища та команди роботи над документами, розрахунковими таблицями тощо.

6.6. COM - ТЕХНОЛОГІЯ РОЗРОБКИ ПРОГРАМ, ЩО РОЗВИВАЮТЬСЯ І РОЗПОДОРОЧЕНИХ КОМПЛЕКСІВ

COM - Component Object Model (модель компонентних об'єктів) - це специфікація методу створення компонентів та побудови з них програм.

У літературних джерелах можна знайти безліч теорій та пропозицій щодо так званої технології еволюційного програмування. Однак до COM практично невідомі вдалі приклади розробки програм, що еволюціонують у часі. Це неможливістю однозначного передбачення людьми майбутнього. Тому поради на кшталт «передбач то в програмі для майбутнього розвитку» виявлялися безглуздими через те, що в ході супроводу з'ясовувалась потреба в якихось інших доробках, але не в апріорі закладених.

Традиційно програма проектувалася з окремих файлів, модулів чи класів, які компілювалися та компонувалися в єдине ціле.

Компоненти COM є виконуваний код, зазвичай поширюється як динамічно компонованих бібліотек (DLL). Компоненти COM підключаються один до одного динамічно.

Розробка програм із компонентів про додатків компонентної архітектури відбувається зовсім інакше. З появою COM єдиного цілого немає. Програми складаються з окремих компонентів. Компонента постачається користувачеві як двійковий код, скомпільований, скомпонований та готовий до використання. Доступ до цього коду здійснюється через документований точно інтерфейс. Під час виконання компоненти підключаються до інших компонентів, утворюючи програму.

COM - це інструмент розробки та розосереджених (багатомашинних) комплексів програм, заснована на моделі компонентних об'єктів.

Головне в COM - дотримання стандартних специфікацій інтерфейсу компонент. Якось прийнятий стандарт специфікацій інтерфейсу ніколи не змінюється. Однак після розробки нового стандарту нові компоненти самі будуть впізнавати старі та нові стандарти. Після заміни деякого числа компонентів програма раптом запрацює по-новому!

У міру розвитку програми компоненти, що становлять програму, можуть замінюватися новими. Програма більше не є статичною, приреченою застаріти ще до появи. Натомість вона поступово еволюціонує із заміною старих компонентів новими. З існуючих компонентів легко створити і абсолютно нові програми.

Користувачі часто хочуть адаптувати програми до своїх потреб. Кінцеві користувачі вважають за краще, щоб програма працювала так, як вони звикли. Програмістам у великих організаціях потрібні програми, що адаптуються, щоб створювати спеціалізовані рішення на основі готових продуктів. Компонентні архітектури добре пристосовані для адаптації, оскільки будь-яку компоненту можна замінити на іншу, більш відповідну потребам користувача. Припустимо, що ми маємо компоненти з урахуванням редакторів vi і Emacs (рис. 6.1). Користувач 1 може налаштувати програми використання vi, тоді як користувач 2 віддасть перевагу Emacs. Програми можна легко налаштовувати, додаючи нові компоненти або замінюючи наявні.

*Мал. 6.1.*Збираються під час виконання програми з окремих компонентів

Один із найперспективніших аспектів впровадження компонентної архітектури — швидка розробка програм. Ви зможете вибирати компоненти з бібліотеки та складати з них, як із деталей конструктора, цілісні додатки методом морфологічного синтезу! Практично всі програми Microsoft, які продаються сьогодні, використовують COM. Технологія ActiveX цієї фірми побудована на основі компонентів COM. Програмісти Visual Basic, C++, Delphi і Java можуть скористатися керуючими елементами ActiveX для прискорення розробки своїх додатків і сторінок Web. Звичайно, кожному додатку, як і раніше, будуть потрібні і деякі спеціалізовані компоненти, але для побудови простих додатків можна обійтися стандартними.

Створити зі звичайної програми розподілену програму легше, якщо ця проста програма складається з компонентів. По-перше, вона вже розділена на функціональні частини, які можуть розташовуватися далеко один від одного. По-друге, оскільки компоненти замінюються, замість деякої компоненти можна підставити іншу, єдиним завданням якої забезпечуватиме зв'язок з віддаленою компонентою.

Переваги використання компонентів безпосередньо випливають із здатності останніх підключатися до додатку і відключатися від нього. Для цього компоненти повинні відповідати двом вимогам. По-перше, вони мають компонуватися динамічно. По-друге, повинні приховувати (або інкапсулювати) деталі реалізації.

Щоб зрозуміти, як це з інкапсуляцією, необхідно визначити деякі терміни. Програма або компонент, що використовує іншу компоненту, називається клієнтом (client). Клієнт приєднується до компонента через інтерфейс (interface). Якщо компонент змінюється без зміни інтерфейсу, то змін у клієнті не потрібно. Аналогічно якщо клієнт змінюється без зміни інтерфейсу, немає необхідності змінювати компоненту. Однак якщо зміна або клієнта, або компоненти викликає зміну інтерфейсу, то й інший бік інтерфейсу також необхідно змінити.

Таким чином, для того, щоб скористатися перевагами динамічного компонування, компоненти та клієнти повинні намагатися не змінювати свої інтерфейси та бути інкапсулюючими. Деталі реалізації клієнта або компоненти не повинні відображатись в інтерфейсі. Чим надійніше інтерфейс ізольований від реалізації, тим менш імовірно, що він зміниться під час модифікації клієнта чи компоненти. Якщо інтерфейс не змінюється, зміна компоненти надає лише незначне впливом геть додаток загалом.

Необхідність ізоляції клієнта від деталей реалізації накладає на компоненти низку важливих обмежень, список яких наведено нижче.

Обмеження 1.Компонента повинна приховувати мову програмування, що використовується. Компоненти можуть бути розроблені за допомогою практично будь-якої процедурної мови, включаючи Ada, C, Java, Modula-3, Oberon та Pascal. Будь-яку мову, у тому числі Smalltalk та Visual Basic, можна пристосувати до використання компонентів COM. Будь-який клієнт повинен мати можливість використовувати компоненту незалежно від мов програмування, на яких написано ту чи іншу.

Обмеження 2.Компоненти повинні поширюватись у двійковій формі. Дійсно, оскільки вони повинні приховувати мову реалізації, їх необхідно постачати вже відкомпільованими та готовими до використання (DLL).

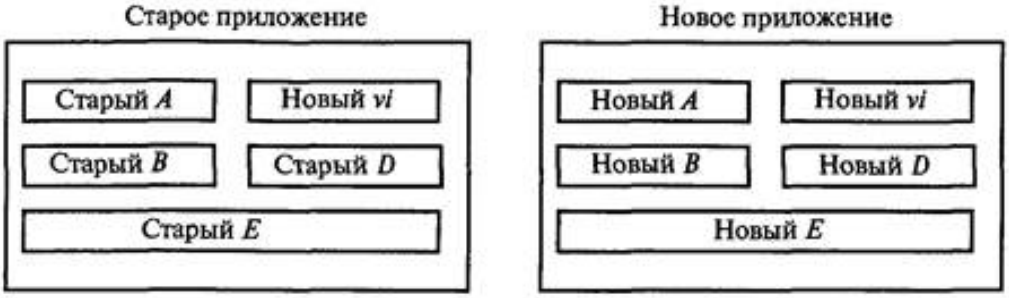
Обмеження 3.Компоненти COM можна модернізувати, не порушуючи роботи старих клієнтів. COM надає стандартний спосіб реалізації різних версій компонентів. Нові версії компонентів повинні працювати як з новими клієнтами, так і старими.

Обмеження 4.Компоненти COM є переміщуються по мережі, причому переміщення по мережі має бути прозорою. Компонент на віддаленій системі розглядається так само, як компонент на локальному комп'ютері. Необхідно, щоб компонент і програма, що використовує її, могли виконуватися всередині одного процесу, в різних процесах або на різних машинах. Клієнт повинен розглядати віддалену компоненту так само, як локальну. Якби з віддаленими компонентами треба було б працювати інакше, ніж з локальними, то

потрібно було б перекомпіляція клієнта щоразу, коли локальна компонента переміщається в інше місце мережі.

Користувач може мати дві клієнтські програми, які використовують одну й ту саму компоненту. Припустимо, що один додаток застосовує нову версію цієї компоненти, а інший стару. Інсталяція нової версії не повинна порушувати роботу програми, яка використовувала стару версію. Стара програма використовує нову компоненту vi абсолютно так само, як це робить нове (рис. 6.2).

Проте зворотна сумісність має обмежувати розвиток компонент. Потрібно, щоб поведінка компоненти нових додатків можна було радикально змінювати, не порушуючи підтримку старих додатків.



*Мал. 6.2.*Поетапне отримання нових додатків

Таким чином, технологія передбачає взаємозамінність компонентів під час виконання за допомогою встановлення стандарту, якому повинні дотримуватися компоненти; практично прозорої підтримки кількох версій компоненти; забезпечення можливості роботи із подібними компонентами однаковим способом; визначення архітектури, незалежної від мови; підтримки прозорих зв'язків із віддаленими компонентами.

ВИСНОВКИ

• Розробка архітектури - це процес розбиття великої системи на дрібніші частини. Процес розробки архітектури — етап, необхідний під час проектування систем чи комплексів, але необов'язковий під час створення програми. Якщо зовнішні специфікації (екранні форми, організація файлів...) описують програмну систему з погляду користувача, наступний крок проектування полягає у розробці архітектури, а потім слідує проектування структури кожної програми.

• Програмна система може складатися з окремих, розроблених різними організаціями програм, що виконуються; із програм, що обмінюються даними через порти; з окремих резидентних програм.

• Традиційно програма проектувалась з окремих файлів, модулів або класів, які компілювалися та компонувалися в єдине ціле.

• Розробка програм з компонентів - так званих додатків компонентної архітектури - відбувається зовсім інакше. З появою технології розробки та розосереджених (багатомашинних) комплексів програм, заснованої на моделі компонентних об'єктів (COM), єдиного цілого більше немає: програми складаються з окремих компонентів. Компонента постачається користувачеві як двійковий код, скомпільований, скомпонований та готовий до використання. Доступ до цього коду здійснюється через документований інтерфейс. Під час виконання компоненти підключаються до інших компонентів, утворюючи програму.

Контрольні питання

1. Що таке архітектура програм?
2. Чи є синонімами поняття «структура» та «архітектура»?
3. У чому полягає процес розробки архітектури програми?

4. Як реалізується архітектура системи окремих програм?
5. Що таке резидентна програма?
6. Як здійснюється обмін даними через порти?
7. Перерахуйте принципи підходу до проектування архітектури системи з позиції рівнів абстракції Дейкстри.
8. Чому зі звичайної програми створити розподілену програму легше, якщо вона складається із компонентів?
9. Перерахуйте низку обмежень, що накладаються на компоненти.
10. За допомогою чого передбачається взаємозамінність компонентів?

Тема 7 ТЕХНОЛОГІЯ СТРУКТУРНОГО ПРОГРАМУВАННЯ

7.1. ПОНЯТТЯ СТРУКТУРИ ПРОГРАМИ

Структура програми — штучно виділені програмістом частини програми, що взаємодіють. Використання раціональної структури усуває проблему складності розробки; робить програму зрозумілою людям; підвищує надійність роботи програми при скороченні терміну її тестування та термінів розробки загалом.

Часто деяку послідовність інструкцій потрібно повторити у кількох місцях програми. Щоб програмісту не доводилося витрачати час і зусилля копіювання цих інструкцій, у більшості мов програмування передбачаються кошти на організацію підпрограм. Таким чином, програміст отримує можливість присвоїти послідовності інструкцій довільне ім'я та використовувати це ім'я як скорочений запис у тих місцях, де зустрічається відповідна послідовність інструкцій. Підпрограма — деяка послідовність інструкцій, яка може бути викликана в кількох місцях програми.

Опис підпрограми (функції чи процедури) складається з двох частин: заголовка та тіла. Заголовок містить ідентифікатор підпрограми. Тіло складається з однієї чи кількох інструкцій. Ідентифікатор підпрограми використовується як скорочений запис у тих місцях програми, де зустрічається відповідна послідовність інструкцій.

Навряд чи варто було докладно говорити про таку просту форму запису, якби за нею не ховалися важливі та основні поняття. Насправді процедури та функції, які називаються підпрограмами, є одним з тих небагатьох фундаментальних інструментів у мистецтві програмування, які надають вирішальний вплив на стиль та якість роботи програміста. Процедура — це спосіб скорочення програмного тексту, а й, що важливіше, засіб розкладання програми на логічно пов'язані, замкнуті елементи, що визначають її структуру. Розкладання на частини суттєво для розуміння програми, особливо якщо програма складна і важко доступна для огляду через велику довжину тексту. Розкладання на підпрограми необхідне як документування, так верифікації програми. Тому бажано оформляти послідовність інструкцій як підпрограми, навіть якщо підпрограма використовується одноразово і, отже, відсутня мотив, що з скороченням тексту програми.

Додаткова інформація про змінні (які передаються та використовуються у процедурі) або про умови, яким мають задовольняти аргументи, задається у заголовку процедури. Про корисність процедури, зокрема про її роль під час структуризації програми, незаперечно свідчать ще два поняття у програмуванні. Деякі змінні (їх зазвичай називають допоміжними або локальними змінними), які використовуються всередині процедури, не мають сенсу за її межами. У програмі значно простіше розібратися, якщо очевидно вказані області дії таких змінних. Процедура постає як природна текстова одиниця, з допомогою якої обмежується сфера існування про локальних змінних.

Ймовірно, найбільш загальна тактика програмування полягає у розкладанні процесу на окремі дії: функціонального опису на підфункції, а відповідних програм – на окремі інструкції. На кожному такому кроці декомпозиції слід переконатися, що розв'язання приватних завдань призводять до вирішення спільного завдання; обрана послідовність окремих дій розумна; обрана декомпозиція дозволяє отримати інструкції, в якомусь сенсі ближчі до мови, якою буде реалізована програма.

Остання вимога виключає можливість прямолінійного просування від початкової постановки завдання до кінцевої програми, яка має вийти зрештою. Кожен етап декомпозиції супроводжується формулюванням приватних підпрограм. У процесі роботи може виявитися, що обрана декомпозиція невдала у тому сенсі хоча б оскільки підпрограми незручно висловлювати з допомогою наявних коштів. У цьому випадку один або кілька попередніх кроків декомпозиції слід переглянути заново.
Якщо бачити в поетапній декомпозиції та одночасному розвитку та деталізації програми поступове просування вглиб, то такий метод при вирішенні завдань можна охарактеризувати як низхідний (згори донизу). І навпаки, можливий такий підхід до вирішення задачі, коли програміст спочатку вивчає обчислювальну машину та/або мову програмування, що є в його розпорядженні, а потім збирає деякі послідовності інструкцій в елементарні процедури, типові для розв'язуваного завдання. Елементарні процедури використовуються на наступному рівні ієрархії процедур. Такий метод переходу від примітивних машинних команд до необхідної реалізації програми називається висхідним (знизу догори).
На практиці розробку програми ніколи не вдається провести строго в одному напрямку (згори донизу або знизу догори). Проте за конструюванні нових алгоритмів низхідний метод зазвичай домінує. З іншого боку, при адаптації програми до дещо змінених вимог перевага найчастіше надається висхідному методу.
Обидва методи дозволяють розробляти програми, яким властива структура — властивість, що відрізняє їхню відмінність від аморфних лінійних послідовностей інструкцій чи команд машини. І надзвичайно важливо, щоб використовувана мова повною мірою відображала цю структуру. Тільки тоді остаточний вид одержаної програми дозволить застосувати систематичні методи верифікації.

7.2. МОДУЛЬ ТА ОСНОВНІ ПРИНЦИПИ СТРУКТУРНОГО ПІДХОДУ

7.2.1. Поняття модуля

Якщо програма розбивається на підпрограми, то представлення результатів і аргументів часто доводиться вводити нові змінні і таким чином встановлювати зв'язок між підпрограмами. Такі змінні слід вводити та описувати на тому етапі розробки, на якому вони були потрібні. Більш того, деталізація опису процесу може супроводжуватися деталізацією опису структури змінних, що використовуються. Отже, в мові мають бути засоби для відображення ієрархічної структури даних. Зі сказаного видно, яку важливу роль грає при покроковій розробці програми поняття процедури, локальності процедур і даних, структурування даних.
Проектування починається із фіксації зовнішніх специфікацій. З зовнішніх специфікацій складається опис внутрішнього алгоритму програми, обов'язково з структурою внутрішніх даних. Далі великі функції розбиваються на підфункції до досягнення підфункції розміру модуля - підпрограми (процедури чи функції) мови програмування, яких пред'являються особливі додаткові вимоги.
Модуль- фундаментальне поняття та функціональний елемент технології структурного програмування.
Модуль -це підпрограма, але оформлена відповідно до особливих правил.
*Правило 1.*Модуль повинен мати один вхід і один вихід і виконувати строго однозначну функцію, яка описується простою поширеною пропозицією природної (російської) мови або навіть пропозицією без присудка.
*Правило 2*Модуль повинен забезпечувати компіляцію, незалежну від інших модулів, із «забуттям» усіх внутрішніх позначень модулів.
*Правило 3*Модуль може викликати інші модулі за їхніми іменами.
*Правило 4*Хороший модуль не використовує глобальні змінні спілкування з іншим модулем, оскільки потім важко знайти модуль, який псує дані. Якщо все ж таки використовуються

глобальні змінні, то потрібно чітко коментувати ті модулі, які тільки читають, і ті модулі, які можуть змінювати дані.

*Правило 5*Модуль кодується лише стандартними структурами та ретельно коментується.

У поняття структури програми включаються склад та опис зв'язків всіх модулів, які реалізують самостійні функції програми та опис носіїв даних, що беруть участь в обміні як між окремими підпрограмами, так і введені та виведені з/на зовнішніх пристроїв.

У разі складної великої програми необхідно опанувати спеціальні прийоми отримання раціональної структури програми. Раціональна структура програми забезпечує майже дворазове скорочення обсягу програмування та багаторазове скорочення обсягів та термінів тестування, а отже, принципово знижує витрати на розробку.

Підпорядкованість модулів зручно зображати схемою ієрархії. Схема ієрархії відбиває лише підпорядкованість підпрограм, але з порядок їх виклику чи функціонування програми. Схема ієрархії може мати вигляд, показаний на рис. 7.1.

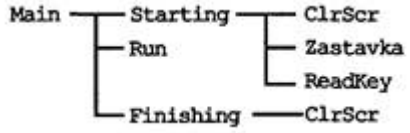


*Рис. 7.1.*Схема ієрархії простої програми

Така схема ієрархії зазвичай доповнюється розшифровкою функцій, що виконуються модулями.

До складання схеми ієрархії доцільно скласти зовнішні специфікації програми та скласти функціональні описи програми разом з описом змінних носіїв даних. Особливу увагу треба приділити ієрархії типів структурованих даних та їхнього коментування. Декомпозиція програми на підпрограми провадиться за принципом від загального до приватного, більш детального. Взагалі процес складання функціонального опису та складання схеми ієрархії є ітераційним. Вибір найкращого варіанта є багатокритеріальним.

Розчленування повинне забезпечувати зручний порядок введення частин в експлуатацію.

Реалізація програми (кодування, складання, тестування) має вестися за розробленим заздалегідь планом і починатися з верхніх модулів схеми ієрархії. Відсутні модулі нижніх рівнів замінюються заглушками, які є найпростішими підпрограмами: або без дій; або вхідні дані, що виводять у файл налагодження; або тестові дані, що повертають у вищестоящі модулі (які зазвичай присвоюються всередині заглушки); або містять комбінацію цих процесів.

Структурний підхід до програмування є як методологією, і технологію створення програм. Свого часу його використання забезпечило підвищення продуктивності праці програмістів під час написання та налагодження програм; отримання програм, які складаються з модулів та придатні для супроводу; створення програм колективом розробників; закінчення створення програми у заданий термін.

Структурний підхід до програмування сприйняв і використовує багато методів в галузі проектування складних технічних систем. Серед них блочно-ієрархічний підхід до проектування складних систем, стадійність створення програм, низхідне проектування, методи оцінки та планування.

Структурний підхід рекомендує дотримуватись наступних принципів при створенні програмного виробу:

- Модульність програм;
- Структурне кодування модулів програм;
- низхідне проектування раціональної ієрархії модулів програм;
- низхідна реалізація програми з використанням заглушок;
- Здійснення планування на всіх стадіях проекту;

- Наскрізний структурний контроль програмних комплексів в цілому і складових їх модулів.
Модульність програм характеризується тим, що програма складається з модулів. Деякі смислові групи модулів зосереджуються на окремих файлах. Наприклад, в окремих файлах (Unit) можуть бути зосереджені модулі текстового редактора та модулі ієрархічного меню.
Структурне кодування модулів програм полягає у особливому оформленні їх текстів. У модуля має бути легко помітний заголовок з коментарем, що пояснює функціональне призначення модуля. Імена змінних мають бути мнемонічними. Суть змінних та порядок розміщення в них інформації мають бути пояснені коментарями, а код модуля закодований з використанням типових алгоритмічних структур із використанням відступів.
Низхідне проектування раціональної ієрархії модулів програм полягає у виділенні спочатку модулів найвищого рівня ієрархії, а потім підлеглих модулів.
Низхідна реалізація програми полягає у первинній реалізації групи модулів верхніх рівнів, які називаються ядром програми, і далі поступово, відповідно до плану, реалізуються модулі нижніх рівнів. Необхідні для лінкування програми, відсутні модулі імітуються заглушками.
Здійснення планування усім стадіях проекту дозволяє спочатку спланувати як склад стадій, і тривалість всіх етапів робіт. Таке планування дозволяє завершити розробку в заданий термін за заданих витрат на розробку. Далі плануються порядок і час інтеграції модулів у ядро, що все розширюється. Плануються заходи щодо тестування програми від ранніх до заключних етапів.
Наскрізний структурний контроль полягає у дотриманні наперед наміченого плану тестування, який охоплює період від розробки зовнішніх специфікацій, далі внутрішніх специфікацій та їх коригування в періоді реалізації аж до приймально-здавальних випробувань. Модулі, що складають програму, тестуються як під час написання їх коду, так і при автономному тестуванні, інспекції їх вихідного коду, при тестуванні відразу після підключення до ядра.

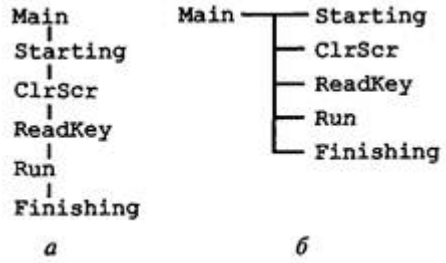
7.2.2. Поняття заглушки модуля

При структурному програмуванні програма переважно реалізується (збирається і тестується) зверху донизу. Спочатку із 20—30 модулів пишеться ядро. Щоб почати тестувати, модулі нижніх рівнів замінюються заглушками. Після закінчення тестування ядра, заглушки замінюються новими готовими модулями, але якщо програма ще не закінчена, то для успішного її лінкування знадобляться все нові заглушки модулів, що відсутні. Тепер можна приступати до тестування зібраної частини тощо.
Заглушка- Це макет модуля. Найпростіша заглушка – це підпрограма чи функція без дій. Більш складна заглушка може виводити повідомлення про те, що відпрацював такий модуль. Ще більш складні заглушки можуть виводити вхідну інформацію в файл налагодження. Нарешті, ще складніші заглушки видають на вихід тестову інформацію, необхідну перевірки вже реалізованих модулів.
Написання заглушок — «зайва» робота, але потрібне мистецтво проектувальника, щоб максимальна кількість заглушок була простою, а тестування вже зібраної частини програми було б повним.

7.2.3. Засоби зміни топології ієрархії програми

У процесі низхідного проектування раціональної ієрархії модулів програми необхідно отримати оптимальну підпорядкованість.
Схемі ієрархії можна надати будь-який топологічний малюнок. Так, схемою ієрархії, зображеної на рис. 7.2, а можна надати вигляд, зображений на рис. 7.2, б.
Фрагменти з вертикальними викликами можуть бути перетворені на виклики з одного рівня за допомогою введення додаткового модуля, який може не виконувати жодних корисних

функцій з точки зору алгоритму програми. Функція нового модуля може перебувати лише моніторингу, т. е. виклику інших модулів у порядку.



Мал. 7.2. Схема ієрархії однієї й тієї програми: а — з вертикальними викликами; б - з горизонтальними викликами

Фрагменти з горизонтальними викликами на одному рівні можуть бути перетворені на вертикальні виклики модулів різних рівнів за допомогою введення додаткових змінних. Ці змінні не можуть бути отримані декомпозицією функціонального опису на підфункції. Ці додаткові змінні мають тип цілий чи логічний і називаються прапорами (семафорами, ключами) подій. Їх зміст зазвичай характеризується фразою: «Залежно від передісторії дій, виконати такі дії».
У процесі проектування необхідно створити кілька проектних ітерацій, генеруючи щоразу нову схему ієрархії, і порівняти ці ієрархії за критеріями оцінки якості схеми ієрархії.

7.2.4. Критерії оцінки якості схеми ієрархії

Проектування структури програми передує розробка зовнішніх функціональних описів. p align="justify"> Функціональні описи (алгоритми виконання програми) для досягнення їх сприйняття повинні бути декомпозіровані від загального до приватного. Також вони повинні включати описи форм подання та обсягу внутрішніх даних.
Отже, спочатку є перший варіант схеми ієрархії, отриманий шляхом простого членування функцій програми на підфункції із зазначенням змінних, необхідних розміщення даних різних кроках обробки. Скоріше цей варіант не є оптимальним і потрібні проектні ітерації (зазвичай виконуються методом «проб і помилок») для покращення топології схеми.
Кожен новий варіант порівнюється з попереднім варіантом за описаними тут критеріями.
Генерація варіантів припиняється за неможливості подальших поліпшень.
Фонд критеріїв оптимальності схем ієрархії є необхідною підмогою при оптимізації схем ієрархії та складається з 13 критеріїв.
Перший- Повнота виконання специфікованих функцій.
Другий -можливість швидкого та дешевого поповнення новими, раніше не специфікованими функціями.
Третій, що випливає з блочно-ієрархічного підходу, — доступність для проектувальника складових частин програми.
Четвертий критерій оцінки якості структури - максимальна незалежність за даними окремих частин програми.
П'ятий можливість пов'язування програми з великою багаторівневою схемою ієрархії конкретним редактором зв'язків (лінковщиком). Якщо починаєте працювати над новою програмою, то дуже корисно виконати на ЕОМ її модель у вигляді порожніх заглушок модулів, які не містять жодних дій.
Шостий- Достатність оперативної пам'яті. Тут розглядаються варіанти з описом особливо структурованих статичних і динамічних змінних різних рівнях схеми ієрархії. Перевірка задоволення цього критерію здійснюється розрахунками з деякими машинними експериментами.

Сьомий- Оцінка впливу топології схеми ієрархії на швидкість виконання програми при використанні оверлеїв (динамічного завантаження програми) і механізму підкачування сторінок при розробці програми, яка цілком не може бути розміщена в оперативній пам'яті.
Восьмий -відсутність різних модулів, що виконують подібні дії. В ідеалі — той самий модуль викликається різних рівнях схеми ієрархії.
Дев'ятий - Досягнення при реалізації програми такого мережевого графіка роботи колективу програмістів, який забезпечує рівномірне завантаження колективу за ключовими датами проекту.
Десятий усіляке скорочення витрат на тестування вже зібраного ядра програми за ключовими датами мережевого графіка реалізації. Характеризується простотою заглушок і якістю тестування по всіх обчислювальних маршрутах модулів. Досягається первинної реалізацією зверху донизу модулів введення та виведення програми з відстрочкою реалізації інших гілок схеми ієрархії. Зазвичай витрати на тестування як за термінами, так і грошима становлять близько 60% вартості всього проекту. Хороша схема ієрархії скорочує витрати на тестування проти початковим варіантом в 2—5 разів і більше.
Одинадцятий- використання в даному проекті якомога більшої кількості розроблених у попередніх проектах модулів і бібліотек при мінімальному обсязі частин, що виготовляються заново.
Дванадцятий— вдалий розподіл модулів за файлами програми та бібліотекам, що компілюються.
Тринадцятий -накопичення вже готових модулів та бібліотек модулів для використання їх у нових розробках.
Отже, хороша структура програми забезпечує скорочення загального обсягу текстів у 2 чи 3 рази, що відповідно здешевлює проект; на кілька порядків здешевлює тестування (на тестування зазвичай припадає щонайменше 60 % від загальних витрат проекту), і навіть полегшує і здешевлює супровід програми.

7.2.5. Рекомендації щодо організації процесу розробки схеми ієрархії

Як правило, складання зовнішніх, потім внутрішніх функціональних описів і надалі структури програми здійснює група від двох до семи кваліфікованих програмістів - системних аналітиків.
Окремі варіанти структури програми розробляються до можливості їх порівняння. При цьому використовуються такі документи:
• опис алгоритму (функціонування) програми чи методів розв'язання задачі разом із описом даних;
• схема ієрархії модулів програми з розшифровкою позначень та функцій модулів;
• паспорти модулів.
На підставі цих документів готуються опис алгоритму програми з урахуванням модульного поділу та мережевий графік реалізації та тестування програми з тестами, складеними до програмування.
Паспорт модуля— внутрішній документ проекту, що є конвертом з ім'ям модуля. Усередині конверта містяться описи: прототипи виклику самого модуля та модулів, що викликаються модулем даних; розшифровки вхідних та вихідних змінних модуля; опис функції, що виконується модулем; принципів реалізації алгоритму модуля з описом основних структур даних У паспорті модуля можуть бути копії літературних джерел з описами основних ідей алгоритму. У процесі виконання проекту паспорт модуля коригується до технічного завдання розробці цього модуля, а після реалізації — до опису модуля.
Група системних аналітиків перевіряє загальну однозначність цих описів і генерує нові варіанти схем ієрархії. При реалізації ця група тестує вже зібране ядро програми з усіх нових ключових дат мережного графіка проекту. У процесі тестування ядра програми з допомогою заглушок уточнюються діапазони даних. У разі потреби системні аналітики коригують

паспорти модулів перед програмуванням модулів за результатами уточнення діапазонів даних.

Найголовніше у схемі ієрархії — мінімізація зусиль зі збирання та тестування програми. При використанні заглушок можна добре тестувати сполучення модулів, але не самі модулі. Тестування самих модулів вимагатиме витончених складних заглушок та астрономічної кількості тестів. Вихід - до інтеграції модулів тестувати модулі з використанням провідних програм. Також рекомендується здійснювати реалізацію з деяким порушенням принципу «зверху-вниз». Рекомендується спочатку з дотриманням принципу «зверху-вниз» реалізувати галузь схеми ієрархії, що відповідає за введення інформації з перевіркою її коректності, заглушивши гілки розрахунків та виведення на найвищому рівні. Далі реалізується галузь виведення інформації й у останню чергу — галузь розрахунків (функціонування програми). Якщо функцій програми багато, можна спочатку реалізувати модулі вибору функцій, заглушивши модулі самих функцій, і далі реалізовувати гілка кожної функції послідовно з дотриманням принципу «зверху—вниз».

Схема ієрархії повинна включати максимальну кількість модулів інших розробок. Багато модулів можна використовувати в інших розробках, однак це не стосується обчислювальних модулів, для яких через похибку рахунку можуть не підійти діапазони даних.

Тут дуже важливим є складання зручного графіка роботи з урахуванням планування загальної кількості програм кодувальників та їх рівномірного завантаження за термінами проекту, а також закінчення проекту у призначений термін.

Схема ієрархії повинна відбиватися на файли з вихідними текстами програм таким чином, щоб кожен файл містив якомога більше готових функцій із загальним призначенням. Це бажано для полегшення їх використання у подальших розробках.

Таким чином, окрім отримання грошей від замовника за розробку, програміст зобов'язаний підвищувати свій інтелектуальний капітал також за гроші замовника.

Існує дуже багато автоматизованих систем формування декомпозиції схем ієрархії, наприклад HIPO, SADT, R-TRAN.

7.3. РЕТРОСПЕКТИВНЕ ПРОЕКТУВАННЯ ДЕМОНСТРАЦІЙНОЇ ПРОГРАМИ MCALC ФІРМИ «BORLAND INC.»

Згідно з ретроспективно проведеним системним аналізом (див. тему 2), фірма «Borland Inc.» ухвалила рішення про реалізацію демонстраційного прикладу програми електронної таблиці. Цілком можливо згенерувати безліч варіантів реалізації електронної таблиці, починаючи від варіанта з усіма клітинами в одному вікні та закінчуючи, наприклад, варіантом Excel. Проте фірма Borland Inc. обрала варіант із прокручуванням інформації клітин у вікні, зміною адрес клітин при вставках рядків і стовпців, а також при їх видаленні. У проект запроваджено вимоги розробки некомерційного виробу. Розмір таблиці обмежений 100-100 клітинами. У програмі відсутня функція копіювання клітин. Вибрана складність варіанта, що реалізується, відповідає багато файловому проекту. Програма має функції підтримки виведення на екран, введення з клавіатури; у ній реалізований інтерпретатор формул із математичними функціями; для збереження інформації таблиці використовується файл складної організації, розглянутий гол. 3. Все це дає змогу продемонструвати можливості компілятора.

Програма Mcalc 1985-1988 гг. (Turbo Pascal 5.0) складається з наступних файлів:

• mcalc.pas - файл основної програми;
• mcvars.pas - файл глобальних описів;
• mcdisply.pas - файл підпрограм роботи з дисплеєм;
• mcmvsmem.asm - асемблерний файл підпрограм запам'ятовування в оперативній пам'яті інформації екрана, а також відновлення раніше збереженої інформації екрана;
• mcinput.pas - файл підпрограм введення даних з клавіатури;
• mcommand.pas — файл підпрограм, які обслуговують систему меню та дій, вибраних за допомогою меню;
• mcutil.pas - файл допоміжних підпрограм;

• mcparser.pas – файл інтерпретатора арифметичних виразів формул клітин.

Всі файли закодовані з дотриманням стандартів оформлення, що розвиваються. Так, у файлах mcdisply.pas, mcinput.pas описи прототипів підпрограм виконані з використанням більш раннього синтаксису мови програмування, що говорить про їхнє запозичення із програм, написаних раніше; при цьому можна виявити їхнє невелике модифікування (рефакторинг).

Хоча фірма Borland Inc. займається розробкою компіляторів, файл mcparser.pas також є запозиченим з UNIX YACC utility та лише частково модифікованим. Інші файли є оригінальними.

Асемблер файл mcmvsmem.asm є штучно доданим. Мета його додавання – демонстрація можливості використання асемблерних вставок. Алгорити, що містяться в ньому, цілком можна було б реалізувати мовою Pascal. Більше того, можна було б взагалі обійтися без реалізованих у ньому підпрограм, щоправда, було б видно деякі затримки виведення інформації на екран.

З метою здійснення нової проектної ітерації, що покращує проект, отримаємо з існуючого проекту проектну документацію, що складається з опису структури даних програми; функціонального опису основного ядра програми; схеми ієрархії модулів основного ядра програми; специфікації призначення модулів основного ядра програми.

Розглянемо організацію файлу mcvars.pas, що містить в основному опис структури внутрішніх даних програми. Файл містить описи в розділі interface. Секція впровадження порожня.

На початку файлу міститься код, який в залежності від наявності співпроцесора транслюється в одному з двох варіантів:

```
{$IFOPT N+}
{Є вбудований співпроцесор}
type
Real = Extended; {Заміна типу Real на Extended}
const
EXPLIMIT = 11356; {Гранове значення аргументу експоненти}
SQRLIMIT = 1E2466; {Гранічне значення аргументу SQRT}
....
{$ELSE}
const
{Тип Real не перевизначено}
EXPLIMIT = 88; {Гранове значення аргументу експоненти}
SQRLIMIT = 1E18; {Граничне значення аргументу SQRT}
....
{$ENDIF}
```

Описи констант містять такі блоки:

- Блок малих констант, що містять інформацію всіх виведених на екран і файли текстових написів (для русифікації всієї програми потрібно змінити тільки цю інформацію);
- блок парних рядків текстів меню та «гарячих» клавіш вибору тем меню;
- блок опису найважливіших констант, що визначають розмірність таблиці та розташування інформації на екрані

```
MAXCOLS = 100; { Maximum is 702 } {Розмір таблиці}
MAXROWS = 100;
MINCOLWIDTH = 3; {Мінімальна ширина стовпця}
MAXCOLWIDTH = 77; {Максимальна ширина стовпця}
....
```

- блок опису кольорів усіх полів екрану, модифікація констант якого дозволяє оперативно змінювати кольори;

— основні константи, мнемоніка імен яких полегшує сприйняття текстів програми

```
HIGHLIGHT = True; {Підсвічена поточна клітина}
NOHIGHLIGHT = False; {Непідсвічена клітина}
{Атрибут вмісту клітини}
TXT = 0;
VALUE = 1;
```

```
FORMULA = 2;
....
{Дозволені літери}
LETTERS: set of Char = ['A'..'Z', 'a'..'z'];
```

— коди клавіш клавіатури.

Слід зазначити, що наведені навіть коди клавіатури, що не використовуються в програмі керуючих клавіш. Це відповідає факту копіювання даних кодів із коду якоїсь іншої розробки.

Далі йдуть описи типу інформації вмісту табличної клітини та типу покажчика на клітину:

```
type
CellRec = record
Error: Boolean;
case Attrib : Byte of
TXT: (T: IString);
VALUE: (Value: Real);
FORMULA: (Fvalue: Real;
Formula: IString);
end;
CellPtr = ^CellRec; {Покажчик на клітку}
```

Даний тип організований так, що клітина завжди може містити ознаку помилки розрахунків Error та розміщувати три варіанти інформації: текст, значення та формулу.

Далі описано основні глобальні змінні. Описи починаються з визначення двовимірного масиву Cell покажчика, що постійно знаходиться в пам'яті, на клітини таблиці. Це дозволяє не витрачати пам'ять на порожні клітини. Пам'ять під інформацію клітини виділяється динамічно в кількості, що суворо відповідає інформації клітини. Без використання пам'яті, що динамічно виділяється, було б неможливо розмістити інформацію клітин таблиці в 640К пам'яті машин того часу.

```
MAXCOLS*MAXROWS*[SizeOf(Istring)+SizeOf(Extened)] = 100*100*[80+10] = 900K
```

Далі йдуть опис змінної, яка є покажчиком на поточну клітину таблиці, опис масиву форматів клітин та змінних позиціонування інформації на екрані.

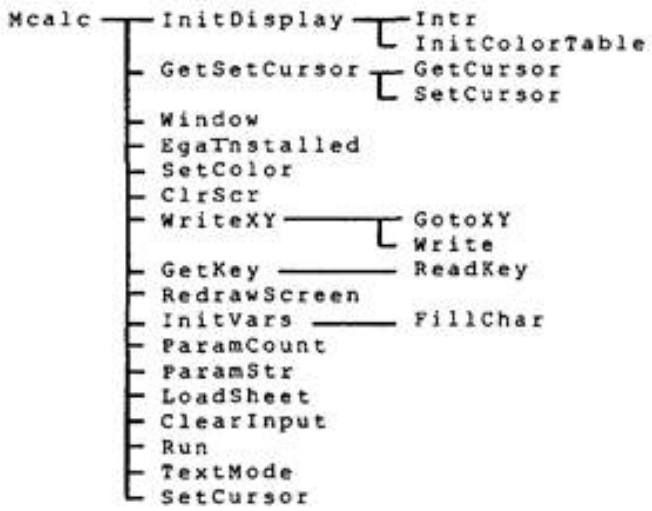
```
Cell : array [1..MAXCOLS, 1..MAXROWS] of CellPtr;
CurCell: CellPtr; {Покажчик на поточну клітинку}
Формат: array [1..MAXCOLS, 1..MAXROWS] of Byte;
LeftCol, RightCol, TopRow, BottomRow,
{Позиціонування}
CurCol, CurRow, LastCol, LastRow: Word;
....
```

Слід зазначити, що виділення окремого масиву форматів інформації клітин не є виправданим. Найпрактичніше ввести байт інформації формату клітини в тип CellRec.

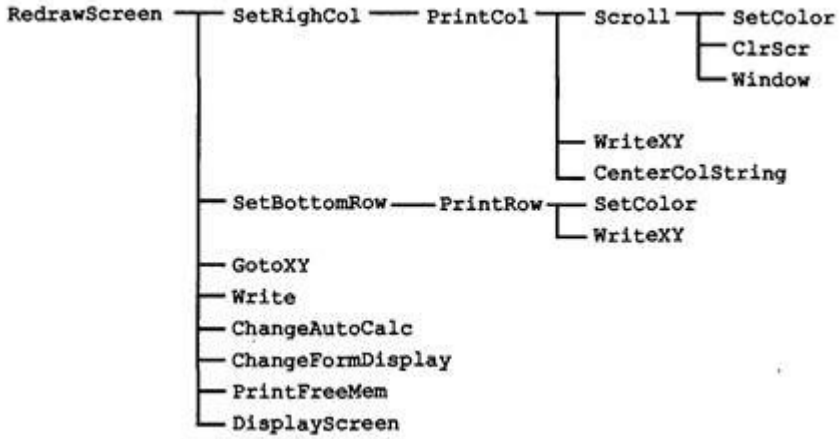
Порівняйте цей проект структури даних із проектом структури даних електронної таблиці, представленої в гол. 3.

Для складання проектної документації, що залишилася, виконаємо трасування програми. Після подвійного натискання клавіші <F7> починає виконуватися налаштування коду, що міститься у файлах *.TPU, і далі починають виконуватися оператори основної програми program Mcalc, що знаходиться у файлі mcalc.pas.

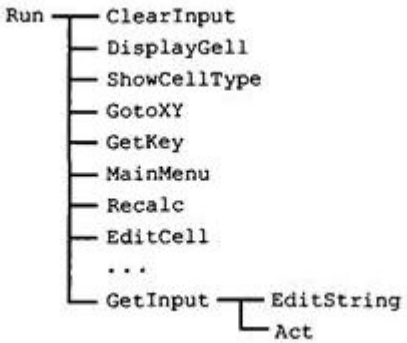
В результаті досліджень було виявлено схему ієрархії модулів програми, зображену на рис. 7.3-7.5. Розшифрування позначень схеми ієрархії представлено у табл. 7.1.



*Мал. 7.3.*Фрагмент схеми ієрархії основних модулів програми



Мал. 7.4. Схема ієрархії модуля RedrawScreen



*Мал. 7.5.*Скорочена схема ієрархії модуля Run

Таблиця 7.1

Розшифрування позначень схеми ієрархії

Ім'я модуля Файл Призначення модуля

Act	Mclib	Обробляє інформацію введеного рядка, заносячи її в клітину
CenterColString	Mcutil	Розраховує X координату центрованого в полі виведення рядка
ChangeAutoCalc	Mclib	Встановлює автоматичний та ручний режими рекалькуляції таблиці
ChangeFormDisplay	Mclib	Встановлює режим видимості значень формул або тексту формул
ClearInput	Mcdisplay	Очищає на екрані поле рядка введення
ClrScr	Crt	Очищає інформацію у вікні екрану
DisplayCell	Mclib	Виводить на екран інформацію клітини
DisplayScreen	Mclib	Відображає на екрані внутрішню інформацію таблиці
EditCell	Mcommand	Здійснює редагування вмісту клітини
EditString	Mcinput	Редактор текстового рядка
EgaInstalled	Mcdisplay	Функція, що визначає наявність відеокарти EGA
FillChar	Dos	Надає елементам масиву значення символу
GetCursor	Mcdisplay	Зчитує товщину курсору у змінну
GetInput	Mcinput	Отримавши перший символ, продовжує введення інформації клітини
GetKey	Mcinput	Формує слово розширеного коду клавіші
GetSetCursor	Mcdisplay	Зчитує товщину курсору в змінну та встановлює нову товщину курсору
GotoXY	Mcdisplay	Переміщує курсор відповідно до заданих координат дисплея
InitColorTable	Mcdisplay	Ініціалізує масив перерахунку кольорів для монохромного монітора
InitDisplay	Mcdisplay	Ініціалізує відеокарту на роботу в режимі 80-25
InitVars	Mcutil	Ініціалізує значення основних змінних програми
Intr	Dos	Викликає переривання MS DOS
LoadSheet	Mcommand	Завантажує інформацію таблиці із файлу
MainMenu	Mcommand	Реалізує вибір тем меню програми
Mcalc	Mcalc	Головна програма
ParamCount	Dos	Лічильник полів командного рядка запуску програми Mcalc
ParamStr	Dos	Повертає значення заданого поля командного рядка запуску програми Mcalc
PrintCol	Mcdisplay	Виводить значення координати колонки таблиці
PrintFreeMem	Mcdisplay	Виводить на екран значення залишку вільної пам'яті
PrintRow	Mcdisplay	Виводить значення координати рядка таблиці
ReadKey	Mcinput	Зчитує короткий код однієї клавіші
Recalc	Mclib	Здійснює перерахунок значень формул клітин таблиці
RedrawScreen	Mclib	Відображає на екрані всю інформацію таблиці
Run	Mcalc	Головний цикл програми
Scroll	Mcdisplay	Прокручує інформацію екрана у вказаному напрямку; встановлює колір фону частини екрана, що звільнилася.
SetBottomRow	Mcdisplay	Виводить на екран стовпець із номерами рядків таблиці
SetColor	Mcdisplay	Встановлює колір виведення рядків на екран

SetCursor	Mcdisplay	Встановлює задану товщину курсору
SetRightCol	Mcdisplay	Виводить на екран рядок із найменуваннями стовпців таблиці
ShowCellType	Mcdisplay	Виводить на екран напис про тип поточної клітини таблиці
TextMode	Dos	Переводить екран у вказаний текстовий режим
Window	Crt	Визначає вікно на екрані дисплея
Write	-	Оператор виведення мови Pascal
WriteXY	Mcdisplay	Здійснює виведення заданого числа символів заданого рядка за заданими координатами дисплея

Розглянемо функціональний опис основного ядра програми. У файлі mcutil.pas виконується рудиментарний код, що залишився від колишніх розробок:

```
HeapError := @HeapFunc;
```

У файлі mcdisplay.pas послідовно виконуються підпрограми: InitDisplay, GetSetCursor, Window, EGAInsalled.
Процедура InitDisplay ініціалізує відеокарту на роботу в режимі 80 • 25 за допомогою переривання 10h і викликом процедури InitColorTable ініціалізує масив перерахунку кольорів для монохромного монітора. Останній масив використовується для викликів процедури SetColor.
Процедура GetSetCursor за допомогою процедури GetCursor зчитує товщину курсора змінну OldCursor і за допомогою процедури SetCursor встановлює нову товщину курсора (NOCURSOR).
Процедура Window визначає вікно на екрані дисплея розміщення інформації всієї таблиці.
Далі починає виконуватись код головної програми Mcalc.
Наданням CheckBreak := False забороняється використання клавіші <Ctrl+Break> негайного завершення програми.
Виведення початкової заставки здійснюється наступними викликами підпрограм.
Процедурами SetColor та ClrScr проводиться очищення вікна програми. Подвійним викликом процедур SetColor та WriteXY виводяться два рядки початкової заставки. Незважаючи на відсутність курсору, відпрацьовується рудиментарний виклик "приховування" курсору GotoXY (80,25). За допомогою функції GetKey здійснюється очікування натискання користувачем будь-якої кнопки.
Процедурами SetColor та ClrScr проводиться очищення вікна програми.
Викликом процедури InitVars ініціалізуються значення основних змінних програм. Масиви ініціалізуються значеннями за промовчанням виклику процедури FillChar.
Присвоюванням Changed := False вказується факт незмінності інформації клітин таблиці після ініціалізації змінних для заборони спрацьовування автозбереження.
Викликом процедури RedrawScreen здійснюється відображення на екрані всієї інформації таблиці.
Якщо значення ParamCount = 1, то командному рядку MS DOS виклику програми було вказано ім'я файлу таблиці. У цьому випадку виконується процедура LoadSheet, яка завантажує інформацію таблиці з файлу з ім'ям файлу, отриманого за допомогою функції ParamStr.
Зрештою, відпрацьовує «зайвий» виклик Clearlnput, який дублюється на початку наступної процедури Run, що містить головний цикл програми.
Після завершення виконання програми послідовно проводиться установка кольору екрана, викликом TextMode переводиться екран у текстовий режим, запам'ятаний змінної OldMode, і, нарешті, викликом SetCursor відновлюється товщина курсора, запам'ятана змінної OldCursor.
Робота процедури RedrawScreen полягає у послідовному виведенні на екран інформації:

• процедурою SetRightCol виводиться на екран рядок із найменуваннями стовпців таблиці;
• процедурою SetBottomRow виводиться на екран колонка із номерами рядків таблиці;
• процедурами GotoXY та Write виводяться написи у верхньому рядку екрана, хоча є зручніша процедура WriteXY;
• виводиться кількість залишку байт пам'яті;
• Процедура DisplayScreen відображає на екрані внутрішню інформацію таблиці.
Зовнішній вигляд програми Mcalc наведено на рис. 7.6.
Робота процедури Run починається із встановлення змінної головного циклу Stop := False та виконання процедури Clearlnput. Головний цикл програми виконується до зміни значення змінної Stop True. Така зміна можлива лише при виборі користувача теми меню Quit — завершення роботи з програмою.
Усередині головного циклу послідовно виконуються такі действия:
- за допомогою процедури DisplayCell виводиться на екран підсвічена клітинним курсором поточна клітина (клітина A1 на рис. 7.6);
- за допомогою процедури ShowCellType виводиться в нижньому лівому куті екрану напис типу поточної клітини таблиці (див. рис. 7.6);
- Оператором Input := GetKey в змінну Input вводиться код символу клавіші, натиснутої користувачем;
— виконуються дії відпрацювання клавіші, натиснутої користувачем.
Дії відпрацювання клавіші, натиснутої користувачем, являють собою ланцюжок альтернативних дій, реалізований структурою ВИБІР. Спочатку відпрацьовуються дії гарячих клавіш. У секції default (якщо клавіша не була гарячою) викликом процедури Getlnput починається занесення інформації в поточну клітинку таблиці. Процедура Getlnput, занісши символ Input в рядок, що редагується, спочатку викликає EditString — редактор текстового рядка інформації клітини і потім викликає процедуру Act, яка обробляє інформацію введеного рядка, заносячи її в клітинку.

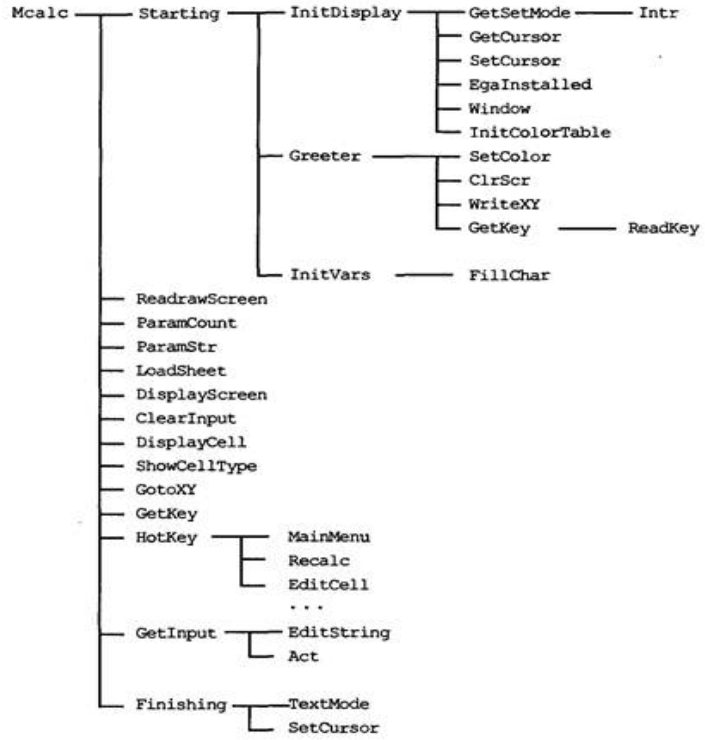
*Мал. 7.6.*Зовнішній вигляд програми Mcalc

Аналіз схеми ієрархії програми та функціонального опису основного ядра програми показав, що основна програма перевантажена допоміжними діями, виділення процедури Run є штучним розподілом основної програми без продуманого структурного розбиття. Усе це призводить до втрати зрозумілості тексту програми.
З метою підвищення зрозумілості програми було ухвалено нові проектні рішення, відображені схемою ієрархії (рис. 7.7).
Виконання основної програми Mcalc починається із запуску нового модуля Starting підготовчих дій програми. Модуль Starting є монітором послідовного виконання модулів InitDisplay, Greeter, InitVars.
Новий модуль InitDisplay тепер є монітором послідовного виконання модулів GetSetMode, GetCursor, SetCursor, Egalnstalled, Window, InitColorTable.

У нового модуля GetSetMode явно як вхідний параметр вказується новий встановлюваний відеорежим, а на виході - старий відеорежим. Така організація краще прямого виклику Intr, оскільки за списком формальних параметрів ясно видно призначення модуля. Реалізація двох функцій виявлення і встановлення відеорежимів в одному модулі тут цілком виправдана, оскільки всі вони реалізуються викликом одного переривання.
Не виправдано використання модуля з двома функціями GetSetCursor, який був монітором послідовного виконання модулів GetCursor, SetCursor. Цей модуль виключено із проекту. Усі функції виведення початкової заставки передані новому модулю Greeter.
З модуля RedrawScreen виключено виклик модуля DisplayScreen. Це дозволило уникнути повторного виклику модуля DisplayScreen у модулі LoadSheet. Також виправлено помилку використання операторів Write для виведення інформації на екран шляхом використання викликів процедури WriteXY.
Далі починає виконуватися головний цикл програми. Модуль Run вилучений із проекту з метою збільшення зрозумілості програми. Довгий текст вибору дій за кодом натиснутої користувачем клавіші замінений однією альтернативою:

```
If (not(HotKey(Input)) and (ConditionalKey(Input)))
then
GetInput(Input).
```

Нова функція HotKey у разі натискання користувачем гарячої клавіші повертає значення TRUE, інакше функція повертає значення FALSE.
Нова функція ConditionalKey у разі натискання користувачем клавіші з кондиційним для занесення в таблицю кодом повертає значення TRUE, інакше функція повертає значення FALSE.



*Мал. 7.7.*Перероблена схема ієрархії модулів програми

Нова процедура WriteXY тепер не використовує виклик повільної процедури GotoXY і оператор Write, що повільно виконується, і використовує прямий доступ до відеопам'яті. Це дозволило значно прискорити виведення інформації на екран. Більше того, в процедуру доданий новий параметр атрибута кольору рядка, що дозволило уникнути ланцюжків початкового виклику SetColor, а потім WriteXY.

Завершується виконання програми викликом нового модуля Finishing. Цей приклад показав самодостатність обраної проектної документації для отримання нового оптимального варіанта побудови структури програми.

ВИСНОВКИ

• Структура програми — штучно виділені програмістом частини програми, що взаємодіють. Використання раціональної структури усуває проблему складності розробки; робить програму зрозумілою людям; підвищує надійність роботи програми при скороченні терміну її тестування та термінів розробки загалом.
• Модуль – функціональний елемент технології структурного програмування. Це підпрограма, але оформлена відповідно до особливих правил.
• Поняття структури програми включає склад і опис зв'язків усіх модулів, які реалізують самостійні функції програми та опис носіїв даних, що беруть участь в обміні як між окремими підпрограмами, так і введеними та виведеними з/на зовнішніх пристроїв.
• Ймовірно, найбільш загальна тактика програмування полягає у розкладанні процесу на окремі дії: функціонального опису на підфункції, а відповідних програм – на окремі інструкції.
• Найголовнішим у схемі ієрархії є мінімізація зусиль зі складання та тестування програми. При використанні заглушок можна добре тестувати сполучення модулів, але не самі модулі. Тестування самих модулів вимагатиме витончених складних заглушок та астрономічного числа тестів. Вихід - до інтеграції модулів тестувати модулі з використанням провідних програм.
• Схема ієрархії повинна відображатися на файлах з вихідними текстами програм таким чином, щоб кожен файл містив якнайбільше готових функцій із загальним призначенням. Це полегшить їх використання у подальших розробках.

Контрольні питання

1. Дайте визначення поняття «структура програми».
2. Що таке модуль програми і які характеристики він повинен мати?
3. Що відбиває схема ієрархії?
4. Яких принципів необхідно дотримуватися, якщо дотримуватися технологій структурного програмування?
5. Дайте визначення поняття «заглушка модуля».
6. Перерахуйте основні засоби зміни топології схеми ієрархії програми.
7. Назвіть критерії оцінки якості схеми ієрархії.
8. Навіщо потрібен паспорт модуля?

Тема 8 ТЕХНОЛОГІЯ ОБ'ЄКТНО-ОРІЄНТОВАНОГО ПРОГРАМУВАННЯ

8.1. ІСТОРІЯ СТВОРЕННЯ ОБ'ЄКТНО-ОРІЄНТОВАНОГО ПРОГРАМУВАННЯ

Практично відразу після появи мов третього покоління (1967) провідні спеціалісти в галузі програмування висунули ідею перетворення постулату фон Неймана: «дані та програми невиразні в пам'яті машини». Їхня мета полягала в максимальному зближенні даних і коду програми. Вирішуючи поставлене завдання, вони зіткнулися із завданням, вирішити яку без декомпозиції виявилося неможливо, а традиційні структурні декомпозиції не спрощували завдання. Зусилля багатьох програмістів і системних аналітиків, створені задля формалізацію підходу, увінчалися успіхом.

Було розроблено три основні принципи того, що потім стало називатися об'єктно-орієнтованим програмуванням (ООПр): успадкування; інкапсуляція; поліморфізм. Результатом їх першого застосування стала мова Сімула-1 (Simula-1), в якій було введено новий тип – об'єкт. У описі цього одночасно вказувалися дані (поля) і процедури, їх обробні — методи. Споріднені об'єкти об'єднувалися в класи, описи яких оформлялися у вигляді блоків програми. При цьому клас можна використовувати як префікс до інших класів, які в цьому випадку стають підкласами першого. Згодом Сімула-1 був узагальнений, і з'явилася перша універсальна ООПр — об'єктно-орієнтована мова програмування — Сімула-67 (67 — за роком створення).

Як з'ясувалося, ООПр виявилися придатними не тільки для моделювання (Simula) і розробки графічних додатків (SmallTalk), але і для створення більшості інших додатків, а їх наближеність до людського мислення та можливість багаторазового використання коду зробили їх найбільш використовуваними у програмуванні.

Об'єктно-орієнтований підхід допомагає подолати такі складні проблеми, як зменшення складності програмного забезпечення; підвищення надійності програмного забезпечення; забезпечення можливості модифікації окремих компонентів програмного забезпечення без зміни інших компонентів; забезпечення можливості повторного використання окремих компонент програмного забезпечення.

8.2. ВСТУП В ОБ'ЄКТНО-ОРІЄНТОВАНИЙ ПІДХІД ДО РОЗРОБКИ ПРОГРАМ

В основу структурного мислення покладено структуризацію та декомпозицію навколишнього світу. Завдання будь-якої складності розбивається на підзавдання, а ті в свою чергу розбиваються далі і т. д., поки кожне підзавдання не стане простим, відповідним модулем.

Модуль у понятті структурного програмування - це підпрограма (функція або процедура), оформлена певним чином і виконує строго одну дію. Методи структурного проектування використовують модулі як будівельні блоки програми, а структура програми представляється ієрархією підпорядкованості модулів.

Модуль ООПр - файл описів об'єктів та дій над ними.

Методи об'єктно-орієнтованого проектування використовують як будівельні блоки об'єкти. Кожна структурна складова є самостійним об'єктом, що містить свої власні коди та дані. Завдяки цьому зменшено або відсутня область глобальних даних.

Об'єктно-орієнтоване мислення адекватно способу природного людського мислення, бо людина мислить образами і абстракціями. Щоб проілюструвати деякі з принципів об'єктно-орієнтованого мислення, звернемося до наступного прикладу, що ґрунтується на аналогії світу об'єктів реальному світу.

Розглянемо ситуацію із повсякденного життя. Допустимо, ви вирішили поїхати в інше місто поїздом. Для цього ви приходите на найближчу залізничну станцію та повідомляєте касиру номер потрібного поїзда та дату, коли плануєте виїхати. Тепер можете бути впевнені, що ваш запит буде задоволений (за умови, що ви купуєте квиток заздалегідь).

Таким чином, для вирішення своєї проблеми ви знайшли об'єкт «касир залізничної каси» та передали йому повідомлення, що містить запит. Обов'язком об'єкта "касир залізничної каси" є задоволення запиту.

Касир має певний певний метод, або евроритм, або послідовність операцій (процедура), які використовують працівники каси для виконання вашого запиту. Є у касира та інші методи, наприклад, по здачі грошей — інкасації.

Вам зовсім не обов'язково знати не тільки детально метод, який використовується касиром, а й навіть весь набір методів роботи касира. Однак якби вас зацікавило питання, як працює касир, то виявили б, що касир надішле своє повідомлення автоматизованій системі залізничного вокзалу. Та, у свою чергу, вживе необхідних заходів і т. д. Тим самим ваш запит, зрештою, буде задоволений через послідовність запитів, що пересилаються від одного об'єкта до іншого.

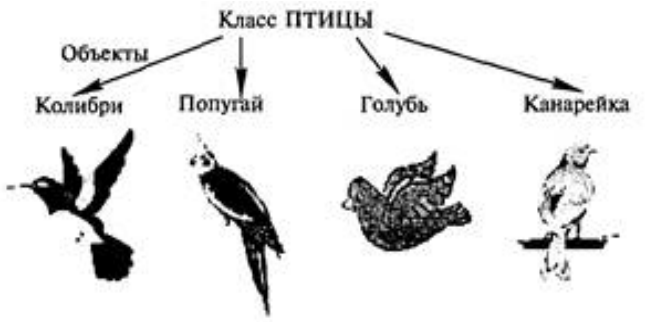
Таким чином, дія в об'єктно-орієнтованому програмуванні ініціюється шляхом передачі повідомлень об'єкту, відповідальному за дію. Повідомлення містить запит на здійснення дії конкретним об'єктом та супроводжується додатковими аргументами, необхідними для його виконання. Приклад аргументів повідомлення: дата від'їзду, номер поїзда, тип вагона.

Повідомлення касира: дайте паспорт, заплатіть таку суму, отримайте квиток та здачу.

Касир, що знаходиться на робочому місці, не зобов'язаний відволікатися від роботи для порожньої балаканини з покупцем квитка, наприклад, повідомляти йому свій домашній телефон або суму грошей, що знаходиться в сейфі каси. Таким чином, касир взаємодіє з іншими об'єктами ("покупець квитка", "автоматизована система", "інкасатор", "бригадир" тощо) тільки за строго регламентованим інтерфейсом. Інтерфейс – це набір форматів допустимих повідомлень. Для виключення можливих, але неприпустимих повідомлень використовується механізм приховування інформації (інструкція, яка забороняє касиру пустувати на робочому місці).

Крім методів, касир для успішної роботи повинен мати у своєму розпорядженні набори чистих бланків квитків, купюрами і монетами готівки (хоча б для здачі покупцю). Такі набори зберігаються в спеціальних відсіках каси, спеціальних коробках. Місця зберігання цих наборів називають полями об'єктів. У програмах полям об'єктів відповідають змінні, які можуть зберігати якісь значення.

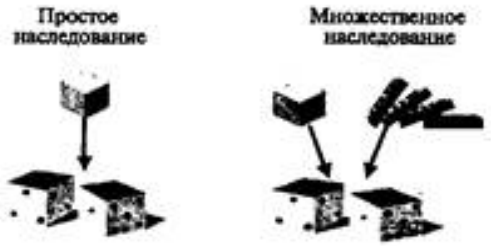
Покупець квитка не може покласти гроші безпосередньо у відсік касового апарату чи сейф касира, а також самостійно відрахувати собі здачу. Таким чином, касир як би укладений в оболонку, або капсулу, яка відокремлює його та покупця від зайвих взаємодій. Приміщення каси (капсула) має особливий пристрій, що унеможливлює доступ покупців квитків до грошей. Це і є інкапсуляція об'єктів, що дозволяє використовувати тільки допустимий інтерфейс - обмін інформацією і предметами тільки за допомогою допустимих повідомлень, а може, ще й поданих у потрібній послідовності. Саме через виклик повідомленнями особливих методів здійснюється обмін даних, відокремлюючи покупців від полів. Завдяки інкапсуляції покупець може лише віддавати як оплату гроші за квиток у формі повідомлення з аргументом «сума». Аналогічно, але у зворотному напрямку касир повертає здачу.
Ви можете передати своє повідомлення, наприклад, об'єкту "свій приятель", і він його, швидше за все, зрозуміє, і як результат - дія буде виконана (а саме квитки будуть куплені). Але якщо ви попросите про те ж об'єкт «продавець магазину», у нього може не виявитися відповідного методу для вирішення поставленого завдання. Якщо припустити, що об'єкт «продавець магазину» взагалі сприйме цей запит, він «видасть» належне повідомлення про помилку. На відміну від програм, люди працюють не за алгоритмами, а за євроритмами. Людина може самостійно змінювати правила методів своєї роботи. Так, продавець магазину, побачивши аргумент «дуже велика сума», може закрити магазин і побігти купувати залізничний квиток. Нагадаємо, що такі ситуації для програм поки що неможливі.
Відмінність між викликом процедури та пересиланням повідомлення полягає в тому, що в останньому випадку існує певний одержувач та інтерпретація (тобто вибір відповідного методу, що запускається у відповідь на повідомлення), яка може бути різною для різних одержувачів.
Зазвичай конкретний об'єкт-одержувач невідомий до виконання програми, отже визначити, який метод, якого об'єкта буде викликано, заздалегідь неможливо (конкретний касир заздалегідь не знає, хто і коли з конкретних покупців звернеться щодо нього). У такому разі говорять, що має місце пізнє зв'язування між повідомленням (ім'ям процедури або функції) та фрагментом коду (методом), що виконується у відповідь на повідомлення. Ця ситуація протиставляється ранньому зв'язуванню (на етапі компілювання або компонування програми) імені із фрагментом коду, що відбувається при традиційних викликах процедур.
Фундаментальною концепцією в об'єктно орієнтованому програмуванні є поняття класів. Усі об'єкти є представниками або екземплярами класів. Наприклад: у вас напевно є зразкове уявлення про реакцію касира на запит про замовлення квитків, оскільки ви маєте загальну інформацію про людей даної професії (наприклад, касира кінотеатру) і очікуєте, що він, будучи представником цієї категорії, загалом відповідатиме шаблону. Те саме можна сказати і про представників інших професій, що дозволяє розділити людське суспільство на певні категорії за професійною ознакою (на класи). Кожна категорія у свою чергу поділяється на представників цієї категорії. Отже, людське суспільство представляється як ієрархічної структури з успадкуванням властивостей класів об'єктів всіх категорій. У корені такої класифікації може бути клас «HomoSapience» або навіть клас «ссавці» (рис. 8.1).
Метод, активізований об'єктом у відповідь повідомлення, визначається класом, якого належить одержувач повідомлення. Усі об'єкти одного класу використовують одні й самі методи у відповідь однакові повідомлення.
Класи можуть бути організовані в ієрархічну структуру з наслідуванням властивостей. Клас-нащадок успадковує атрибути батьківського класу, розташованого нижче в ієрархічному дереві (якщо дерево ієрархії успадкування росте вгору). Абстрактний батьківський клас - це клас, що не має екземплярів об'єктів. Він використовується лише для породження нащадків. Клас HomoSapience, швидше за все, буде абстрактним, оскільки для практичного застосування, наприклад роботодавцю, екземпляри його об'єктів не цікаві.



*Мал. 8.1.*Поняття створення об'єкта як екземпляра класу ПТАХИ

Отже, нехай абстрактним батьківським класом у роботодавця буде клас «працездатна людина», який включає методи доступу до внутрішніх даних, а також поля самих внутрішніх даних: прізвище; ім'я; по батькові; дата народження; домашню адресу; домашній телефон; відомості про освіту; відомості про трудовий стаж і т. д. Від цього класу можуть бути успадковані класи: "касир", "водій автомобіля", "музикант". Клас «касир» має в своєму розпорядженні методи роботи: спілкування з клієнтом за правилами, отримання грошей, видача грошей, спілкування з інкасатором і т.д. . Касир залізничної каси відрізняється від касира, який видає зарплату, додатковими знаннями та навичками роботи.
Від класу «касир залізничної каси» можна отримати екземпляри об'єктів: «касир каси №1», «касир каси № 2», «касир каси № 3» тощо.
У приміщенні великого вокзалу можна знайти безліч однаково обладнаних об'єктів — кас. Однак серед кас можна виділити каси, що розрізняються: добові, попередні, військові, що працюють з бронювання квитків і т. д. Для того щоб начальнику вокзалу поміняти один вид каси на інший, немає необхідності перебудовувати приміщення каси і змінювати обладнання. Йому достатньо замінити в касі касира з одними навичками касира з іншими навичками. Касир вставляє табличку із новим написом виду каси — і все. Зауважимо, що зміна функції кас відбулася без припинення роботи вокзалу. Така заміна стає простою саме тому, що всі приміщення кас мають однаковий інтерфейс із касирами та клієнтами. Тепер різні об'єкти, які підтримують однакові інтерфейси, можуть виконувати у відповідь запити різні операції. Асоціація запиту з об'єктом та однією з його операцій під час виконання називається динамічним зв'язуванням. Динамічне зв'язування дозволяє під час виконання підставити замість одного об'єкта інший, якщо він має такий самий інтерфейс. Така взаємозамінність називається поліморфізмом та є ще однією фундаментальною особливістю об'єктно-орієнтованих систем (рис. 8.2).
Нехай, згідно з проведеною класифікацією, об'єкти «скрипаль із прізвищем Петров» та «водій автомобіля Сидорів» будуть екземплярами різних класів. Для того щоб отримати об'єкт «Іванів, який є одночасно скрипалем і водієм», необхідний особливий клас, який може бути отриманий із класів «скрипач» та «водій автомобіля» множинним успадкуванням (рис. 8.3). Тепер роботодавець, надіславши особливе повідомлення делегування, може доручити (делегувати) об'єкту Іванів виконувати функцію або водія, або скрипаля. Об'єкт «Іванів», що знаходиться за кермом автомобіля, не повинен грати на скрипці. Для цього має бути реалізований механізм самоделегування повноважень — об'єкт «Іванів», перебуваючи за кермом, забороняє собі гру на скрипці. Таким чином, поняття обов'язку чи відповідальності за виконання дії є фундаментальним у об'єктно-орієнтованому програмуванні.

Рис. 8.2. Понятие полиморфизма



*Мал. 8.3.*Приклад простого та множинного успадкування

У системах програмування з відсутнім множинним успадкуванням завдання, що вимагають множинного успадкування, завжди можуть бути вирішені композицією (агрегуванням) з подальшим делегуванням повноважень.

*Композиція об'єктів*це реалізація складового об'єкта, що складається з кількох спільно працюючих об'єктів і утворюють єдине ціле з новою, складнішою функціональністю.

*Агрегований об'єкт*об'єкт, складений із подобъектов. Подіб'єкти називаються частинами агрегату, і агрегат відповідає за них. Наприклад, у системах з множинним успадкуванням шахова фігура ферзь може бути успадкована від слона та човна. У системах з відсутнім множинним наслідуванням можна отримати ферзя двома способами. Згідно з першим способом, можна створити клас «люба_фігура» і далі, в періоді виконання, делегувати повноваження кожному об'єкту-примірнику даного класу бути човном, слоном, ферзою, пішаком і т.д. їх можна об'єднати композицією в клас «ферзь». Тепер об'єкт класу "ферзь" можна використовувати як об'єкт "ферзь" або навіть як об'єкт "слон", для чого об'єкту "ферзь" делегується виконання повноважень слона. Більше того, можна делегувати об'єкту «ферзь» повноваження стати об'єктами «король» чи навіть «пішака»! Для композиції потрібно, щоб об'єднані об'єкти мали чітко визначені інтерфейси. І успадкування, і композиція має переваги і недоліки.

Спадкування класу визначається статично на етапі компіляції; його простіше використовувати, оскільки воно безпосередньо підтримане мовою програмування.

Але успадкування класу має й мінуси. По-перше, не можна змінити успадковану від батька реалізацію під час виконання програми, оскільки саме успадкування фіксовано на етапі компіляції. По-друге, батьківський клас нерідко, хоч би частково, визначає фізичне уявлення своїх підкласів. Оскільки підклас доступні деталі реалізації батьківського класу, то часто кажуть, що успадкування порушує інкапсуляцію. Реалізації підкласу та батьківського класу настільки тісно пов'язані, що будь-які зміни останньої вимагають змінювати та реалізацію підкласу.

Композиція об'єктів визначається динамічно під час виконання завдяки тому, що об'єкти отримують посилання інші об'єкти. Композицію можна застосувати, якщо об'єкти дотримуються інтерфейсів один одного. Для цього, у свою чергу, потрібно ретельно проектувати інтерфейси, щоб один об'єкт можна було використовувати разом з широким

спектром інших. Але й виграш великий, оскільки доступ до об'єктів здійснюється лише через їх інтерфейси, ми не порушуємо інкапсуляцію. Під час виконання програми будь-який об'єкт можна замінити іншим, аби він мав той самий тип. Понад те, оскільки за реалізації об'єкта кодуються передусім його інтерфейси, залежність від реалізації різко знижується.
Композиція об'єктів впливає дизайн системи і ще одному аспекті. Віддаючи перевагу композиції об'єктів, а не успадкування класів, ви інкапсулюєте кожен клас і даєте можливість виконувати тільки своє завдання. Класи та його ієрархії залишаються невеликими, і можливість їх розростання до некерованих розмірів невелика. З іншого боку, дизайн, заснований на композиції, міститиме більше об'єктів (хоча кількість класів, можливо, зменшиться), і поведінка системи почне залежати від їхньої взаємодії, тоді як за іншого підходу було б визначено в одному класі.
Це підводить ще до одного правила об'єктно-орієнтованого проектування: віддайте перевагу композиції успадкування класу.
В ідеалі, щоб домогтися повторного використання коду, взагалі не слід створювати нові компоненти. Добре, щоб можна було отримати всю потрібну функціональність, просто збираючи разом вже існуючі компоненти. На практиці, однак, так виходить рідко, оскільки набір наявних компонентів все ж таки недостатньо широкий. Повторне використання з допомогою успадкування спрощує створення нових компонентів, які можна було б застосовувати зі старими. Тому успадкування та композиція часто використовуються разом.
Проте досвід показує, що проектувальники зловживають успадкуванням. Нерідко програми могли б стати простішими, якби їхні автори більше покладалися на композицію об'єктів.
За допомогою делегування композицію можна зробити настільки ж потужним інструментом повторного використання, як і успадкування. При делегуванні до процесу обробки запиту залучено два об'єкти: одержувач доручає виконання операцій іншому об'єкту – уповноваженому. Приблизно також підклас делегує відповідальність своєму батьківському класу. Але успадкована операція завжди може звернутися до об'єкта-отримувача через змінну-член (C++) або змінну self (Smalltalk). Щоб досягти того ж ефекту для делегування, одержувач передає покажчик на себе відповідному об'єкту, щоб при виконанні делегованої операції останній міг звернутися до безпосереднього адресату запиту.
Наприклад, замість того щоб робити клас Window (вікно) підкласом класу Rectangle (прямокутник) - адже вікно є прямокутником, - ми можемо скористатися всередині Window поведінкою класу Rectangle, помістивши в клас Window змінну екземпляра типу Rectangle і делегуючи їй операції, специфічні для прямокутника. Інакше кажучи, вікно перестав бути прямокутником, а містить його. Тепер клас Window може явно перенаправляти запити своєму члену Rectangle, а чи не успадковувати його операції.
Головне достоїнство делегування у цьому, що його спрощує композицію поведінки під час виконання. При цьому спосіб комбінування поведінки можна змінювати. Внутрішню область вікна дозволяється зробити круговою під час виконання простою підставкою замість екземпляра класу Rectangle екземпляра класу Circle. Передбачається, звичайно, що обидва ці класи мають однаковий тип.
У делегування є і недолік, властивий та іншим підходам, які застосовуються для підвищення гнучкості за рахунок композиції об'єктів. Полягає він у тому, що динамічну, високою мірою параметризовану програму важче зрозуміти, ніж статичну. Є, звичайно, і деяка втрата машинної продуктивності, але неефективність роботи проектувальника набагато суттєвіша.
Делегування можна вважати добрим вибором лише тоді, коли воно дозволяє досягти спрощення, а не ускладнення. Нелегко сформулювати правила, що чітко говорять, коли слід користуватися делегуванням, оскільки ефективність його залежить від контексту та особистого досвіду програміста.
Таким чином, можна виділити такі фундаментальні характеристики об'єктно-орієнтованого мислення:
Характеристика1.Будь-який предмет чи явище може розглядатися як об'єкт.

Характеристика2.Об'єкт може розміщувати у своїй пам'яті (у полях) особисту інформацію, незалежну від інших об'єктів. Рекомендується використовувати інкапсульований (через спеціальні методи) доступ до інформації полів.
Характеристика3.Об'єкти можуть мати відкриті за інтерфейсом методи обробки повідомлень. Самі повідомлення викликів методів надсилаються іншими об'єктами, але для здійснення розумного інтерфейсу між об'єктами деякі методи можуть бути приховані.
Характеристика4.Обчислення здійснюються шляхом взаємодії (обміну даними) між об'єктами, коли один об'єкт вимагає, щоб інший об'єкт виконав деяку дію (метод). Об'єкти взаємодіють, посилаючи та отримуючи повідомлення. Повідомлення — це запит виконання дії, доповнений набором аргументів, які можуть знадобитися під час виконання дії. Об'єкт – одержувач повідомлення – обробляє повідомлення своїми внутрішніми методами.
Характеристика5.Кожен об'єкт є представником класу, який виражає загальні властивості об'єктів даного класу у вигляді однакових списків набору даних (полів) у своїй пам'яті та внутрішніх методів, що обробляють повідомлення. У класі методи задають поведінку об'єкта. Тим самим усі об'єкти, які є екземплярами одного класу, можуть виконувати ті самі дії.
Характеристика6.Класи організовані в єдину квазідревоподібну структуру із загальним коренем, що називається ієрархією спадкування. Зазвичай корінь ієрархії спрямований нагору. При множинні спадкування гілки можуть зростатися, утворюючи мережу спадкування. Пам'ять і поведінка, пов'язані з екземплярами певного класу, автоматично доступні будь-якому класу, розташованому нижче в ієрархічному дереві.
Характеристика 7. Завдяки поліморфізму — здатності підставляти під час виконання замість одного об'єкта інший, з сумісним інтерфейсом, у періоді виконання одні й самі об'єкти можуть різними методами виконувати одні й самі запити повідомлень.
Характеристика8.Композиція є кращою альтернативою множинному успадкування і дозволяє змінювати склад об'єктів агрегату у процесі виконання програми.
Характеристика9.Структура об'єктно-орієнтованої програми на етапі виконання часто має мало спільного зі структурою вихідного коду. Остання фіксується на етапі компіляції. Її код складається з класів, відносини наслідування між якими незмінні. На етапі виконання структура програми — мережа, що швидко змінюється, з взаємодіючих об'єктів. Ці дві структури майже незалежні.

8.3. ПОРІВНЯЛЬНИЙ АНАЛІЗ ТЕХНОЛОГІЙ СТРУКТУРНОГО ТА ОБ'ЄКТНО-ОРІЄНТОВАНОГО ПРОГРАМУВАННЯ

Для проведення порівняльного аналізу технологій структурного та об'єктно-орієнтованого програмування розроблено спеціальну методику, засновану на таких об'єктивних принципах, як арифметичний підрахунок елементів тексту програми, аналіз алгоритмів програм. Арифметичний підрахунок виконувався ручним рахунком та був доповнений статистичними даними, що видаються компіляторами та текстовими редакторами. Підсумкові таблиці та його візуалізація здійснювалася з допомогою програми Excel. Таблиці включають інформацію щодо окремих файлів та розрахунок підсумкової інформації по всій програмі. Інформація щодо окремих файлів представлена:
1) ім'ям файлу;
2) загальною кількістю рядків файлу (показується текстовим редактором);
3) кількістю рядків операторів описів даних у всьому файлі;
4) загальною кількістю коментарів у файлі (виявляється контекстним пошуком ознаки коментаря у тексті файлу);
5) кількістю рядків окремих коментарів у файлі;
6) кількістю порожніх рядків у файлі (виявляється візуальним аналізом тексту файлу);
7) кількістю підпрограм у файлі (є контекстним пошуком заголовків procedure і function у тексті файла);
8) кількістю операторів опису підпрограм у файлі;

9) кількістю рядків коду, розрахованих за формулою: кількість рядків коду = 2) - 3) - 5) - 6) - 8).

Кількість операторів опису підпрограм у файлі виявляється за принципом підрахунку всіх термінів, наприклад, у наступному прикладі виявлено чотири рядки:

```
function CellString (Col, Row: Word; var Color: Word;
Formatting; Boolean): String;
Begin
End; {CellStrung}
```

Для проведення об'єктивного порівняльного аналізу був потрібний вибір функціонально схожих програм:

- Mcalc - розглянута раніше в гол. 2 та 7 демонстраційна програма, реалізована за технологією структурного програмування;
- Tcalc - демонстраційна програма, реалізована за технологією об'єктно-орієнтованого програмування - функціонально повний аналог програми Mcalc.

Результати арифметичного аналізу тексту програми MCalc, розробленої за технологією структурного програмування, представлені у табл. 8.1.

Таблиця 8.1

Результати аналізу тексту програми MCalc

Ім'я файлу	Усього рядків	Кількість описових операторів	Коментарі		Порожніх рядків	Кількість процедур	Кількість описових операторів процедур	Код
			Усього	Рядок				
Mcalc	143	8	11	7	5	2	6	117
Mcdisply	357	54	47	15	49	18	64	175
Mcinput	240	33	18	8	19	7	25	155
Mclib	503	68	47	20	46	21	73	296
Mcommand	873	88	63	19	54	24	86	626
Mcparser	579	51	33	21	16	12	36	455
Mcutil	413	62	46	16	45	18	75	215
mcvars	124	96	9	5	19	0	0	0
Разом:	3232	460	274	111	253	102	365	2043
		15,4%		3,7%			12,3%	68,6%

Аналіз демонстраційної програми TCalc «Borland Inc.»

Програма TCalc 1993 (Turbo Pascal 6.0) складається з наступних файлів:

tcalc.pas - файл основної програми;
tcell.pas - файл роботи з клітинами;
tcellsp.pas - файл доповнень роботи з клітинами (зміна значень);
tchash.pas - файл доповнень роботи з клітинами (значення в клітинах);
tcinput.pas - файл підпрограм введення даних з клавіатури;
tclstr.pas - файл підпрограм роботи з рядками;
tcmenu.pas - файл підпрограм, що обслуговують систему меню;
tcparser.pas - файл інтерпретатора арифметичних виразів формул клітин;
tcrun.pas - файл ініціалізації та запуску основних об'єктів;
tcscreen.pas - файл підпрограм роботи з дисплеєм;
tcsheet.pas - файл підпрограм, що обслуговують дії, вибраних за допомогою меню;
tcutil.pas - файл допоміжних підпрограм;
mcmvsmem.asm - асемблерний файл підпрограм запам'ятовування в оперативній пам'яті інформації екрана, а також відновлення раніше збереженої інформації екрана.

Усі файли закодовані з дотриманням стандартів оформлення.

Хоча фірма Borland Inc. займається розробкою компіляторів, файл mcparser.pas також є запозиченим з UNIX YACC utility і лише частково модифікований. Інші файли є оригінальними.

Асемблер файл mcmvsmem.asm є штучно доданим. Мета його додавання – демонстрація можливості використання асемблерних вставок. Результати арифметичного аналізу тексту програми представлені у табл. 8.2.

Таблиця 8.2

Результати аналізу тексту програми TCalc

Ім'я файлу	Усього рядків	Кількість описових операторів	Коментарі		Порожніх рядків	Кількість процедур	Кількість описових операторів процедур	Код
			Усього	Рядок				
Tcalc	21	2	9	3	5	1	3	8
Tcell	1962	490	206	20	153	46	152	1147
Tcellsp	228	39	24	5	18	6	25	141
Tchash	262	50	47	23	23	14	43	123
Tcinput	334	63	39	15	22	9	32	202
Tclstr	243	45	120	20	12	15	52	114
Tcmenu	234	48	40	20	21	22	66	79
Tcparser	677	73	29	5	17	9	64	518
Tcrun	1367	146	128	59	57	47	163	942
Tcscreen	523	215	92	37	16	8	96	159
Tcsheet	1722	240	170	40	44	32	101	1297
Tcutil	379	114	55	38	70	29	115	42
Разом:	7952	1525	959	285	458	238	912	4772
		20,3%		3,8%			12,2%	63,7%

У табл. 8.3 та на рис. 8.3 відображено результати порівняльного аналізу технологій структурного та об'єктно-орієнтованого програмування.

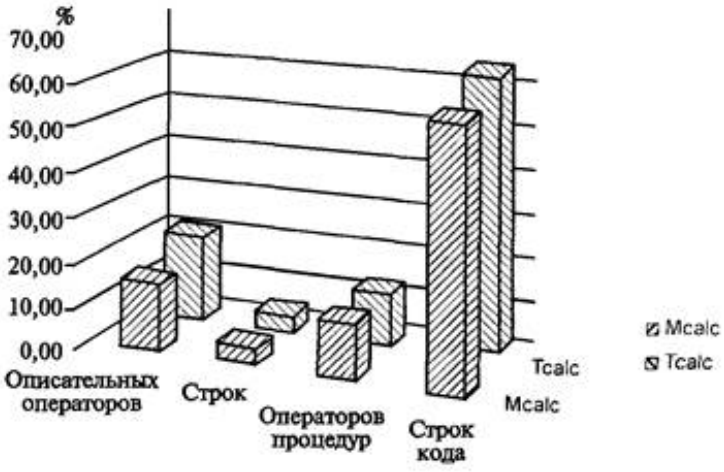
Таблиця 8.3

Результати порівняльного аналізу технологій структурного та об'єктно-орієнтованого програмування

Ім'я програми	Усього рядків	Кількість описових операторів	Коментарі		Порожніх рядків	Кількість процедур	Кількість операторів процедур	Код
			Усього	Рядок				
MCalc	3232	460	274	111	253	102	365	2043
		15,4%		3,7%			12,3%	68,6%
TCalc	7952	1525	959	285	458	238	912	4772
		20,3%		3,8%			12,2%	63,7%

Порівняльний аналіз технологій структурного та об'єктно-орієнтованого програмування встановив, що для цих технологій спостерігається практично повний збіг:

- відсотковий склад описових операторів;
- відсотковий склад кількості коментарів;
- відсотковий склад описових операторів процедур;
- процентний склад операторів коду програми.



*Мал. 8.3.*Результати порівняльного аналізу технологій структурного та об'єктно-орієнтованого програмування

При проведенні розробки за технологією об'єктно-орієнтованого програмування в порівнянні з технологією структурного програмування обсяг коду збільшився в 2,34 рази з урахуванням тільки коду, що виконує ті самі функції (для цього було виключено код функцій, аналогічних функцій роботи з clipboard Windows). Загальна кількість рядків збільшилась у 2,46 разу. У стільки і навіть більше разів зросла трудомісткість розробки.
Власне функціонально корисний код програм Mcalc та Tcalc – однаковий і становить близько 1500 рядків.
Майже 2,3-3,5 кратне збільшення трудомісткості розробки пояснюється платою за організацію самостійності поведінки об'єктів та їхню завершену функціональність для повторного використання.

8.4. ОСНОВНІ ПОНЯТТЯ ОБ'ЄКТНО-ОРІЄНТУВАНОЇ ТЕХНОЛОГІЇ

З чого починається створення об'єктно-орієнтованої програми?
Звісно, з об'єктно-орієнтованого аналізу (ООА—object-oriented analysis), спрямований створення моделей реальної дійсності з урахуванням об'єктно-орієнтованого світогляду.
Об'єктно-орієнтований аналіз (ООА) — це методологія, за якої вимоги до системи сприймаються з погляду класів та об'єктів, прагматично виявлених у предметній галузі.
На результатах ООА формуються моделі, на яких ґрунтується об'єктно-орієнтоване проектування (object-oriented design, OOD).
Об'єктно-орієнтоване проектування (ООП) - це методологія проектування, що поєднує в собі процес об'єктної декомпозиції та прийоми уявлення логічної та фізичної, а також статичної та динамічної моделей проектованої системи.
Що таке об'єктно-орієнтоване програмування (ООПр) (object-oriented programming)?
Програмування передусім має на увазі правильне та ефективне використання механізмів конкретних мов програмування. Об'єктно-орієнтоване програмування - це процес реалізації програм, заснований на представленні програми у вигляді сукупності об'єктів. ООПр передбачає, будь-яка функція (процедура) у програмі є метод об'єкта деякого класу, причому клас має формуватися у програмі природним чином, як у програмі виникає необхідність опису нових фізичних предметів чи його абстрактних понять (об'єктів програмування).
Кожен новий крок у розробці алгоритму також повинен бути розробкою нового класу на основі вже існуючих класів, тобто технологія ООП інакше може бути названа як програмування «від класу до класу».
Чи можна реалізувати об'єктно-орієнтовану програму не об'єктно-орієнтованими мовами?
Відповідь, швидше за все, позитивна, хоча доведеться подолати низку труднощів. Адже

головне, що потрібно, це реалізувати об'єктну модель. Приховування інформації під час використання звичайних мов, в принципі, можна реалізувати приховуванням доступності дзвінків підпрограм у файлах (Unit). Інкапсуляцію об'єктів можна досягти як і в об'єктно орієнтованих мовами написанням окремих підпрограм. Далі можна вважати, кожен об'єкт породжується від свого унікального класу. Звичайно, ієрархії класів у такому проекті не буде і для досягнення паралелізму доведеться писати код для організації виклику до виконання одразу кількох копій процедур, але програма при цьому буде цілком об'єктно-орієнтованою.

8.5. ОСНОВНІ ПОНЯТТЯ, ЩО ВИКОРИСТОВУЮТЬСЯ В ОБ'ЄКТНО-ОРІЄНТУВАНИХ МОВАХ

Клас в одному з значень цього терміну означає тип структурованих даних.
Об'єкт – це структурована змінна типу клас. Кожен об'єкт є представником (примірником) певного класу. У програмі може бути кілька об'єктів, які є екземплярами одного й того самого класу. Усі об'єкти — екземпляри даного класу — аналогічні один одному, оскільки мають однаковий інтерфейс, той самий набір операцій (методів) і полів, що визначаються в їхньому класі. Інтерфейс класу іноді називають особливостями класу.
Клас є описом того, як виглядатиме і поводитиметься його представник. Зазвичай проектують клас як освіту (матрицю), що відповідає за створення своїх нових представників (примірників чи об'єктів). Примірник об'єкта створюється за допомогою особливого методу класу, званого конструктором, оскільки необхідно створити екземпляр, перш ніж він стане активним і почне взаємодіяти з навколишнім світом. Знищення екземплярів підтримує сам активний екземпляр, який має відповідний метод – деструктор.
Об'єкт- Це структурована змінна типу клас, що містить всю інформацію про деякий фізичний предмет або реалізується в програмі поняття.
*Об'єкт*це логічна одиниця, яка містить дані та правила (методи з кодом алгоритму) (див. рис. 1.8). Іншими словами, об'єкт - це розташовані в окремій ділянці пам'яті:
— порція даних об'єкта чи атрибути вихідних даних, звані полями, членами даних (data members), значення яких визначають поточний стан об'єкта;
- методи об'єкта (methods, у різних мовах програмування ще називають підпрограмами, діями, member functions або функціями-членами), що реалізують дії (виконання алгоритмів) у відповідь на їх виклик у вигляді переданого повідомлення;
- Частина методів, званих властивостями (property), які, у свою чергу, визначають поведінку об'єкта, тобто його реакцію на зовнішні впливи (у ряді мов програмування властивості оформляються особливими операторами).
Об'єкти у програмах відтворюють всі відтінки явищ реального світу: «народжуються» та «вмирають»; змінюють свій стан; запускають та зупиняють процеси; «вбивають» та «відроджують» інші об'єкти.
Оголошення класів визначають описані три характеристики об'єктів: поля об'єкта, методи об'єкта, властивості об'єктів. Також в оголошеннях може вказуватись предок даного класу. Відповідно до описом класу всередині об'єкта дані та методи можуть бути як відкритими за інтерфейсом public, так і прихованим private.
Під час виконання програми об'єкти взаємодіють один з одним за допомогою виклику методів об'єкта, що викликається - в цьому і полягає передача повідомлень. Для того, щоб об'єкт надіслав повідомлення іншому об'єкту, у більшості мов програмування потрібно після вказівки імені об'єкта, що викликається, записати виклик підпрограми (методу) з відповідним ім'ям та вказівкою необхідних фактичних параметрів (аргументів). Отримавши повідомлення, об'єкт-отримувач починає виконувати код викликаної підпрограми (методу) з отриманими значеннями аргументів. Таким чином, функціонування програми (виконання всього алгоритму програми) здійснюється послідовним викликом методів від одного об'єкта до іншого.
Хоча можна отримати прямий доступ до полів об'єкта, використання такого підходу не заохочується. Одна з великих переваг ООПр — інкапсуляція, призначена для дозволу роботи

з даними в полях об'єктів тільки через повідомлення. p align="justify"> Для реалізації методів обробки таких повідомлень використовуються властивості. Властивості - це особливим чином оформлені методи, призначені як для читання та контрольованої зміни внутрішніх даних об'єкта (полів), так і виконання дій, пов'язаних із поведінкою об'єкта.
Так, наприклад, якщо в заданому місці екрана вже відображено якийсь рядок і ми хочемо змінити положення рядка на екрані, то ми надсилаємо об'єкту нове значення властивості у вигляді набору потрібних координат. Далі властивість автоматично трансформується у виклик методу, який змінить значення поля координат відображення рядка та виконає дії зі знищення зображення рядка на попередньому місці екрана, а також відображення рядка в новому місці екрана.
Можна виділити кілька переваг інкапсуляції.
*Перевага 1.*Надійність даних. Можна запобігти зміні елемента даних, виконавши у властивості (методі) додаткову перевірку значення допустимість. Таким чином, можна гарантувати надійний стан об'єкта.
*Перевага 2.*Цілісність посилань. Перед доступом до об'єкта, пов'язаного з цим об'єктом, можна переконатися, що непряме поле містить коректне значення (посилання на екземпляр).
*Перевага 3.*Передбачені побічні ефекти. Можна гарантувати, що кожного разу, коли виконується звернення до поля об'єкта, синхронно з ним виконується спеціальна дія.
*Перевага 4.*Приховування інформації. Коли доступом до даних здійснюється лише через методи, можна приховати деталі реалізації об'єкта. Пізніше, якщо реалізація зміниться, доведеться змінити лише реалізацію методів доступу до полів. Ті ж частини програми, які використовували цей клас, не торкнуться.
Дуже зручно розглядати об'єкти як спробу створення активних даних. Сенс, що вкладається в слова «об'єкт є активними даними», заснований на об'єктно-орієнтованій парадигмі виконання операцій, що полягає в посиланні повідомлень. У повідомленнях, що посилаються, вказується, що ми хочемо, щоб він виконав. Так, наприклад, якщо ми хочемо вивести на екран рядок, то ми посилаємо об'єкту рядка повідомлення, щоб він зобразив себе. У цьому випадку рядок – це вже не пасивний шматок тексту, а активна одиниця, яка знає, як правильно робити над собою різні дії.
Однією з фундаментальних концепцій ООП є поняття успадкування класів, яке встановлює між двома класами відносини «батько-нащадок».
Спадкування — відношення найвищого рівня та відіграє важливу роль на стадії проектування. Спадкування - це визначення класу і потім використання його для побудови ієрархії похідних класів, причому кожен клас-нащадок успадковує від класу-предка інтерфейс всіх предків-класів у вигляді доступу до коду їх методів і даних. При цьому можливо перевизначення або додавання як нових даних, так і методів.
Клас-предок -це клас, що надає свої можливості та характеристики іншим класам через механізм успадкування. Клас, який використовує характеристики іншого класу у вигляді спадкування, називається його класом-нащадком.
Отже, успадкування виявляється в тому, що будь-який клас-нащадок має доступ або, іншими словами, успадковує практично всі ресурси (методи, поля та властивості) батьківського класу та всіх предків до найвищого рівня ієрархії.
Розглянемо, як інформація, що міститься в класі-нащадку, може перевизначати інформацію, що успадковується від предків. Дуже часто при реалізації такого підходу метод, що відповідає підкласу, має те саме ім'я, що й відповідний метод у батьківському класі. При цьому для пошуку методу, що підходить для обробки повідомлення, використовується наступне правило. Пошук методу, який викликається у відповідь певне повідомлення, починається з методів, що належать класу одержувача. Якщо відповідний метод не знайдено, пошук триває до батьківського класу. Пошук просувається вгору ланцюжком батьківських класів до того часу, доки знайдено потрібний метод чи доки вичерпана послідовність батьківських класів. У першому випадку виконується знайдений метод, у другому видається повідомлення про помилку. У багатьох мовах програмування вже на етапі компілювання, а

не при виконанні програми визначається, що відповідного методу немає взагалі і видається повідомлення про помилку.

Семантично успадкування визначає ставлення типу «is-a». Наприклад, ведмідь є ссавець, будинок є нерухомість і «швидке сортування» є алгоритм, що сортує. Таким чином, спадкування породжує ієрархію «узагальнення — спеціалізація», в якій підклас є спеціалізованим окремим випадком свого суперкласу. «Лакмусовий папірець» успадкування — зворотна перевірка: так, якщо В немає А, то В не варто виробляти від А.

Повторне використання — це використання в програмі класу для створення екземплярів або як базовий для створення нового класу, що успадковує частину або всі характеристики батьків. Породжуючи класи від базових, ви ефективно повторно використовуєте код базового класу для потреб. Повторне використання скорочує обсяг коду, який необхідно написати та відтестувати під час реалізації програми, що скорочує обсяги праці.

Таким чином, успадкування виконує в ООП кілька важливих функцій:

• моделює концептуальну структуру предметної галузі;

• заощаджує описи, дозволяючи використовувати їх багаторазово для завдання різних класів;

• забезпечує покрокове програмування великих систем шляхом багаторазової конкретизації класів.

Ряд мов, наприклад Object Pascal, опис якого дається у додатку 4, підтримує модель спадкування, відому як просте спадкування і яка обмежує кількість батьків конкретного класу одним. Іншими словами, певний користувачем клас має лише одного з батьків. Схема ієрархії класів у цьому випадку є рядом одиночних дерев (hierarchical classification).

Більш потужна модель складного успадкування, звана множинним успадкуванням, у якій кожен клас може, у принципі, породжуватися відразу від кількох батьківських класів, успадковуючи поведінку всіх своїх предків. Множинне успадкування не підтримується у Delphi, але підтримується у Visual C++ та інших мов. При множинному успадкування складається вже не схема ієрархії, а мережа, яка може включати дерева з кронами, що зрослися.

Зазвичай, якщо об'єкти відповідають конкретним сутностям реального світу, то класи є абстракціями, що виступають у ролі понять. Між класами, як між поняттями, існує ієрархічне відношення конкретизації, що пов'язує клас із класом-нащадком. Це ставлення реалізується у системах ОВП механізмом наслідування. Спадкування – це здатність одного класу використовувати характеристики іншого.

Спадкування дозволяє практично без обмежень послідовно будувати та розширювати класи, створені вами чи кимось. Починаючи з найпростіших класів можна створювати похідні класи за складністю, що не тільки легкі при налагодженні, але і прості за внутрішньою структурою.

Множинне успадкування багато аналітиків вважають «шкідливим» механізмом, що призводить до складних проблем проектування та реалізації. У мовах з відсутнім множинним успадкуванням цілей множинного успадкування досягають агрегування об'єктів з додатковим делегуванням повноважень.

На агрегуванні засновано роботу таких систем візуального програмування, як Delphi, C++ Builder. У цих системах є об'єкт користувача, що породжує клас-форма (порожнє вікно Windows). Системи забезпечують підключення до форми через покажчики необхідних користувачеві об'єктів, наприклад кнопок, вікон редакторів і т. д. При перемальовуванні форми на екрані монітора одночасно з нею перемальовуються зображення агрегованих об'єктів. Більш того, при активізації форми агреговані об'єкти також стають активними: кнопки починають натискатися, а вікна редакторів можна починати вводити інформацію.

Одним із базових понять технології ООП є поліморфізм. Термін "поліморфізм" має грецьке походження і означає приблизно "багато форм" (poly - багато, morphos - форма).

*Поліморфізм*це засіб надання різних значень одному й тому події залежно від типу оброблюваних даних, т. е. поліморфізм визначає різні форми реалізації однойменного дії (див. рис. 8.2.).

Метою поліморфізму стосовно об'єктно-орієнтованого програмування є використання одного імені завдання загальних для класу дій, причому кожен об'єкт має можливість по-своєму реалізувати цю дію своїм власним, відповідним йому кодом.
Поліморфізм є передумовою для розширюваності об'єктно-орієнтованих програм, оскільки він надає спосіб старим програмам сприймати нові типи даних, які були визначені під час написання програми.
Протилежність поліморфізму називається мономорфізмом; він характерний для мов із сильною типізацією та статичним зв'язуванням (Ada).
У більш загальному трактуванні поліморфізм - це здатність об'єктів, що належать до різних типів, демонструвати однакову поведінку; здатність об'єктів, що належать до одного типу, демонструвати різну поведінку.
Розглянемо «вироджений приклад» поліморфізму. У MS DOS є поняття "номер переривання", за яким ховається адреса в пам'яті. Помістіть в ту ж комірку іншу адресу — і програми почнуть викликати процедуру з іншим ім'ям і з іншого модуля. Як очевидно з прикладу, принцип поліморфізму можна реалізувати й над об'єктно-орієнтованих програмах.
Ряд авторів книг з теорії об'єктно-орієнтованого проектування співвідносять термін «поліморфізм» з різними поняттями, наприклад, поняттям навантаження; для позначення одного-двох чи більшої кількості механізмів поліморфізму; чистий поліморфізм.
*Навантаження функцій.*Одним із застосувань поліморфізму C++ є навантаження функцій. Вона дає тому самому імені функції різні значення. Наприклад, вираз a + b має різні значення, залежно від типів змінних a і b (припустимо, якщо це числа, то «+» означає додавання, а якщо рядки, то склеювання цих рядків або взагалі додавання комплексних чисел, якщо a і b комплексного типу). Перевантаження оператора «+» для типів, які визначають користувач, дозволяє використовувати їх у більшості випадків так само, як і вбудовані типи. Двом або більше функцій (операція - це теж функція) може бути дано те саме ім'я. Але при цьому функції повинні відрізнятися сигнатурою (або типами параметрів або їх числом).
Поліморфний метод C++ називається віртуальною функцією, що дозволяє отримувати відповіді на повідомлення, адресовані об'єктам, точний вид яких невідомий. Така можливість є наслідком пізнього зв'язування. При пізньому зв'язуванні адреси визначаються динамічно під час виконання програми, а не статично під час компіляції як у традиційних мовах, що компілюються, в яких застосовується раннє зв'язування. Сам процес зв'язування полягає у заміні віртуальних функцій на адреси пам'яті.
Віртуальні методи визначаються батьківському класі, а похідних класах відбувається їх довизначення і їм створюються нові реалізації. Основою віртуальних методів та динамічного поліморфізму є покажчики на похідні класи. Працюючи з віртуальними методами повідомлення передаються як покажчики, які вказують на об'єкт замість прямої передачі об'єкту.
Практичний зміст поліморфізму полягає в тому, що він дозволяє надсилати загальне повідомлення про збір даних будь-якому класу, причому і батьківський клас, і класи-нащадки дадуть відповідне відповідне повідомлення, оскільки похідні класи містять додаткову інформацію. Програміст може зробити регулярним процес обробки несумісних об'єктів різних типів за наявності такого поліморфного методу.

8.6. ЕТАПИ І МОДЕЛІ ОБ'ЄКТНО-ОРІЄНТУВАНОЇ ТЕХНОЛОГІЇ

Чому на початку процесу проектування роботу починають із аналізу функціонування чи поведінки системи? Справа в тому, що поведінка системи зазвичай відома задовго до інших її властивостей. Програма повинна виконувати набір дій згідно з виявленими її функціями. Процес розробки моделі у формі функціональної специфікації вже було викладено раніше в гол. 5.

Об'єктно-орієнтована технологія створення програм ґрунтується на так званому об'єктному підході. Одним із проявів цього підходу є те, що спочатку досить довго створюються та оптимізуються об'єктна модель та інші моделі і лише потім здійснюється кодування.
Зазвичай проектована програмна система спочатку представляється як трьох взаємозалежних моделей:
1) об'єктної моделі, що представляє статичні, структурні аспекти системи;
2) динамічної моделі, що визначає роботу окремих частин системи;
3) функціональної моделі, в якій розглядається взаємодія окремих частин системи (як за даними, так і з управління) у процесі її роботи.
Ці три види моделей повинні дозволити розглядати три взаємно-ортогональні уявлення системи в одній системі позначень.
Об'єктна модель на пізніших етапах проектування доповнюється моделями, що відображають як логічну (класи та об'єкти), так і фізичну структуру системи (процеси та поділ на компоненти, файли чи модулі).
Оскільки розробки об'єктно-орієнтованого проекту використовується безліч моделей, які необхідно ув'язати в єдине ціле, далі в гол. 10 розглядаються засоби автоматизації складання, верифікації (перевірки) та графічної візуалізації цих моделей.
Процес побудови об'єктної моделі включає наступні, можливо, повторювані до досягнення прийнятної якості моделі етапи:
1) визначення об'єктів;
2) підготовку словника об'єктів з метою виключення подібних (синонімічних) понять та уточнення імен, класифікацію об'єктів, виділення класів;
3) визначення взаємозв'язків між об'єктами;
4) визначення атрибутів об'єктів та методів (визначення рівнів доступу та проектування інтерфейсів класів);
5) Вивчення якості моделі.
Тепер, використовуючи функціональну модель, можна розпочинати роботу з динамічною моделлю, наділяючи об'єкти необхідними методами та даними.
Моделі, розроблені на першій фазі життєвого циклу системи, продовжують використовуватися на всіх наступних його фазах, полегшуючи програмування системи, її налагодження та тестування, супровід та подальшу модифікацію.
Об'єктна модель визначає структуру об'єктів, що становлять систему, їх атрибути, операції, взаємозв'язки коїться з іншими об'єктами. В об'єктній моделі повинні бути відображені ті поняття та об'єкти реального світу, які важливі для системи, що розробляється. В об'єктній моделі відображається насамперед прагматика системи, що розробляється, що виражається у використанні термінології прикладної області, пов'язаної з використанням системи, що розробляється.
Прагматика визначається метою розробки програмної системи: для обслуговування покупців залізничних квитків, управління роботою аеропорту, обслуговування чемпіонату світу з футболу тощо. У формулюванні мети беруть участь предмети та поняття реального світу, що мають відношення до програмної системи, що розробляється.
Об'єктну модель можна описати так:
1) основні елементи моделі - об'єкти та повідомлення;
2) об'єкти створюються, використовуються і знищуються подібно до динамічних змінних у звичайних мовах програмування;
3) виконання програми полягає у створенні об'єктів та передачі їм послідовності повідомлень.
Об'єктна модель виходить з чотирьох основних принципах: абстрагуванні; інкапсуляції; модульності; ієрархії.
Ці принципи є головними в тому сенсі, що без кожного з них модель не буде по-справжньому об'єктно-орієнтованою.

Абстрагування концентрує увагу на зовнішніх особливостях об'єкта і дозволяє відокремити найсуттєвіші особливості поведінки від несуттєвих. Вибір правильного набору абстракцій для заданої предметної області є головне завдання об'єктно-орієнтованого проектування. Усі абстракції мають як статичними, і динамічними властивостями. Наприклад, файл як об'єкт потребує певного обсягу пам'яті на конкретному пристрої, має ім'я та зміст. Ці атрибути статичні властивості. Конкретні значення кожного з перерахованих властивостей динамічні і змінюються в процесі використання об'єкта: файл можна збільшити або зменшити, змінити його ім'я і вміст.

Абстракція та інкапсуляція доповнюють один одного: абстрагування спрямоване на поведінку об'єкта, що спостерігається, а інкапсуляція займається внутрішнім пристроєм. Найчастіше інкапсуляція доповнюється приховуванням інформації, т. е. маскуванням всіх внутрішніх деталей, які впливають зовнішнє поведінка. Об'єктний підхід передбачає, що власні ресурси, якими можуть лише маніпулювати методи самого об'єкта, приховані від зовнішніх компонент.

При об'єктно-орієнтованому проектуванні необхідно фізично розділити класи та об'єкти, що становлять логічну структуру проекту. Такий поділ робить можливим повторно використовувати в нових проектах код модулів, написаних раніше. Модулю у цьому контексті відповідає окремий файл вихідного тексту. На вибір розбиття на модулі можуть впливати деякі зовнішні обставини. При колективній розробці програм розподіл роботи здійснюється, як правило, за модульним принципом, та правильний поділ проекту мінімізує зв'язки між учасниками.

Таким чином, принципи абстрагування, інкапсуляції та модульності є взаємодоповнювальними. Об'єкт логічно визначає межі певної абстракції, а інкапсуляція та модульність роблять їх фізично непорушними.

Маючи виявлені об'єкти, можна розпочати виявлення класів. Класи найчастіше будуються поступово, починаючи від простих батьківських класів і до складніших. Безперервність процесу заснована на спадкування. Щоразу, коли з попереднього класу виробляється наступний, похідний клас успадковує якісь чи всі батьківські якості, додаючи до них нові. Завершений проект може включати десятки і сотні класів, але часто всі вони виготовлені від кількості батьківських класів.

8.7. ЯКИМИ БУВАЮТЬ ОБ'ЄКТИ З ПРИСТРОЮ

Під патернами проектування розуміється опис взаємодії об'єктів і класів, адаптованих для вирішення спільної задачі проектування в конкретному контексті. Паттерн проектування - це зразок, типове рішення будь-якого механізму об'єктно-орієнтованої програми. Паттерни створювалися кілька років колективом із єдиною метою зрівнювання шансів хороший проект досвідчених і дуже досвідчених проектувальників. За словами архітектора Крістофера Олександра, «будь-який патерн описує завдання, яке знову і знову виникає в нашій роботі, а також принцип її вирішення, причому таким чином, що це рішення можна потім використати мільйон разів, нічого не винаходячи заново».

У випадку патерн складається з чотирьох основних елементів: імені, завдання, рішення, результатів.

Ім'я. Послався на нього, можна одразу описати проблему проектування, її вирішення та їх наслідки. За допомогою словника патернів можна вести обговорення з колегами, згадувати патерни у документації.

Завдання - Опис того, коли слід застосовувати патерн. Тут може описуватися конкретна проблема проектування, наприклад спосіб представлення алгоритмів як об'єктів. Іноді зазначається, які структури класів чи об'єктів свідчать про негнучкий дизайн. Також може включатися перелік умов, при виконанні яких є сенс застосовувати цей патерн.

Рішення - Опис елементів дизайну, відносин між ними, функцій кожного елемента. Конкретний дизайн або реалізація не маються на увазі, оскільки патерн - це шаблон, що застосовується в різних ситуаціях. Просто дається абстрактний опис задачі проектування і

того, як вона може бути вирішена за допомогою якогось дуже узагальненого поєднання елементів (у нашому випадку класів та об'єктів).

Результати — це наслідки застосування патерну та різноманітних компромісів. Хоча при описі проектних рішень про наслідки часто не згадують, знати про них необхідно, щоб можна було здійснити вибір різних варіантів та оцінити переваги та недоліки обраного патерну. Тут йдеться про вибір мови та реалізації. Оскільки в об'єктно-орієнтованому проектуванні повторне використання найчастіше є важливим фактором, то до результатів слід відносити і вплив на рівень гнучкості, розширюваності та переносимості системи. Перелік усіх наслідків допоможе вам зрозуміти та оцінити їхню роль. Нижче наведено повний список розділів опису патерну:

- Назва та класифікація патерну (назва патерну має чітко відображати його призначення);
- призначення (коротка відповідь на такі питання: які функції патерну, його обґрунтування та призначення, яку конкретну задачу проектування можна вирішити за його допомогою);
- відомий також під ім'ям (інші поширені назви патерну, якщо такі є);
- Мотивація (сценарій, що ілюструє завдання проектування і те, як вона вирішується даною структурою класу або об'єкта. Завдяки мотивації, можна краще зрозуміти наступний, більш абстрактний опис патерну);
- застосовність (опис ситуацій, в яких можна застосовувати даний патерн; приклади проектування, які можна покращити за його допомогою);
- структура (графічне представлення класів у патерні з використанням нотації, заснованої на методиці ЗМТ, а також з використанням діаграм взаємодій для ілюстрації послідовностей запитів та відносин між об'єктами);
- Учасники (класи або об'єкти, задіяні в даному патерні проектування, та їх функції);
- Відносини (взаємодія учасників для виконання своїх функцій);
- результати (наскільки патерн задовольняє поставленим вимогам? Результати застосування, компроміси, на які доводиться йти. Які аспекти поведінки системи можна незалежно змінювати, використовуючи цей патерн?);
- реалізація (складності і так звані «підводні камені» при реалізації патерну; поради та рекомендовані прийоми; чи є у даного патерну залежність від мови програмування?);
— приклад коду (фрагмент коду, що ілюструє можливу реалізацію мовами C++ чи Smalltalk);
- відомі застосування (можливості застосування патерну в реальних системах; даються щонайменше два приклади з різних областей);
- Споріднені патерни (зв'язок інших патернів проектування з даними, важливі відмінності, використання даного патерну в поєднанні з іншими).

Каталог містить 23 патерни.

Нижче для зручності перераховані їхні імена та призначення.

1. Abstract Factory (абстрактна фабрика). Надає інтерфейс створення сімейств, пов'язаних між собою, чи незалежних об'єктів, конкретні класи яких невідомі.
2. Adapter (Адаптер). Перетворює інтерфейс класу на деякий інший інтерфейс, очікуваний клієнтами. Забезпечує спільну роботу класів, яка була б неможлива без цього патерну через несумісність інтерфейсів.
3. Bridge (міст). Відокремлює абстракцію від реалізації, завдяки чому з'являється можливість незалежно змінювати те й інше.
4. Builder (будівельник). Відокремлює конструювання складного об'єкта від його уявлення, дозволяючи використовувати той самий процес конструювання до створення різних уявлень.
5. Chain of Responsibility (ланцюжок обов'язків). Можна уникнути жорсткої залежності відправника запиту від його одержувача, при цьому запит починає оброблятися один з декількох об'єктів. Об'єкти-одержувачі зв'язуються в ланцюжок, і запит передається ланцюжком, поки якийсь об'єкт його не обробить.
6. Command (команда). Інкапсулює запит як об'єкта, дозволяючи цим параметризовывать клієнтів типом запиту, встановлювати черговість запитів, протоколювати їх і підтримувати скасування виконання операцій.

7. Composite (компонувальник). Групує об'єкти в деревоподібні структури уявлення ієрархій типу «частина-целое». Дозволяє клієнтам працювати з одиничними об'єктами так само, як із групами об'єктів.
8. Декоратор (декоратор). Динамічно покладає нові функції. Декоратори застосовуються для розширення наявної функціональності та є гнучкою альтернативою породженню підкласів.
9. Facade (фасад). Надає уніфікований інтерфейс до множини інтерфейсів в деякій підсистемі. Визначає інтерфейс вищого рівня, що полегшує роботу з підсистемою.
10. Factory Method (фабричний метод). Визначає інтерфейс для створення об'єктів, при цьому вибраний клас інстанцується підкласами.
11. Flyweight (пристосуванець). Використовує поділ для ефективної підтримки великої кількості дрібних об'єктів.
12. Interpreter (інтерпретатор). Для заданої мови визначає подання її граматики, а також інтерпретатор речень мови, що використовує це уявлення.
13. Iterator (ітератор). Дає можливість послідовно обійти всі елементи складового об'єкта, не розкриваючи його внутрішнього уявлення.
14. Mediator (посередник). Визначає об'єкт, в якому інкапсульовано знання про те, як взаємодіють об'єкти з деякої множини. Сприяє зменшенню кількості зв'язків між об'єктами, дозволяючи їм працювати без явних посилань один на одного. Це, своєю чергою, дає можливість незалежно змінювати схему взаємодії.
15. Memento (зберігач). Дозволяє, не порушуючи інкапсуляції, отримати та зберегти у зовнішній пам'яті внутрішній стан об'єкта, щоб пізніше об'єкт можна було відновити точно у такому стані.
16. Observer (спостерігач). Визначає між об'єктами залежність типу "один до багатьох", так що при зміні стану одного об'єкта всі залежні від нього отримують повідомлення і автоматично оновлюються.
17. Prototype (прототип). Описує види об'єктів, що створюються за допомогою прототипу і створює нові об'єкти шляхом його копіювання.
18. Proxy (заступник). Підміняє інший об'єкт контролю доступу до нього.
19. Singleton (одинак). Гарантує, що певний клас може мати лише один екземпляр і надає глобальну точку доступу до нього.
20. State (стан). Дозволяє об'єкту варіювати свою поведінку при зміні внутрішнього стану. У цьому складається враження, що змінився клас об'єкта.
21. Strategy (стратегія). Визначає сімейство алгоритмів, інкапсулюючи їх усі і дозволяючи підставляти один замість одного. Можна змінювати алгоритм незалежно від клієнта, який користується ним.
22. Template Method (шаблонний метод). Визначає скелет алгоритму, перекладаючи відповідальність деякі його кроки на підкласи. Дозволяє підкласам перевизначати кроки алгоритму, не змінюючи його загальну структуру.
23. Visitor (відвідувач). Подає операцію, яку треба виконати над елементами об'єкта. Дозволяє визначити нову операцію, не змінюючи класи елементів, яких він застосовується.

8.8. ПРОЕКТНА ПРОЦЕДУРА ОБ'ЄКТНО-ОРІЄНТОВАНОГО ПРОЕКТУВАННЯ ПО Б. СТРАУСТРУПУ

8.8.1. Укрупнений виклад проектної процедури Б. Страуструпа

Б. Страуструп - автор об'єктно-орієнтованої мови програмування С++ з множинним успадкуванням. У Б. Страуструпа в описах методики проектування вводиться одиниця проектування — «компонента». Під компонентою розуміється безліч класів, об'єднаних деякою логічною умовою, іноді це загальний стиль програмування або опису, іноді сервіс, що надається. Ряд авторів замість терміна "компонента" використовують термін "модуль".

Структура компонентів проектується використанням ітераційного наростаючого процесу. Зазвичай для отримання проекту, який можна впевнено використати для первинної реалізації або повторної, потрібно кілька разів зробити послідовність з наступних чотирьох кроків.
*Крок 1*Виділення понять (класів, що породжують об'єкти) та встановлення основних зв'язків між ними.
*Крок 2.*Уточнення класів із визначенням наборів операцій (методів) для кожного.
*Крок 3*Уточнення класів із точним визначенням їх залежностей з інших класів. З'ясовується успадкування та використання залежностей.
*Крок 4.*Завдання класових інтерфейсів. Точніше визначаються відносини класів. Методи поділяються на загальні та захищені. Визначаються типи операцій із класами.

8.8.2. Крок 1. Виділення понять та встановлення основних зв'язків між ними

Виділення об'єктів проводиться під час процесу уявного системи. Часто це відбувається як цикл питань «що/хто». Команда програмістів визначає: що робити? Це негайно призводить до питання: хто виконуватиме дію? Тепер програмна система значною мірою стає схожою на якусь організацію. Дії, які мають бути виконані, присвоюються деякому програмному об'єкту як його обов'язки.
Поняття (об'єкти) відповідають класам, що породжують, і можуть мати форму у вигляді іменників і, як екзотика, дієслів та прикметників.
Часто кажуть, що поняття у формі іменників відіграють роль класів та об'єктів, що використовуються у програмі. Наприклад: трактор, редуктор, гайка, редактор, кнопка, файл, матриця. Це справді так, але це лише початок.
Дієсловаможуть представляти операції над об'єктами або звичайні (глобальні) функції, що виробляють нові значення, виходячи зі своїх параметрів або навіть класи. Як приклад можна розглядати маніпулятори, запропоновані А. Кенігом. Суть ідеї маніпулятора в тому, що створюється об'єкт, який можна передавати будь-куди і який використовується як функція. Такі дієслова, як «повторити» або «вчинити», можуть бути представлені ітеративним об'єктом або об'єктом, що є операцією виконання програми в базах даних.
Навіть прикметники можна успішно представляти за допомогою класів. Наприклад, такими класами можуть бути: «зберігаючий», «паралельний», «реєстровий», «обмежений», а також класи, які допоможуть розробнику або програмісту, задавши віртуальні базові класи, специфікувати та вибрати потрібні властивості для класів, що проектуються пізніше.
Усі класи можна умовно поділити на дві групи: класи з предметної (прикладної) області та класи, які є артефактами реалізації або абстракціями періоду реалізації.
Класи з предметної (прикладної) області-безпосередньо відображають поняття з прикладної області, тобто поняття, які використовує кінцевий користувач для описів своїх завдань та методів їх вирішення.
Найкращий засіб для пошуку цих понять/класів — грифельна дошка, а найкращий метод першого уточнення — бесіда зі спеціалістами з програми або просто з друзями. Обговорення необхідно створити початковий словник термінів і понятійну структуру.
Головне в хорошому проекті — прямо відобразити якесь поняття «реальності», тобто вловити поняття в галузі додатка класів, уявити взаємозв'язок між класами строго певним способом, наприклад, за допомогою успадкування, і повторити ці дії на різних рівнях абстракції.
Класи, що є артефактами реалізації або абстракціями періоду реалізації,-це ті поняття, які застосовують програмісти та проектувальники для опису методів реалізації:
• класи, що відбивають ресурси устаткування (оперативна пам'ять, механізми управління ресурсами, дисковий простір);
• класи, які мають системні ресурси (процеси, потоки вводу-вывода);
• класи, що реалізують програмні структури (стеки, черги, списки, дерева, словники тощо);
• інші абстракції, наприклад елементи керування програмою (кнопки, меню тощо).

Добре спроектована система має містити класи, які дають можливість розглядати систему з логічно різних точок зору.
Приклад:
1) класи, що представляють поняття користувача (наприклад, легкові машини і вантажівки);
2) класи, що представляють узагальнення користувальницьких понять (засоби, що рухаються);
3) класи, які мають апаратні ресурси (наприклад, клас управління пам'яттю);
4) класи, які мають системні ресурси (наприклад, вихідні потоки);
5) класи, що використовуються для реалізації інших класів (наприклад, списки, черги);
6) вбудовані типи даних та структури управління.
У великих системах дуже важко зберігати логічний поділ типів різних класів та підтримувати такий поділ між різними рівнями абстракції. У наведеному вище перерахунку представлено три рівні абстракції:
(1+2) — представляє відображення системи користувача системи;
(3+4) - представляє машину, на якій працюватиме система;
(5+6) — є низькорівневим (з боку мови програмування) відображенням реалізації.
Чим більша система, тим більше рівнів абстракції необхідно для її опису і тим важче визначати та підтримувати ці рівні абстракції. Зазначимо, що таким рівням абстракції є пряма відповідність у природі та у різних побудовах людського інтелекту. Наприклад, можна розглядати будинок як об'єкт, який складається з атомів; молекул; дощок та цегли; стін, підлоги та стель; кімнат.
Поки вдається зберігати окремо уявлення цих рівнів абстракції, можна підтримувати цілісне уявлення про будинок. Однак якщо змішати їх, виникне нісенітниця.
Взаємини, про які ми говоримо, природно встановлюються в області додатку або (у разі повторних проходів по кроках проектування) виникають із подальшої роботи над структурою класів. Вони відбивають наше розуміння основ області застосування і найчастіше є класифікацією основних понять. Приклад такого відношення - машина з висувними сходами є вантажівка, є пожежна машина, є засіб, що рухається.

8.8.3. Крок 2. Уточнення класів із визначенням набору операцій (методів) для кожного

Насправді не можна розділити процеси визначення класів та з'ясування того, які операції для них потрібні. Однак на практиці вони різняться, оскільки при визначенні класів увага концентрується на основних поняттях, не зупиняючись на програмістських питаннях їх реалізації, тоді як при визначенні операцій насамперед зосереджуються на тому, щоб задати повний та зручний набір операцій. Часто дуже важко поєднати обидва підходи, особливо, враховуючи, що пов'язані класи треба проектувати одночасно.
Можливо кілька підходів до процесу визначення набору операцій. Пропонуємо таку стратегію:
— розгляньте, яким чином об'єкт класу створюватиметься, копіюватиметься (якщо потрібно) і знищуватиметься;
- Визначте мінімальний набір операцій, необхідний для поняття, представленого класом;
— розгляньте операції, які можуть бути додані для зручності запису, та увімкніть лише кілька справді важливих;
- Розгляньте, які операції можна вважати тривіальними, тобто такими, для яких клас виступає в ролі інтерфейсу для реалізації похідного класу;
- Розгляньте, якої спільності іменування і функціональності можна досягти для всіх класів компонента.
Очевидно, що це стратегія мінімалізму. Набагато простіше додати будь-яку функцію, яка приносить відчутну користь, і зробити всі операції віртуальними. Але чим більше функцій, тим більша ймовірність, що вони не будуть використовуватися, накладуть певні обмеження

на реалізацію та ускладнять еволюцію системи. Набагато легше включити в інтерфейс ще одну функцію, як тільки встановлено потребу в ній, ніж видалити її звідти, коли вона стала звичною.
Причина, через яку ми вимагаємо явного прийняття рішення про віртуальність цієї функції, не залишаючи його на стадію реалізації, у тому, що, оголосивши функцію віртуальної, суттєво вплинемо на використання її класу та на взаємини цього класу з іншими.
При визначенні набору операцій (методів) більше уваги слід приділяти тому, що треба зробити, а чи не тому, як це зробити.
Іноді корисно класифікувати операції класу після того, як вони працюють з внутрішнім станом об'єктів:
1) базові операції: конструктори, деструктори, операції копіювання;
2) селектори: операції, що не змінюють стану об'єкта;
3) модифікатори: операції, що змінюють стан об'єкта;
4) операції перетворень, т. е. операції, які породжують об'єкт іншого типу, з значення (стану) об'єкта, якого вони застосовуються;
5) повторювачі: операції, які відкривають доступ до об'єктів класу чи використовують послідовність об'єктів.
Крім уже перерахованих груп методів, класи можуть бути введені додаткові методи самотестування та перевірки коректності даних. Це не є розбиття на ортогональні групи операцій. Наприклад, повторювач може бути спроектований як селектор чи модифікатор. Виділення цих груп просто призначено допомогти у процесі проектування класу інтерфейсу. Звісно, допустима та інша класифікація.

8.8.4. Крок 3. Уточнення класів з точним визначенням їхньої залежності від інших класів

Види взаємин між класами може бути такими: відносини наслідування; відносини включення; відносини використання; запрограмовані відносини.
Ще одне взаємини - відношення включення {агрегування) - клас містить у вигляді члена об'єкт або покажчик на об'єкт іншого класу. Дозволяючи об'єктам містити покажчики інші об'єкти, можна створювати звані «ієрархії об'єктів». Такі реалізації альтернативно доповнюють можливість використання ієрархії класів.
Дуже важливим при проектуванні є питання: яке відношення вибрати – агрегації (включення) чи спадкування. У принципі, ці методи взаємозамінні, крім випадку, коли використовується пізніше зв'язування. Найбільш кращим є той варіант, в якому найбільш точно моделюється навколишня дійсність, тобто якщо поняття X є частиною поняття Y, то використовується включення. Якщо поняття X більш загальне, ніж Y, то спадкування.
Для складання та розуміння проекту часто необхідно знати, які класи та яким способом вони використовуються, іншими словами, відносини використання. Можливо, таким чином класифікувати ті способи, за допомогою яких клас X може використовувати клас Y.
- *X* використовує Y;
- *X* викликає функцію-член (метод) Y;
- *X* читає член Y;
- *X* пише член Y;
- *X* створює Y;
- *X* розміщує змінну з Y

Аналіз подібних взаємозв'язків дозволяє виявити потреби у певних методах класів чи, навпаки, виявити їх непотрібність.
Запрограмовані відносини- Ті відносини проекту, які не можуть бути прямо представлені у вигляді конструкцій мови. Допустимо, у проекті застережено, що кожна операція, не реалізована в класі A, повинна обслуговуватися об'єктом класу B. До запрограмованих відносин відносять також операції перетворення типів. Слід, наскільки можна, уникати

застосування цього виду відносин через ускладнення реалізації. Ідеальний клас повинен мінімальною мірою залежати від решти світу. Отже, слід намагатися мінімізувати залежність.

8.8.5. Крок 4. Завдання класових інтерфейсів

Схуємо подробиці реалізації за фасадом інтерфейсу. Об'єкт інкапсулює поведінку, якщо він вміє виконувати деякі дії, але подробиці, як це робиться, залишаються прихованими за фасадом інтерфейсу. Ця ідея була сформульована фахівцем з інформатики Девідом Парнасом у вигляді правил, які часто називають принципами Парнаса.
Правило 1. Розробник програми повинен надавати користувачеві всю інформацію, яка потрібна для ефективного використання програми, і нічого, крім цього.
Правило 2 Розробник програмного забезпечення повинен знати тільки потрібну поведінку об'єкта і нічого, крім цього.
Наслідок принципу відокремлення інтерфейсу від у тому, що програміст може експериментувати з різними алгоритмами, не торкаючись інші класи об'єктів програми.
На цьому кроці дається чіткий опис класів, їх даних та методів (опускаючи реалізацію та, можливо, приховані методи). Всі методи визначають точні типи параметрів.
Ідеальний інтерфей спредставляє користувачу повний та послідовний набір понять; узгоджений з усіма частинами компоненти; не відкриває подробиці реалізації та може бути реалізований різними способами; обмежено та чітко певним чином залежить від інших інтерфейсів.
Інтерфейси класів надають повну інформацію для реалізації класів на етапі кодування.
Існує золоте правило: якщо клас не допускає принаймні двох істотно відмінних реалізацій, то щось явно не в порядку з цим класом, це просто замаскована реалізація, а не уявлення абстрактного поняття. У багатьох випадках для відповіді на запитання: «Достатньо інтерфейс класу незалежний від реалізації?» - Треба вказати, чи можлива для класу схема звичайних обчислень.

8.8.6. Розбудова ієрархії класів

Намагаючись провести класифікацію деяких нових об'єктів, ставимо такі питання: У чому схожість цього об'єкта з іншими об'єктами загального класу? У чому його розходження? Кожен клас має набір поведінок та характеристик, що його визначають. Почнемо з верхівки фамільного дерева зразка і спускатимемося по гілках, ставлячи ці питання протягом усього шляху. Вищі рівні є більш спільними, а питання більш простими. Кожен рівень є специфічнішим, ніж попередній рівень, і менш загальним.
Безперечно, це тривіальне завдання, але встановити ідеальну ієрархію класів для певного застосування дуже важко. Перш ніж написати рядок коду програми, необхідно добре подумати про те, які класи необхідні та на якому рівні. У міру того, як збільшується розуміння, може виявитися, що необхідні нові класи, які фундаментально змінюють всю ієрархію класів.
На другому та третьому кроках ітеративної процедури проектування проводиться виявлення того, наскільки адекватно класи та їхня ієрархія підходять по суті проекту. Проектувальники змушені реорганізовувати, покращувати проект та повторювати всі кроки спочатку, і так доти, доки якість проекту не буде задовільною.
При перебудові ієрархії класів використовуються чотири процедури: розщеплення класу на два і більше; абстрагування (узагальнення); злиття; аналіз можливості використання існуючих розробок.
Розщеплення застосовується у таких випадках:
1) якщо є складний клас, іноді має сенс поділити його на кілька простих класів і тим самим забезпечити поетапну розробку;

2) клас містить низку незв'язаних між собою функцій або набір незалежних один від одного даних.
Узагальнення — виявлення групи класів загальних властивостей і винесення в загальний базовий клас. Ознаки необхідності узагальнення такі:
1) загальна схема використання;
2) подібність між наборами операцій;
3) подібність реалізацій;
4) ці класи часто фігурують разом у дискусіях щодо проекту.
Злиття- Об'єднання кількох невеликих, але тісно взаємодіючих класів в один. Таким чином, взаємодія буде прихована у реалізації нового класу.
Використання існуючих розробок.Відокремлений клас або група класів із вже існуючого проекту може бути легко інтегрована до нового класу. Проте така інтеграція вносить певні обмеження у структуру системи і може зашкодити ефективності розробки самої програми. Виробники систем об'єктно-орієнтованого програмування постачають системи із сумісними бібліотеками класів. Очевидно, що більше готових бібліотечних класів буде використано у програмі, то менше коду доведеться писати при реалізації програми.

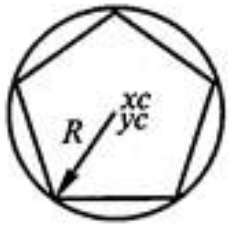
8.8.7. Звід правил

У розглянутих раніше темах не було дано настійних та конкретних рекомендацій щодо проектування. Це відповідає переконанню, що немає «єдино вірного рішення». Принципи та прийоми слід застосовувати такі, які найкраще підходять для вирішення конкретних завдань. Для цього потрібен смак, досвід та розум. Проте можна вказати деяке зведення правил (евристичних прийомів), яке розробник може використовувати як орієнтири, поки не буде достатньо досвідчений, щоб виробити найкращі правила. Нижче наведено зведення таких евристичних правил.
Правило 1. Дізнайтеся, що вам належить створити.
Правило 2 Ставте певні та відчутні цілі.
Правило 3 Не намагайтеся за допомогою технічних прийомів вирішити соціальні проблеми.
Правило 4 Розраховуйте на великий термін у проектуванні та управлінні людьми.
Правило 5 Використовуйте існуючі системи як моделі, джерело натхнення та відправну точку.
Правило 6 Проектуйте з розрахунку на зміни: гнучкість, розширюваність, переносимість, повторне використання.
Правило 7 Документуйте, пропонуйте та підтримуйте повторно використовувані компоненти.
Правило 8 Заохочуйте та винагороджуйте повторне використання: проектів, бібліотек, класів.
Правило 9 Зосередьтеся на проектуванні компонентів.
Правило 10 Використовуйте класи для представлення понять.
Правило 11 Визначайте інтерфейси так, щоб відкрити мінімальний обсяг інформації, необхідної для інтерфейсу.
Правило 12 Проводіть строгу типізацію інтерфейсів завжди, коли це можливо.
Правило 13 Використовуйте в інтерфейсах типи в області програми завжди, коли це можливо.
Правило 14 Багаторазово досліджуйте та уточнюйте як проект, так і реалізацію.
Правило 15 Використовуйте найкращі доступні засоби для перевірки та аналізу проекту та реалізації.
Правило 16 Експериментуйте, аналізуйте та проводьте тестування на найможливішому ранньому етапі.
Правило 17 Прагніть до простоти, максимальної простоти, але не більше.
Правило 18 Не розростайтеся, не додавайте можливості «про всяк випадок».
Правило 19 Не забувайте про ефективність.

Правило 20 Зберігайте рівень формалізації, що відповідає розміру проекту.
Правило 21 Не забувайте, що розробники, програмісти та навіть менеджери залишаються людьми.

8.8.8. Приклад найпростішого проекту

Б. Страуструп придумав реалізацію механізму множинного успадкування і при цьому відкидав агрегування, хоч і реалізував це своєю мовою C++.
Наведений далі приклад показує неможливість здійснення вирішення наступного простого завдання двома способами рішення - з використанням множинного успадкування та агрегування. У процесі розв'язання завдань було виявлено, що у низці завдань без виконання третього кроку неможливе коректне виконання другого кроку. Таким чином, при вирішенні того самого прикладу двома способами другий і третій кроки проекту були взаємно переставлені. Також додано крок «класифікація об'єктів» (складання словника).
Перший спосіб розв'язання задачі- Використання множинного успадкування.
Постановка завдання прикладу. Вивести на екран фігуру, показану на рис. 8.4.



*Мал. 8.4.*Зображення фігури, що виводиться

Зображена на рис. 8.4 фігура складається з правильного п'ятикутника та описаного навколо нього кола, де xc, yc - координати центру описаного навколо п'ятикутника кола; R - радіус описаного навколо п'ятикутника кола.
Крім того, фігура малюється заданим кольором.
Слід зазначити, що завдання можна вирішити декількома способами.
*Крок 1а.*Визначення об'єктів та виявлення їх властивостей.
Об'єкт- Малюнок. Властивості об'єкта:
- Радіус кола (R);
-координати центру кола (xc; yc);
-колір ліній.
Об'єкт- П'ятикутник. Властивості об'єкта:
- радіус описаного навколо нього кола (R);
-координати центру описаного навколо нього кола (xc; yc):
- Колір лінії.
Об'єкт- Коло. Властивості об'єкта:
- Радіус (R);
-координати центру (xc; yc);
-колір лінії.
Розв'язання задачі прикладу з використанням множинного успадкування.
*Крок 1б.*Класифікація об'єктів (складання словника).
П'ятикутник – центрально-симетрична фігура з п'ятьма вершинами.
Окружність - центрально-симетрична фігура, кожна точка якої віддалена від заданої точки - центру, на задану величину - радіус кола.
Отриманий граф спадкування класів зображено на рис. 8.5.

*Крок 2.*Уточнення класів із точним визначенням їх залежностей з інших класів. З'ясовується успадкування та використання залежностей.

*Мал. 8.5.*Граф успадкування класів згідно з першим способом

Оскільки П'ятикутник і Окружність – це різновиди центрально-симетричних фігур, то їм може відповідати наступна ієрархія класів. Базовий клас: Центрально-симетрична фігура із даними R, xc, yc. Класи П'ятикутник та Окружність є спадкоємцями цього класу, а клас Малюнок є спадкоємцем класів Окружність та П'ятикутник, оскільки в даній задачі малюнок є поєднанням п'ятикутника та кола.
*Крок 3*Уточнення класів із визначенням наборів операцій для кожного. Тут аналізується потреба у конструкторах, деструкторах та операціях копіювання. При цьому береться до уваги мінімальність, повнота та зручність.
*Клас Малюнок.*Примірник цього класу повинен створюватися і малюватись, а отже, в інтерфейсі класу Малюнок мають бути конструктори та функція — член малювання малюнка. Тоді отримуємо:
• конструктор без параметрів;
• конструктор з параметрами (радіус, х-координата, у-координата, колір);
• функцію-член виведення малюнка – «Накреслити».
*Клас П'ятикутник.*Примірник цього класу повинен створюватися і малюватись, а отже, в інтерфейсі класу П'ятикутник повинні бути конструктори та функція-член малювання п'ятикутника. Тоді отримуємо:
• конструктор без параметрів;
• конструктор з параметрами (радіус, х-координата, у-координата);
• функцію-член виведення п'ятикутника на екран – «Накреслити».
*Клас Окружність.*Примірник цього класу повинен створюватися і малюватись, а отже, в інтерфейсі класу Окружність повинні бути присутніми конструктори та функція-член виведення кола на екран. Тоді отримуємо:
• конструктор без параметрів;
• конструктор з параметрами (радіус, х-координата, у-координата);
• функцію-член виведення кола на екран — «Накреслити».
*Клас Центрально-симетрична фігура.*Примірник даного класу повинен містити інформацію про центрально-симетричну фігуру у вигляді даних із захищеним доступом (не інтерфейсна частина класу) та мати чисто-віртуальну функцію перемальовки разом із конструкторами. Тоді отримуємо:
• конструктор без параметрів;
• конструктор з параметрами (радіус, х-координата, у-координата);
• чисто-віртуальну функцію-член виведення зображення на екран.
*Крок 4.*Завдання класових інтерфейсів. Точніше визначаються відносини класів. Методи поділяються на загальні та захищені методи. Визначаються типи операцій із класами.
Дані, розташовані в класі Центрально-симетрична фігура (R, xc, yc), мають бути доступні класам-спадкоємцям П'ятикутник та Окружність, але недоступні «ззовні», отже рівень доступу — «захищений». У класі Центрально-симетрична постать потрібно розташувати функцію «Намалювати», яку передбачається зробити чисто-віртуальною. Класи, що

успадковують у класу Центрально-симетрична фігура, зможуть перевизначити функцію «Намалювати» малювання себе.
Оскільки обом об'єктам — екземплярам класів П'ятикутник та Окружність потрібен лише один центр на двох, то, отже, екземпляр класу Центрально-симетрична фігура має створюватися лише один, а отже, при описі успадкування у мові С++ потрібно додати зарезервоване слово virtual. Спадкування класами П'ятикутник та Окружність ознак у класу Центрально-симетрична фігура має відбуватися з відкритим рівнем доступу, інакше під час створення класу «Малюнок» ми не зможемо запустити конструктор класу верхнього рівня. Спадкування класом Малюнок ознак класів П'ятикутник та Окружність має відбуватися закрито, щоб до методів цих класів не можна було звернутися через об'єкт класу Малюнок. До успадкованих ознак додається властивість «Колір лінії», значення якого зберігатиметься у класі Малюнок. У класі Малюнок, як і у класах П'ятикутник і Окружність, можна перевизначити метод «Намалювати». Цей метод виводить зображення на екран, у ньому якраз і встановлюватиметься колір ліній, у якому малюватимуться постаті.
Другий спосіб розв'язання задачііз використанням агрегування. Оскільки кроки 1а і 1б виконуються повністю аналогічно до попереднього способу рішення, починаємо з кроку 2.
*Крок 2.*Уточнення класів із точним визначенням їх залежностей з інших класів. З'ясовується успадкування та використання залежностей.
Об'єкт малюнок складається з об'єктів п'ятикутник і коло, форма і розмір яких визначаються налаштуваннями, що задаються при створенні об'єкта малюнок, тобто можна створити два незалежні класи П'ятикутник (правильний) і Окружність, а потім екземпляри цих класів агрегувати в об'єкт малюнок - екземпляр класу Малюнок.
*Крок 3*Уточнення класів із визначенням наборів операцій для кожного. Тут аналізується потреба у конструкторах, деструкторах та операціях копіювання. При цьому береться до уваги мінімальність, повнота та зручність.
*Клас Малюнок.*Об'єкт цього класу повинен вміти створити, знищити та намалювати себе, тому інтерфейсна частина класу буде такою:
• конструктор без параметрів;
• конструктор з параметрами (радіус, х-координата, у-координата, колір);
• метод виведення малюнка на екран;
• деструктор для знищення утворених включених об'єктів.
Примітка. Увімкнення об'єктів типів П'ятикутник та Окружність відбувається у закритій, не інтерфейсній частині класу.
*Клас П'ятикутник.*Об'єкт класу П'ятикутник повинен уміти створити і малювати себе, тому інтерфейсна частина класу виглядатиме так:
• конструктор без параметрів;
• конструктор з параметрами (радіус, х-координата, у-координата);
• метод виведення п'ятикутника на екран.
*Клас Окружність.*Об'єкт класу Окружність повинен створювати і малювати сам себе, тому інтерфейсна частина класу виглядатиме так:
• конструктор без параметрів;
• конструктор з параметрами (радіус, х-координата, у-координата);
• метод виведення кола на екран.
*Крок 4.*Завдання класових інтерфейсів. Точніше визначаються відносини класів. Методи поділяються на загальні та захищені. Визначаються типи операцій із класами.
Класи Окружність і П'ятикутник повинні містити в собі змінні R, xc, yc, які повинні бути закриті для доступу; функцію-член виведення фігури на екран для доступу – відкриту (як і конструктори).
Для класу Малюнок включаються екземпляри класів П'ятикутник та Окружність є полями, тому їх потрібно приховати, щоб командувати цими об'єктами міг лише екземпляр класу Малюнок. Функцію виведення малюнка на екран, як і конструктори, необхідно зробити відкритими.

Аналіз результатів кроків 2 і 3 показує, що проектна процедура допускає попереднє виконання визначення набору операцій до визначення класових залежностей від інших класів з подальшим уточненням наборів операцій класів.

8.9. ТЕХНОЛОГІЯ ПРОЕКТУВАННЯ НА ОСНОВІ ОБОВ'ЯЗКІВ

8.9.1. RDD-технологія проектування на основі обов'язків

Далі буде викладено технологію проектування на основі обов'язків (або RDD-проектування - Responsibility-Driven-Design), запропоновану Т. Бадтом. Технологія орієнтована на малі та середні проекти. Вона заснована на поведінці систем. Ця технологія за способом мислення аналогічна розробці структури служб якоїсь організації: директора, заступників директора, служб та підрозділів.
Щоб виявити окремі об'єкти і визначити їх обов'язки, команда програмістів опрацьовує сценарій системи, тобто подумки відтворюється запуск додатка, якби воно вже було готове. Будь-яка дія, яка може статися, приписується деякому об'єкту як його обов'язок.
Як складова цього процесу корисно зображати об'єкти за допомогою CRC-карток. Назва CRC-картки утворена від слів: Component, Responsibility, Collaborator - компонент (об'єкт), обов'язки, співробітники. У міру того, як для об'єктів виявляються обов'язки, вони записуються на лицьовій стороні CRC-картки (рис. 8.6).

Компонента (название)
Greeter
(начальный экран)

Описание обязанностей, приписанных данной компоненте
Вывести на экран заставку
Передать управление другой компоненте:
1) базе данных рецептов:
2) менеджеру планирования

Компоненты, сотрудничающие с данной компонентой:
база данных рецептов (*Recipe Database*)
менеджер планирования (*Plan Manager*)

*Мал. 8.6.*Зразок CRC-картки

Під час опрацювання сценарію корисно розділити CRC-картки між різними членами проектної групи. Людина, яка має картку, яка є певним об'єктом, записує його обов'язки та виконує функції замінника програмної системи, передаючи «управління» наступному члену команди, коли програмна система потребує послуг інших об'єктів.
Переваги CRC-карток у тому, що вони недорогі і можна прати з них інформацію. Це стимулює експериментування, оскільки альтернативні проекти можуть бути випробувані, вивчені та відкинуті з мінімальними витратами. Фізичне поділ карток стимулює інтуїтивне розуміння важливості логічного поділу класів об'єктів. Невеликий розмір картки служить гарною оцінкою приблизної складності окремого класу об'єкта. Об'єкт, якому приписується більше завдань, ніж може поміститися на картці, ймовірно, є надмірно складним. Можливо, слід переглянути поділ обов'язків чи розбити об'єкт на два.

8.9.2. Починаємо з аналізу функціонування. Навчальний приклад об'єктно-орієнтованого проекту середньої складності

Чому процес проектування починають із аналізу функціонування чи поведінки системи? Проста відповідь полягає в тому, що поведінка системи зазвичай відома задовго до інших її властивостей.

Поведінка - це щось, що може бути описано в момент виникнення ідеї програми та (на відміну від формальної специфікації системи) виражено в термінах, зрозумілих як для програміста, так і для клієнта.

Уявімо, що ви є головним архітектором програмних систем у провідній комп'ютерній фірмі. Якось з'являється ваш начальник із ідеєю, яка, як він сподівається, буде черговим успіхом компанії. Вам доручають розробити систему під назвою "Інтерактивний розумний кухонний помічник" (РКП).

Завдання, поставлене перед вашою командою програмістів, сформульовано в кількох скупих словах. Програма "Розумний кухонний помічник" (РКП) призначена для домашніх персональних комп'ютерів. Її мета – замінити собою набір карток із рецептами, який можна зустріти майже в кожній кухні.

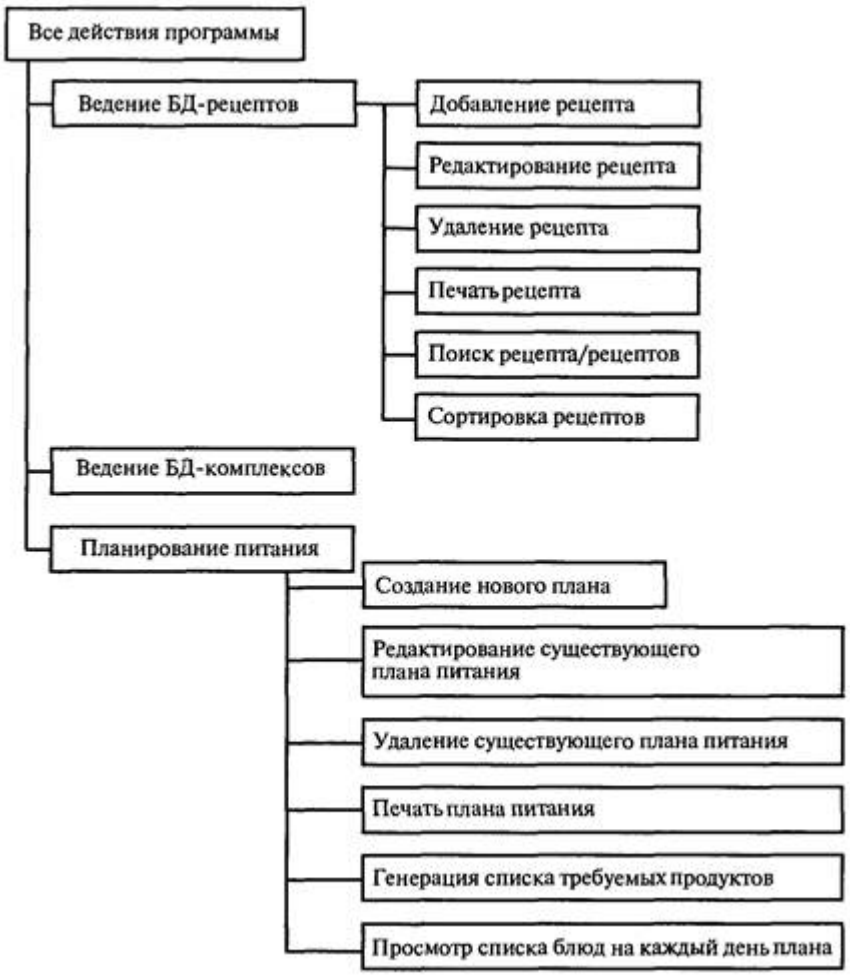
Аналіз аналогів виявив, що вже відома низка програмних реалізацій електронних куховарських книг із рецептами страв. У цій галузі застосування нової була програма, що дозволяє планувати харчування на заданий період. План харчування на заданий період складається із щоденних планів харчування з три- або чотириразовим прийомом їжі. Що треба врахувати під час розробки щоденних планів харчування? Число людей, калорійність харчування кожної людини, улюблені та зненавиджені страви, витрати на харчування. Ранні, описані в літературі спроби оптимізації харчування з урахуванням лише продуктів, їхньої калорійності та цін призвели до рішень виду: оптимальний сніданок — 12 чашок оцту. Генерація меню обіду з використанням датчика випадкових чисел може призвести до несумісних страв: молочний суп, оселедець з гороховим гарніром, квас. Вирішення проблеми - використання набору комплексних сніданків, обідів та вечерь. Чи є у літературі достатній опис можливих комплексів? Чи потрібно залучити спеціалістів із харчування для розробки необхідної кількості комплексів? Скільки коштуватиме база даних комплексів? Чи слід реалізувати функцію автоматичної передачі замовлення на продукти магазину? На ці та інші питання необхідно дати відповідь, щоб укластися у відпущені кошти та терміни.

Як це зазвичай буває при початковому описі багатьох програмних систем, первинні специфікації дуже двозначні.

Команда розробників вирішує, коли система починає роботу, користувач бачить привабливе інформаційне вікно. Відповідальність за його відображення приписується об'єкту Greeter. CRC-картка Greeter представлена на рис. 8.6. Деяким, поки невизначеним чином (за допомогою кнопок, спливаючого меню і т. д.) користувач вибирає одну з наступних восьми дій.

1. Переглянути базу даних із рецептами, але без посилань на якийсь план харчування.
2. Додати новий рецепт до бази даних.
3. Редагувати чи додати коментар до існуючого рецепту.
4. Переглянути базу даних комплексів.
5. Додати новий комплекс до бази даних.
6. Редагувати чи додати коментар до існуючого комплексу.
7. Створити новий план харчування.
8. Переглянути існуючий план щодо деяких дат, страв та продуктів.

Більш детальний опис функцій програми представлено на рис. 8.7.

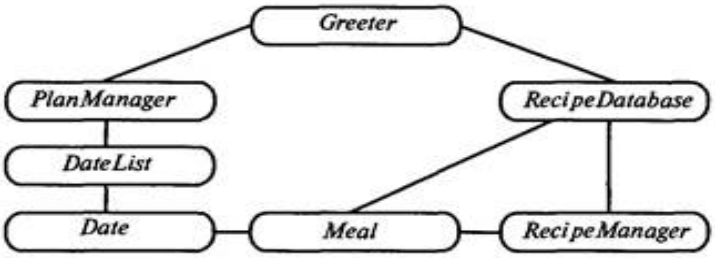


*Мал. 8.7.*Детальний опис функцій програми

Програма має забезпечувати ведення бази даних (додавання, видалення та інші дії з окремим рецептом або набором рецептів). Це загалом стандартні функції СУБД. Що стосується функції планування, то передбачається, що програма на запит користувача складатиме план харчування на певний період часу (тиждень, місяць, рік) для всієї сім'ї або окремих її членів, виходячи із заданих обмежень (наприклад, обмеження на калорійність). Після створення плану користувачеві буде надано такі можливості:
— перегляд плану харчування на кожен день із заданого періоду дії плану, причому користувач зможе не тільки переглядати пропоновані набори страв на обід, вечерю тощо, але й редагувати рецепти їх приготування, вибирати рецепт страви із запропонованого програмою списку або додавати свій бази даних рецептів;
- Отримання списку продуктів, які необхідно закупити на розрахунковий період;
- Здійснення роздруківки даного плану харчування або списку необхідних продуктів;
— створення нового плану на цей період, але з іншим обмеженням (наприклад, обмеження на продукти харчування — церковна посада) або створення нового плану з тим самим обмеженням, але на інший період.
Важливим завданням є уточнення специфікації. У вихідних специфікаціях найбільше зрозумілі лише загальні положення. Ймовірно, що специфікації для кінцевого продукту будуть змінюватися під час розробки програмної системи. Далі дії, які здійснюються програмною системою, приписуються об'єктам.
Перші три дії пов'язані з базою даних рецептів; наступні три дії пов'язані з базою даних комплексів, останні дві - з плануванням харчування. У результаті команда приймає таке рішення: створити об'єкти, які відповідають цим двом обов'язкам.

Таким чином, може бути сформульована постановка завдання: розробити та реалізувати систему ведення бази даних рецептів з можливістю планування харчування членів сім'ї. Система повинна містити:
• стандартні засоби для ведення бази даних рецептів (перегляд, додавання, редагування, видалення записів рецептів);
• стандартні засоби для ведення бази даних комплексів (перегляд, додавання, редагування, видалення записів комплексів);
• засоби розробки плану харчування (створення, коригування) на певний період (тиждень, місяць, рік), виходячи із заданих групових та індивідуальних обмежень (на калорійність, утримання певних компонентів) для кожного члена сім'ї;
• можливість виведення інформації щодо страв, що готуються відповідно до плану живлення (на екран, принтер) на весь розрахунковий період або на необхідний день;
• можливість виведення інформації про склад продуктів (на екран, принтер) як за весь період, так і за датами закупівель, виходячи зі строків зберігання.
Створення складної фізичної системи, подібної до будівлі чи автомобіля, спрощується за допомогою розбиття проекту на структурні одиниці. Так само розробка програмного забезпечення полегшується після виділення окремих об'єктів програми. Об'єкт — це абстрактна одиниця, яка може виконувати певну роботу (тобто мати певні обов'язки). На цьому етапі немає необхідності знати точно як задається об'єкт або як він виконуватиме свою роботу. Об'єкт може в кінцевому підсумку бути перетворений на окрему функцію, структуру або сукупність інших об'єктів. У цьому рівні розробки є дві важливі особливості: об'єкт повинен мати невеликий набір чітко визначених обов'язків; об'єкт повинен взаємодіяти з іншими об'єктами настільки слабко, наскільки це можливо.
*Відстрочені дії.*Зрештою доведеться вирішувати, як користувач переглядатиме базу даних. Наприклад, чи повинен він спочатку входити до списку таких категорій, як "супи", "салати", "гарячі страви", "десерти"? З іншого боку, чи може користувач задавати ключові слова пошуку інгредієнтів, наприклад "полуниця", "сир". Чи слід застосовувати смуги прокручування або закладки у віртуальній книжці?
Розмірковувати про ці предмети приносить задоволення, але важливо те, що немає необхідності приймати конкретні рішення на даному етапі проектування. Оскільки вони впливають тільки на окремий об'єкт і не торкаються функціонування інших частин системи, то все, що потрібно для продовження роботи над сценарієм, — це інформація про те, що користувач має обрати комплекс із конкретними рецептами.
Що таке план харчування? План живлення - це список об'єктів DateList - дат. Дата це об'єкт Date із включеними кулінарними рецептами, з дотриманням правил комплексів сніданків, обідів та вечерь.
Кожен кулінарний рецепт ідентифікуватиметься з конкретним об'єктом. Якщо рецепт вибраний користувачем, керування передається об'єкту, асоційованому з рецептом. Рецепт повинен містити певну інформацію, яка в основному складається зі списку інгредієнтів та дій, необхідних для трансформування складових у кінцевий продукт. Згідно з нашим сценарієм, об'єкт-рецепт повинен виконувати й інші дії. Наприклад, він відображатиме рецепт на екрані. Користувач отримає можливість постачати рецепт анотацією, змінювати список інгредієнтів або набір інструкцій, а також може вимагати роздрукувати рецепт на принтері. Всі ці дії є обов'язком об'єкту Recipe. На етапі проектування можна розглядати Recipe як прототип численних об'єктів-рецептів.
Визначивши вчорне, як здійснити перегляд бази даних, повернемося до її блоку управління і припустимо, що користувач хоче додати новий рецепт. У блоці управління базою даних певним чином визначається, який розділ помістити новий рецепт (нині нас цікавлять деталі), запитується ім'я рецепту і виводиться вікно для набору тексту. Таким чином, це завдання природно віднести до об'єкта, який відповідає за редагування рецептів.
Повернемося до блоку Greeter (див. мал. 8.6). Планування меню, як пам'ятаєте, було доручено об'єкту PlanManager. Користувач повинен мати можливість зберегти план. Отже,

об'єкт PlanManager може запускатися або внаслідок відкриття вже існуючого плану харчування, або під час створення нового. У разі користувача необхідно попросити ввести інтервали часу (список дат) нового плану. Кожна дата асоціюється із окремим об'єктом типу Date. Користувач може вибрати дату для детального дослідження. У цьому випадку керування передається відповідному об'єкту Date. Об'єкт PlanManager повинен вміти роздруковувати меню харчування на період, що планується. Нарешті користувач може попросити об'єкт PlanManager згенерувати список продуктів на зазначений період.
В об'єкті Date зберігаються такі дані: список страв на відповідний день та (необов'язково) текстові коментарі, додані користувачем (наприклад, ювілейні дати). Об'єкт повинен виводити на екран перелічені вище дані. Крім того, у ньому має бути передбачена функція друку. У разі бажання користувача детальніше ознайомитися з тією чи іншою стравою слід передати керування об'єкту Meal.
В об'єкті Meal зберігається інформація про страву. Не виключено, що користувач має кілька рецептів однієї страви. Тому необхідно видаляти та додавати рецепти. Крім того, бажано мати можливість роздрукувати інформацію про ту чи іншу страву. Зрозуміло, має бути забезпечене виведення інформації на екран. Користувачеві, найімовірніше, захочеться звернутися ще до якихось рецептів. Отже, необхідно налагодити контакт із базою даних рецептів, отже, об'єкти Meal і база даних повинні взаємодіяти між собою.



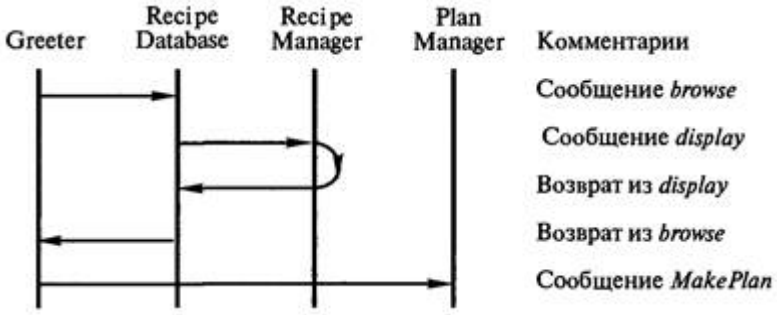
*Мал. 8.8.*Схема статичних зв'язків між об'єктами програми РКП

Далі команда розробників продовжує досліджувати усі можливі сценарії. Необхідно передбачити опрацювання виняткових ситуацій. Наприклад, що відбувається, якщо користувач задає ключове слово для пошуку рецепта, а відповідний рецепт не знайдено? Як користувач зможе перервати дію (наприклад, введення нового рецепта), якщо не хоче продовжувати далі? Все це має бути вивченим. Відповідальність за розробку таких ситуацій слід розподілити між об'єктами.
Вивчивши різні сценарії, команда розробників наприкінці вирішує, що це дії належним чином можуть бути розподілені між сімома об'єктами (рис. 8.8). Об'єкт Greeter взаємодіє лише з PlanManager та Recipe Database. Об'єкт PlanManager «зачіпляється» лише з DateList, DateList з Date, а Date, своєю чергою, — з Meal. Об'єкт Meal звертається до RecipeManager і через цей об'єкт до конкретних рецептів (див. рис. 8.8).
Окремі слова мають дуже багато інтерпретацій. Тому необхідно на самому початку проектування підготувати словник, що містить чіткі та недвозначні визначення всіх об'єктів (класів), атрибутів, операцій, ролей та інших сутностей, що розглядаються у проекті. Без такого словника обговорення проекту з колегами з розробки та замовниками системи не має сенсу, оскільки кожен може по-своєму інтерпретувати терміни, що обговорюються.

8.9.3. Динамічна модель системи

Об'єктна модель представляє статичну структуру проектованої системи (підсистеми). Однак знання статичної структури недостатньо, щоб зрозуміти та оцінити роботу підсистеми. Схема, зображена на рис. 8.8, не підходить для опису динамічної взаємодії під час виконання програми.

Динамічна модель підсистеми будується після того, як об'єктна модель підсистеми побудована та попередньо узгоджена та налагоджена.
Динамічна модель системи представляється діаграмою послідовності та діаграмою станів об'єктів.
На рис. 8.9 показано частину діаграми послідовності РКП. Час змінюється зверху донизу. Кожен об'єкт представлений вертикальною лінією. Повідомлення від одного об'єкта до іншого зображується горизонтальною стрілкою між вертикальними лініями. Повернення управління (і, можливо, результату) в об'єкт представлене стрілкою у зворотному напрямку. Деякі автори використовують з цією метою пунктирну стрілку. Коментар праворуч від малюнка докладніше пояснює взаємодію.
Завдяки наявності осі часу діаграма послідовності краще визначає послідовність подій у процесі роботи програми. Тому діаграми послідовності є корисним засобом документування складних програмних систем.
Стан визначається сукупністю поточних значень атрибутів. Наприклад, банк може мати стан — платоспроможний і неплатоспроможний (коли більшість банків одночасно виявляється у другому стані, настає банківська криза). Стан визначає реакцію об'єкта на подію, що надходить (в тому, що реакція різна, неважко переконатися за допомогою банківської картки: залежно від стану банку обслуговування або реакція банку на пред'явлення картки буде різним). Реакція об'єкта на подію може включати певну дію та/або переведення об'єкта в новий стан.
При визначенні станів ми не розглядаємо ті атрибути, які не впливають на поведінку об'єкта, і об'єднуємо в один стан усі комбінації значень атрибутів та зв'язків, що дають однакові реакції на події.



*Мал. 8.9.*Приклад діаграми послідовності

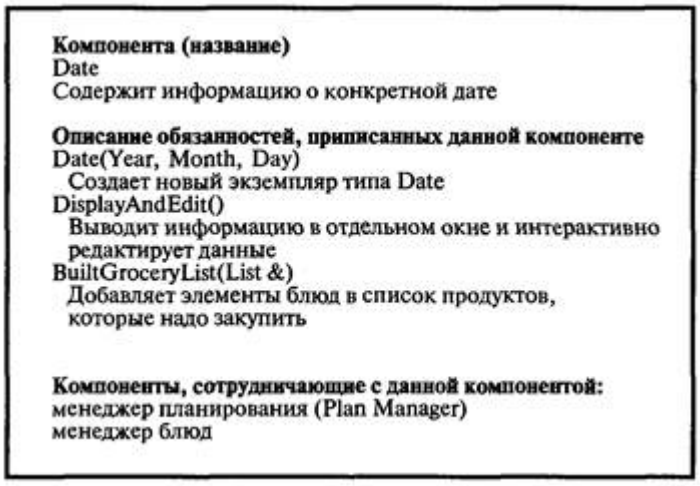
*Мал. 8.10.*Приклад діаграми станів

Діаграма станів пов'язує події та стани. При прийомі події наступний стан системи залежить від її поточного стану, і від події (рис. 8.10). Зміна стану називається переходом. Діаграма станів - це граф, вузли якого представляють стани, а спрямовані дуги, позначені іменами відповідних подій - переходи. Діаграма станів дозволяє отримати послідовність станів заданої послідовності подій.
Як опис поведінки об'єкта, діаграма станів повинна описувати, що робить об'єкт у відповідь на перехід в деякий стан або на виникнення деякої події. Для цього діаграму станів включаються описи активностей і дій.

Активністю називається операція, пов'язана з будь-яким станом об'єкта (вона виконується, коли об'єкт потрапляє у вказаний стан); виконання активності потребує певного часу. Приклади активностей: видача картинки на екран телевізора, телефонний дзвінок, зчитування порції файлу буфер і т. п.; іноді активністю буває просто призупинення виконання програми (пауза), щоб забезпечити необхідний час перебування у відповідному стані (це буває особливо важливим для паралельної асинхронної програми).

8.9.4. Уточнення класів із точним визначенням їх залежностей з інших класів. Продовження навчального прикладу

Продовжимо розробку програми РКП. На таких етапах уточнюється опис об'єктів. Спочатку формалізуються методи взаємодії.
Слід визначити, як буде реалізовано кожен із об'єктів. Об'єкт, що характеризується лише поведінкою (що має внутрішнього стану — внутрішніх даних), може бути оформлений як функції. Наприклад, об'єкт, що замінює у рядку всі великі літери на малі, краще уявити у вигляді функції. Об'єкти з багатьма функціями краще реалізувати як класів. Кожному обов'язку, перерахованому на CRC-картці, надається ім'я. Ці імена стануть потім назвами функцій чи процедур. Разом з іменами визначаються типи аргументів, що передаються функціям та процедурам. Потім описується вся інформація, що міститься всередині об'єкта класу. Якщо об'єкту потрібні деякі дані для виконання конкретного завдання, їх джерело (аргумент функції, глобальна або внутрішня змінна) має бути явно описано.
Як тільки для всіх дій вибрано імена, CRC-картка для кожного об'єкта переписується заново із зазначенням імен функцій та списку формальних параметрів (рис. 8.11). Тепер CRC-картка відображає всю інформацію для запису опису класу, що породжує об'єкт, відображений на цій картці.
Ідея класифікації класів об'єктів програми через їхню поведінку має надзвичайний наслідок. Програміст знає, як використовувати об'єкт, розроблений іншим програмістом, і при цьому немає необхідності знати, як він реалізований. Нехай класи шести об'єктів РКП розробляються шістьма програмістами. Програміст, який розробляє клас об'єкта Meal, повинен забезпечити перегляд бази даних із рецептами та вибір окремого рецепту при складанні страви. Для цього об'єкта Meal просто викликає функцію browse, прив'язану до об'єкту RecipeDatebase. Функція browse повертає окремий рецепт Recipe із бази даних. Все це справедливо незалежно від того, як конкретно реалізований всередині Recipe Database перегляд бази даних.


Компонента (название)
Date
Содержит информацию о конкретной дате

Описание обязанностей, приписанных данной компоненте
Date(Year, Month, Day)
Создает новый экземпляр типа Date
DisplayAndEdit()
Выводит информацию в отдельном окне и интерактивно редактирует данные
BuiltGroceryList(List &)
Добавляет элементы блюд в список продуктов, которые надо закупить

Компоненты, сотрудничающие с данной компонентой:
менеджер планирования (Plan Manager)
менеджер блюд

Мал. 8.11. Уточнена CRC-картка

Ймовірно, у реальному додатку буде багато рецептів. Проте всі вони поводитимуться однаково. Відрізняється лише стан: список інгредієнтів та інструкцій із приготування. На ранніх стадіях розробки нас має цікавити поведінка, спільна для всіх рецептів. Деталі, специфічні для окремого рецепту, не є важливими. Зауважимо, що поведінка асоційована з класом, а чи не з індивідуальним представником, т. е. все екземпляри класу сприймають одні й самі команди і виконують їх подібним чином. З іншого боку, стан є індивідуальним, і це видно з прикладу різних екземплярів класу Recipe. Всі вони можуть виконувати ті самі дії (редагування, виведення на екран, друк), але використовують різні дані.
Двома важливими поняттями розробки програм є зачеплення (cohesion) і пов'язаність (coupling). Пов'язаність це міра того, наскільки окремий об'єкт утворює логічно закінчену, осмислену одиницю. Висока пов'язаність досягається об'єднанням в одному об'єкті співвідносяться (у тому чи іншому сенсі) один з одним функцій. Найчастіше функції виявляються пов'язаними друг з одним за необхідності мати доступом до загальним даним. Саме це поєднує різні частини об'єкта Recipe.
Зокрема, зачеплення виникає, якщо один об'єкт повинен мати доступ до даних (стану) іншого об'єкта. Слід уникати таких ситуацій. Покладіть обов'язок здійснювати доступ до даних на об'єкт, що ними володіє. Наприклад, за редагування рецептів відповідальність повинна лежати на об'єкті RecipeDatabase, оскільки саме в ньому вперше виникає необхідність. Але об'єкт RecipeDatabase повинен безпосередньо маніпулювати станом окремих рецептів (їх внутрішніми даними: списком інгредієнтів та інструкціями з приготування). Краще уникнути такого тісного зчеплення, передавши обов'язок редагування безпосередньо рецепту.
З іншого боку, зачеплення характеризує взаємозв'язок між об'єктами програми. У загальному випадку бажано, як тільки можна, зменшити ступінь зачеплення, оскільки зв'язки між об'єктами програми перешкоджають їхній модифікації та заважають подальшій розробці або повторному використанню в інших програмах.

8.9.5. Спільний розгляд трьох моделей

В результаті аналізу отримуємо три моделі: об'єктну, динамічну та функціональну. У цьому об'єктна модель становить основу, навколо якої здійснюється подальша технологія. При побудові об'єктної моделі у ній який завжди вказуються операції над об'єктами, оскільки з погляду об'єктної моделі об'єкти — це структури даних. Тому розробка системи починається зі зіставлення дій та активностей динамічної моделі та процесів функціональної моделі операцій та внесення цих операцій до об'єктної моделі. З цього починається процес розробки програми, що реалізує поведінку, яка описується моделями, побудованими в результаті аналізу вимог до системи.
Поведінка об'єкта задається його діаграмою стану; кожному переходу на цій діаграмі відповідає застосування до об'єкта однієї з операцій; можна кожній події, отриманої об'єктом, зіставити операцію над цим об'єктом, а кожній події, надісланій об'єктом, зіставити операцію над об'єктом, якому подію було надіслано. Активності, яка запускається переходом на діаграмі станів, може відповідати ще одна (вкладена) діаграма станів.
Результатом цього етапу проектування є уточнена об'єктна модель, що містить всі класи програмної системи, що проектується, в яких специфіковані всі операції над їх об'єктами.

8.10. ПРИКЛАД РЕТРОСПЕКТИВНОЇ РОЗРОБКИ ІЄРАРХІЇ КЛАСІВ БІБЛІОТЕКИ ВІЗУАЛЬНИХ КОМПОНЕНТ DELPHI І C++ BUILDER

Delphi і C++ Builder є візуальним засобом розробки корпоративних інформаційних систем. У C++ Builder використовується мова об'єктно-орієнтованого програмування C++, а Delphi — Object Pascal. Незважаючи на це, обидва середовища використовують одні й ті самі модулі бібліотеки візуальних компонентів, написаних на Object Pascal.

Кожен тип органів управління системи описується класом, а конкретні органи управління, що містяться на форми, є об'єктами відповідних класів. Приміром, Button1, Button2, ..., ButtonN є об'єктами класу TButton; Edit1, Edit2, ..., EditM - об'єктами класу TEdit і т. п. Коли користувач створює форму у візуальному інтегрованому середовищі, він, по суті (на відміну від інших органів управління), створює новий клас, об'єктом якого буде форма, що з'являється під час виконання програми (наприклад, клас — TForm1, об'єкт класу — Form1).
З метою з'ясування процесів розробки ієрархії класів спробуємо ретроспективного аналізу ієрархії класів системи Delphi/C++ Builder.
У процесі аналізу була розписана ієрархія класів, обраних для прикладу органів управління, виділено деякі обов'язки, які міг би накласти на них розробник, а потім на основі порівняння списків виділених обов'язків зроблено спробу обґрунтувати ієрархію класів, прийняту серед Delphi/C++ Builder.
Слід зазначити, що цей аналіз проводиться з суто навчальними цілями, тому тут не розглянуті всі органи управління, а також всі обов'язки, які може накласти розробник того чи іншого орган управління.
Розглянемо органи управління, які можна отримати транспортуванням їх з допомогою миші з палітри компонент Delphi/C++ Builder.
-TButton - звичайна кнопка;
- TRadioButton - радіокнопка (група кнопок із залежною фіксацією, що забезпечує можливість вибору лише однієї кнопки із групи);
- TListBox - звичайний список;
- TDBListBox - список для роботи з таблицями даних;
- TDataSource - джерело даних (є посередником між елементами DataAccess: Table, Query, - і органами управління базами даних DataControls: DBGrid, DBEdit тощо).
Спробуємо загалом прокоментувати цю ієрархію та обґрунтувати її на основі методу розподілу обов'язків. Нижче наведено набір обов'язків для цих компонентів.
Обов'язки об'єктів класу TDataSource:
-контролювати доступ користувача до елементів TDataSet;
- Забезпечувати можливість визначення, чи підключений TDataSource до деякого елементу TDataSet;
- Забезпечувати можливість роботи з дочірніми компонентами;
- Забезпечувати можливість копіювання даних в інший об'єкт того ж класу;
- Забезпечувати можливість знищення об'єкта з вивільненням пам'яті;
- Забезпечувати можливість посилати повідомлення;
- Визначати ім'я класу, об'єктом якого є даний елемент.
Обов'язки об'єктів класу TButton:
- обробляти повідомлення WMLBUTTONDOWN та WMLBUTTONDBLCLK (натискання та подвійне натискання лівої кнопки миші);
- програмно емулювати натискання кнопки;
- обробляти повідомлення від клавіатури;
- Отримувати фокус введення;
- Визначати, чи є фокус введення;
— визначати, чи може об'єкт мати фокус уведення (наприклад, якщо елемент невидимий, йому не можна передати фокус уведення);
- обробляти повідомлення BMCLICK (відбувається після натискання кнопки миші);
- Ставати видимим і невидимим;
- Перемальовуватися;
- Забезпечувати можливість переведення точки з системи координат вікна в систему координат екрана;
- Зберігати ідентифікатор батька (з можливістю змінити батька);
- Забезпечувати можливість роботи з дочірніми компонентами;
- Забезпечувати можливість копіювання даних в інший об'єкт того ж класу;

- Забезпечувати можливість знищення об'єкта з вивільненням пам'яті;
- Забезпечувати можливість посилати повідомлення;
- Визначати ім'я класу, об'єктом якого є даний елемент.

Обов'язки об'єктів класу TRadioButton:

-обробляти повідомлення WMLBUTTONDOWN та WMLBUTTONDBLCLK (натискання та подвійне натискання лівої кнопки миші);
- Визначати, яка обрана кнопка з групи кнопок;
- обробляти повідомлення від клавіатури;
- Отримувати фокус введення;
- Визначати, чи є фокус введення;
— визначати, чи може об'єкт мати фокус уведення (наприклад, якщо елемент невидимий, йому не можна передати фокус уведення);
- обробляти повідомлення BMCLICK. (відбувається після натискання кнопки миші);
- Ставати видимим і невидимим;
- Перемальовуватися;
- Забезпечувати можливість переведення точки з системи координат вікна в систему координат екрана;
- Зберігати ідентифікатор батька (з можливістю змінити батька);
- Забезпечувати можливість роботи з дочірніми компонентами;
- Забезпечувати можливість копіювання даних в інший об'єкт того ж класу;
- Забезпечувати можливість знищення об'єкта з вивільненням пам'яті;
- Забезпечувати можливість посилати повідомлення;
- Визначати ім'я класу, об'єктом якого є даний елемент.

Обов'язки об'єктів класу TListBox:

-очищати список;
- Забезпечувати можливість знаходження декількох елементів зі списку;
- Визначати номер елемента списку за координатами точки, що належить об'єкту класу TListBox;
- обробляти повідомлення від клавіатури;
- Отримувати фокус введення;
- Визначати, чи є фокус введення;
— визначати, чи може об'єкт мати фокус уведення (наприклад, якщо елемент невидимий, йому не можна передати фокус уведення);
- обробляти повідомлення BM_CLICK (відбувається після натискання кнопки миші);
- Ставати видимим і невидимим;
- Перемальовуватися;
- Забезпечувати можливість переведення точки з системи координат вікна в систему координат екрана;
- Зберігати ідентифікатор батька (з можливістю змінити батька);
- Забезпечувати можливість роботи з дочірніми компонентами;
- Забезпечувати можливість копіювання даних в інший об'єкт того ж класу;
- Забезпечувати можливість знищення об'єкта з вивільненням пам'яті;
- Забезпечувати можливість посилати повідомлення;
- Визначати ім'я класу, об'єктом якого є даний елемент.

Обов'язки об'єктів класу TDBListBox:

- Забезпечувати зв'язок з джерелом даних (TDataSource);
- Очищати список;
- Забезпечувати можливість виведення декількох елементів зі списку;
- Визначати номер елемента списку за координатами точки, що належить об'єкту класу TDBListBox;
- обробляти повідомлення від клавіатури;
- Отримувати фокус введення;

- Визначати, чи є фокус введення;
— визначати, чи може об'єкт мати фокус уведення (наприклад, якщо елемент невидимий, йому не можна передати фокус уведення);
- обробляти повідомлення BMCLICK (відбувається після натискання кнопки миші);
- Ставати видимим і невидимим;
- Перемальовуватися;
- Забезпечувати можливість переведення точки з системи координат вікна в систему координат екрана;
- Зберігати ідентифікатор батька (з можливістю змінити батька);
- Забезпечувати можливість роботи з дочірніми компонентами;
- Забезпечувати можливість копіювання даних в інший об'єкт того ж класу;
- Забезпечувати можливість знищення об'єкта з вивільненням пам'яті;
- Забезпечувати можливість посилати повідомлення;
- Визначати ім'я класу, об'єктом якого є даний елемент.

При уважному розгляді обов'язків, обумовлених об'єктам перерахованих класів, можна побачити, деякі з них збігаються, т. е. виявляються загальними об'єктів різних класів. Слід зазначити, що з обов'язків є спільними для об'єктів всіх класів, а деякі — лише обмеженого набору.

Логічно було б запровадити додаткові класи, об'єкти яких мали загальні для розглянутих класів обов'язки. Тоді розглянуті класи (TDataSource, TButton, TRadioButton, TListBox, TDBListBox) могли б успадкувати функції, що забезпечують виконання цих обов'язків у введених додаткових класів (в об'єктно-орієнтованому програмуванні є механізм, який так і називається механізм спадкування, який робить доступними з дочірніх класів якості та способи, з урахуванням прав доступу, очевидно, з батьківських (чи базових) классов).

Спробуємо виділити згадані класи та призначити їм їхні обов'язки:

Обов'язки об'єктів класу Клас_1:
- Забезпечувати можливість роботи з дочірніми компонентами;
- Забезпечувати можливість копіювання даних в інший об'єкт того ж класу;
- Забезпечувати можливість знищення об'єкта з вивільненням пам'яті;
- Забезпечувати можливість посилати повідомлення;
- Визначати ім'я класу, об'єктом якого є даний елемент.

Обов'язки об'єктів класу Клас_2:
- обробляти повідомлення від клавіатури;
- Отримувати фокус введення;
- Визначати, чи є фокус введення;
— визначати, чи може об'єкт мати фокус уведення (наприклад, якщо елемент невидимий, йому не можна передати фокус уведення);
- обробляти повідомлення BM_CLICK (відбувається після натискання кнопки миші);
- Ставати видимим і невидимим;
- Перемальовуватися;
- Забезпечувати можливість переведення точки з системи координат вікна в систему координат екрана;
- Зберігати ідентифікатор батька (з можливістю змінити батька).

Обов'язок об'єктів класу Клас_3:
-обробляти повідомлення WM_LBUTTONDOWN та WM_LBUTTONDBLCLK (натискання та подвійне натискання лівої кнопки миші).

Обов'язки об'єктів класу Клас_4:
- Очищати список;
- Забезпечувати можливість знаходження декількох елементів зі списку;
— визначати номер списку за координатами точки, що належить об'єкту класу TDBListBox.

Тепер, виключивши з безлічі обов'язків об'єктів класів, що розглядаються, ті, які передані додатковим класам, перерахуємо інші обов'язки:

Обов'язки класу TDataSource:
- контролювати доступ користувача до елементів TDataSet;
- Забезпечувати можливість визначення, чи підключений TDataSource до деякого елементу TDataSet.

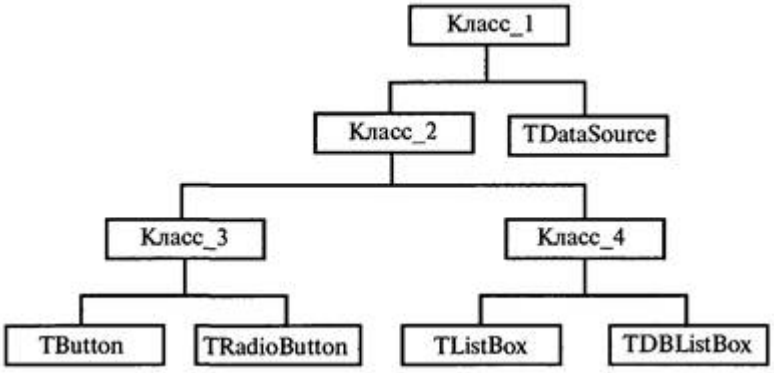
Отже, обов'язки класу Tbutton - програмно емулювати натискання кнопки; класу TradioButton - визначати, яка з кнопок із залежною фіксацією обрана; класів TlistBox і TDBListBox - забезпечувати зв'язок із джерелом даних (TDataSource).

В результаті отримана ієрархія класів (рис. 8.12), яка забезпечує виключення надмірності коду (функції, що здійснюють виконання об'єктами різних класів однакових обов'язків, кодуються приблизно, а іноді абсолютно однаково), підвищує доступність для огляду коду програми, а отже, потенційно скорочує час на її налагодження .

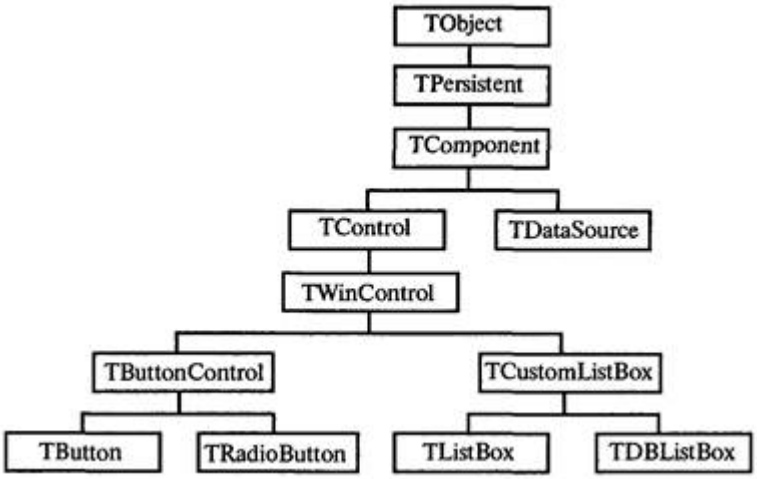
Тепер розглянемо рис. 8.13 фрагмент схеми ієрархії класів для перерахованих елементів (до кореневого суперкласу).

Порівнюючи малюнки 8.12 та 8.13, можна помітити:
1. Клас_1 у нашій ієрархії відповідає гілки TObject → TPersistent → TComponent;
2. Клас_2 - TControl -> TWinControl;
3. Клас_3 - TbuttonControl;
4. Клас_4 - TCustomListBox.

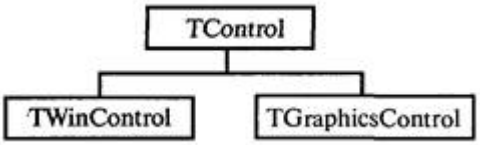
Якщо з двома останніми класами все зрозуміло, виникає питання: чому перші два класи нашої ієрархії відповідають не одному, а цілим ланцюжкам класів C++ Builder? Справа в тому, що в цьому прикладі описані не всі класи C++ Builder. Тому ті з них, які призводять до розгалуження двох перших ланцюжків, тут просто не враховано. Наприклад, елемент TImage, призначений для розташування на формах графічних зображень, має наступний ланцюжок успадкування класів: TObject → TPersistent → TComponent → TControl → TGraphicsControl, — тобто ланцюжок TControl → TWinControl перетворюється на дерево, на якому класи TWinControl і TGraphicsControl виявляються одному рівні. Цей фрагмент схеми ієрархії класів зображено на рис. 8.14.



*Мал. 8.12.*Попередня ієрархія класів



*Мал. 8.13.*Ієрархія деяких класів C++Builder



*Мал. 8.14.*Фрагмент схеми ієрархії

8.11. АЛЬТЕРНАТИВНИЙ ПРОЕКТ ГРАФІЧНОГО ІНТЕРФЕЙСУ

При розвитку програм виникає проблема збільшення функціональних можливостей одного об'єкта з допомогою функціональних можливостей іншого. Найактуальніша проблема програмування - написання гнучких програм, пристосованих для модифікації та розвитку. Спочатку треба запровадити лише одне поняття, запропоноване Олександром Усовим: контейнер-менеджер, або просто контейнер. Слід зазначити, що не йдеться про контейнері C++. Отже, контейнер — це клас, який дозволяє об'єднувати (агрегувати) у собі різні класи об'єктів, у тому числі й інші контейнери. Однією з найскладніших завдань проектування є агрегація різнорідних елементів на нове єдине ціле. Контейнер — один із механізмів вирішення проблеми гнучкої агрегації.

Найпростіший контейнер – це список посилань на об'єкти. Далі якщо скористатися механізмом повідомлень, то... всіх цих труднощів можна уникнути! Жодного рядка нового коду! Повідомлення, що надходять контейнеру, проектуються на об'єкти, що йому належать. Але допустима і складніша логіка обробки запитів, як вони потраплять до объекту-обработчику.

Повідомлення, які може обробляти клас, утворюють його інтерфейс. При використанні таких контейнерів немає потреби оголошувати поля класу private чи protected або ще якось, оскільки їх взагалі не повинно бути видно (вихідні тексти класу більше не треба постачати разом з його кодом). Для всіх розробників, які використовують цей клас, достатньо знати його типи та структури повідомлень, тобто повідомлення забезпечують максимальний захист полів об'єктів і при цьому не потребують накладних витрат.

Повідомлення дозволяють збільшити віртуалізацію коду, що позитивно впливає на зниження його обсягу. Повідомлення на відміну від виклику процедури простіше перехопити, щоб виконати над ними попередню обробку, наприклад, фільтрацію або сортування. Нарешті повідомлення дозволяють максимально збільшити продуктивність системи, що недосяжно при виклику процедур.

Контейнери бувають двох типів: однорідні (динамічні) та різнорідні (статичні).

Однорідний контейнер може включати довільне безліч об'єктів одного чи класів, похідних від цього класу. Логіка роботи такого контейнера гранично проста, наприклад розподіляти

повідомлення, що надходять, по всіх включених до нього об'єктах. Оскільки включені до нього об'єкти належать одному класу, то, отже, вони мають єдиний інтерфейс, але тоді стає зовсім неважливо, скільки об'єктів включено до контейнера у будь-який момент часу, тобто це число довільно. Логіка роботи такого контейнера з включеними до нього об'єктами однакова і залежить від конкретного об'єкта. Типовий представник такого контейнера – список (наприклад, рядків). При додаванні (видаленні) нових об'єктів (рядків) логіка роботи самого контейнера залишається незмінною.

Навпаки, контейнер різнорідних елементів може складатися з об'єктів різних класів. Його можна як схему, де кожен елемент (об'єкт) має своє смислове (функціональне) навантаження. Події, що надходять на такий контейнер, не примітивно транслюються на всі об'єкти, а розподіляються між ними за заданою схемою. Для цього типу контейнера застосовується поняття «конструювання».

Іншою відмінністю контейнера від множинного успадкування є те, що можна довільно під час роботи або проектування включати нові або виключати старі об'єкти, наприклад, щоб забезпечити їх перенесення з одного контейнера до іншого. При цьому стан об'єктів залишається тим самим, ми просто змінюємо посилання у контейнерів. Можна динамічно підвантажувати нові логічні схеми роботи контейнера чи змінювати старі, що з множинного успадкування, напевно, недосяжно у принципі. Отже, контейнер може гнучко реалізовувати поліморфізм у найбільш загальному значенні!

Зазначимо ще раз, що взаємозв'язок між об'єктами здійснюється у вигляді повідомлень. Але тут повідомлення – спеціальний клас. Саме цей клас несе відповідальність за поліморфізм властивостей, але не класи основної ієрархії. У такому разі ми маємо можливість оголосити певний клас-повідомлення та створити набір поліморфних класів-спадкоємців, які будуть оброблятися об'єктами основної ієрархії класів.

Зручність роботи з повідомленнями не означає, що можна змінювати (додавати чи модифікувати) набір властивостей класу основний ієрархії. Ні, характеристики кожного класу задаються на етапі проектування ієрархії.

При використанні контейнерів у жодному об'єкті не використовуються ні конструктори, ні деструктори. Це невипадково. У чому полягає суть конструктора? Реально він має виконати дві дії: проініціалізувати покажчик на таблицю віртуальних методів (VMT) та проініціалізувати власні дані.

Розглянемо приклад проекту із використанням контейнерів. Припустимо, перед вами стоїть завдання розробки графічного інтерфейсу, аналогічного GUI Microsoft Windows. Аналогічний інтерфейс створювали розробники Delphi і раніше ми ретроспективно виконували даний проект.

У вас кілька розробників (проектувальників та програмістів), і завдання треба вирішити у максимально короткий термін. Тут слід зазначити наступний важливий момент: ви не одразу пишете програму, а скоріше створюєте інструментарій її вирішення.

Насамперед ви визначаєте все різноманіття елементів GUI: labels, shapes, edit fields, buttons, check radio buttons, list combo boxes, bitmap тощо. буд. Нескладно помітити, більшість елементів є прості комбінації з двох чи більше візуальних елементів: наприклад рядок і кадру. Інтуїтивно зрозуміло, що візуальний елемент і елемент інтерфейсу - це не те саме. Головною функцією елемента інтерфейсу є отримання інформації від користувача, тоді як візуальний елемент служить для її відображення. Це важливо.

Тепер роздробимо нашу команду на чотири підкоманди.

Перша команда займеться графікою, тобто візуальними елементами. Їм необхідно побудувати ієрархію об'єктів - графічних примітивів, починаючи від точки і закінчуючи фонтами, довільними багатокутниками тощо.

Друга команда має специфікувати ієрархію елементів інтерфейсу.

Третя команда займеться побудовою дерева повідомлень, за допомогою якого елементи інтерфейсу взаємодіятимуть не тільки між собою, але і з ядром операційної системи.

І нарешті, функцією четвертої команди буде створення ієрархії об'єктів введення-виведення (клавіатура, миша, дисплей тощо).
Завдання кожної з підкоманд достатньо незалежні один від одного і можуть виконуватися паралельно. Це також важливо.
Тепер перейдемо до контейнерів, для чого виріжемо невеликий фрагмент роботи ваших команд. Припустимо, що перша група специфікувала (зазначте, лише специфікувала, але ще, можливо, не створила жодного об'єкта) дерево візуальних елементів. Нехай десь у цій ієрархії знайдеться місце, скажімо, для прямокутника та рядка. Тепер друга команда може створити свій елемент інтерфейсу – припустимо, що це буде банальна кнопка. Що таке кнопка - прямокутна рамка та рядок. Оскільки ми припускаємо обійтися без множини, то розумно припустити, що це контейнер. Отже, ієрархія елементів інтерфейсу повинна включати контейнери для візуальних елементів. Контейнер розподіляє вхідний вплив за складовими його елементами, отже, контейнер є менеджером об'єктних запитів.
Як уявити графи реакцій, які можна назвати кодом контейнера? Тепер для нас дуже важливо досягти швидкої реакції на кожну подію. Проблема могла бути вирішена множинним успадкуванням. Але вчинимо інакше.
У нас було виділено спеціальну команду, яка мала розробити механізм об'єктних повідомлень. Дамо їм слово. Коли ми їм сказали, який тип об'єктів будуть використовуватися в нашій системі, вони розробили ієрархію повідомлень. Так, кожне повідомлення є класом, але дивно не тільки це, а й те, що повідомлення, що обробляються кожним класом, компілюються разом із кодом цього класу. Це у першому наближенні можна як таблицю віртуальних методів, лише роздроблену на шматочки. Таким чином, кожне повідомлення несе в собі адресу функції, що його обробляє. Коли контейнер отримує таке повідомлення, він підставляє посилання на належний екземпляр об'єкта даного класу і здійснює виклик. І все...
Що ж тепер маємо? Припустимо, що набридли прямокутні кнопки і захотілося круглих, багатокутних чи взагалі довільних кнопок. «Ну, вже ні», — сказав би фахівець із множинного спадкування. Але ми запитаємо: «Вам у runtime чи спеціально налаштувати?» Дійсно, будь-який спадкоємець від плоскої фігури може бути підставлений у контейнер у будь-який час, включаючи час виконання. І тут ви з подивом помічаєте, що можна вважати проект готовим до вживання, налагодивши його схеми взаємодії всього на одному-двох реальних об'єктах і додаючи все інше за необхідності.
Запропонована Олександром Усовим агрегація є одним із механізмів реалізації в рамках ОВП, який вдало перетинається та доповнює механізми спадкування, інкапсуляції та поліморфізму.
Ймовірно, для забезпечення динаміки буде зроблено наступний крок – використовувати теорію ролей. Теорія ролей - це просто зручна людська назва багато разів тут згаданого поділу оголошеного інтерфейсу та його реалізації деяким об'єктом (актором), який вміє цю роль виконувати.

8.12. ПРОЕКТ АСУ ПІДПРИЄМСТВА

Розвиваючи ідею використання контейнерів А. Усова, можна отримати ідею системи генерації нових програм із використовуваними «кубиками» — готовими об'єктами, які під час формування програми автоматично витягуються об'єктно-орієнтованої СУБД з бази даних об'єктів.
Створивши систему програмування з використанням бази даних об'єктів та генератором схем властивостей контейнерів, А. Усов розробив ядро типової АСУ підприємства, що дозволяє за короткий термін і за малої кількості програмістів генерувати АСУ нових підприємств.
Інформаційний простір будь-якого підприємства складається із двох частин — залежної та незалежної від профілю підприємства. Незалежна частина базується на спільності властивостей, які притаманні будь-якому підприємству. Завдяки цьому, по-перше, можна побудувати класифікатор підприємств будь-якого профілю так, як це заведено в технології

об'єктно-орієнтованого проектування. По-друге, це дозволяє об'єднувати підприємства різного профілю у єдину корпорацію. По-третє, можна створювати абстрактні підприємства, які потребують мінімального налаштування на конкретний профіль. Нарешті завдяки наявності загальних властивостей у всіх підприємств зовнішні організації можуть контролювати діяльність підприємства.

Кожне підприємство має, зазвичай, ієрархічну структуру підрозділів. Структурний підрозділ (СП) включає три інформаційних класу: службовці, обладнання та матеріали. Тут під обладнанням будуть розумітись основні фонди підприємства або даного СП. Терміном «матеріали» позначаються ті сутності, які споживаються у процесі виробництва. Базові інформаційні класи — службовці, обладнання та матеріали — можуть мати загальний суперклас (ЩОСЬ, ЩО БЕРЕТЬ У ВИРОБНИЦТВІ) чи ні (справа смаку).

Таким чином, створивши необхідні інформаційні класи, склавши їх у контейнер СП та представивши набір цих контейнерів у вигляді ієрархії володіння, ми цим створюємо абстрактне підприємство. Так, це підприємство нічого не виробляє, бо виробництво є специфічним та визначає профіль підприємства. Але такий клас дозволяє створювати підкласи підприємств чи то промислові, муніципальні, транспортні, фінансові чи інші. Кожен із цих класів підприємств може утворювати своє піддерево класів.

Є ще низка моментів, на яких зупинимося. Існуючі системи досить громіздкі та важкі у налаштуванні. Перед їх встановленням, зазвичай, проводяться дослідження з організації бізнес-процесів. За наслідками цих обстежень видаються рекомендації, метою яких є оптимізація основних процесів. Однак після впровадження систем переорганізація виробництва потребує значних зусиль щодо налаштування системи на нові умови. Зазвичай до цієї роботи залучаються спеціальні фірми, які займаються супроводом АСУ. Але сучасні умови ведення бізнесу вимагають високої гнучкості, яка поки що залишається недосяжною мрією.

У запропонованому рішенні кожен структурний підрозділ виділено самостійну сутність, це дозволяє, по-перше, моделювати і прораховувати нові схеми управління виробництвом, а по-друге, дає можливість впроваджувати ці схеми «на ходу». Справді, підприємство, як було зазначено, є контейнер із наявністю низки властивостей. Розкладання цих властивостей СП є генерація схем. Маючи механізм версійності схем, можна будувати моделі, оптимізуючи їх за різними критеріями та використовуючи суворі математичні методи.

Тут можна відзначити, що сучасна теорія управління підприємствами базується на BPR (bussiness process re-engineering) і TQM (total quality managment). Одне з основних положень BPR говорить про необхідність перенесення точки прийняття тактичних рішень якомога ближче до виконавців, тобто СП має бути максимально самостійним, самодостатнім і компетентним у прийнятті рішень.

Знову ж таки, набуваючи можливості розглядати кожне СП як самостійну частину підприємства, нам набагато легше вирішити це завдання. Не важко оцінити, у що обходиться кожне СП і яку воно дає віддачу, наскільки продумана внутрішня структура СП та його місце у загальній структурі виробництва. Так само як і на рівні виробництва, можна оптимізувати бізнес-процеси на рівні окремого підрозділу. Нарешті, перенесення точки прийняття тактичних рішень всередину СП дозволяє якщо не скасувати зовсім, то принаймні суттєво полегшити роботу багатьох відділів, що функціонують на рівні підприємства (відділ кадрів, планування закупівель обладнання та проведення ремонтів тощо).

Функціональна частина підприємств різна і залежить від профілю підприємства. Тому візьмемо за основу розгляду типове (узагальнене) промислове підприємство, виробничий цикл якого можна уявити наступною схемою, показаною на рис. 8.15.

*Мал. 8.15.*Виробничий цикл промислового підприємства

Кожна фаза виробництва дробиться більш дрібні, наприклад, стадія «Сировина» полягає у пошуку постачальників, укладанні договорів, отриманні й оплаті рахунків, отриманні та складуванні сировини тощо. п. Поділ відбувається до отримання елементарних операцій, реалізованих як наборів сервісів.
Коли виконано розкладання вихідного завдання сервіси, можна розпочати комплектування посад. Посада визначається набором доступних та необхідних сервісів, тобто посада представима контейнером сервісів. У свою чергу, посади з'єднуються в структурні підрозділи. Таким чином, відбулося з'єднання функціональної та функціонально-незалежної частин. Ми зберегли можливість динамічної зміни як окремої посади, так і структурного підрозділу, отже, нам доступне і динамічне перепрофілювання підприємства загалом.
Система підтримує довільну кількість логічних шарів (аналог – багаторівневі системи клієнт – сервер). Шар зберігання інформації представлений середовищем зберігання (СУБД), шар відображення - середовищем відображення, заснованої на GUI (прикладами користувача), шар бізнес правил - схемами і т. д.
Кожен сервіс є групою класів (можливо, ієрархій). Класи можуть бути об'єднані у контейнери, властивості яких реалізуються у вигляді схем. Додаток, взаємодіючи з контейнерами явно чи опосередковано, запускає ті чи інші схеми, реалізуючи власну логіку роботи.

8.13. ОГЛЯД ОСОБЛИВОСТЕЙ ПРОЕКТІВ ПРИКЛАДНИХ СИСТЕМ

Проектуючи систему одного з перерахованих нижче типів, має сенс звернутися до одного з відповідних рішень. Далі розглядаються такі типи систем:
-системи пакетної обробки - обробка даних проводиться один раз для кожного набору вхідних даних;
- системи безперервної обробки - обробка даних проводиться безперервно над вхідними даними, що змінюються;
- Системи з інтерактивним інтерфейсом - Системи, керовані зовнішніми впливами;
- системи динамічного моделювання - системи, що моделюють поведінку об'єктів зовнішнього світу;
- Системи реального часу - системи, в яких переважають суворі тимчасові обмеження;
- Системи управління транзакціями - системи, що забезпечують сортування та оновлення даних; мають колективний доступ (типовою системою управління транзакціями є СУБД).
При розробці системи пакетної обробки необхідно виконати такі кроки:
- Розбиваємо повне перетворення на фази, кожна з яких виконує деяку частину перетворення; система описується діаграмою потоку даних, що будується розробки функціональної моделі;
- визначаємо класи проміжних об'єктів між кожною парою послідовних фаз, при цьому кожна фаза знає про об'єкти, розташовані на об'єктній діаграмі до та після неї (ці об'єкти представляють відповідно вхідні та вихідні дані фази);
- Складаємо об'єктну модель кожної фази (вона має таку ж структуру, що і модель всієї системи в цілому: фаза розбивається на підфази) і далі розробляємо кожну підфазу.
При розробці системи безперервної обробки необхідно виконати такі кроки:
— будуємо діаграму потоку даних (активні об'єкти на її початку та наприкінці відповідають структурам даних, значення яких безперервно змінюються, а сховища даних, пов'язані з її внутрішніми фазами, відображають параметри, що впливають на залежність між вхідними та вихідними даними фази);
- визначаємо класи проміжних об'єктів між кожною парою послідовних фаз, при цьому кожна фаза знає про об'єкти, розташовані на об'єктній діаграмі до та після неї (ці об'єкти представляють відповідно вхідні та вихідні дані фази);

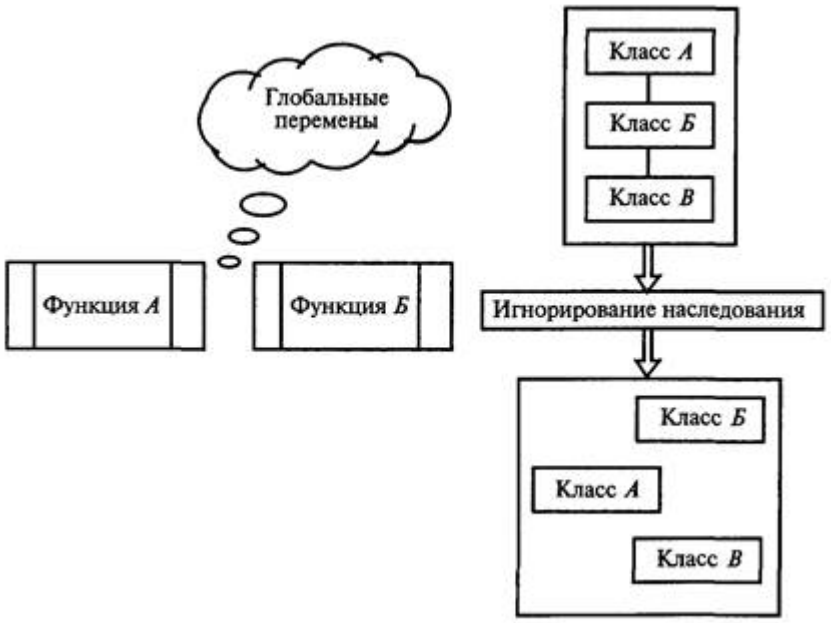
- представляємо кожну фазу як послідовність змін значень елементів вихідної структури даних залежно від значень елементів вхідної структури даних та значень, що одержуються зі сховища даних (значення вихідної структури даних формується частинами).
Під час розробки системи з інтерактивним інтерфейсом необхідно виконати такі кроки:
- Виділяємо об'єкти, що формують інтерфейс;
— якщо є можливість використовувати готові об'єкти для організації взаємодії (наприклад, для організації взаємодії системи з користувачем через екран дисплея можна використовувати бібліотеку системи X-Window, що забезпечує роботу з меню, формами, кнопками тощо);
— структуру програми визначаємо за її динамічною моделлю, а для реалізації інтерактивного інтерфейсу використовуємо паралельне управління (багатозадачний режим) або механізм спів-
буття (переривання), а не процедурне управління, коли час між виведенням чергового повідомлення користувачу та його відповіддю система проводить у режимі очікування;
- З безлічі подій виділяємо фізичні (апаратні, прості) події і намагаємося при організації взаємодії використовувати в першу чергу їх.
Під час розробки системи динамічного моделювання необхідно виконати такі кроки:
- за об'єктною моделлю визначаємо активні об'єкти; ці об'єкти мають атрибути з значеннями, що періодично оновлюються;
- Визначаємо дискретні події; такі події відповідають дискретним взаємодіям об'єкта (наприклад, включення живлення) та реалізуються як операції об'єкта;
- Визначаємо безперервні залежності (наприклад, залежності атрибутів від часу), при цьому значення таких атрибутів повинні періодично оновлюватися відповідно до залежності;
— моделювання управляється об'єктами, які відстежують тимчасові цикли послідовностей подій.
Розробка системи реального часу аналогічна до розробки системи з інтерактивним інтерфейсом.
Під час розробки системи управління транзакціями необхідно виконати такі кроки:
- Відобразити об'єктну модель на базу даних;
- Визначити асинхронно працюючі пристрої та ресурси з асинхронним доступом; у разі потреби визначити нові класи;
- Визначити набір ресурсів (у тому числі структур даних), до яких необхідний доступ під час транзакції (учасники транзакції);
- Розробити паралельне управління транзакціями; система може знадобитися кілька разів повторити невдалу транзакцію, перш ніж видати відмову.

8.14. ГІБРИДНІ ТЕХНОЛОГІЇ ПРОЕКТУВАННЯ

8.14.1. Ігнорування класів

Процедурно-орієнтований та об'єктно-орієнтований підходи до програмування різняться за своєю суттю і зазвичай ведуть до різних рішень одного завдання. Цей висновок вірний як для стадії реалізації, так і для стадії проектування: ви концентруєте увагу або на діях, що вживаються, або на сутності, що представляють, але не на тому й іншому одночасно.
Тоді чому метод об'єктно-орієнтованого проектування кращий за метод функціональної декомпозиції? Головна причина у тому, що функціональна декомпозиція не дає достатньої абстракції даних. А звідси вже випливає, що проект буде менш податливим до змін; менш пристосованим для використання різних допоміжних засобів; менш придатним для паралельного розвитку; менш придатним для паралельного виконання.
Справа в тому, що функціональна декомпозиція змушує оголошувати «важливі» дані глобальними, оскільки якщо система структурована як дерево функцій, будь-яке дане, доступне двом функцій, має бути глобальним по відношенню до них. Це призводить до того,

що «важливі» дані «спливають» до вершини дерева в міру того, як все більше функцій потребує доступу до них.
Так само відбувається у разі ієрархії класів з одним коренем, коли «важливі» дані спливають у напрямку до базового класу (рис. 8.16).



*Мал. 8.16.*Ігнорування класів та їх наслідування

8.14.2. Ігнорування наслідування

Розглянемо другий варіант – проект, який ігнорує спадкування. Вважати успадкування лише деталлю реалізації — означає ігнорувати ієрархію класів, яка може безпосередньо моделювати відносини між поняттями в галузі додатку. Такі відносини мають бути явно виражені у проекті, щоб дати можливість розробнику їх продумати.
Таким чином, політика «ніякого спадкування» призведе лише до того, що в системі буде відсутня цілісна загальна структура, а використання ієрархії класів буде обмежено певними підсистемами.

8.14.3. Ігнорування статичного контролю типів

Розглянемо третій варіант, що стосується проекту, у якому ігнорується статичний контроль типів. Поширені аргументи на користь відмови на стадії проектування від статичного контролю типів зводяться до того, що «типи — це продукт мов програмування» або що «природніше розмірковувати про об'єкти, не піклуючись про типи», або «статичний контроль типів змушує нас думати про реалізацію на ранньому етапі». Такий підхід цілком допустимий доти, доки він працює і не завдає шкоди.
Розглянемо таку аналогію: у фізичному світі ми постійно з'єднуємо різні пристрої, і існує уявлення нескінченним число стандартів на з'єднання. Головна особливість цих з'єднань - вони спеціально спроектовані таким чином, щоб унеможливити з'єднання двох пристроїв, не розрахованих на нього, тобто з'єднання має бути зроблено єдиним правильним способом. Ви не можете підключити радіотрансляційний приймач до розетки з високою напругою. Якби ви змогли це зробити, то спалили б приймач або згоріли самі.

Тут практично пряма аналогія: статичний контроль типів еквівалентний сумісності лише на рівні з'єднання, а динамічні перевірки відповідають захисту чи адаптації ланцюга. Результатом невдалого контролю як у фізичному, так і в програмному світі буде серйозна шкода. У великих системах використовуються обидва види контролю (рис. 8.17).
На ранньому етапі проектування цілком достатньо простого твердження:

*Мал. 8.17.*Ігнорування статичного контролю типів

«Ці два пристрої необхідно з'єднати», але незабаром стає суттєвим, як саме їх слід з'єднати: «Які гарантії дає з'єднання щодо поведінки пристроїв?» або «Виникнення якихось помилкових ситуацій можливе?», або «Яка приблизна ціна такого з'єднання?»

8.14.4. Гібридний проект

Перехід на нові методи роботи може бути болісним для будь-якої організації. Оскільки в об'єктно-орієнтованих мовах можливі кілька схем програмування, мова допускає поступовий перехід нею, використовуючи такі переваги такого переходу:
1) вивчаючи об'єктно-орієнтоване проектування, програмісти можуть продовжувати працювати за технологією структурного програмування;
2) в оточенні, бідному на програмні засоби, використання об'єктно-орієнтованих мов може принести значні вигоди.

*Мал. 8.18.*Гібридний проект

Ідея поступового, покрокового оволодіння об'єктно-орієнтованими мовами та технологій їх застосування, а також можливість змішування об'єктно-орієнтованого коду з кодом структурного програмування, природно, призводить до проекту, що має гібридний стиль. Більшість інтерфейсів можна поки що залишити на процедурному рівні, оскільки щось складніше не принесе негайного виграшу (рис. 8.18).

ВИСНОВКИ

• Процедурно-орієнтований та об'єктно-орієнтований підходи до програмування різняться за своєю суттю і зазвичай ведуть до різних рішень одного завдання.
• Об'єктно-орієнтований підхід допомагає подолати такі складні проблеми, як:
- Зменшення складності програмного забезпечення;
- Підвищення надійності програмного забезпечення;

- Забезпечення можливості модифікації окремих компонентів програмного забезпечення без зміни інших його компонентів;
- Забезпечення можливості повторного використання окремих компонентів програмного забезпечення.

• Методи об'єктно-орієнтованого проектування використовують як будівельні блоки об'єкти.
• Принципи абстрагування, інкапсуляції та модульності є взаємодоповнювальними. Об'єкт логічно визначає межі певної абстракції, а інкапсуляція та модульність роблять їх фізично непорушними.
• Спадкування виконує в ООП кілька важливих функцій:
- Модельує концептуальну структуру предметної області;
- Заощаджує описи, дозволяючи використовувати їх багаторазово для завдання різних класів;
- Забезпечує покрокове програмування великих систем шляхом багаторазової конкретизації класів.

• Класи з предметної (прикладної) області безпосередньо відображають поняття, які використовує кінцевий користувач для опису своїх завдань та методів їх вирішення.
• Ідеальний клас повинен мінімально залежати від решти світу. Кожен клас має набір поведінок та характеристик, що його визначають.
• При розбудові ієрархії класів застосовуються чотири процедури:
1) розщеплення класу на два і більше;
2) абстрагування (узагальнення);
3) злиття;
4) аналіз можливості використання існуючих розробок.

• Розробка проекту починається із складання функціональної моделі.
• Об'єктна модель представляє статичну структуру проектованої системи (підсистеми).
• Динамічна модель системи є діаграмою послідовності та діаграмою станів об'єктів.

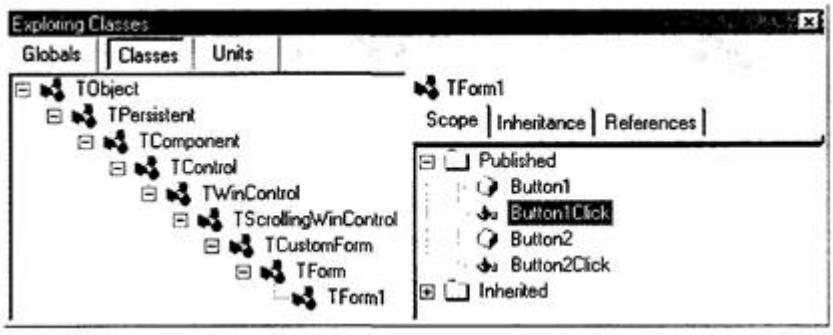
Контрольні питання

1. Під час вирішення яких проблем краще використовувати об'єктно-орієнтований підхід?
2. Які характеристики є фундаментальними в об'єктно-орієнтованому мисленні?
3. На яких засадах базується об'єктна модель?
4. Що таке патерн проектування?
5. Якому патерну відповідає динамічний та статичний контейнер А. Усова?
6. Які переваги надає об'єктна модель?
7. У чому полягають переваги інкапсуляції?
8. У чому важливість успадкування?
9. Навіщо корисний поліморфізм?
10. Що таке агрегування об'єкта?
11. З яких етапів складається процес побудови об'єктної моделі?
12. Як взаємодіють між собою об'єкти у програмі?
13. Які процедури застосовують при перебудові схеми успадкування класів?
14. Чому такий важливий аналіз функціонування системи?
15. У чому полягає зручність використання CRC-карток?
16. Які діаграми використовують у проектах середньої складності?

Тема 9 ВІЗУАЛЬНЕ ПРОГРАМУВАННЯ

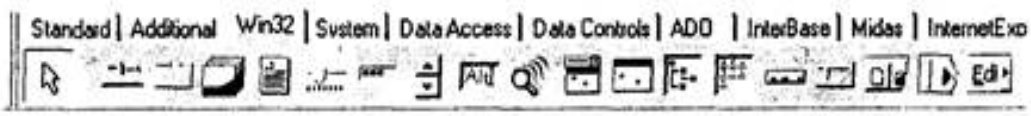
9.1. ЗАГАЛЬНЕ ПОНЯТТЯ ВІЗУАЛЬНОГО ПРОГРАМУВАННЯ

Візуальне програмування є в даний час. час однієї з найпопулярніших парадигм програмування. Візуальне програмування полягає в автоматизованій розробці програм із використанням особливої діалогової оболонки. Розглядаючи системи візуального програмування, легко побачити, що вони базуються на об'єктно-орієнтованому програмуванні і його логічним продовженням. Найчастіше візуальне програмування використовується для створення інтерфейсу програм та систем управління базами даних.
З об'єктно-орієнтованими системами асоціюється програма Browser (рис. 9.1). Цей засіб разом із системою екранних підказок дозволяє програмісту за бажанням переглядати деякі частини програмного оточення та бачити весь проект вже створеної програми. Під проектом програми тут розуміється структура програми — склад файлів, об'єктів та їх класів, що породжують, які складають програму в цілому.

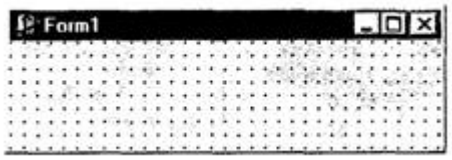


*Мал. 9.1.*Browser у Delphi 5.0

Одну з ключових можливостей програми Browser надає вікно, де знаходиться список всіх класів системи. При виборі одного із класів у спеціальних вікнах відображаються його локальні функції та змінні. Потім при виборі одного з методів окремої панелі висвічується його код. Зазвичай у системі є засоби для додавання та видалення класів з проекту. Програма Browser – це не просто візуалізатор. Це основний інтегруючий інструмент, який допомагає одночасно розглядати існуючу систему та розробляти документацію програмного проекту.
Структурною одиницею візуального програмування Delphi і C++Builder є компонента. Компонента є різновидом об'єкта, який можна перенести (агрегувати) у додаток зі спеціальної Палітри компонент (рис. 9.2). Компонент має набір властивостей, які можна змінювати, не змінюючи вихідний код програми.
Компоненти бувають візуальними та невізуальними. Перші призначені організації інтерфейсу з користувачем. Це різні кнопки, списки, статичний та редагований текст, зображення та багато іншого. Ці компоненти відображаються при виконанні програми, що розробляється. Невізуальні компоненти відповідають за доступ до системних ресурсів: драйверів баз даних, таймерів і т. д. Під час розробки вони відображаються своєю піктограмою, але при виконанні програми зазвичай невидимі. Компонента може належати або до іншої компоненти, або до форми.
Формою називається візуальна компонента, що має властивість вікна Windows (рис. 9.3). При розробці на формі містяться необхідні компоненти (наприклад, елементи необхідного діалогу). Форм у додатку може бути кілька - за необхідним числом вікон, що відкриваються при виконанні діалогу, їх можна додавати і видаляти.

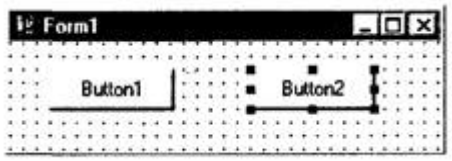


*Мал. 9.2.*Палітра компонент Delphi 5.0

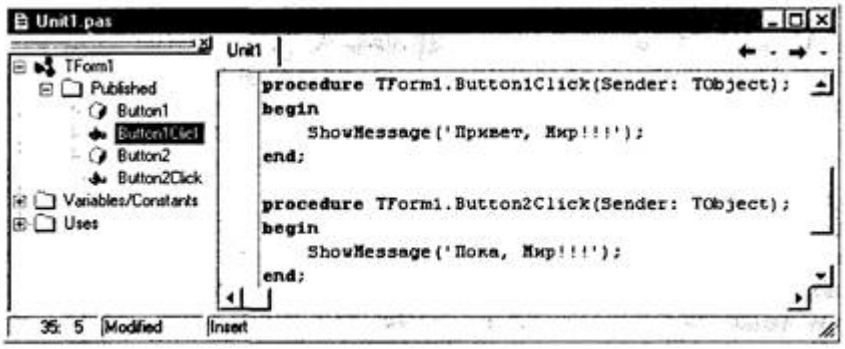


*Мал. 9.3.*Порожня форма

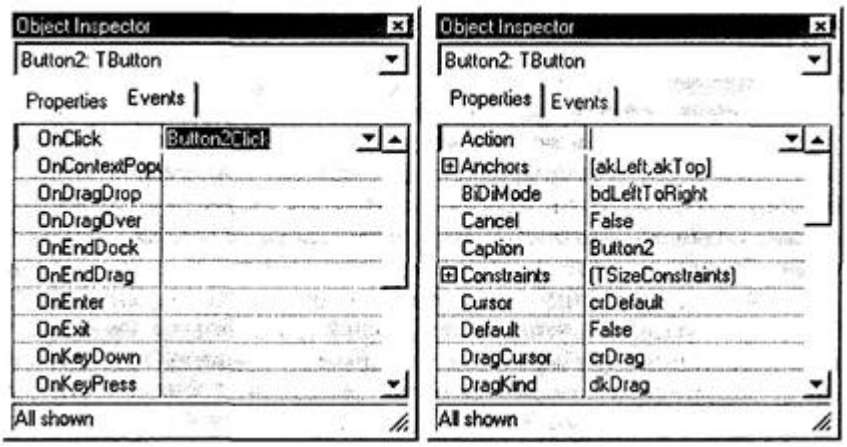
Програмні засоби розробки додатків, що відносяться до попереднього покоління, пропонують інтерактивні засоби вирішення типових завдань (майстри Borland C++ і Wizards або чарівники Visual C++), які дозволяють в діалозі з програмістом створювати і вставляти в програми готові фрагменти вихідного коду.
Технологія створення додатків у середовищі Delphi вийшла новий рівень розвитку цих ідей. У Delphi розробник з меню палітри компонент вибирає необхідну компоненту, наприклад кнопку, і буксирує її за допомогою миші в потрібне місце вікна форми, що розробляється. При цьому кнопці автоматично надається назва (ім'я або ідентифікатор), і вона описується в модулі форми (рис. 9.4).
Клацнувши на зображенні компоненти на формі, можна зробити її активною. Потім, переміщуючи за допомогою миші межі кнопки та працюючи у вікні Inspector, можна задати напис (наприклад, OK) та/або графічну піктограму на кнопці, задати кольори та інші параметри налаштування кнопки. Подвійне клацання по кнопці - і у вихідному тексті форми з'явиться шаблон підпрограми (методу) потрібного типу реакції на клацання (рис. 9.5). Працюючи у вікні редактора тексту, можна оформити тіло підпрограми реакції кнопки на клацання.



*Мал. 9.4.*Форма з двома компонентами - кнопками



*Мал. 9.5.*Оформлення події (методу) натискання кнопки Delphi 5.0



*Мал. 9.6.*Object Inspector у Delphi 5.0

Програма Inspector дозволяє входити у вихідні тексти методів (підпрограми обробки подій, названих Events), наприклад, натискання Enter, а також задавати початкові значення поля даних, названих Properties (рис. 9.6).

9.2. ТЕХНОЛОГІЯ ВІЗУАЛЬНОГО ПРОГРАМУВАННЯ

Початкові кроки технології візуального програмування визначаються оболонкою середовища візуального програмування. Спочатку створюються екранні форми найпростішим буксируванням миші. В інспекторі об'єктів провадиться налаштування їх властивостей шляхом заповнення окремих полів. На головну форму крім візуальних компонентів наносяться невізуальні компоненти. Форми об'єднуються у єдиний проект. Далі відповідно до сценарію діалогу програмуються методи події основної та підлеглих форм. Програми порожніх методів подій з'являються у вікні редактора після натискання відповідних клавіш або дій миші. «Порожні» методи доповнюються певними операторами активації та дезактивації форм. Після закінчення початкових кроків виходить працюючий «скелет» програми з джерелами даних із файлових баз даних та зі згенерованими формами документів, що виводяться на друк. Дослідник (Browser) забезпечує візуалізацію схеми ієрархії класів одержаного «скелета» програми. Інакше кажучи, технічний проект реалізованої частини програми формується автоматично.

Подальша розробка програми ведеться за технологією об'єктно-орієнтованого програмування. Можна частину програми реалізувати за технологією структурного програмування. Деякі відсутні візуальні та невізуальні компоненти виходять модифікацією вихідних текстів найближчих прототипів наявних компонентів. Рекомендується нові компоненти розміщувати на панелі компонентів. Це полегшить їх повторне використання у цій чи наступних разработках. Код, що стосується лише даної розробки, набирається за текстом програми.

ВИСНОВКИ

• Візуальне програмування багато в чому автоматизує працю програміста написання програм.

• Візуальне програмування – одна з найпопулярніших парадигм програмування на даний момент. Воно базується на технології ООП.

• Середовище візуального програмування підтримує роботу браузерів (Browser), за допомогою яких можна автоматично отримати документацію структури програми.

• Основним елементом у засобах візуального програмування є компонент. Компоненти бувають візуальними та невізуальними.
• Технологія візуального програмування полягає в наступному: створення екранних форм, нанесення візуальних та невізуальних компонентів, програмування подій та методів віконних форм.

Контрольні питання

1. Назвіть основні функції браузера.
2. Які компоненти бувають?
3. Що таке палітра компонент?
4. Опишіть функціональні можливості інспектора об'єктів.
5. Назвіть основні кроки технології візуального програмування.

Тема 10 CASE-ЗАСОБИ ТА ВІЗУАЛЬНЕ МОДЕЛЮВАННЯ

10.1. ПЕРЕДУМОВИ ПОЯВИ CASE ЗАСОБІВ

Тенденції розвитку сучасних інформаційних технологій спричиняють постійне ускладнення автоматизованих систем (АС). Для боротьби зі складністю проектів нині створено системи автоматизованого проектування (САПР) самих програмних проектів.

Для успішної реалізації проекту об'єкт проектування (АС) має бути насамперед адекватно описаний, мають бути побудовані повні, а також несуперечливі функціональні та інформаційні моделі АС. Накопичений на сьогодні досвід проектування АС показує, що це трудомістка і тривала робота.

Все це сприяло появі програмно-технологічних засобів особливого призначення - CASE-коштів, що реалізують CASE-інженерію створення та супроводу АС. Термін "CASE" (Computer Aided Software Engineering) використовується в даний час у широкому сенсі. Початкове значення терміна CASE, обмежене питаннями автоматизації розробки тільки програмного забезпечення (ПЗ), в даний час набуло нового змісту, що охоплює процес розробки складних АС в цілому.

Тепер під терміном «CASE-засоби» розуміються програмні засоби, що підтримують процеси створення та супроводу АС, включаючи аналіз та формулювання вимог, проектування прикладного ПЗ (додатків) та баз даних, генерацію коду, тестування, документування, забезпечення якості, конфігураційне управління та управління проектом , і навіть інші процеси. CASE-засоби разом із системним ПЗ та технічними засобами утворюють середовище розробки АС.

CASE-технологіяє методологією проектування АС, а також набір інструментальних засобів, що дозволяють у наочній формі моделювати предметну область, аналізувати цю модель на всіх етапах розробки та супроводу АС та розробляти додатки відповідно до інформаційних потреб користувачів. Більшість існуючих CASE-засобів засновані на методологіях структурного (в основному) або об'єктно-орієнтованого аналізу та проектування, що використовують специфікації у вигляді діаграм або текстів для опису зовнішніх вимог, зв'язків між моделями системи, динаміки поведінки системи та архітектури програмних засобів.

На підставі анкетування понад тисячу американських фірм фірмою Systems Development Inc. у 1996 р. було складено огляд передових технологій (Survey of Advanced Technology). Згідно з цим оглядом CASE-технологія наразі потрапила до розряду найбільш стабільних інформаційних технологій (її використала половина всіх опитаних користувачів більш ніж у третині своїх проектів, з них 85% завершилися успішно). Однак, незважаючи на всі потенційні можливості CASE-засобів, існує безліч прикладів їх невдалого впровадження. У зв'язку з цим слід зазначити таке:

- CASE-засоби не обов'язково дають негайний ефект, він може бути отриманий тільки через якийсь час;

— реальні витрати на використання CASE-коштів зазвичай набагато перевищують витрати на їх придбання;

— CASE-кошти забезпечують можливості для отримання суттєвої вигоди лише після успішного завершення процесу їх впровадження.
Для успішного впровадження CASE-засобів організація повинна мати такі якості, як:
• технологія – розуміння обмеженості існуючих можливостей та здатність прийняти нову технологію;
• культура – готовність до впровадження нових процесів та взаємовідносин між розробниками та користувачами;
• управління — чітке керівництво та організованість по відношенню до найважливіших етапів та процесів впровадження.

10.2. ОГЛЯД CASE-СИСТЕМ

На сьогоднішній день ринок програмного забезпечення має у своєму розпорядженні наступні найбільш розвинені CASE-засоби:
• Vantage Team Builder (Westmount I-CASE);
• Designer/2000;
• Silverrun;
• ERwin+BPwin;
• S-Designor;
• CASE.Аналітик;
• Rational Rose.
Крім того, на ринку постійно з'являються як нові для вітчизняних користувачів системи, так і нові версії та модифікації цих систем.
CASE-засіб Silverrun американської фірми "Computer Systems Advisers, Inc." (CSA) використовується для аналізу та проектування АС бізнес-класу та орієнтовано переважно на спіральну модель життєвого циклу. Воно застосовується для підтримки будь-якої методології, заснованої на окремій побудові функціональної та інформаційної моделей (діаграм потоків даних та діаграм «сутність-зв'язок»).
Платою за високу гнучкість та різноманітність образотворчих засобів побудови моделей є така вада Silverrun, як відсутність жорсткого взаємного контролю між компонентами різних моделей (наприклад, можливості автоматичного поширення змін між ДПД різних рівнів декомпозиції).
Для автоматичної генерації схем баз даних у Silverrun існують мости до найпоширеніших СУБД: Oracle, Informix, DB2, SQL Server та ін. Для передачі даних у засоби розробки додатків є мости до мов: , Delphi. Всі мости дозволяють завантажити в Silverrun RDM інформацію з каталогів відповідних СУБД або 4GL.
Vantage Team Builderє інтегрованим програмним продуктом, орієнтованим на реалізацію каскадної моделі життєвого циклу (ЖЦ) ПЗ та підтримку повного життєвого циклу ПЗ.
Система забезпечує виконання таких функцій:
- Проектування діаграм потоків даних, «сутність-зв'язок», структур даних, структурних схем програм та послідовностей екранних форм;
— проектування діаграм архітектури системи SAD (проектування складу та зв'язку обчислювальних засобів, розподіл завдань системи між обчислювальними засобами, моделювання відносин типу «клієнт-сервер», аналіз використання менеджерів транзакцій та особливостей функціонування систем в реальному часі);
- генерацію коду програм мовою 4GL цільової СУБД з повним забезпеченням програмного середовища та генерація SQL-коду для створення таблиць БД, індексів, обмежень цілісності та збережених процедур;
- програмування мовою C із вбудованим сервером SQL;
- Управління версіями та конфігурацією проекту;
- генерацію проектної документації за стандартними та індивідуальними шаблонами.
Vantage Team Builder функціонує на всіх основних UNIX-платформах (Solaris, SCO UNIX, AIX, HP-UX) та VMS.

CASE-засіб Designer/2000 2.0 фірми «ORACLE» є інтегрованим CASE-засобом, що забезпечує разом із засобами розробки додатків Developer/2000 підтримку повного життєвого циклу ПЗ для систем, що використовують СУБД ORACLE.
Базова методологія Designer/2000 (CASE Method) - структурна методологія проектування систем, що повністю охоплює всі етапи життєвого циклу АС. Designer/2000 забезпечує графічний інтерфейс розробки різних моделей (діаграм) предметної області. У процесі побудови моделей інформація про них заноситься до репозитарію.
Середовище функціонування Designer/2000 – Windows xt та ін.
ERwin - Засіб концептуального моделювання БД, що використовує методологію IDEF1X. ERwin реалізує проектування схеми БД, генерацію її опису мовою цільової СУБД (ORACLE, Informix, Ingres, Sybase, DB/2, Microsoft SQL Server, Progress та ін.) та реінженіринг існуючої БД. ERwin випускається в різних конфігураціях, орієнтованих на найбільш поширені засоби розробки додатків 4GL. Версія ERwin/OPEN повністю сумісна з засобами розробки програм PowerBuilder і SQLWindows і дозволяє експортувати опис спроектованої БД безпосередньо до репозитарію даних коштів.
Для ряду засобів розробки додатків (PowerBuilder, SQLWindows, Delphi, Visual Basic) виконується генерація форм та прототипів додатків.
Мережа версія Erwin ModelMart забезпечує узгоджене проектування БД та додатків у робочій групі.
BPwin - Засіб функціонального моделювання, що реалізує методологію IDEF0.
S-Designor 4.2 є CASE-засіб для проектування реляційних баз даних. За своїми функціональними можливостями і вартістю він близький до CASE-засобу ERwin, відрізняючись нотацією, що зовні використовується на діаграмах. S-Designor реалізує стандартну методологію моделювання даних та генерує опис БД для таких СУБД, як ORACLE, Informix, Ingres, Sybase, DB/2, Microsoft SQL Server та ін. Для існуючих систем виконується реінженіринг БД.
S-Designor сумісний із низкою засобів розробки додатків (PowerBuilder, Uniface, TeamWindows та інших.) і дозволяє експортувати опис БД у репозитарії даних. Для PowerBuilder виконується пряма генерація шаблонів програм.
CASE.Аналітик 1.1є практично єдиним у цей час конкурентоспроможним вітчизняним CASE-засобом функціонального моделювання. Його основні функції:
• аналіз діаграм та проектних специфікацій на повноту та несуперечність;
• отримання різноманітних звітів щодо проекту;
• генерація макетів документів відповідно до вимог ДСТУ 19.XXX та 34.XXX.
Середовище функціонування: процесор 386 і вище, основна пам'ять 4 Мб, дискова пам'ять 5 Мб, MS Windows xt та ін.
За допомогою окремого програмного продукту (Catherine) виконується обмін даними із CASE-засобом ERwin. При цьому з проекту, виконаного в CASE. Аналітик, експортується опис структур даних та накопичувачів даних, який за певними правилами формує опис сутностей та їх атрибутів.
І нарешті, **Rational Rose** — CASE-засіб фірми Rational Software Corporation (США) — призначений для автоматизації етапів аналізу та проектування ПЗ, а також для генерації кодів різними мовами та випуску проектної документації. Rational Rose використовує об'єднану методологію об'єктно-орієнтованого аналізу та проектування, засновану на підходах трьох провідних фахівців у цій галузі: Буча, Рамбо та Джекобсона. Розроблена ними універсальна нотація для моделювання об'єктів (UML – Unified Modeling Language) претендує на роль стандарту в галузі об'єктно-орієнтованого аналізу та проектування. Конкретний варіант Rational Rose визначається мовою, якою генеруються коди програм (C++, Smalltalk, PowerBuilder, Ada, SQL Windows і ObjectPro). Основний варіант – Rational Rose/C++ – дозволяє розробляти проектну документацію у вигляді діаграм та специфікацій, а також генерувати програмні коди на C++. Крім того, Rational Rose містить засоби

реінженірингу програм, що забезпечують повторне використання програмних компонентів у нових проектах.
Rational Rose працює на різних платформах: IBM PC (в середовищі Windows), Sun SPARC stations (UNIX, Solaris, SunOS), Hewlett-Packard (HP UX), IBM RS/6000 (AIX).

10.3. ВІЗУАЛЬНЕ МОДЕЛЮВАННЯ В RATIONAL ROSE

Вивчаючи вимоги до системи, ви берете за основу запити користувачів і перетворюєте їх на таку форму, яку ваша команда зможе зрозуміти і реалізувати. На основі цих вимог ви генеруєте код. Формально перетворюючи вимоги в код, ви гарантуєте їх відповідність один одному, а також можливість у будь-який момент повернутися від коду до вимог, що його породили. Цей процес називається моделюванням. Моделювання дозволяє простежити шлях від запитів користувачів до вимог моделі і потім коду і назад, не втрачаючи при цьому своїх напрацювань.
Візуальним моделюванням називають процес графічного представлення моделі з допомогою деякого стандартного набору графічних елементів. Наявність стандарту життєво необхідна реалізації однієї з переваг візуального моделювання — комунікації. Спілкування між користувачами, розробниками, аналітиками, тестувальниками, менеджерами та іншими учасниками проекту є основною метою графічного візуального моделювання.
Створені моделі надаються всім зацікавленим сторонам, які можуть отримати з них цінну інформацію. Наприклад, дивлячись на модель, користувачі візуалізують свою взаємодію з системою. Аналітики побачать взаємодію між об'єктами моделі. Розробники зрозуміють, які об'єкти потрібно створити та що ці об'єкти мають робити. Тестувальники візуалізують взаємодію між об'єктами, що дозволить їм збудувати тести. Менеджери побачать як всю систему загалом, і взаємодія її частин. Нарешті, керівники інформаційної служби, дивлячись на високорівневі моделі, зрозуміють, як взаємодіють друг з одним системи у тому організації. Таким чином, візуальні моделі надають потужний інструмент, що дозволяє показати систему, що розробляється всім зацікавленим сторонам.

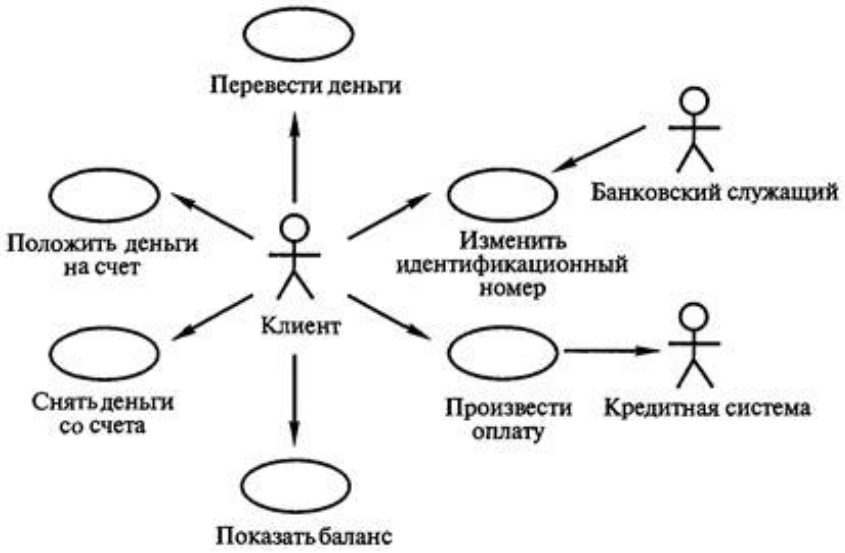
10.4. ДІАГРАМИ UML

10.4.1. Типи візуальних діаграм UML

UML дозволяє створювати кілька типів візуальних діаграм:
• діаграми варіантів використання;
• діаграми послідовності;
• кооперативні діаграми;
• діаграми класів;
• діаграми станів;
• діаграми компонентів;
• діаграми розміщення.
Діаграми ілюструють різні аспекти системи. Наприклад, кооперативна діаграма показує, як повинні взаємодіяти об'єкти, щоб реалізувати певну функціональність системи. Кожна діаграма має свою мету.

10.4.2. Діаграми варіантів використання

Діаграми варіантів використання відображають взаємодію між варіантами використання, що представляють функції системи, та дійовими особами, які представляють людей або системи, які отримують або передають інформацію до цієї системи. Приклад діаграми варіантів використання банківського автомата (ATM) показаний на рис. 10.1.

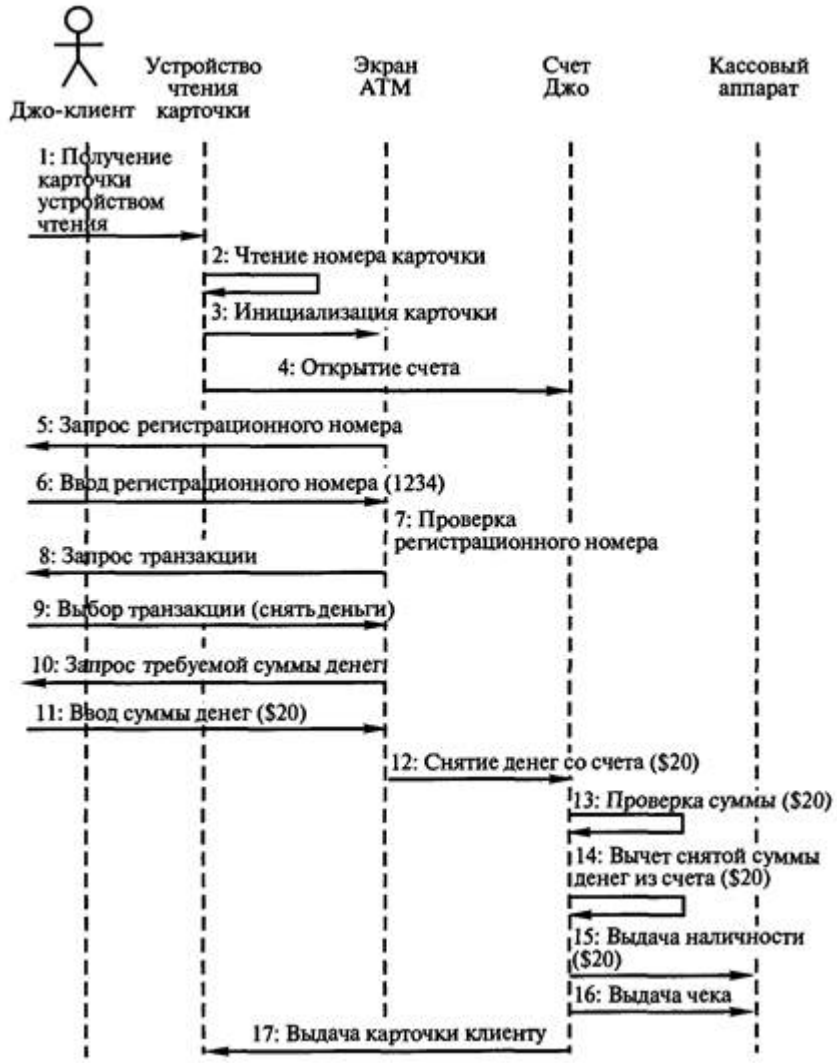


Мал. 10.1. Діаграма варіантів використання

На діаграмі представлено взаємодію між варіантами використання та дійовими особами. Вона відображає вимоги до системи з погляду користувача. Таким чином, варіанти використання - це функції, що виконуються системою, а дійові особи - це зацікавлені особи по відношенню до створюваної системи. Діаграми показують, які дійові особи ініціюють варіанти використання. З них також видно, коли дійова особа одержує інформацію від варіанта використання. По суті, діаграма варіантів використання може ілюструвати вимоги до системи. У прикладі клієнт банку ініціює різні варіанти використання: «Зняти гроші з рахунку», «Перевести гроші», «Покласти гроші на рахунок», «Показати баланс», «Змінити ідентифікаційний номер», «Здійснити оплату». Банківський службовець може ініціювати варіант використання "Змінити ідентифікаційний номер". Від варіанта використання «Здійснити оплату» йде стрілка до Кредитної системи. Діючими особами можуть бути й зовнішні системи, у цьому випадку Кредитна система показана саме як дійова особа — вона є зовнішньою для системи АТМ. Стрілка, спрямована від варіанта використання до дійової особи, показує, що варіант використання надає деяку інформацію чинній особі. В даному випадку варіант використання «Здійснити оплату» надає Кредитній системі інформацію про оплату за кредитною карткою.
З діаграм варіантів використання можна отримати багато інформації про систему. Цей тип діаграм визначає загальну функціональність системи. Користувачі, менеджери проектів, аналітики, розробники, фахівці з контролю якості та всі, кого цікавить система загалом, можуть, вивчаючи діаграми варіантів використання, зрозуміти, що система має робити.

10.4.3. Діаграми послідовності

Діаграми послідовності відбивають потік подій, які у рамках варіанта використання. Наприклад, варіант використання «Зняти гроші» передбачає кілька можливих послідовностей: зняття грошей, спроба зняти гроші за відсутності їх достатньої кількості на рахунку, спроба зняти гроші за неправильним ідентифікаційним номером та деякі інші. Нормальний сценарій зняття $20 з рахунку (за відсутності таких проблем, як неправильний ідентифікаційний номер або нестача грошей на рахунку) показано на рис. 10.2.



*10.2.*Діаграма послідовності зняття клієнтом Джо $20 з рахунку

У верхній частині діаграми показані всі дійові особи та об'єкти, необхідні системі для виконання варіанта використання «Зняти гроші». Стрілки відповідають повідомленням, що передаються між дійовою особою та об'єктом або між об'єктами для виконання потрібних функцій. Слід зазначити також, що у діаграмі послідовності показані саме об'єкти, а чи не класи. Класи є типами об'єктів. Об'єкти є конкретними; замість класу Клієнт на діаграмі послідовності подано конкретний клієнт Джо.
Варіант використання починається, коли клієнт вставляє свою картку в пристрій для читання – цей об'єкт показано у прямокутнику у верхній частині діаграми. Він зчитує номер картки, відкриває об'єкт "рахунок Джо" та ініціалізує екран АТМ. Екран запитує Джо його реєстраційний номер. Клієнт вводить число 1234. Екран перевіряє номер об'єкта «рахунок Джо» і виявляє, що він правильний. Потім екран надає Джо меню для вибору і той вибирає пункт «Зняти гроші». Екран запитує скільки він хоче зняти, і Джо вказує $20. Екран знімає гроші з рахунку. При цьому він ініціює серію процесів, які виконує об'єкт «рахунок Джо». У той самий час здійснюється перевірка, що у цьому рахунку лежать, по крайнього заходу, $20 і з рахунку віднімається необхідна сума. Потім касовий апарат отримує інструкцію "видати чек і $20 готівкою". Нарешті, той самий об'єкт «рахунок Джо» дає пристрою для читання карток інструкцію повернути картку.
Отже, дана діаграма послідовності ілюструє послідовність дій, що реалізують варіант використання "Зняти гроші з рахунку" на конкретному прикладі зняття клієнтом Джо $20.

Дивлячись на цю діаграму, користувачі знайомляться зі специфікою своєї роботи. Аналітики бачать послідовність (потік) дій, розробники - об'єкти, які треба створити, та їх операції. Фахівці з контролю якості зрозуміють деталі процесу та зможуть розробити тести для їхньої перевірки. Таким чином, діаграми послідовності корисні для всіх учасників проекту.

10.4.4. Кооперативні діаграми

Кооперативні діаграми відображають ту саму інформацію, що і діаграми послідовності. Однак роблять вони це інакше і з іншими цілями. Показана на рис. 10.2 діаграма послідовності представлена на рис. 10.3 як кооперативної діаграми.
Як і раніше, об'єкти зображені у вигляді прямокутників, а дійові особи у вигляді фігур. Якщо діаграма послідовності показує взаємодію між дійовими особами та об'єктами у часі, то на кооперативній діаграмі зв'язок з часом відсутній. Так, можна бачити, що пристрій для читання картки видає "рахунку Джо" інструкцію відкритися, а "рахунок Джо" змушує цей пристрій повернути картку власнику. Безпосередньо об'єкти, що взаємодіють, з'єднані лініями. Якщо, наприклад, пристрій читання картки спілкується безпосередньо з екраном АТМ, між ними слід провести лінію. Відсутність лінії означає, що безпосереднє повідомлення між об'єктами відсутнє.

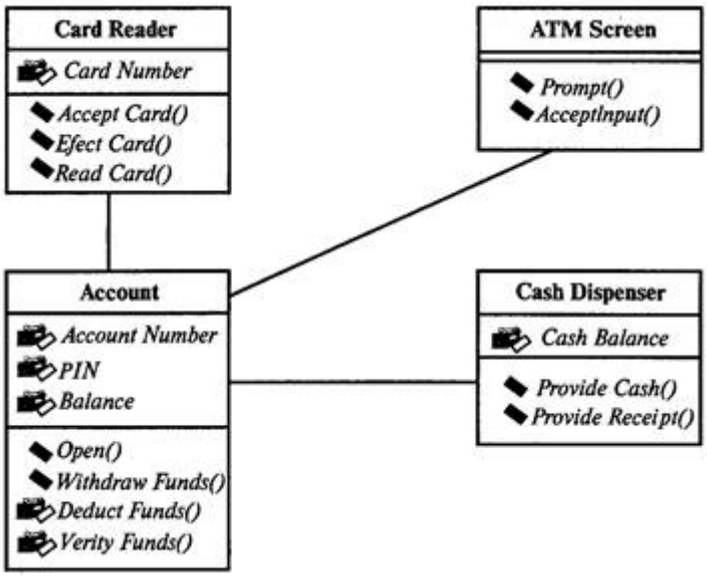
*Мал. 10.3.*Кооперативна діаграма, що описує процес зняття грошей з рахунку

Отже, на кооперативній діаграмі відображається та сама інформація, що і на діаграмі послідовності, але потрібна вона для інших цілей. Фахівці з контролю якості та архітектори системи зможуть зрозуміти розподіл між об'єктами. Припустимо, якась кооперативна діаграма нагадує зірку, де кілька об'єктів пов'язані з одним центральним об'єктом. Архітектор системи може зробити висновок, що система дуже залежить від центрального об'єкта, і необхідно перепроектувати її для більш рівномірного розподілу процесів. На діаграмі послідовності такий тип взаємодії було важко побачити.

10.4.5. Діаграми класів

Діаграми класів відбивають взаємодію між класами системи. Наприклад, "рахунок Джо" - це об'єкт. Типом такого об'єкта вважатимуться рахунок взагалі, т. е. «Рахунок» — це клас.

Класи містять дані та поведінку (дії), що впливає на ці дані. Так, клас Рахунок містить ідентифікаційний номер клієнта та дії, що перевіряють. На діаграмі класів клас створюється для кожного типу об'єктів із діаграм послідовності або кооперативних діаграм. Діаграма класів для варіанта використання "Зняти гроші" показана на рис. 10.4.
На діаграмі показані зв'язки між класами, які реалізують варіант використання "Зняти гроші". У цьому процесі задіяно чотири класи: Card Reader (пристрій для читання карток), Account (рахунок), ATM (екран ATM) та Cash Dispenser (касовий апарат). Кожен клас на діаграмі класів зображується у вигляді прямокутника, розділеного на три частини. У першій частині вказується ім'я класу, у другій його атрибути. Атрибут - це деяка інформація, що характеризує клас. Наприклад, клас Account (рахунок) має три атрибути: Account Number (номер рахунку), PIN (ідентифікаційний номер) та Balance (баланс). В останній частині містяться операції класу, що відображають його поведінку (дії класу). Сполучні класи лінії показують взаємодію між класами.



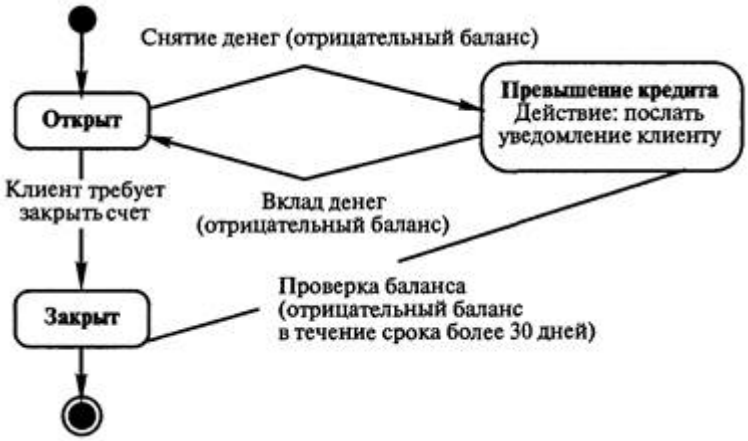
*Мал. 10.4.*Діаграма класів

Розробники використовують діаграми класів для створення класів. Такі інструменти, як Rose, генерують основу коду класів, яку програмісти заповнюють деталями обраною ними мовою. За допомогою цих діаграм аналітики можуть показати деталі системи, а архітектори зрозуміти її проект. Якщо, наприклад, якийсь клас несе надто велике функціональне навантаження, це буде видно на діаграмі класів, і архітектор зможе перерозподілити її між іншими класами. За допомогою діаграми можна виявити випадки, коли між сполученими класами не визначено жодних зв'язків. p align="justify"> Діаграми класів слід створювати, щоб показати взаємодіючі класи в кожному варіанті використання. Можна будувати також загальніші діаграми, що охоплюють усі системи чи підсистеми.

10.4.6. Діаграми станів

Діаграми станів призначені для моделювання різних станів, у яких може бути об'єкт. У той час, як діаграма класів показує статичну картину класів та їх зв'язків, діаграми станів застосовуються при описі динаміки поведінки системи.
Діаграми станів відображають поведінку об'єкта. Так, банківський рахунок може мати кілька різних статків. Він може бути відкритий, закритий або може бути перевищений кредит щодо нього. Поведінка рахунка змінюється залежно стану, у якому перебуває. На діаграмі станів

показують цю інформацію. На рис. 10.5 наведено приклад діаграми станів для банківського рахунку.



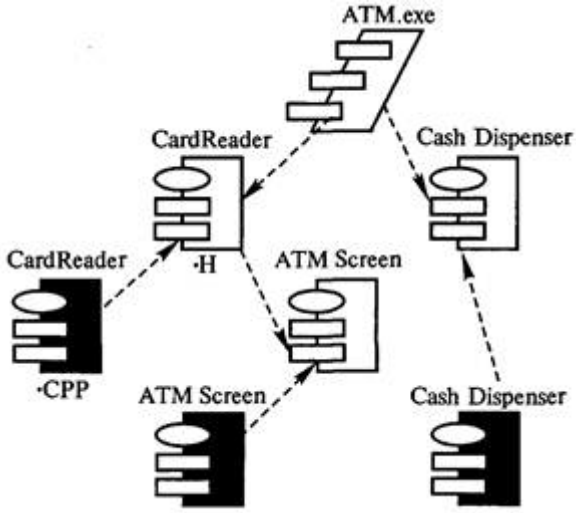
*Мал. 10.5.*Діаграма станів для класу Account

На цій діаграмі показані можливі стани рахунку, а також процес переходу рахунку з одного стану до іншого. Наприклад, якщо клієнт вимагає закрити відкритий рахунок, останній перетворюється на стан «Закритий». Вимога клієнта називається подією, саме події викликають перехід із одного стану в інший.
Коли клієнт знімає гроші з відкритого рахунку, рахунок може перейти у стан "Перевищення кредиту". Це відбувається, тільки якщо баланс по рахунку менший за нуль, що відображено умовою [негативний баланс] на нашій діаграмі. Укладене в квадратні дужки умова визначає, коли може чи може статися перехід із одного стану до іншого.
На діаграмі є два спеціальні стани - початковий і кінцевий. Початковий стан виділяється чорною точкою: він відповідає стану об'єкта на момент створення. Кінцевий стан позначається чорною точкою у білому гуртку: він відповідає стану об'єкта безпосередньо перед його знищенням. На діаграмі станів може бути один і лише один початковий стан. У той же час може бути стільки кінцевих станів, скільки вам потрібно, або їх може не бути взагалі.
Коли об'єкт перебуває у певному стані, можуть виконуватися ті чи інші процеси. У прикладі при перевищенні кредиту клієнту надсилається відповідне повідомлення. Процеси, що відбуваються, коли об'єкт перебуває у певному стані, називаються діями.
Діаграми станів не потрібно створювати для кожного класу, вони використовуються тільки в дуже складних випадках. Якщо об'єкт класу може існувати в кількох станах і в кожному з них поводиться по-різному, для нього, ймовірно, буде потрібно таку діаграму. Однак у багатьох проектах вони взагалі не використовуються. Якщо ж діаграми станів таки були побудовані, розробники можуть застосовувати їх під час створення класів.
Діаграми станів необхідні в основному для документування.

10.4.7. Діаграми компонент

Діаграми компонент показують, як виглядає модель фізично. На ній зображуються компоненти програмного забезпечення вашої системи та зв'язку між ними. При цьому виділяють два типи компонентів: виконувані компоненти та бібліотеки коду.
На рис. 10.6 зображено одну з діаграм компонент для системи ATM. На цій діаграмі показано компоненти клієнта системи ATM. У разі команда розробників вирішила будувати систему з допомогою мови C++. Кожен клас має свій власний заголовний файл і файл з розширенням .CPP, отже кожен клас перетворюється на власні компоненти на діаграмі. Виділена темна компонента називається специфікацією пакета і відповідає файлу тіла класу

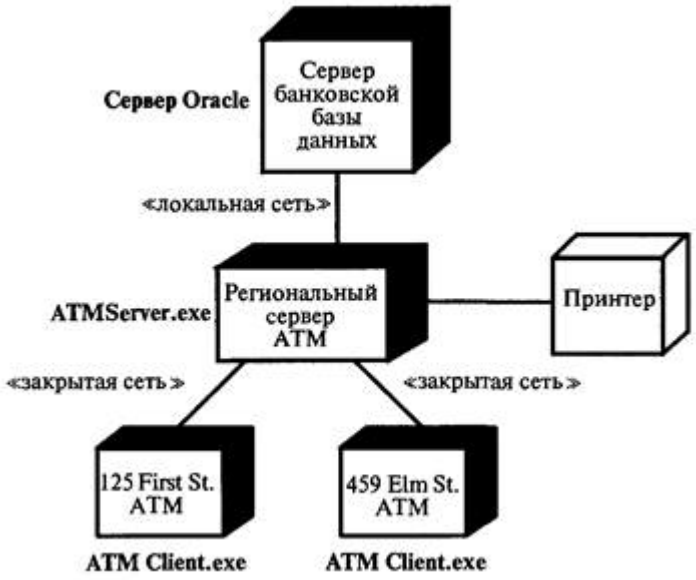
ATM мовою C++ (файл із розширенням .CPP). Невиділена компонента називається специфікацією пакета, але відповідає заголовному файлу класу мови C++ (файл з розширенням .H). Компонента ATM.exe є специфікацією завдання та представляє потік обробки інформації. У разі потік обробки — це виконувана програма.
Компоненти з'єднані штриховою лінією, що відображає залежність між ними. Система може мати кілька діаграм компонентів залежно від кількості підсистем або виконуваних файлів. p align="justify"> Кожна підсистема є пакетом компонент.
Діаграми компонентів застосовуються тими учасниками проекту, хто відповідає за компіляцію системи. Діаграма компонент дає уявлення про те, в якому порядку треба компілювати компоненти, а також які компоненти, що виконуються, будуть створені системою. Діаграма показує відповідність класів реалізованим компонентам. Отже, вона потрібна там, де починається створення коду.



*Мал. 10.6.*Діаграма компонент

10.4.8. Діаграми розміщення

Діаграми розміщення показують фізичне розташування різних компонентів системи в мережі. У прикладі система ATM складається з великої кількості підсистем, виконуваних на окремих фізичних пристроях чи вузлах. Діаграма розміщення системи ATM представлена на мал. 10.7.
З цієї діаграми можна дізнатися про фізичне розміщення системи. Клієнтські програми ATM працюватимуть у кількох місцях різних сайтах. Через закриті мережі буде здійснюватись повідомлення клієнтів з регіональним сервером ATM. На ньому працюватиме програмне забезпечення сервера ATM. У свою чергу, за допомогою локальної мережі регіональний сервер взаємодіятиме з сервером банківської бази даних, який працює під управлінням Oracle. Нарешті, з регіональним ATM сервером з'єднаний принтер.
Отже, ця діаграма показує фізичне розташування системи. Наприклад, наша система ATM відповідає трирівневій архітектурі, коли на першому рівні розміщується база даних, на другому регіональний сервер, а на третьому клієнт.



*10.7.*Діаграма розміщення

Діаграма розміщення використовується менеджером проекту, користувачами, архітектором системи та експлуатаційним персоналом для з'ясування фізичного розміщення системи та розташування її окремих підсистем. Менеджер проекту пояснить користувачам, як виглядатиме готовий продукт. Експлуатаційний персонал зможе планувати роботу із встановлення системи.

10.5. ВІЗУАЛЬНЕ МОДЕЛЮВАННЯ І ПРОЦЕС РОЗРОБКИ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

10.5.1. Переваги та недоліки типів процесу розробки

Програмне забезпечення може бути створене різними способами. Існує кілька різних типів процесу розробки, які можна використовувати у проекті: від «водоспаду» до объектно-ориентированного підходу. У кожного є свої переваги та недоліки. Тут не вказується, який саме процес проектування необхідно застосовувати розробникам своєї роботи, а представляємо лише короткий опис процесу, пов'язаного з візуальним моделюванням.
Довгий час програмне забезпечення розробляли, дотримуючись так званої моделі «водоспаду». Відповідно до неї необхідно було спочатку проаналізувати вимоги до майбутньої системи, спроектувати, створити та протестувати систему, а потім встановити її у користувачів. Як видно з назви, рух у зворотний бік по цьому ланцюжку було неможливим — вода не потече вгору. Цей метод був офіційною методологією, що застосовувалась у тисячах проектів, але в чистому вигляді, як його прийнято розуміти, він не використовувався жодного разу. Одним із головних недоліків моделі «водоспаду» є неможливість повернення назад на пройдені етапи. На початку проекту, наступного такої моделі, перед розробниками стоїть завдання, що бентежить — повністю визначити всі вимоги до системи. Для цього необхідно ретельно та всебічно обговорити з користувачами та дослідити бізнес-процеси. Користувачі повинні погодитися з усім тим, що з'ясовується в ході такого обстеження, хоча вони можуть до кінця не ознайомлюватися з його результатами. У такий спосіб на стадії аналізу при деякому везенні вдається зібрати близько 80% вимог до системи.
Потім розпочинається етап проектування. Необхідно сісти та визначити архітектуру майбутньої системи. Ми з'ясовуємо, де будуть встановлені програми та яка апаратура потрібна для досягнення прийнятної продуктивності. На даному етапі можуть виявитися нові

проблеми, їх необхідно знову обговорювати з користувачами, що виявиться у появі нових вимог до системи. Отже, доводиться повертатись до аналізу.
Після кількох таких кіл нарешті настає етап написання програмного коду. На цьому етапі виявляється, що деякі з ухвалених раніше рішень неможливо здійснити. Доводиться повертатися до проектування та переглядати ці рішення. Після завершення кодування настає етап тестування, у якому з'ясовується, що вимоги були досить деталізовані та його реалізація некоректна. Потрібно повертатись назад, на етап аналізу, та переглядати ці вимоги.
Нарешті, система готова та поставлена користувачам. Оскільки минуло вже багато часу і бізнес, ймовірно, вже встиг змінитися, користувачі сприймають її без великого ентузіазму, відповідаючи приблизно так: «Так, це те, що я просив, але не те, що хочу!» Ця магічна фраза, заклинання разом зістарить всю команду розробників щонайменше на 10 років!
Чи проблема полягає в тому, що правила бізнесу змінюються занадто швидко? Можливо, користувачі не можуть пояснити, чого вони хочуть чи не розуміють команду розробників? Чи сама команда не дотримується певного процесу? Відповіді на ці питання: так, та й ні. Бізнес змінюється дуже швидко, і професіонали-програмісти повинні встигати за цим. Користувачі не завжди можуть сказати, чого вони хочуть, оскільки їхня робота стала для них другою натурою. Запитувати про це банківського клерка, що прослужив 30 років, — приблизно те саме, що питати, як він дихає. Робота стала для нього настільки звичною, що її важко описати. Ще одна проблема полягає в тому, що користувачі не завжди розуміють команду розробників. Команда показує їм графіки, випускає томи тексту, що описує вимоги до системи, але користувачі не завжди розуміють, що їм дають. Чи є спосіб оминути цю проблему? Так, тут допоможе візуальне моделювання. І нарешті, команда розробників точно дотримується процесу — методу «водоспаду». На жаль, планування та реалізація методу – дві різні речі.
Отже, одна з проблем полягає в тому, що розробники використовували метод «водоспаду», який полягає в акуратному та послідовному проходженні через усі етапи проекту, але їм доводилося повертатися на пройдені етапи. Чи це відбувається через погане планування? Мабуть, ні. Розробка програмного забезпечення - складний процес, і його поетапне, акуратне виконання не завжди можливе. Якщо ж ігнорувати необхідність повернення, система буде містити дефекти проектування, і деякі вимоги будуть втрачені, можливі і серйозніші наслідки. Минули роки, допоки ми не навчилися заздалегідь планувати повернення на пройдені етапи.
Таким чином, ми дійшли ітеративної розробки. Ця назва означає лише, що ми збираємося повторювати ті самі етапи знову і знову. В об'єктно-орієнтованому процесі потрібно по багато разів невеликими кроками проходити етапи аналізу, проектування, розробки, тестування та встановлення системи.
Неможливо виявити всі вимоги ранніх етапах проектування. Ми враховуємо появу нових вимог у процесі розробки, плануючи проект у кілька ітерацій. У межах такої концепції проект можна як послідовність невеликих «водопадиков». Кожен із них досить великий, щоб означати завершення будь-якого важливого етапу проекту, але малий, щоб мінімізувати необхідність повернення назад. Ми проходимо чотири фази (етапу) проекту: початкова фаза, уточнення, конструювання та введення в дію.
Початкова фаза- Це початок проекту. Ми збираємо інформацію та розробляємо базові концепції. Наприкінці цієї фази приймається рішення продовжувати (чи продовжувати) проект.
У фазі уточнення деталізуються варіанти використання та приймаються архітектурні рішення. Уточнення включає деякий аналіз, проектування, кодування та планування тестів. У фазі конструювання розробляється переважна більшість коду.
Введення в дію— це завершальне компонування системи та встановлення її у користувачів.
Далі розглянемо, що означає кожна з цих фаз об'єктно-орієнтованому проекті.

10.5.2. Початкова фаза

Початкова фаза— це початок роботи над проектом, коли хтось каже: «А добре, щоб система робила...» Потім хтось ще вивчає ідею, і менеджер запитує, скільки часу вимагатиме її реалізація, скільки це коштуватиме і наскільки вона здійсненна. Початкова фаза таки полягає у тому, щоб знайти відповіді на ці питання. Ми досліджуємо властивості системи на високому рівні та документуємо їх. Визначаємо дійових осіб та варіанти використання, але не заглиблюємося в деталі варіантів використання, обмежуючись однією або двома пропозиціями. Готуємо також оцінки для найвищого керівництва. Отже, застосовуючи Rose для підтримки нашого проекту, створюємо дійових осіб та варіанти використання, а також будуємо діаграми варіантів використання. Початкова фаза завершується, коли це дослідження закінчено і для роботи над проектом виділені необхідні ресурси.
Початкова фаза проекту переважно послідовна і ітеративна. На відміну від неї інші фази повторюються кілька разів у процесі роботи над проектом. Оскільки проект може бути розпочато лише один раз, початкова фаза також виконується лише одного разу, тому в початковій фазі має бути вирішене ще одне завдання – розробка плану ітерацій. Це план, який описує, які варіанти використання, на яких ітераціях мають бути реалізовані. Якщо, наприклад, у початковій фазі виявлено 10 варіантів використання, можна створити наступний план:
Ітерація 1 Варіанти Використання 1, 5, 6
Ітерація 2 Варіанти Використання 7, 9
Ітерація 3 Варіанти Використання 2, 4, 8
Ітерація 4 Варіанти Використання 3, 10
План визначає, які варіанти використання треба реалізувати насамперед. Побудова плану вимагає розгляду залежностей між варіантами використання. Якщо для того, щоб міг працювати Варіант Використання 5, необхідна реалізація Варіанта Використання 3, то описаний вище план нездійсненний, оскільки Варіант Використання 3 реалізується на четвертій ітерації значно пізніше Варіанта Використання 5 з першої ітерації. Такий план слід переглянути, щоб врахувати всі залежності.

10.5.3. Використання Rose у початковій фазі

Деякі завдання початкової фази включають визначення варіантів використання і дійових осіб. Rose можна застосовувати для документування цих варіантів використання та дійових осіб, а також для створення діаграм, що показують зв'язок між ними. Отримані діаграми варіантів використання можна показати користувачам, щоб переконатися, що вони дають повне уявлення про властивості системи.
У фазі уточнення проекту виконуються деяке планування, аналіз та проектування архітектури. Наслідуючи план ітерації, уточнення проводиться для кожного варіанту використання в поточній ітерації. Уточнення включає такі аспекти проекту, як кодування прототипів, розробка тестів і прийняття рішень по проекту.
Основне завдання фази уточнення – деталізація варіантів використання. Вимоги низького рівня, що пред'являються до варіантів використання, передбачають опис потоку обробки даних усередині них, виявлення дійових осіб, розробку діаграм Взаємодії для графічного відображення потоку обробки даних, а також визначення всіх переходів станів, які можуть мати місце в рамках варіанту використання. p align="justify"> З вимог, визначених у формі деталізованих варіантів використання, складається документ під назвою «Специфікація вимог до програмного забезпечення».
У фазі уточнення виконуються такі завдання, як уточнення попередніх оцінок, вивчення моделі варіантів використання з погляду якості проектованої системи, аналіз ризиків. Можна уточнити модель варіантів використання, а також розробити діаграми послідовності та кооперативні діаграми для графічного представлення потоку обробки даних. Крім того, у цій фазі проектуються діаграми класів, що описують об'єкти, які необхідно створити.

Фаза уточнення завершується, коли варіанти використання повністю деталізовані та схвалені користувачами, прототипи завершені настільки, щоб зменшити ризики, розроблені діаграми класів. Іншими словами, ця фаза пройдена, коли система спроектована, розглянута та готова для передачі розробникам.

Фаза уточнення пропонує кілька можливостей для застосування Rational Rose. Оскільки уточнення – це деталізація вимог до системи, модель варіантів використання може вимагати оновлення. Діаграми послідовності та кооперативні діаграми допомагають проілюструвати потік обробки даних при його деталізації. З їхньою допомогою можна також спроектувати необхідні для системи об'єкти. Уточнення передбачає підготовку проекту системи передачі розробникам, які розпочнуть її конструювання. У середовищі Rose може бути виконано шляхом створення діаграм класів і діаграм станів.

Конструювання- Процес розробки та тестування програмного забезпечення. Як і у разі уточнення, ця фаза виконується для кожного набору варіантів використання кожної ітерації. Завдання конструювання включають визначення всіх вимог, розробку і тестування програмного забезпечення (ПО). Так як ПЗ повністю проектується у фазі уточнення, конструювання не передбачає великої кількості рішень щодо проекту, що дозволяє команді працювати паралельно. Це означає, що різні групи програмістів можуть одночасно працювати над різними об'єктами, знаючи, що після завершення фази система «зійдеться». У фазі уточнення ми проектуємо об'єкти системи та їхню взаємодію. Конструювання лише запускає проект у дію, а нових рішень щодо нього, здатних змінити цю взаємодію, не приймається.

Ще однією перевагою такого підходу до моделювання системи є те, що середовище Rational Rose здатне генерувати «скелетний код» системи. Для використання цієї можливості слід розробити компоненти та діаграму компонентів на ранньому етапі конструювання. Генерацію коду можна розпочати відразу після створення компонентів та нанесення на діаграму залежностей між ними. В результаті буде автоматично побудований код, який можна створити, ґрунтуючись на проекті системи. Це не означає, що за допомогою Rose можна отримати будь-який код, що реалізує бізнес-логіку програм. Результат сильно залежить від мови програмування, але в загальному випадку передбачає визначення класів, атрибутів, областей дії. Це дозволяє заощадити час, тому що написання коду вручну — досить копітка та стомлююча робота. Отримавши код програмісти можуть сконцентруватися на специфічних аспектах, пов'язаних з бізнес-логікою. Ще одна група розробників має виконати експертну оцінку коду, щоб переконатися в його функціональності та відповідності стандартам та угодам щодо проекту. Потім об'єкти повинні бути оцінені якістю. Якщо у фазі конструювання були додані нові атрибути або функції або якщо були змінені взаємодії між об'єктами, код слід перетворити на модель Rose за допомогою зворотного перетворення. Конструювання можна вважати завершеним, коли програмне забезпечення готове та протестоване. Важливо переконатися в адекватності моделі та програмного забезпечення. Модель буде надзвичайно корисною у процесі супроводу ПЗ.

У фазі конструювання пишеться більшість коду проекту. Rose дозволяє створити компоненти відповідно до проектування об'єктів. Щоб показати залежність між компонентами на етапі компіляції, створюються діаграми компонентів. Після вибору мови програмування можна здійснити генерацію скелетного коду кожного компонента. Після завершення роботи над кодом модель можна привести у відповідність з ним за допомогою зворотного проектування.

Фаза введення в дію настає, коли готовий програмний продукт передають користувачам. Завдання у цій фазі передбачають завершення роботи над фінальною версією продукту, завершення приймального тестування, завершення складання документації та підготовку до навчання користувачів. Щоб відобразити останні внесені зміни, слід оновити специфікацію вимог до програмного забезпечення, діаграми варіантів використання, класів, компонентів та розміщення. Важливо, щоб ваші моделі були синхронізовані з готовим продуктом, оскільки вони використовуватимуться під час його супроводу. Крім того, моделі будуть неоціненними

при внесенні удосконалень у створену систему вже за кілька місяців після завершення проекту. У фазі введення в дію Rational Rose не така корисна, як в інших фазах. У цей момент програма вже створена. Rose призначена для надання допомоги при моделюванні та розробці програмного забезпечення і навіть при плануванні його розміщення. Однак Rose не є інструментом тестування і не здатна допомогти у плануванні тестів або процедур, пов'язаних із розміщенням програмного забезпечення. Для цього створені інші продукти. Отже, у фазі введення в дію Rose застосовується передусім для оновлення моделей після завершення роботи над програмним продуктом.

10.6. РОБОТА НАД ПРОЕКТОМ У СЕРЕДОВИЩІ RATIONAL ROSE

З усіх розглянутих видів канонічних діаграм серед Rational Rose 98/98i не підтримується лише діаграма діяльності.

У ході роботи над діаграмами проекту є можливість видалення та додавання відповідних графічних елементів, встановлення відносин між цими елементами, їх специфікації та документування.

Загальна послідовність роботи над проектом аналогічна послідовності розгляду канонічних діаграм у книзі.

Однією з найпотужніших властивостей середовища Rational Rose є можливість генерації програмного коду після побудови та перевірки моделей. Загальна послідовність дій, які необхідно виконати для цього, складається із шести етапів:

- Перевірки моделі незалежно від вибору мови генерації коду;
- Створення компонентів для реалізації класів;
- Відображення класів на компоненти;
- Налаштування властивостей генерації програмного коду;
- вибору класу, компонента чи пакета;
- генерації програмного коду.

ВИСНОВКИ

• CASE-кошти дозволяють в автоматизованому режимі реалізувати проектні моделі.

• Реалізовані проектні моделі повинні бути повними, відображати як функціональні, так і інформаційні аспекти автоматизованих систем, що проектуються.

• CASE-засоби включають набір інструментальних засобів, що дозволяють у наочній формі моделювати предметну область, аналізувати цю модель на всіх етапах розробки та супроводу програмного проекту та розробляти програми відповідно до інформаційних потреб користувачів.

• Більшість існуючих CASE-засобів засновані на методологіях структурного (в основному) або об'єктно-орієнтованого аналізу та проектування, що використовують специфікації у вигляді діаграм або текстів для опису зовнішніх вимог, зв'язків між моделями системи, динаміки поведінки системи та архітектури програмних засобів.

• Передові засоби SESE здатні не тільки складати специфікації, але і їх перевіряти, а також генерувати вихідний код програм.

• CASE-засоби виробляються безліччю виробників і лише найвдаліші з них проходять перевірку практикою.

• CASE-засіб Rational Rose фірми «Software Corporation» (США) призначений для автоматизації етапів аналізу та проектування ПЗ, а також для генерації кодів різними мовами та випуску проектної документації. Rational Rose використовує об'єднану методологію об'єктно-орієнтованого аналізу та проектування, засновану на підходах трьох провідних фахівців у цій галузі: Буча, Рамбо та Джекобсона.

• Мова UML CASE-засоби Rational Rose дозволяє створювати кілька типів візуальних діаграм:

- Діаграми варіантів використання;
- Діаграми послідовності;
- Кооперативні діаграми;

- Діаграми класів;
- Діаграми станів;
- Діаграми компонент;
- Діаграми розміщення.

Контрольні питання

1. Що таке CASE-кошти?
2. Навіщо необхідні CASE-кошти?
3. У чому полягає суть візуального моделювання?
4. Що відображає діаграми варіантів використання?
5. Що відображають діаграми послідовності?
6. Що відображають кооперативні діаграми?
7. Що відображають діаграми класів?
8. Що відображає діаграми станів?
9. Що відображають діаграми компонентів?
10. Що відображають діаграми розміщення?
11. У чому полягає суть моделі розробки програмного забезпечення «водоспад», її особливості та недоліки?
12. Викладіть кроки методики розробки програм із використанням Rational Rose.

Тема 11 ТЕСТУВАННЯ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

11.1. ОСНОВНІ ВІДОМОСТІ

Будь-який замовник хоче отримати надійний програмний виріб, який повністю задовольняє його потреби. Різні рівні надійності забезпечуються різними інженерними підходами до тестування. Іншими словами, за достатні кошти можна досягти рівня надійності як такої фірми або навіть рівня, необхідного для атомної енергетики або космічних досліджень. Отже, рівень надійності програмних виробів визначається інженерією тестування. Як досягти найвищого рівня надійності? Зрозуміло, що тексти програм, написані однієї організації, можна заново інспектувати, тестувати автономне у складі ядра в інший організації. У самій програмі можна одночасно проводити розрахунки за різними алгоритмами з використанням різних обчислювальних методів і злічувати результати в контрольних точках, використовувати підстановки розрахованих результатів у вихідні рівняння і цим контролювати результати рішення. З викладеного видно, що рівень надійності програмних виробів безпосередньо з витратами як коштів, і часу проекту.

Як відомо, при створенні типового програмного проекту близько 50% загального часу і понад 50—60% загальної вартості витрачається на перевірку (тестування) програми або системи, що розробляється. Крім того, частка вартості тестування у загальній вартості програм має тенденцію зростати зі збільшенням складності програмних виробів та підвищення вимог до їх якості.

Враховуючи це, при виборі способу тестування програм слід чітко виділяти певну (по можливості не дуже велику) кількість правил налагодження, що забезпечують високу якість програмного продукту та знижують витрати на його створення.

Тестування здійснюється шляхом виконання тестів. Сенс тесту програм показано на мал. 11.1.

Аксіоми тестування, висунуті провідними програмістами:

— добрий той тест, для якого висока ймовірність виявлення помилки;

- Головна проблема тестування - вирішити, коли закінчити (зазвичай вирішується просто - закінчуються гроші);
- Неможливо тестувати свою власну програму;
- Необхідна частина тестів - опис вихідних результатів;
- уникайте невідтворюваних тестів;
- готуйте тести як для правильних, так і неправильних даних;
- не тестуйте «з літа»;
- Детально вивчайте результати кожного тесту;

— у міру виявлення дедалі більшої кількості помилок у певному модулі чи програмі, зростає ймовірність виявлення у ній ще більшої кількості помилок;

— тестують програми найкращі уми;

- вважають тестованість головним завданням розробників програми;
- не змінюй програму, щоб полегшити тестування;
- Тестування має починатися з постановки цілей.

Якщо в програмі ставлять коментарі на місці виклику модуля замість використання заглушки, виключають можливість перевірки типів даних, а також часто забувають знімати коментарі. Для пошуку «забутих» коментарів необхідне трудомістке налагодження.
Серед прийомів тестування варто виділити також так званий друк налагодження. Якщо налагоджувальні печатки вилучаються з тексту, супровід ускладнюється. Висновок: ніколи не вилучай налагоджувальні печатки навіть з використанням препроцесора:

```
{$IFDEF DEBUG THEN}
...
{$ELSE}
...
{$ENDIF}
```

*Мал. 11.1.*Сенс тесту програм

Краще опишіть глобальну змінну DebugLevel та програмуйте умовні налагоджувальні друку:

```
var
DebugLevel: слово; {0-немає жодного налагоджувального друку, чим більше
значення, тим докладніше налагоджувальний друк}
DebugFile: text; {файл налагоджувального друку}
DebugLevel: = 4; {завдання рівня налагодження}
if DebugLevel >= 3 then
WriteLn(DebugFile, 'Модуль:', 'MyModule, 'результат:');
```

11.2. ВЛАСТИВОСТІ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

Слід виділити такі властивості програмного забезпечення.
Коректність програмного забезпечення — властивість безпомилкової реалізації необхідного алгоритму за відсутності таких факторів, що заважають, як помилки вхідних даних, помилки операторів ЕОМ (людей), збої та відмови ЕОМ.
В інтуїтивному сенсі під коректністю розуміють властивості програми, що свідчать про відсутність у ній помилок, допущених розробником на різних етапах проектування (специфікації, проектування алгоритму та структур даних, кодування). Коректність самої програми розуміють по відношенню до цілей, поставлених перед її розробкою (тобто це відносна властивість).
Стійкість - Властивість здійснювати необхідне перетворення інформації при збереженні вихідних рішень програми в межах допусків, встановлених специфікацією. Стійкість характеризує поведінка програми під впливом неї таких чинників нестійкості, як помилки операторів ЕОМ, і навіть не виявлені помилки програми.
Відновлюваність - властивість програмного забезпечення, що характеризує можливість пристосовуватися до виявлення помилок та їх усунення.
Надійність можна уявити сукупністю наступних характеристик:
- *Цілісністю програмного засобу* (здатністю його до захисту від відмов);
- *живучістю* (здатністю до вхідного контролю даних та їх перевірки під час роботи);
- **Завершеністю** (бездефектністю готового програмного засобу, характеристикою якості його тестування);
- *Працездатністю* (здатністю програмного засобу до відновлення своїх можливостей після збоїв).

Відмінність поняття коректності та надійності програм полягає в наступному:
- надійність характеризує як програму, так і її «оточення» (якість апаратури, кваліфікацію користувача тощо);

— говорячи про надійність програми, зазвичай допускають певну, хоч і малу частку *помилок* у ній і оцінюють *ймовірність* їх появи.

Повернемося до поняття коректності. Вочевидь, що ні всяка синтаксично правильна програма є коректною у зазначеному вище сенсі, т. е. коректність характеризує семантичні властивості програм.

З урахуванням специфіки появи помилок у програмах можна виділити дві сторони поняття коректності:

- коректність як точна відповідність цілям розробки програми (які відображені у специфікації) за умови її завершення або часткова коректність;

— завершення програми, т. е. досягнення програмою у її виконання своєї кінцевої точки.

Залежно від виконання чи невиконання кожного із двох названих властивостей програми розрізняють шість завдань аналізу коректності:

1) доказ часткової коректності;
2) доказ часткової некоректності;
3) доказ завершення програми;
4) доказ не завершення програми;
5) доказ тотальної (повної) коректності (тобто одночасне рішення 1-ї та 3-ї задач);
6) доказ некоректності (вирішення 2-го чи 4-го завдання). Методи доказу часткової коректності програм, як

правило, спираються на аксіоматичний підхід до формалізації семантики мов програмування. Аксіоматична семантика мови програмування є сукупністю аксіом і правил виведення. За допомогою аксіом задається семантика простих операторів мови (присвоєння, введення-виведення, виклику процедур). З допомогою правил виведення описується семантика складових операторів чи керуючих структур (послідовності, умовного вибору, циклів). Серед цих правил виведення слід зазначити правило виведення для операторів циклу, оскільки воно вимагає знання інваріанту циклу (формули, істинність якої не змінюється за будь-якого проходження циклу).

Найбільш відомим із методів доказу часткової коректності програм є метод індуктивних тверджень, запропонований Флойдом та вдосконалений Хоаром. Один із важливих етапів цього методу – отримання анотованої програми. На цьому етапі для синтаксично правильної програми повинні бути задані твердження мовою логіки предикатів першого порядку: вхідний предикат; вихідний предикат.

Ці твердження задаються для вхідної точки циклу та мають характеризувати семантику обчислень у циклі.

Доказ несправжності умов коректності свідчить про неправильність програми або її специфікації, або програми та специфікації.

За впливом на результати обробки інформації до надійності та стійкості програмного забезпечення близька і точність програмного забезпечення, що визначається помилками методу та помилками представлення даних.

Найбільш простими методами оцінки надійності програмного забезпечення є емпіричні моделі, що ґрунтуються на досвіді розробки програм: якщо до початку тестування на 1000 операторів припадає 10 помилок, а прийнятною якістю є 1 помилка, то в ході тестування треба знайти:

$$N \text{ ошибок} = \frac{N \text{ операторов}}{1000}.$$

Точніша модель Холстеда: N помилок = N операторів * log2 (N операторів - N операндів), де N операторів - число операторів у програмі; N операндів - число операндів у програмі.

Емпірична модель фірми IBM:

N помилок = 23 M(10) + 2 M(1),

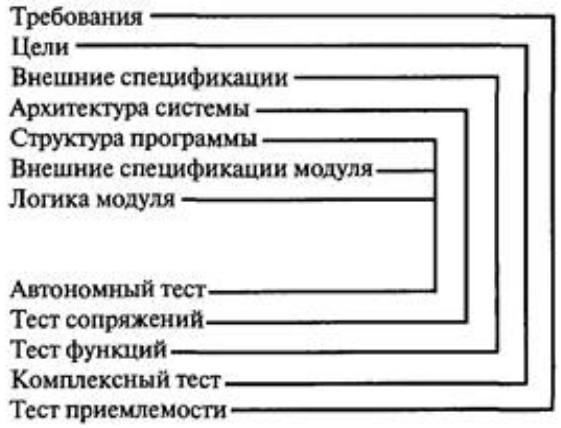
де M(10) — число модулів із 10 і більше виправленнями; M(1) — число модулів із менше 10 виправленнями.
Якщо в модулі виявлено більше 10 помилок, його програмують заново.
За методом Мілса в програму, що розробляється, вносять заздалегідь відому число помилок. Далі вважають, що темпи виявлення помилок (відомих та невідомих) однакові.

11.3. ЗВ'ЯЗОК ПРОЦЕСІВ ТЕСТУВАННЯ З ПРОЦЕСОМ ПРОЕКТУВАННЯ

З рис. 11.2 видно, що помилки на ранніх етапах проекту вичерпно можуть бути виявлені наприкінці роботи.
Тестування програм охоплює низку видів діяльності:
• постановку задачі;
• проектування тестів;
• написання тестів;
• тестування тестів;
• виконання тестів;
• Вивчення результатів тестування.
Тут найважливішим є проектування тестів.
Отже, тестування це процес виконання програми з метою виявлення помилок.

11.4. ПІДХОДИ ДО ПРОЕКТУВАННЯ ТЕСТІВ

Розглянемо два найбільш протилежні підходи до проектування тестів.
Прихильник першого підходу орієнтується лише на стратегію тестування, яка називається стратегією «чорної скриньки», тестуванням з управлінням за даними або тестуванням з управлінням по входу-виходу. При використанні цієї стратегії програма розглядається як чорна скринька. Тестові дані використовуються лише відповідно до специфікації програми (тобто без урахування знань про її внутрішню структуру). Недосяжний ідеал прихильника першого підходу – перевірити всі можливі комбінації та значення на вході. Зазвичай їх дуже багато навіть для найпростіших алгоритмів. Так, для програми розрахунку середнього арифметичного чотирьох чисел треба готувати 107 тестових даних.



*Мал. 11.2.*Взаємозв'язок процесів проектування та тестування

При першому підході виявлення всіх помилок у програмі є критерієм вичерпного вхідного тестування. Останнє може бути досягнуто, якщо як тестові набори використовувати всі можливі набори вхідних даних. Отже, приходимо до висновку, що для вичерпного тестування програми потрібна нескінченна кількість тестів, а отже, побудова вичерпного вхідного тесту неможлива. Це підтверджується двома аргументами: по-перше, не можна створити тест, який би гарантував відсутність помилок; по-друге, розробка таких тестів суперечить економічним вимогам. Оскільки вичерпне тестування виключається, нашою метою має стати максимізація результативності капіталовкладень у тестування (максимізація

числа помилок, які виявляються одним тестом). Для цього необхідно розглядати внутрішню структуру програми і робити деякі розумні, але, звісно, припущення, що не мають повної гарантії достовірності.

Прихильник другого підходу використовує стратегію «білої скриньки», або стратегію тестування, керовану логікою програми, що дозволяє досліджувати внутрішню структуру програми. І тут тестувальник отримує тестові дані шляхом аналізу лише логіки програми; прагне, щоб кожну команду було виконано хоча б один раз. При достатній кваліфікації домагається, щоб кожна команда умовного переходу виконувалася в кожному напрямі хоча б один раз. Цикл повинен виконуватися один раз, жодного разу, максимальну кількість разів.

Мета тестування всіх шляхів ззовні також недосяжна. У програмі двох послідовних циклів усередині кожного з них включено розгалуження на десять шляхів, є 1018 шляхів розрахунку. При цьому виконання всіх шляхів розрахунку не гарантує виконання всіх специфікацій. Для довідки: вік Всесвіту 1017с.

Порівняємо спосіб побудови тестів за цієї стратегії з вичерпним вхідним тестуванням стратегії «чорного ящика». Невірно припущення, що достатньо побудувати такий набір тестів, у якому кожен оператор виконується хоча б один раз. Вичерпне вхідне тестування може бути поставлене у відповідність вичерпне тестування маршрутів. Мається на увазі, що програма перевірена повністю, якщо за допомогою тестів вдається здійснити виконання цієї програми по всіх можливих маршрутах потоку (графа) передач управління.

Останнє твердження має два слабкі пункти: по-перше, число маршрутів, що не повторюють один одного, — астрономічне; по-друге, навіть якщо кожен маршрут може бути перевірений, сама програма може містити помилки (наприклад, деякі маршрути пропущено).

Властивість шляху виконуватися правильно для одних даних і неправильно для інших - зване чутливістю до даних, що найчастіше проявляється за рахунок чисельних похибок і похибок усічення методів. Тестування кожного зі всіх маршрутів одним тестом не гарантує виявлення чутливості до даних.

В результаті всіх викладених вище зауважень зазначимо, що ні вичерпне вхідне тестування, ні вичерпне тестування маршрутів не можуть стати корисними стратегіями, тому що вони не реалізуються. Тому реальним шляхом, який дозволить створити хорошу, але, звісно, не абсолютну стратегію, є поєднання тестування програми кількома методами.

Розглянемо приклад тестування оператора

```
if A and B then ...
```

під час використання різних критеріїв повноти тестування.

При критерії покриття умов були б потрібні два тести: A = true, B = false і A = false, B = true. Але в цьому випадку не виконується then-пропозиція оператора if.

Існує ще один критерій, названий покриттям рішень/умов. Він вимагає такого достатнього набору тестів, щоб усі можливі результати кожної умови у вирішенні виконувались принаймні один раз; всі результати кожного рішення виконували також один раз і кожній точці входу передавалося управління, принаймні один раз.

Недоліком критерію покриття рішень/умов є неможливість застосування для виконання всіх результатів всіх умов. Часто таке виконання має місце через те, що певні умови приховані іншими умовами. Наприклад, якщо умова AND є брехня, то жодна з наступних умов у виразі не буде виконана. Аналогічно, якщо умова OR є істиною, то жодна з наступних умов не буде виконана. Отже, критерії покриття умов та покриття рішень/умов недостатньо чутливі до помилок у логічних виразах.

Критерієм, який вирішує ці та деякі інші проблеми, є комбінаторне покриття умов. Він вимагає створення такого числа тестів, щоб усі можливі комбінації результатів умови в кожному рішенні та всі точки входу виконувались принаймні один раз.

У разі циклів кількість тестів задоволення критерію комбінаторного покриття умов зазвичай більше, ніж кількість шляхів.

Легко бачити, що набір тестів, що задовольняє критерію комбінаторного покриття умов, задовольняє також критеріям покриття рішень, покриття умов і покриття рішень/умов.

Таким чином, для програм, що містять лише одну умову на кожне рішення, мінімальним є критерій, набір тестів якого викликає виконання всіх результатів кожного рішення, принаймні один раз; передає керування кожній точці входу (наприклад, оператор CASE). Для програм, що містять рішення, кожне з яких має більше однієї умови, мінімальний критерій складається з набору тестів, що викликають усі можливі комбінації результатів умов у кожному рішенні та передають управління кожній точці входу програми, принаймні один раз.

Розподіл алгоритму на типові стандартні структури, згідно з проектною процедурою кодування тексту модуля (методу) з п'ятого розділу підручника, дозволяє мінімізувати зусилля програміста, що витрачаються на тестування. Заборона на вкладені структури таки пояснюється зайвими витратами на тестування. Використання ланцюжка простих альтернатив з одним дією чи структури ВИБІР замість вкладених простих АЛЬТЕРНАТИВ значно скорочує кількість тестів!

11.5. ПРОЕКТУВАННЯ ТЕСТІВ ВЕЛИКИХ ПРОГРАМ

Проектування тестів великих програм поки що більшою мірою залишається мистецтвом і меншою мірою є наукою. Щоб побудувати розумну стратегію тестування, треба розумно поєднувати обидва ці крайні підходи і користуватися математичними доказами.

Висхідне тестування. Спочатку автономно тестуються модулі нижніх рівнів, які викликають інших модулів. При цьому досягається така ж їхня висока надійність, як і у вбудованих у компілятор функцій. Потім тестуються модулі вищих рівнів разом із вже перевіреними модулями тощо за схемою ієрархії.

При висхідному тестуванні для кожного модуля потрібна провідна програма. Це монітор або драйвер, який подає тести відповідно до специфікацій тестів. Ряд фірм випускає промислові драйвери чи монітори тестів.

Низхідне тестування. У цьому підході ізольовано тестується головний модуль чи група модулів головного ядра. Програма збирається та тестується зверху вниз. Відсутні модулі замінюються заглушками.

Переваги низхідного тестування: цей метод поєднує тестування модуля з тестуванням пар і частково тестує функції модуля. Коли вже починає працювати введення/виведення модуля, зручно готувати тести.

Недоліки низхідного тестування: модуль рідко досконально тестується одразу після його підключення. Для ґрунтовного тестування потрібні витончені заглушки. Часто програмісти відкладають ретельне тестування і забувають про нього. Інший недолік - бажання розпочати програмування ще до кінця проектування. Якщо ядро вже запрограмоване, виникає опір будь-яким його змін, навіть поліпшення структури програми. Зрештою, саме раціоналізація структури програми за рахунок проведення проектних ітерацій сприяє досягненню більшої економії, ніж дасть раннє програмування.

Модифікований низхідний метод. Згідно з цим методом, кожен модуль автономно тестується перед включенням до програми, що збирається зверху донизу.

Метод великого стрибка - Кожен модуль тестується автономно. Після закінчення автономного тестування всіх модулів модулі легко інтегруються в готову програмну систему. Як правило, цей метод небажаний. Однак якщо програма мала і добре спроектована по сполучення, то метод великого стрибка цілком прийнятний.

*Метод сандвіча*є компромісом між низхідним і висхідним підходами. За цим методом реалізація та тестування ведуться одночасно зверху та знизу, і два ці процеси зустрічаються в заздалегідь наміченій тимчасовій точці.

Модифікований метод сандвіча: нижні модулі тестуються строго знизу вгору, а модулі верхніх модулів спочатку тестуються автономно, а потім збираються низхідним методом.

11.6. КРИТЕРІЇ ВИБОРУ НАЙКРАЩОЇ СТРАТЕГІЇ РЕАЛІЗАЦІЇ

Критеріями вибору найкращої стратегії реалізації програми є:

• час до повного складання програми;

• час реалізації скелета програми;
• існуючий інструментарій тестування;
• міра паралелізму ранніх етапів реалізації;
• можливість перевірки будь-яких шляхів програми даними із заглушок;
• складність планування та дотримання графіка реалізації;
• складність тестування.

11.7. СПОСОБИ ТА ВИДИ ТЕСТУВАННЯ ПІДПРОГРАМ. ПРОЕКТУВАННЯ ТЕСТІВ

Існують два способи тестування: публічний та приватний.
Громадське тестування означає, що кожен бажаючий може отримати цей товар (або безкоштовно, або за ціною дисків і доставки).
Приватний спосіб тестування означає, що перевірку продукту проводить обмежене коло тестувальників, які висловили згоду активно працювати з продуктом та оперативно повідомляти про всі виявлені дефекти. Часто використовуються обидва способи: спочатку програма проходить приватне тестування, а потім коли всі великі проколи вже виявлені, починається її публічне тестування, щоб зібрати відгуки широкого кола користувачів.
Більшість компаній-розробників вимагають, щоб приватний бета-тестувальник підписав «Угоду про нерозголошення» (NDA, Non-Disclosure Agreement). Тим самим він зобов'язується не розголошувати подробиці про продукт, що тестується, і не передавати його копії третім особам. Подібні угоди підписуються з метою збереження комерційної таємниці та щоб уникнути недобросовісної конкуренції.
Так, фірма "Microsoft" використовує всі перелічені вище підходи для тестування своїх продуктів. З її сайтів завжди можна безкоштовно завантажити велику кількість продуктів, що проходять публічне тестування бета. Проте їхній появі передує багатомісячний процес приватного тестування. Фірма Microsoft проводить політику відкритого набору приватних бета-тестувальників, при якій кожен бажаючий може повідомити корпорації про своє бажання взяти участь у тестуванні того чи іншого продукту. Насправді коло приватних бета-тестерів Microsoft дуже вузький, і шанс, що вас включать до їхнього числа, невеликий. Тим не менш, він існує, а потрапляють до списку бета-тестерів практично довічно.
Приватне тестування підпрограм починається з етапу контролю, основними різновидами якого є: візуальний, статичний та динамічний.
Візуальний контроль — це перевірка текстів за столом, без використання комп'ютера.
На першому етапі візуального контролю здійснюється читання тексту підпрограми, причому особлива увага приділяється: коментарям та їх відповідності до тексту програми; умовам в операторах умовного вибору та циклу; складним логічним виразам; можливості незавершення ітераційних циклів
Другий етап **візуального контролю** - наскрізний контроль тексту підпрограми (його ручне прокручування на кількох заздалегідь підібраних простих тестах). Поширена думка, що вигіднішим є перекладання більшої частини роботи з контролю програмних засобів на комп'ютер помилково. Основний аргумент на користь цього такий: при роботі на комп'ютері головним чином удосконалюються навички у використанні клавіатури, у той час як програмістська кваліфікація набувається, перш за все, за столом.
Статичний контроль - Це перевірка тексту підпрограми (без виконання) за допомогою інструментальних засобів.
Першою, найбільш відомою формою статичного контролю є синтаксичний контроль програми з допомогою компілятора, у якому перевіряється відповідність тексту програми синтаксичним правилам мови програмування.
Повідомлення компілятора зазвичай поділяються на кілька груп залежно від рівня тяжкості порушення синтаксису мови програмування:
1) інформаційні повідомлення та попередження, при виявленні яких компілятор, як правило, будує коректний об'єктний код і подальша робота з програмою (компонування, виконання) можлива (проте повідомлення цієї групи повинні ретельно аналізуватися, тому що їх поява

також може свідчити про помилку у програмі - наприклад, через неправильне розуміння синтаксису мови);
2) повідомлення про помилки, при виявленні яких компілятор намагається їх виправити та будує об'єктний код, але його коректність малоймовірна та подальша робота з ним, швидше за все, неможлива;
3) повідомлення про серйозні помилки, за наявності яких побудований компілятором об'єктний код свідомо некоректний та його подальше використання неможливе;
4) повідомлення про помилки, виявлення яких призвело до припинення синтаксичного контролю та побудови об'єктного коду.
Проте будь-який компілятор пропускає деякі види синтаксичних помилок. Місце виявлення помилки може знаходитись далеко за текстом підпрограми від місця справжньої помилки, а текст повідомлення компілятора може не вказувати на справжню причину помилки. Одна синтаксична помилка може спричинити генерацію компілятором кількох повідомлень про помилки (наприклад, помилка в описі змінної призводить до появи повідомлення про помилку в кожному операторі підпрограми, який використовує цю змінну).
Другою формою **статичного контролю** може бути контроль структурованості тексту підпрограми, тобто перевірка виконання угод та обмежень структурного програмування. Прикладом подібної перевірки може бути виявлення у тексті підпрограми ситуацій, коли цикл утворюється за допомогою оператора безумовного переходу (використання оператора GOTO для переходу до тексту програми). Для проведення контролю структурованості можуть бути створені спеціальні інструментальні засоби, а за їх відсутності ця форма статичного контролю може поєднуватися з візуальним контролем.
Третя форма **статичного контролю** - контроль правдоподібності підпрограми, тобто виявлення в її тексті конструкцій, які хоч і синтаксично коректні, але швидше за все містять помилку або свідчать про неї. Основні неправдоподібні ситуації:
• використання у програмі не ініціалізованих змінних (тобто змінних, які отримали початкового значення);
• наявність у програмі описів елементів, змінних, процедур, міток, файлів, які надалі не використовуються в її тексті;
• наявність у тексті підпрограми фрагментів, які ніколи не виконуються;
• наявність у тексті програми змінних, які жодного разу не використовуються для читання після присвоєння їм значень;
• наявність у тексті підпрограми свідомо нескінченних циклів.
Навіть якщо присутність у тексті програми неправдоподібних конструкцій не призводить до її неправильної роботи, виправлення цього фрагмента підвищить ясність та ефективність програми, тобто благотворно позначиться на її якості.
Для можливості контролю правдоподібності в повному обсязі також повинні бути створені спеціальні інструментальні засоби, хоча ряд можливостей контролю правдоподібності є в існуючих налагоджувальних і звичайних компіляторах.
Слід зазначити, що створення інструментальних засобів контролю структурованості та правдоподібності програм може бути спрощено суттєво при застосуванні наступних принципів:
1) проведення додаткових форм статичного контролю після завершення компіляції та тільки для синтаксично коректних програм;
2) максимальне використання результатів компіляції та лінкування програми та, зокрема, інформації, що включається до лістингу компілятора та лінкеру;
3) замість повного синтаксичного розбору тексту програми, що перевіряється, необхідна побудова для неї списку ідентифікаторів та списку операторів із зазначенням усіх їх необхідних ознак.
За відсутності інструментальних засобів контролю правдоподібності ця фаза статичного контролю може об'єднуватися з візуальним контролем.

Четвертою формою **статичного контролю** програм є їхня верифікація, тобто аналітичний доказ їх коректності.
Незважаючи на достатню складність процесу верифікації програми і на те, що навіть успішно завершена верифікація не дає гарантій якості програми (оскільки помилка може міститися і у верифікації), застосування методів аналітичного доказу правильності дуже корисне для уточнення сенсу програми, що розробляється, а знання цих методів благотворно позначається на кваліфікації програміста.
Динамічний контроль програми- Це перевірка правильності програми при її виконанні на комп'ютері, тобто тестування. Мінімальне автономне тестування підпрограми має забезпечувати проходження всіх шляхів обчислень.
Проектна процедура тестування підпрограми полягає в наступному:
- за зовнішніми специфікаціями модуля підготуйте тести для кожної ситуації та кожної неприпустимої умови;
— перегляньте текст підпрограми, щоб переконатися, що всі умовні переходи виконуватимуться у кожному напрямку; за потреби додайте тести;
- Перевірте текст підпрограми, що тести охоплюють досить багато шляхів; для циклів повинні бути тести без повторення, з одним повторенням та з кількома повтореннями;
— перевірте за текстом підпрограми її чутливість до особливих значень даних (найнебезпечніші числа — це нуль і одиниця), у разі потреби додайте тести.

11.8. ПРОЕКТУВАННЯ КОМПЛЕКСНОГО ТЕСТА

У комплексному тесті мають проводитися такі види тестування:
• працездатності;
• стресів;
• граничного обсягу даних, що вводяться;
• конфігурації різних технічних засобів;
• сумісності;
• захисту;
• необхідної пам'яті;
• продуктивність;
• налаштування;
• надійність;
• засобів відновлення при відмові;
• зручності обслуговування;
• програмної документації;
• психологічних факторів;
• зручність експлуатації.

11.9. ЗАСОБИ АВТОМАТИЗАЦІЇ ТЕСТУВАННЯ

Генератори тестів (automatic unit test) випадково генерують дані.
Статичні аналізатори програм, аналізують вихідний тест та будують діаграми маршрутів; аналізують привласнення даних та роблять спроби побудов даних, що призводять до помилки.
Засоби періоду виконання зазвичай роблять статистичний підрахунок кількості виконання кожного оператора та дозволяють контролювати повноту тестів.

ВИСНОВКИ

• Тестування програм – головне, що визначає найважливішу якість програм – надійність.
• На тестування витрачається основна частина коштів та часу проекту.
• Під час розробки будь-якої програми або системи тестування відбирає більшу частину часу та грошей. З огляду на це необхідно визначити не дуже велику кількість тестів, що забезпечують високу ймовірність виявлення тих чи інших помилок.
• Аксіоми тестування визначають основні цілі та принципи тестування.

• Якщо з тексту виключити налагоджувальні печатки, суттєво ускладниться супровід.
• Існують два крайні підходи до проектування тестів: стратегія «чорної скриньки» та стратегія «білої скриньки». Марно слідувати лише одному підходу. Необхідно будувати стратегію тестування лише з урахуванням поєднання підходів.
• При проектуванні багатомодульних програм використовується висхідне тестування (автономне тестування нижніх модулів, які не викликають інших модулів) та низхідне тестування (застосування заглушок нижніх рівнів). Але у кожного з них є свої переваги та недоліки. Можливі також варіанти:
- модифікований низхідний метод - згідно з цим методом кожен модуль автономно тестується перед включенням до програми, що збирається зверху донизу;
- метод великого стрибка - кожен модуль тестується автономно, далі модулі легко інтегруються в готову програмну систему;
- метод сандвіча - за цим методом реалізація та тестування ведуться одночасно зверху та знизу, і два ці процеси зустрічаються в заздалегідь наміченій тимчасовій точці;
- Модифікований метод сандвіча - нижні модулі тестуються строго знизу вгору, а модулі верхніх модулів спочатку тестуються автономно, а потім збираються низхідним методом. Цей метод апробовано під час створення ОС.
• Існує розподіл тестування на публічне та приватне. Часто використовуються обидва способи: спочатку програма проходить приватне тестування, а потім, коли всі великі проколи вже виявлено, починається її публічне тестування, щоб зібрати відгуки широкого кола користувачів.

Контрольні питання

1. Назвіть основні аксіоми тестування.
2. У чому перевага налагоджувального друку?
3. Які властивості програмного забезпечення мають найбільший вплив на процес виявлення помилок при тестуванні?
4. Навіщо потрібні емпіричні моделі? Як проводиться їхній аналіз?
5. Який зв'язок між процесами тестування та проектування?
6. У чому переваги та недоліки висхідного та низхідного проектування? Відповідь обґрунтувати на прикладі конкретної програми.
7. Який тест максимально швидко виявить зациклювання?
8. Виділіть кілька основних критеріїв під час вибору параметрів тестування.

Тема 12 МЕНЕДЖМЕНТ ПРОГРАМНИХ РОЗРОБОК

12.1. УПРАВЛІННЯ РОЗРОБКОЮ ПРОГРАМНИХ СИСТЕМ

Управління розробкою програмних систем (software management)- Це діяльність, спрямована на забезпечення необхідних умов для роботи колективу розробників програмного забезпечення (ПЗ), на планування та контроль діяльності цього колективу з метою забезпечення необхідної якості ПЗ, виконання термінів та бюджету розробки ПЗ. Часто цю діяльність називають управлінням програмним проектом (software project management). Тут під програмним проектом (software project) розуміють всю сукупність робіт, що з розробкою ПЗ, а хід виконання цих робіт називають розвитком програмного проекту (software project progress).

До необхідних умов роботи колективу відносять приміщення, апаратно-програмні засоби розробки, документацію та матеріально-фінансове забезпечення. Планування та контроль передбачають розбиття всього процесу розробки ПЗ на окремі конкретні роботи (завдання), підбір та розстановку виконавців, встановлення строків та порядку виконання цих робіт, оцінку якості виконання кожної роботи. Фінальною частиною цієї діяльності є організація та проведення атестації (сертифікації) ПЗ, якою завершується стадія розробки ПЗ.

Хоча види діяльності з управління розробкою ПЗ можуть бути досить різноманітними, залежно від специфіки ПЗ, що розробляється, і організації робіт з його створення можна виділити деякі загальні процеси (види діяльності) з управління розробкою ПЗ:

- Складання плану-проспекту з розробки ПЗ;
- Планування та складання розкладів з розробки ПЗ;
- Управління витратами з розробки ПЗ;
- поточний контроль та документування діяльності колективу з розробки ПЗ;
- Підбір та оцінка персоналу колективу розробників ПЗ.

Складання плану-проспекту з розробки ПЗвключає формулювання пропозицій у тому, як виконувати розробку ПЗ. Насамперед має бути зафіксовано, для кого розробляється ПЗ:

• для зовнішнього замовника;
• для інших підрозділів тієї самої організації;
• є ініціативною внутрішньою розробкою.

У плані-проспекті повинні бути встановлені загальні обриси робіт зі створення ПЗ та оцінена вартість розробки, а також матеріально-фінансові ресурси та тимчасові обмеження, що надаються для розробки ПЗ. Крім того, він повинен включати обґрунтування, якого роду колективом має розроблятися ПЗ (спеціальною організацією, окремою бригадою тощо). І, нарешті, мають бути сформульовані необхідні технологічні вимоги (включаючи, можливо, і вибір відповідної технології програмування).

Планування та складання розкладів з розробки ПЗ— це діяльність, пов'язана з розподілом робіт між виконавцями та за часом їх виконання у межах намічених термінів та наявних ресурсів.

Управління витратами розробки ПЗ- Це діяльність, спрямована на забезпечення відповідної вартості розробки в рамках виділеного бюджету. Вона включає оцінювання вартості розробки проекту в цілому або окремих його частин, контроль за виконанням бюджету, вибір відповідних варіантів його витрачання. Ця діяльність тісно пов'язана з плануванням та складанням розкладів протягом усього періоду виконання проекту. Основними джерелами витрат є витрати на апаратне обладнання (hardware), вербування та навчання персоналу, оплату праці розробників.

Поточний контроль та документування діяльності колективу з розробки ПЗ- Це безперервний процес стеження за ходом розвитку проекту, порівняння дійсних стану та витрат із запланованими, а також документування різних аспектів розвитку проекту. Цей процес допомагає вчасно виявити труднощі та передбачити можливі проблеми у розвитку проекту.

Підбір та оцінка персоналу колективу розробників ПЗ- Це діяльність, пов'язана з формуванням колективу розробників ПЗ. Наявний штат розробників далеко не завжди буде відповідним за кваліфікацією та досвідом роботи для даного проекту. Тому доводиться частково вербувати відповідний персонал та частково організовувати додаткове навчання існуючих розробників. У будь-якому випадку в колективі, що формується, хоча б один його член повинен мати досвід розробки програмних засобів (систем), порівнянних з ПЗ, яке потрібно розробити. Це допоможе уникнути багатьох простих помилок у розвитку проекту.

Забезпечення якості.Якість ПЗ формується поступово в процесі всієї розробки ПЗ у кожній окремій роботі, що виконується за програмним проектом. Не можна вносити зміни щодо покращення якості у вже створену програму.

Для керівництва цією діяльністю призначається спеціальний менеджер, підпорядкований безпосередньо директору, менеджер з якості. Йому безпосередньо підпорядковані бригади, що формуються з контролю якості. Для кожної роботи організується огляд відповідною бригадою. Дивлюся підлягають всі програмні компоненти та документи, що включаються до ПЗ, а також процеси їх розробки. Огляд контролю якості є функцією управління розробкою та пов'язаний з оцінкою того, наскільки результати цієї роботи узгоджуються з декларованими вимогами щодо якості ПЗ.

Для оцінки є програмні стандарти. Вони фіксують вдалий досвід висококваліфікованих фахівців з розробки програмного забезпечення для різних його класів та для різних моделей якості.

Розрізняють два види таких стандартів:

• стандарти ПЗ (програмного продукту);

• стандарти процесу створення та використання ПЗ.

Стандарти ПЗвизначають деякі властивості, якими повинні мати програми або документи ПЗ, тобто визначають якоюсь мірою якість ПЗ. До стандартів ПЗ належать передусім стандарти мови програмування, склад документації, структуру різних документів, різні формати та інших.

Стандарти процесу створення та використання ПЗвизначають, як має проводитися цей процес, тобто підхід до розробки ПЗ, структуру життєвого циклу ПЗ та його технологічні процеси. Хоча ці стандарти безпосередньо не визначають якості ПЗ, проте вважається, що якість ПЗ істотно залежить від якості процесу його розробки.

12.2. СТРУКТУРА УПРАВЛІННЯ РОЗРОБКИ ПРОГРАМНИХ ЗАСОБІВ

Розробка ПЗ зазвичай проводиться в організації, в якій одночасно можуть вестися розробки ряду інших програмних засобів. Для управління цими програмними проектами використовується ієрархічна структура управління (рис. 12.1).

*Мал. 12.1.*Структура управління розробкою програмних засобів

На чолі ієрархії знаходиться директор програмістської організації, який відповідає за управління усіма розробками програмних засобів. Йому безпосередньо підпорядковані кілька менеджерів сфери розробок та один менеджер з якості програмних засобів. В результаті спілкування з потенційними замовниками директор ухвалює рішення про початок виконання будь-якого програмного проекту, доручаючи його одному з менеджерів сфери розробок, а також рішення про припинення того чи іншого проекту. Він бере участь в обговоренні загальних організаційних вимог до програмного проекту та проблем, що виникають, вирішення яких вимагає використання загальних ресурсів програмістської організації або зміни замовником загальних вимог.

Менеджер сфери розробок відповідає за управління розробками програмних засобів (систем) певного типу, наприклад програмні системи у сфері бізнесу, експертні системи, програмні інструменти та інструментальні системи, що підтримують процеси розробки програмних засобів та ін. Йому безпосередньо підпорядковані менеджери проектів, що належать до його сфери. Отримавши доручення директора з виконання деякого проекту, організує формування колективу виконавців з цього проекту, бере участь у обговоренні плану-проспекту програмного проекту, що належить до сфери розробок, яку він відповідає, соціальній та обговоренні та вирішенні виникаючих проблем у розвитку цього проекту. Він організує узагальнення досвіду розробок програмних засобів у його сфері та накопичення програмних засобів та документів для повторного використання.

За кожним програмним проектом призначається свій менеджер, який керує розвитком цього проекту. Йому безпосередньо підпорядковані лідери бригад розробників.

Менеджер проекту здійснює планування та складання розкладів роботи бригад щодо розробки відповідного програмного засобу.

Вважається вкрай недоцільним розробка великої програмної системи однією великою єдиною бригадою розробників. Для цього є низка серйозних причин. Зокрема, у великій бригаді час, що витрачається на спілкування між її членами, може бути більшим за час, що витрачається на власне розробку. Негативний вплив має велика бригада на будову ПЗ і інтерфейс між окремими його частинами. Усе це призводить до зниження надійності ПЗ. Тому зазвичай великий проект розбивається на кілька відносно незалежних підпроектів таким чином, щоб кожен підпроект міг бути виконаний окремою невеликою бригадою розробників (зазвичай вважається, що в бригаді не повинно бути більше 8—10 членів). При

цьому архітектура ПЗ повинна бути такою, щоб між програмними підсистемами, які розробляють незалежні бригади, був досить простий і добре визначений системний інтерфейс.

Найбільш часто застосовуються чотири підходи до організації бригад розробників: звичайні бригади; неформальні демократичні бригади; бригади провідного програміста; бригада з контролю за якістю.

У звичайній бригаді старший програміст (лідер бригади) безпосередньо керує роботою молодших програмістів. Недоліки такої організації безпосередньо пов'язані зі специфікою розробки: програмісти розробляють сильно пов'язані частини програмної підсистеми; сам процес розробки складається з багатьох етапів, кожен із яких вимагає особливих здібностей від програміста; помилки окремого програміста можуть перешкоджати роботі інших програмістів. Успіх роботи такої бригади досягається у тому випадку, коли її керівник є компетентним програмістом, здатним пред'являти до членів бригади розумні вимоги та вміє заохочувати добру роботу.

У неформальній демократичній бригаді доручена їй робота обговорюється разом усіма її членами, а завдання між її членами розподіляються узгоджено, залежно від здібностей та досвіду членів бригади. Один із членів такої бригади є керівником бригади. Він також виконує деякі завдання, що розподіляються між членами бригади. Неформальні демократичні бригади можуть успішно справлятися з дорученою їм роботою, якщо більшість членів бригади є досвідченими і компетентними фахівцями. Якщо ж неформальна демократична бригада складається з недосвідчених і некомпетентних членів, у діяльності бригади можуть бути великі труднощі. Без наявності в бригаді хоча б одного кваліфікованого та авторитетного члена, здатного координувати та спрямовувати роботу членів бригади, ці труднощі можуть призвести до невдачі проекту.

У бригаді провідного програміста за розробку дорученої програмної підсистеми несе повну відповідальність одна людина — провідний програміст (chief programmer), який є лідером бригади: він сам конструює цю підсистему, складає та налагоджує необхідні програми, пише документацію до підсистеми. Провідний програміст вибирається з-поміж досвідчених і обдарованих програмістів. Решта членів такої бригади в основному створюють умови для найбільш продуктивної роботи провідного програміста. Організацію такої бригади зазвичай порівнюють із хірургічною бригадою. Ядро бригади провідного програміста складають три члени бригади: крім провідного програміста до нього входить дублер провідного програміста та адміністратор бази даних розробки.

Дублер провідного програміста також є кваліфікованим та досвідченим програмістом, здатним виконати будь-яку роботу провідного програміста, але сам він цю роботу не робить. Головний його обов'язок — знати все, що робить провідний програміст. Він виступає в ролі опонента провідного програміста при обговоренні його ідей та пропозицій, але рішення з усіх питань, що обговорюються, приймає одноосібно провідний програміст.

Адміністратор бази даних розробки (librarian) відповідає за супровід усієї документації (включаючи версії програм), що виникає в процесі розробки програмної підсистеми, та забезпечує членів бригади інформацією про поточний стан розробки. Ця робота виконується за допомогою відповідної інструментальної підтримки комп'ютера. Залежно від обсягу та характеру дорученої роботи до бригади можуть бути включені додаткові члени:

• розпорядник бригади, який виконує адміністративні функції;

• технічний редактор, який здійснює доопрацювання та технічне редагування документів, написаних провідним програмістом;

• інструментальщик, який відповідає за підбір та функціонування програмних засобів, що підтримують розробку програмної підсистеми;

• тестувальник, який готує відповідний набір тестів для налагодження програмної підсистеми, що розробляється;

• один або кілька молодших програмістів, які здійснюють кодування окремих програмних компонентів за специфікаціями, розробленими провідним програмістом.

Крім того, до роботи бригади може залучатись для консультації експерт з мови програмування.

Бригада з контролю якості складається з помічників (рецензентів) за якістю ПЗ. До її обов'язків входять огляди тих чи інших частин ПЗ або всього ПЗ в цілому з метою пошуку проблем, що виникають у процесі його розробки. Дивлюся підлягають всі програмні компоненти та документи, що включаються до ПЗ, а також процеси їх розробки. У процесі огляду враховуються вимоги, сформульовані у специфікації якості ПЗ, зокрема, перевіряється відповідність досліджуваного документа чи технологічного процесу стандартам, зазначеним у цій специфікації. В результаті огляду формулюються зауваження, які можуть фіксуватися письмово або передаватися розробникам усно.

12.3. ПІДБІР КОМАНДИ

Кожна розробка збирає довкола себе команду проекту. Ця команда проекту складається з осіб кількох типів: - кінцеві користувачі; розробники; начальник відділу; начальник відділу інформаційних систем; відповідальний за гарантію якості; група, відповідальна за бета-тестування.

*Кінцеві користувачі*здійснюють введення тестів у систему, що розробляється, забезпечують зворотний зв'язок у проекті інтерфейсу, проводять бета-тестування та допомагають керувати визначенням досягнення кінцевої мети.

*Розробники*відповідають за дослідження, проект та створення програмного забезпечення, включаючи альфа-тестування своєї роботи. Один із розробників є керівником команди проекту, який регулює потік інформації між членами команди.

Начальник відділу відповідає за достовірність даних, що видаються цим відділом інформаційної системи. Крім того, начальник відділу відповідає за те, чи відповідає закінчений додаток поставленим у проекті завданням.

Начальник відділу інформаційних систем визначає цілі (плани) для розробників, засновані на інформації, отриману від менеджерів інших відділів та головного управління.

Крім того, встановлює пріоритети між проектами та працює як джерело інформації між відділами та між розробниками, розподіляє обсяги ресурсів, необхідних для виконання кожного проекту.

Відповідальним за гарантію якості є одна людина, але стежать за якістю усі члени команди проекту. Відповідальний стежить, щоб проект програми досяг намічених цілей; проект програми відповідав опису системи; план тестування та план перетворення даних відповідали вимогам; стандарти розробки інформаційних систем дотримувалися.

Група відповідальних за бета-тестування складається з програмістів та можливих кінцевих користувачів та здійснює два типи тестування. Перший тип використовує плани спеціального тестування, які розроблені відповідальним за гарантію якості. При другому типі використовуються тести, які розробили відповідальні за бета-тестування, застосовуючи критерії відповідального за гарантію якості.

Незалежні консультанти зазвичай концентрують увагу до вартості проекту. Часто вони не беруть до уваги витрати на проведення системного аналізу та розроблення проекту та дають неправильну оцінку часу реалізації даного проекту. Хоча відомо, що необхідно виконати детальний аналіз завдання перед тим, як проект буде затверджений, користувачі не схильні витрачати додаткові кошти на дослідження. На жаль, це часто призводить до великої кількості труднощів у процесі розробки, інколи ж до розвалу всього проекту.

12.4. МЕТОДОЛОГІЯ УПРАВЛІННЯ ПРОЕКТОМ

Взаємодія у команді. Відповідальність за проект несе розробник, а чи не начальник відділу. Начальники є членами команди — останнє слово у деяких важливих питаннях належить їм. Проте насправді проектом керує розробник. Коли розробники виконують «власний» проект, їхній інтерес в успіху цього проекту і, отже, ймовірність його успішного виконання збільшується в геометричній прогресії.

Якщо розробники повністю вивчать типи зв'язків між усіма частинами проекту (на відміну від його частини), їх кваліфікація підвищиться. Це забезпечує тип перехресного навчання у дуже важливих чинниках успішної розробки програмного забезпечення. Чи означає це, що розробник виконує менше власне програмістської роботи? Звісно. Коли розробнику потрібна допомога у програмуванні для конкретного проекту, залучається відділ інформаційних систем та сторонніх розробників, які співпрацюють за контрактом.

Усі члени команди точно знають свої обов'язки. Це досягається з допомогою зустрічей (сесій), у яких обговорюються окремі частини проекту. Кожного моменту часу на певній стадії проекту всі члени команди знають точно, що вони мають робити. Це досягається за допомогою постановки завдань та визначення локальних завдань.

Певна методологія розробки програмне забезпечення усуває розбіжності та відсутність зв'язку між членами команди розробників та між програмістами та кінцевими користувачами.

Нові люди «безболісно» підключаються до проекту на будь-якій стадії розробки.

Кінцева програма базується на фундаментальному аналізі завдання, що дозволяє звести до мінімуму витрати на подальше доопрацювання, модифікацію та супровід продукту.

У процесі розробки створюється потужний пакет документації, що дозволяє спростити неминучий супровід і доповнення програмного продукту.

12.5. СКЛАДНІ МЕТОДОЛОГІЇ РОЗРОБКИ

Отримавши певне уявлення необхідність розгляду методології управління проектом, розглянемо окремі її складові: попередній аналіз; чітке формулювання мети; складені моделі даних та словники; вихідні форми; безпека системи та даних; платформа та оточення; контингент майбутніх користувачів.

Попередній аналізє дуже важливим етапом. Ви повинні бути впевнені, що маєте всю необхідну інформацію про клієнта перед тим, як візьметеся за реалізацію проекту.

Чітке формулювання мети має відповідати на запитання: «Що система має робити?»; «Чи було чітко сформульовано мету створення системи?»; Чи знає кінцевий користувач, що система дійсно повинна робити? Звичайно, дуже важливо знайти справжню мету програми, щоб мати можливість визначити межі проекту. Це необхідно зробити настільки швидко, наскільки це можливо.

Моделі даних та словники необхідні для того, щоб дані, що обробляються в додатку, були виділені та визначені у поняттях, доступних як кінцевим користувачам, так і команді розробників. Часто трапляється, що заздалегідь не існує будь-якої моделі даних, що сформувалася, і проектувальник повинен створити словник і модель даних, а потім повернутися до користувача і обговорити з ним розроблену схему, щоб користувач розумів її.

Вихідні форми. При попередньому опитуванні користувача необхідно зробити начерки всіх вихідних форм, оскільки може знадобитися додаткове нарощування словника задля забезпечення реалізації тієї чи іншої.

Безпека системи та даних. Перш ніж розпочати розробку, кінцевий користувач повинен визначити необхідність забезпечення безпеки системи та даних. Включення системи забезпечення безпеки має розглядатися на ранній стадії проектування.

Платформа та оточення. Важливо оцінити оточення, у якому працюватиме система. Клієнти витрачають великі кошти на придбання апаратних засобів до того, як звертаються до вас. Ви повинні з'ясувати всі деталі: про мережні апаратні та програмні ресурси; про типи комп'ютерів; про операційну систему; типи принтерів, моніторів, дисководів, інших периферійних пристроїв.

Контингент майбутніх користувачів. Часто поняття «хто» значно важливіше за поняття «що». Хороше розуміння категорій кінцевих користувачів може дати вам важливу стартову інформацію для створення проекту. Ви повинні постійно вивчати, що хочуть ваші кінцеві

користувачі. Різні типи груп користувачів мають різні вимоги, які повинні бути враховані при проектуванні програмного забезпечення.

Якщо ви робите програми для спільного ринку, то можете створити лише дуже грубе уявлення про кінцевого користувача. Якщо ви пишете будь-яку загальну програму обліку, то можете лише припустити, що мій клієнт має загальне уявлення про комп'ютер і він має необхідність що-небудь враховувати.

Якщо ви пишите програму підтримки офісу бюро подорожей та екскурсій, то знаєте, що кінцеві користувачі будуть використовувати цей додаток саме в цій галузі. Таким чином, ви можете оптимізувати систему обліку та розрахунків з огляду на конкретну специфіку. Однак ви не знаєте ні досвіду роботи кінцевих користувачів з обчислювальною технікою, ні рівня їхнього професіоналізму в їхньому власному бізнесі.

Якщо ви пишете додаток користувача, наприклад офісну систему для офісу «Іванів і сини», то можете безпосередньо спілкуватися з кінцевими користувачами і з'ясовувати їх рівень пізнання обчислювальної техніки та досвід роботи у власному бізнесі, що дасть можливість розробнику заздалегідь передбачити більшість конфліктних ситуацій між вашим додатком та кінцевими користувачами.

Що чекають на вас кінцеві користувачі?

Кожна група кінцевих користувачів має різні вимоги та очікування від вашої системи. Перед проектуванням системи необхідно з'ясувати, на що розраховує кінцевий користувач. Необхідно звернути увагу на такі аспекти: початкове обстеження та складання технічного завдання, інсталяція, навчання, підтримка, допомога в експлуатації.

Резюме Як проектувальник системи, ви повинні повернутися на рівень попереднього аналізу завдання та переконатися, що вся необхідна інформація отримана. У разі недотримання цієї вимоги ви можете значно уповільнити реалізацію проекту внаслідок багаторазового повторного звернення до користувача за уточненням неправильно трактованих деталей та необговорених умов.

12.6. АНАЛІЗ ПОБАЖАНЬ І ВИМОГ ЗАМОВНИКА

Існує величезна прірва між ідеями користувачів та уявленням про можливі способи реалізації цих ідей конкретними розробниками. Містом між цими двома поняттями має бути первинний етап обстеження проекту та складання технічного завдання на даний проект. Це завдання ділиться на три стадії: вивчення вимог замовника, уточнення функціональної специфіки задачі та технічне проектування задачі.

Аналіз вимог та побажань замовникапочинається з отримання замовлення на нову розробку (або модифікацію існуючої) і закінчується складанням документа, в деталях описує цю розробку. Це повинен бути інтерактивний процес, в результаті якого з'являється документ, що повністю описує завдання і задовольняє обидві сторони, що включає розгляд всіх проблем і задач, що вирішуються, безліч аркушів з вимогами і побажаннями замовника та іншу необхідну інформацію.

Найважливіша мета, якої необхідно досягти на цьому першому етапі, — знайти і зрозуміти, що ж НА САМОМ СПРАВІ ХОЧЕ КОРИСТУВАЧ. Іноді зробити це не так просто, оскільки користувач не завжди точно уявляє, ЩО він дійсно хоче отримати. Банальним прикладом можуть бути користувачі, які замовляють, наприклад, одночасно кілька великих завдань типу «Облік заробітної плати», «Ведення складського обліку», «Складання табеля» тощо, називаючи все це «Бухгалтерією». Якщо проігнорувати даний етап, то проект може зрештою бути засуджений на велику кількість доопрацювань, добудовування коду «на коліні» та неодмінне сидіння програмістів у вихідні, щоб зробити клієнту дійсно те, що він хоче і що не було заздалегідь обумовлено.

Очевидно, що будь-який проект розпочинається з ідеї. Як тільки з'являється ідея, одна чи кілька людей починають її розвивати. Ці люди — замовники чи потенційні користувачі. Вони визначають початкові вимоги та приймають рішення про створення того чи іншого

програмного продукту. Таким чином, необхідно з'ясувати, що ці люди хочуть отримати від програмного продукту.

Перед початком обговорення майбутнього проекту дуже важливо переконатися, що по обидва боки столу переговорів сидять саме ті люди, які потрібні для спільного обговорення проекту. Три найбільш поширені помилки припускаються на даному етапі.

Помилка 1. Користувачі, які починають обговорення проекту, не є людьми, які прийматимуть остаточне рішення про вимоги до обговорюваної системи (тобто вони не є людьми, які мають повне уявлення про завдання, що ними описується).

Помилка 2. Учасники обговорення з боку розробників не є людьми, які стосуються технічної розробки проекту.

Помилка 3. Технічні фахівці не розуміють користувачів (або не докладають зусиль до розуміння), або розробники погано знаються на діловодстві та бізнесі, або вони останню частину життя провели не відходячи від монітора і можуть розмовляти тільки мовою бітів і байтів.

У першому випадку користувачі, які беруть участь в обговоренні проекту, описують усі, з їхньої точки зору, деталі майбутнього завдання і залишаються задоволеними думкою, що вони точно виклали всі вимоги та побажання до завдання. На жаль, якщо користувачі, які брали участь в обговоренні, не є кінцевими користувачами даної системи, то після складання кінцевого документа, який в деталях описує задачу, що вирішується, у людей, які підписують цей документ, виникають питання і будь-які нові пропозиції щодо вдосконалення окремих деталей або їхню зміну. Ця ситуація виникає у більшості подібних випадків. Така ситуація відкидає процес розробки програми стадію аналізу майбутнього проекту. Наявна втрата часу та коштів.

Аналогічна проблема виникає за участю у складанні проекту людей, які ніколи не використовуватимуть створювану програму. Це загальна проблема проектування програмного забезпечення. Коли весь проект розроблений, реалізований, відтестований і представлений замовнику, кінцеві користувачі, ті, хто дійсно використовуватиме створений додаток, з'ясовують, що він, скоріше, перешкода, ніж допомога в їх роботі.

Третя з описаних вище проблем полягає в тому, що користувачі, що пред'явили мінімальні вимоги до системи на стадії системного проектування і розробку проекту на розгляд виробника, починають обурюватися, що продукт не задовольняє тим чи іншим вимогам, а тому працює некоректно і вимагає переробки.

Ще однією поширеною помилкою є вибір керівника проекту, який не має відповідних технічних знань для реалізації даного проекту. Ця проблема зазвичай зустрічається при розробці великих проектів, де потрібна велика команда програмістів. Часто існує технічний лідер, який може керувати проектом так само добре, як вирішувати технічні питання. Використання його як менеджера проекту краще, ніж використання простого адміністратора.

У процесі аналізу вимог замовника важливо, щоб у переговорах взяв участь один із членів команди розробників, а в кращому разі провідний технічний спеціаліст або технічний керівник проекту. На жаль, досить важко зібрати в одній кімнаті та одночасно всіх людей, яким необхідно брати участь в обговоренні проекту.

Якщо у процесі обговорення бере участь лише адміністративна особа або керівник проекту, далекий від проблем безпосереднього кодування, виникає безліч проблем і питань, пов'язаних з можливою оптимізацією окремих операцій, створенням словника баз даних, системними вимогами до створюваного програмного забезпечення і т. д. у разі окремим членам доводиться повторно спілкуватися з кінцевими користувачами для з'ясування неврахованих чи погано продуманих питань, що врешті-решт може зіпсувати відносини між командою розробників та кінцевими користувачами. З іншого боку, участь в обговоренні проекту технічних фахівців може призвести до помітного спрощення проекту за рахунок приведення окремих вимог користувача до існуючих і раніше розроблених технологій задоволення цих вимог. Коли необхідний технічний персонал просто не може бути присутнім на всіх засіданнях обговорення проекту або технічний фахівець повинен

тимчасово перейти на інші дії в процесі обговорення, може допомогти так званий протокол засідань. Цей протокол містить нотатки про всі питання, що обговорюються на цьому засіданні, а також імена, посади та телефони учасників обговорення як з одного, так і з іншого боку. Цей протокол також має містити інформацію про прийняті рішення, обумовлені нюанси та будь-які деталі, обговорення яких уже проводилося. Зрештою, дані нотатки повинні перерости в кінцевий документ, який описує результат, отриманий у процесі обговорення всіх частин і деталей проекту.

Якщо зазначені рекомендації будуть дотримані, то технічна сторона розробки буде розглянута повніше, що дозволить згодом уникнути багатьох помилок, пов'язані з нерозумінням тій чи іншій стороною технічних особливостей проекту.

Уникайте програмістів, які можуть просидіти кілька діб над створенням функції, яка фактично не потрібна кінцевому користувачеві. Програмісти, які не вміють працювати з користувачами, не розуміють питань, пов'язаних зі специфікою роботи кінцевого користувача, або не вміють викласти більш менш складні технічні питання простою російською мовою, не повинні брати участь у процесі обговорення проекту. Вони можуть створити додаткові труднощі технічного та тимчасового характеру, починаючи детально з'ясовувати несуттєві питання.

На жаль, багато програмістів не дуже добре розуміються на навколишньому діловому світі. Їхня спеціалізація — комп'ютери та програми, а не створення кінофільмів або управління госпітальним господарством. Виникає питання: «Чи справді необхідно команді розробників детально розбиратися у діловодстві та специфіці бізнесу кінцевих користувачів?»

Недосвідчений програміст подумає: «Користувачі - професіонали у своїй галузі, я - професіонал у своїй; якщо ми почнемо навчати один одного нашим професіям, чи знадобимося ми один одному врешті-решт?»

Неправильно. Професійні знання у тій чи іншій галузі не набуваються у процесі спільного обговорення будь-якого проекту. Не купуються вони і в процесі написання програми на задану тематику. Професійні знання часто набуваються в процесі багатьох років навчання, що йдуть за не менш тривалим періодом спроб і помилок.

Коли користувачі намагаються описати, що вони хочуть від окремих частин програми, не треба відразу переводити їхні побажання до коду. Необхідно зрозуміти, що хоче користувач, а потім постаратися зробити це найбільш оптимальним способом.

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 та розробника, приблизно половина питань одразу знімається з обговорення за непотрібністю.

12.7. АНАЛІЗ ВИМОГ ДО ПРОЕКТУ

Одне підключення до процесу розробки необхідних осіб з обох сторін не спричинить створення повноцінного документа, що описує завдання. Важливо зберегти простоту процесу аналізу вимог та уникати обмірковування, як буде реалізована та чи інша функція чи процедура. Необхідно пам'ятати, що аналіз вимог замовника може тривати від двох годин до кількох тижнів, залежно від складності завдання.

Може існувати велика кількість способів розпочати та проводити аналіз вимог, але всі вони повинні призводити до одного й того самого результату - складання документа, що описує всі вимоги та побажання користувача.

Найпростіший спосіб – почати обстеження зверху вниз. Що головна мета системи? Визначення основних компонентів системи може бути корисним для введення користувача в потрібне русло обговорення проблеми. Майже всі системи вимагають введення певної інформації та виведення якихось звітних форм (у вигляді звітів та запитів), деякого виду конфігурації, можливості імпорту та експорту даних, архівування та, можливо, сервісний розділ. Виходячи з цих даних, можете отримати інформацію про те, що має перебувати в

головному меню програми, та обміркувати деякі деталі розробки ще до повного визначення проекту.

Незалежно від прийнятого підходу до розгляду вимог користувача результатом аналізу має бути ясне розуміння того, що вимагає користувач і що хоче. Тонка відмінність між цими двома поняттями є важливою. Вимоги користувача обмежуються його уявленням про запропоноване їм завдання. Ці вимоги користувач явно обговорює у процесі дискусії. Побажання користувача нерідко залишаються за кадром не тому, що користувач не обговорює їх спеціально, а тому, що він підсвідомо вважає деякі вимоги природними і не вимагають спеціального виділення.

Головна мета цього етапу — переконатися, що ви розумієте потреби користувача та пріоритети напрямів розробки.

Далі слідує функціональна специфікація - це міст між початковим оглядом вимог і технічною специфікацією, що розробляється пізніше. Документ повинен складатися з логічних розділів типу короткого огляду системи, що супроводжується коротким описом основних фрагментів або функціональних об'єктів. Демонстрація планованих екранних форм має показувати основні напрями дій із головними функціональними об'єктами та модулями програми. Розділ опису звітів повинен містити всі звітні форми, які ви плануєте створювати. У великих системах основні модулі можуть бути розбиті на простіші з описом того, що ці, простіші модулі будуть робити.

Плануйте цей документ таким чином, щоб користувач, який не зацікавлений у розгляді детальних особливостей системи, міг би прочитати лише першу частину документа з описом основних функцій системи. Користувачі, зацікавлені у розгляді більш детальних деталей, можуть читати документ далі.

12.8. ВИМОГИ КОРИСТУВАЧА

Ваші специфікації мають відповідати всім вимогам користувачів. Переконайтеся, що знову зверніться до початкового аналізу перед завершенням специфікації, що враховані всі вимоги та запити користувачів. Якщо вимога користувача не може бути задоволена, поясніть чому, а не просто виключіть її зі специфікації.

Ви також повинні обговорити з користувачем обмежені ресурси, які є у користувача. Дев'яносто дев'ять відсотків проблем, що виникають під час програмування, можуть бути вирішені шляхом використання специфічних зовнішніх пристроїв, драйверів та сторонніх програм.

Припустимо, функціональна специфікація розроблена, підписана та покладена на полицю. Але вона може виявитися цілком марною з низки причин. При неправильному ставленні до розробки функціональної специфікації вона може бути погано написана, погано організована або, що найімовірніше, обтяжена томами опису непотрібних технічних подробиць. Іншими словами, працювати з таким документом буде неможливо.

Однією з найнебезпечніших хвороб розробки програм є синдром «повзуючого проекту», або «зсуву ґрунту». Він проявляється, коли функціональна специфікація неповно розглядає окремі аспекти проекту. У цьому випадку, в міру створення системи, користувачі, розглядаючи окремі готові модулі, будуть просити внести деякі вдосконалення, посилаючись на неясні описи цього модуля функціональної специфікації. Поступово система набуватиме вигляду величезного динозавра в латках, оскільки глобальні зміни розроблених структур програми проводити вже не можна, а зміни та вдосконалення необхідно вносити. Це може призвести до перевитрати тимчасового ліміту на створення окремих модулів та нестабільності роботи системи через випадання окремих функціональних шматків програми із суворої загальної схеми усієї системи.

Швидке макетування – метод проектування, розробки та зміни інтерфейсів користувача «на льоту». Кінцеві користувачі повинні активно включатися в цей процес, оскільки розробка інтерфейсу разом з користувачем відбувається значно швидше, ніж без нього. Спільна технологія дає можливість «підігнати» інтерфейс під користувача за кілька коротких сесій.

Для цього є спеціальні засоби, зокрема CASE-засоби. З потужними CASE-засобами процес розробки програм помітно спрощується. Проектувальник використовує програмні засоби для створення та компонування словників даних, потоків даних та діаграм об'єктів, а в деяких випадках прототипів процесів обробки даних та функціонального коду.
Однак використання CASE-засобів розробки програм не дуже поширене у сфері розробки промислових програм. Це відбувається з двох причин. По-перше, це обмеженість можливостей CASE-систем. По-друге, якщо CASE-система досить потужна і багатофункціональна, вона вимагає великих тимчасових витрат за її освоєння.
Наприкінці цього етапу, якщо було написано хорошу, легко розуміємо, неперевантажену і непусту функціональну специфікацію, системний аналітик чи технічна група зможе перейти до наступного етапу — створення технічної специфікації, — ґрунтуючись на інформації, отриманій на всіх попередніх етапах.

12.9. ТЕХНІЧНЕ ПРОЕКТУВАННЯ

Технічне проектування - це міст між функціональною специфікацією та фактичною стадією кодування. Часто команда розробників намагається скоротити та об'єднати стадію розробки функціональної специфікації та технічне проектування та розробити один документ. Це є помилка.
По-перше, користувач читатиме документ із великою кількістю незрозумілих йому технічних подробиць. У цьому випадку користувач відкине ваш документ, що може призвести до недостатньої закінченості цього документа.
По-друге, якщо функціональна специфікація концентрує увагу вимогах і побажаннях користувача, то технічне проектування має орієнтуватися створення методів реалізації даних вимог. Тільки після того, як обидві ці фази завершені та акценти розставлені, програміст може приступати до безпосереднього кодування.
Коли обидві ці стадії об'єднані, розробник неспроможна сконцентруватися на якомусь напрямі мислення й у результаті виходить неясний і погано відпрацьований документ. Або ще гірше, програміст починає реалізовувати ідею, яка ще не визначена до кінця користувачем.
Середовище розробкидозволяє всім членам команди знати розміщення всіх файлів проекту, бібліотек, документів та іншої пов'язаної з проектом інформації. Вона повинна бути створена таким чином, щоб усі члени команди розробників з мінімальними витратами часу могли звернутися до будь-якої інформації щодо проекту.
Створення тимчасової діаграми проекту є найважливішим етапом робіт, на якому необхідно скласти детальний розклад термінів початку та закінчення розробки кожного модуля, частин проекту та всього проекту загалом. Необхідно враховувати час, який витрачається на додаткові контакти із замовником, розробку специфічних інструментальних засобів, а також можливі проблеми, пов'язані з непередбаченими обставинами (наприклад, хвороба співробітника або часткова втрата даних внаслідок збоїв апаратного забезпечення).

12.10. РЕАЛІЗАЦІЯ

Зазвичай на етапі кодування спливають усі неприємні проблеми, які тільки можна собі уявити. Чим більший проект, тим більше проблем. Ось чому перші три кроки такі важливі. Якщо всі з описаних вище кроків повністю пройдені, то реалізація програми значно спрощується. В ідеалі всі потенційні вузли та пастки мають бути передбачені та обійдені. Технічна специфікація може і повинна бути передана команді програмістів, які виконують безпосереднє кодування, щоб вони могли записувати код згідно з детальним проектом завдання. Будь-які проблеми, що виникають на цьому етапі, повинні відстежуватися програмістами і поміщатися у документи, що стосуються проблеми, щоб відображати всі зміни, що виникають.
Огляд кодуробиться програмістами - кодувальниками програм на спеціальній сесії (зустрічі). Як і на етапі огляду проекту, під час огляду коду «відловлюється» велика кількість неточностей та помилок, виявляються неоптимальні ділянки програми. Огляд коду дозволяє

побачити різним членам групи розробників фактичний код, виконаний колегами за проектом. Оскільки програмування є творчим процесом, кожен член команди представляє бачення однієї й тієї проблеми по-різному. Хтось вирішує це питання краще, хтось гірше. Огляд коду дозволяє виявити гірше написані ділянки програми та за необхідності переписати їх, скориставшись порадою досвідченішого члена команди. Також розгляд різних прийомів, технологій та підходів до програмування дозволяє скористатися ними для вирішення майбутніх проблем у подальших проектах. Особливо це корисно для новачків команди, хоча, як відомо, «навіть старого собаку можна навчити новим трюкам».

12.11. СИСТЕМНЕ ТЕСТУВАННЯ

Стадія тестування системи починається після того, як більшість модулів системи вже завершено. Тестування може складатися з трьох окремих фаз:
- Системний тест, або лабораторні випробування;
- Досвідчена експлуатація;
- приймальний тест.

Альфа-тест(Лабораторні випробування). Ця фаза тестування має дві мети. По-перше, цей тест повинен підтвердити, що всі фрагменти правильно інтегровані у систему. Це дозволяє групі тестування розпочати повне тестування усієї системи. Зазвичай використовується деяка однорідна технологія тестування всім компонент системи, що дозволяє визначити відповідність всіх частин певним, заздалегідь передбаченим параметрам. Один із шляхів створення сценаріїв тестування — створювати методи тестування у процесі безпосереднього кодування.

*Лабораторне тестування*остання можливість розробників виправити всі виявлені помилки, перш ніж система буде передана кінцевим користувачам. Бета-тестування — це не та стадія, на якій програмісти хотіли б виявляти серйозні збої розробленої системи, тому лабораторне тестування має проходити максимально повно. Якщо альфа-тестування проведено неякісно, загальний процес тестування може зайняти тривалий час, оскільки виправлення помилок, виявлених на наступних стадіях тестування, займає значно більше часу через неможливість їх виправлення «на льоту». Будь-які виявлені проблеми повинні протоколюватися, щоб хронологія проблем та їх усунення була доступна при виникненні наступних питань про проблеми, що існували раніше.

Бажано, щоб програмне забезпечення не передавалося для дослідної експлуатації, доки всі відомі проблеми не будуть вирішені. Насправді програмне забезпечення часто випускається для бета-тестування з вже знайденими, але ще не вирішеними проблемами через брак часу і закінчення термінів розробки проекту. Це недопущення призводить до значних непередбачених затримок, пов'язаних з труднощами подальшого тестування через наявність будь-яких помилок або неточностей.

Бета-тестування-це наступна фаза загального тестування, коли програмне забезпечення поставляється обмеженому колу кінцевих користувачів більш жорсткого тестування. Добре відомо, що користувачі іноді використовують програмне забезпечення не для тих цілей, для яких воно призначалося. Тому вони часто можуть знаходити помилки в тих місцях програми, над якими протягом цього часу проводилися лабораторні випробування, які не знайшли жодних порушень. Це слід очікувати і не заперечувати можливості повернення до попередньої фази — лабораторного тестування. У цих випадках часто допомагають протоколи виявлених та фіксованих помилок.

12.12. ПРИЙМАЛЬНИЙ ТЕСТ

Приймальний тест.Приймальний тест стає простою формальністю, якщо попередні стадії тестування успішно завершено. Використовуючи інформацію про те, що всі виявлені помилки вже виправлені, приймальний тест просто підтверджує, що нових проблем не виявлено і програмне забезпечення готове для випуску. Очевидно, що чим більше існує реальних чи потенційних користувачів вашого продукту, тим важливішим є приймальний тест. Коли виробляється промислове тиражування і розсилання більше трьох сотень тисяч

дисків, хочеться подвійно переконатися, що програма написана без помилок і всі явні та неявні проблеми було вирішено. Звичайно, якщо попередні стадії не пройшли успішно, то приймальний тест є єдиною можливістю запобігти витратам на зміну та доповнення продукту, що поставляється.

12.13. ПІСЛЯ РЕАЛІЗАЦІЙНИЙ ОГЛЯД

Даний етап – найкраща можливість здійснити огляд створеного програмного забезпечення, перш ніж буде розпочато новий проект. Типові питання, що виникають після здачі програмного проекту користувачу-замовнику:

• Що ми робили правильно?
• Що ми робили неправильно?
• Які етапи були найкориснішими, а які непотрібними?
• Чи не було що-небудь на якомусь етапі розробки, щоб допомогло вдосконалити програмний продукт?

12.14. СУПРОВІД ПРОГРАМ

Супровід програм - "ложка дьогтю" для кожного програміста. Це завжди перешкода на початку розробки будь-якого нового проекту, що змушує відволікатися від розробки проекту та повертатися до старих програм та старих проблем. Ніщо не робить супровід настільки непривабливим, як погано документований код, недостатньо повне початкове проектування та відсутність зовнішньої документації.

Якщо більшість кроків розробки виконано правильно, супровід не викликатиме серйозних проблем, а буде елементарною технічною підтримкою та модифікацією програмного забезпечення.

ВИСНОВКИ

• Розробка програмних систем – складний захід. Можна виділити такі загальні процеси з управління розробкою ПЗ: складання плану-проспекту з розробки ПЗ - планування та складання розкладів з розробки ПЗ; управління витратами розробки ПЗ; поточний контроль та документування діяльності колективу з розробки ПЗ; підбір та оцінка персоналу колективу розробників ПЗ.
• Зазвичай, в організації одночасно розробляється кілька програмних проектів. Для оптимальної якості та швидкості роботи необхідно правильно структурувати управління організацією.
• Кожна розробка проекту збирає довкола себе команду фахівців (команду проекту), що складається з кінцевого користувача; розробників; начальника відділу; начальник відділу інформаційних систем; відповідального за гарантію якості; групи, відповідальної за бета-тестування
• Необхідно дотримуватись методології управління проектом, яка ділиться на складові: попередній аналіз; чітке формулювання мети; складені моделі даних та словники; вихідні форми; безпека системи та даних; платформу та оточення; контингент майбутніх користувачів.
• Різниця між поняттями «бажання замовника» та «кінцевий продукт» зазвичай дуже велика. Мостом для їх з'єднання має бути первинний етап обстеження проекту та складання технічного завдання на цей проект. Це завдання ділиться на три стадії: вивчення вимог замовника, уточнення функціональної специфіки задачі та технічне проектування задачі. Якщо говорити про вимоги користувача, то їх необхідно дотримуватися неухильно.
• Технічне проектування — своєрідний міст між функціональною специфікацією та фактичною стадією кодування. Це вкрай важлива стадія і недбало до неї ставитися не можна.
• Системне тестування може складатися із трьох окремих фаз: системний тест або лабораторні випробування; дослідна експлуатація; приймальний тест.
• Супровід - нелюбима програмістами, але необхідна частина, що дає можливість для вдосконалення продукту.

Контрольні питання

1. Що таке програмний проект?
2. Що включає складання плану-проспекту з розробки ПЗ?
3. Назвіть основні джерела витрат розробки ПЗ.
4. Перерахуйте обов'язки членів ядра бригади програмістів.
5. Чим займаються незалежні консультанти?
6. Назвіть складові методології розробки.
7. У чому полягає аналіз вимог та побажань замовника?
8. Що таке швидке макетування?
9. У чому полягає технічне проектування?
10. Назвіть три фази тестування. І. Навіщо потрібен приймальний тест?
12. Назвіть фактор, який ускладнює супровід найбільшою мірою.

Додаток 1 СТАДІЇ ТА ЕТАПИ РОЗРОБКИ ПРОГРАМ

Цей текст не замінює сам стандарт, який може змінитися, і наводиться лише для пояснення порядку роботи з цим та іншими стандартами.
Таблиця 1

Стадії розробки, етапи та зміст робіт

Стадії розробки	Етапи робіт	Зміст робіт
1. Технічне завдання	Обґрунтування необхідності розробки програми	Постановка задачі. Збір вихідних матеріалів. Вибір та обґрунтування критеріїв ефективності та якості програми, що розробляється. Обґрунтування необхідності проведення науково-дослідних робіт.
	Науково-дослідні роботи	Визначення структури вхідних та вихідних даних. Попередній вибір методів розв'язання задач. Обґрунтування доцільності застосування раніше розроблених програм. Визначення вимог до технічних засобів. Обґрунтування принципової можливості вирішення поставленого завдання.
	Розробка та затвердження технічного завдання	Визначення вимог до програми. Розробка техніко-економічного обґрунтування розробки програми. Визначення стадій, етапів та термінів розробки програми та документації на неї. Вибір мов програмування. Визначення необхідності проведення науково-дослідницьких робіт на наступних стадіях. Погодження та затвердження технічного завдання.
2. Ескізний проект	Розробка ескізного проекту	Попередня розробка структури вхідних та вихідних даних. Уточнення методів розв'язання задачі. Розробка загального опису алгоритму розв'язання задачі. Розробка техніко-економічного обґрунтування.
	Затвердження ескізного проекту	Розробка пояснювальної записки. Узгодження та затвердження ескізного проекту.
3. Технічний проект	Розробка технічного проекту	Уточнення структури вхідних та вихідних даних. Розробка алгоритму розв'язання задачі. Визначення форми подання вхідних та вихідних даних. Визначення семантики та синтаксису мови. Розробка структури програми. Остаточне визначення конфігурації технічних засобів.
	Твердження технічного проекту	Розробка пояснювальної записки. Узгодження та затвердження ескізного проекту.
4. Робочий проект	Розробка програми	Програмування та налагодження програми
	Розробка програмної документації	Розробка програмних документів відповідно до вимог ДСТУ
	Випробування програми	Розробка, погодження та затвердження порядку та методики випробувань. Проведення попередніх державних, міжвідомчих, приймально-здавальних та інших видів випробувань. Коригування програми та програмної документації за результатами випробувань.

Стадії розробки	Етапи робіт	Зміст робіт
5. Використання	Підготовка та передача програми	Підготовка та передача програми та програмної документації для супроводу та (або) виготовлення. Оформлення та затвердження акта про передачу програми на супровід та (або) виготовлення. Передача програми до фонду алгоритмів та програм.

Примітки:
1. Допускається виключати другу стадію розробки, а в технічно обґрунтованих випадках – другу та третю стадії. Необхідність проведення цих стадій зазначається у технічному завданні.
2. Допускається об'єднувати, виключати етапи робіт та (або) їх зміст, а також запроваджувати інші етапи робіт за погодженням із замовником.

Додаток 2 ПРИКЛАД ВИКОНАННЯ НАВЧАЛЬНОГО ТЕХНІЧНОГО ЗАВДАННЯ

1. ВСТУП

1.1. Найменування програмного виробу

Повне найменування програми - "Простий консольний редактор текстових файлів ". Коротка назва програми - редактор.

1.2. Область застосування

Редактор призначений для коригування наявних і створення нових текстових файлів у діалоговому консольному режимі роботи .

Редактор може застосовуватись для роботи з короткими текстовими файлами під час написання вихідних текстів програм.

2. ПІДСТАВИ ДЛЯ РОЗРОБКИ

2.1. Документ, на підставі якого ведеться розробка

Розробка ведеться виходячи з завдання з дисципліни «Технологія програмування».

2.2. Організація, яка затвердила цей документ, та дата його затвердження

Завдання затверджено на засіданні кафедри та видано викладачем кафедри.

2.3. Назва теми розробки Назва теми розробки - EDIT.

3. ПРИЗНАЧЕННЯ РОЗРОБКИ

Розробка є атестаційною під час підготовки бакалавра.

4. ВИМОГИ ДО ПРОГРАМИ

4.1. Вимоги до функціональних характеристик

4.1.1. Склад виконуваних функцій

4.1.1.1. Редактор повинен забезпечити коригування наявних на диску та створення нових текстових файлів з КОНСОЛІ в діалоговому режимі роботи.

4.1.1.2. Зовнішній вигляд програми повинен відповідати макетам екранів та сценарію роботи, наведеним у ДОДАТКУ 1.

4.1.1.3. Список керуючих клавіш програми редактора та кодів символів, що заносяться в текстовий файл, повинен відповідати ДОДАТКУ 2.

4.1.1.4. При запуску редактора командою КОНСОЛІ EDIT.EXE із зазначенням через символ пропуску імені файлу, що редагується, програма редактора повинна забезпечити завантаження файлу, що редагується. Програма редактора повинна запускатися командою КОНСОЛІ EDIT.EXE і без вказівки імені файлу, що редагується.

4.1.1.5. У будь-який момент роботи програми при натисканні клавіші <F1> повинні виводитись тексти допомоги зі списком всіх можливих команд редактора на даний момент.

4.1.1.6. Програма має забезпечити виведення на принтер вмісту текстового файлу стандартними символами принтера з числом рядків на сторінці, заданим користувачем.

4.1.2. Організація вхідних та вихідних даних

Організація вхідних та вихідних файлів редактора повинна відповідати Додатку 3. Розмір файлу, що редагується, не повинен перевищувати 64 Кбайт. Число символів у рядку не повинно перевищувати 255.

В процесі роботи редактора вхідною інформацією для програми повинні бути коди клавіш, що натискаються користувачем на клавіатурі ЕОМ, відповідно до режимів, що визначаються вихідною екранною інформацією.

4.1.3. Тимчасові характеристики та розмір займаної пам'яті

Час реакції програми на натискання будь-якої клавіш не повинен перевищувати 0,25 с, за винятком реакцій на читання і запис вхідних і вихідних файлів. Обсяг займаної оперативної пам'яті має перевищувати 200 Кбайт.

4.2. Вимоги до надійності

4.2.1. Вимоги до надійного функціонування

Програма має нормально функціонувати при безперебійній роботі ЕОМ. При виникненні збою в роботі апаратури відновлення нормальної роботи програми повинно проводитись після:

1) перезавантаження операційної системи;

2) запуску виконуваного файлу програми; повторне виконання дій, втрачених до останнього збереження інформації у файл на магнітному диску.
Рівень надійності програми повинен відповідати технології програмування, яка передбачає:
1) інспекцію вихідних текстів програми;
2) автономне тестування модулів (методів) програми;
3) тестування сполучення модулів (методів) програми;
4) комплексне тестування програми.
4.2.2. Контроль вхідної та вихідної інформації
Програма повинна контролювати вибір користувачем пункту меню «Вихід» та попереджати його про втрату «незбережених змін».
4.2.3. Час відновлення після відмови
Час відновлення після відмови має складатися з:
1) часу перезапуску користувачем операційної системи;
2) часу запуску користувачем файлу програми, що виконується;
3) час повторного введення втрачених даних.
4.3. Умови експлуатації
Програма повинна зберігатися у вигляді двох маркованих дискетних копій – еталонної та робочої. Періодична перезапис інформації має здійснюватися згідно з нанесеним маркуванням. Умови зберігання дискет повинні відповідати нанесеному на них маркуванню.
4.4. Вимоги до складу та параметрів технічних засобів
Програма має коректно працювати на наступному або сумісному з ним обладнанні:
1) ПЕОМ IBM PC моделі 300 GL;
2) принтер Epson Stylus 800+ моделі P780B.
4.5. Вимоги до інформаційної та програмної сумісності
4.5.1. Вимоги до інформаційних структур на вході та виході
Вимоги до інформаційних структур на вході та виході визначено у пункті (див. п. 4.1.2.).
4.5.3. Вимоги до методів вирішення
Вимоги до методів вирішення визначено у підпункті (див. пп. 4.1.1.2). Внутрішній буфер редактора повинен поміщати найдовший редагований файл повністю. Вибір інших способів рішення здійснюється розробником без погодження із замовником.
4.5.4. Вимоги до мов програмування
Мова програмування має вибиратися розробником без узгодження із замовником.
4.5.5. Вимоги до програмних засобів, що використовуються програмою
Для роботи програми потрібна операційна система MS DOS версії 6.22. або консоль windows
4.6. Вимоги до маркування та упаковки
Дискети з еталонними та робочими екземплярами програми повинні мати маркування, що складається з напису EDIT, напису «еталон» або «робочий», дати останнього перезапису програми. Упаковка має відповідати умовам зберігання дискети. На упаковці мають бути зазначені умови транспортування та зберігання дискети.
4.7. Вимоги до транспортування та зберігання
Умови транспортування та зберігання дискети повинні відповідати підрозділу (див. підрозділ 4.6).
5. ВИМОГИ ДО ПРОГРАМНОЇ ДОКУМЕНТАЦІЇ
Склад програмної документації повинен включати такі документи:
1) технічний проект програми з ДСТУ у машинописному виконанні;
2) опис програми з ДСТУ на машинному носії;
3) текст програми з ДСТУ на машинному носії;
4) керівництво програміста з ДСТУ на машинному носії як файл README.TXT.
Пояснювальна записка «технічний проект програми» має містити такі розділи:
1) Розділ «ВХІДНІ ДАНІ» (Характер, організація та попередня підготовка вхідних даних);
2) Розділ «ВИХІДНІ ДАНІ» (Характер та організація вихідних даних);
3) Розділ «ОПИС ЛОГІЧНОЇ СТРУКТУРИ»;

4) Розділ «ВИКОРИСТОВУВАНІ ТЕХНІЧНІ ЗАСОБИ» (Типи ЕОМ, на яких можливе виконання програми; пристрої ЕОМ, що використовуються при виконанні програми);
5) Розділ «ВИКЛИК І ЗАВАНТАЖЕННЯ» (Види носіїв програми, їх обсяг, що використовується; способи виклику програми з відповідних носіїв даних; вхідні точки в програму — запуск програми);
6) Розділ «ПЛАН ЗАХОДІВ З РОЗРОБКИ ТА ВПРОВАДЖЕННЯ ПРОГРАМИ» (План заходів розробляється для реалізації програми колективом програмістів — дві людини. Планом мають бути передбачені контрольні тимчасові точки реалізації, наприклад, через кожні десять днів або тиждень, протягом яких відбувається інтеграція розроблених модулів та тестування вже розробленої частини програми. Наводиться склад тестів та принципи їх підготовки для тестування вже створеного фрагмента програми для кожної з контрольних точок).
Розділ «ОПИС ЛОГІЧНОЇ СТРУКТУРИ» за технологією структурного програмування повинен включати такі матеріали:
1) опис зв'язків програми з іншими програмами;
2) опис внутрішніх масивів та змінних, що використовуються в міжмодульному обміні даними;
3) схема ієрархії програми (наводиться малюнок чи малюнки);
4) розшифрування найменувань модулів (наводиться таблиця з переліком найменувань модулів в алфавітному порядку із зазначенням виконуваної кожним модулем функції);
5) опис функціонування програми з урахуванням її модульного поділу (наводиться словесний опис виконання програми з урахуванням викликів модулів);
6) опис модулів програми (підрозділ заповнюється з урахуванням паспортів модулів).

6. ТЕХНІКО-ЕКОНОМІЧНІ ПОКАЗНИКИ

Техніко-економічні показники мають визначатися замовником без участі виконавця.

7. СТАДІЇ ТА ЕТАПИ РОЗРОБКИ

Розробка програми має виконуватися за такими етапами:
1) розробка, погодження та затвердження технічного проекту програми з пояснювальною запискою - 5 тижнів;
2) розробка робочого проекту програми з комплексним тестуванням – 6 тижнів;
3) приймання-здавання з виправленням виявлених недоліків у програмі та програмної документації - 2 тижні;
4) Використання.

8. ПОРЯДОК КОНТРОЛЮ І ПРИЙМАННЯ

8.1. Види випробувань

Випробування програми та верифікація документації повинні проводитись в організації замовника із залученням сторонніх експертів. Перевірочні випробування повинні готуватися замовником.

8.2. Загальні вимоги до приймання

Приймання програми має здійснюватися замовником. Програма повинна вважатися придатною, якщо вона задовольняє всі пункти цього технічного завдання.

Додаток З ФОНД ЕВРИСТИЧНИХ ПРИЙОМ ПРОЕКТУВАННЯ ПРОГРАМ

1. ВИБІР СТРАТЕГІЇ ПРОЕКТУВАННЯ ПРОГРАМ

1.1. Замінити висхідний спосіб проектування програм низхідним.
1.2. Інверсія прийому.
1.3. Використовувати комбінований (висхідно-низхідний) спосіб проектування. У разі головна частина програми розробляється низхідним методом, а окремі модулі і підсистеми — висхідним.
1.4. Використовувати спосіб проектування шляхом розширення ядра системи. У разі спочатку створюється оболонка, реалізує мінімальний набір функцій проектованої системи, потім до цієї оболонці (ядру) системи послідовно додаються нові модулі, що розширюють набір функцій, що реалізуються.

2. ВИБІР ПІДХОДУ У ПРОГРАМУВАННІ (методології проектування)

2.1. Замінити методологію, орієнтовану на обробку (модульне програмування; функціональна декомпозиція; проектування з використанням потоку даних; структурне проектування; технологія структурного аналізу проекту SADT; проектування, засноване на використанні структур даних; методологія Джексона; методологія Уорнера та ін.), на методологію, орієнтовану на дані (абстракції даних Дейкстри, об'єктно-орієнтована методологія; методологія, орієнтована проектування концептуальних баз даних та інших.).
2.2. Інверсія прийому.

3. ВИБІР МОВИ

3.1. Вибрати «улюбленішу» мову програмування.
3.2. Вибрати мову програмування, спеціально призначену для вирішення конкретної проблеми.
3.3. Замінити проблемно-орієнтовану мову на об'єктно-орієнтовану.
3.4. Інверсія прийому.
3.5. Замінити мову високого рівня мовою низького рівня.
3.6. Інверсія прийому.
3.7. Використовувати у проекті дві та більше мов програмування.
3.8. Підключати об'єктний код (відкомпільований компілятором іншої мови програмування або асемблер) за допомогою директиви компілятора.
3.9. Використовувати вбудований асемблер системи програмування.

4. ПЕРЕТВОРЕННЯ АРХІТЕКТУРИ, АБО СТРУКТУРИ ПРОГРАМНОЇ СИСТЕМИ

4.1. Збільшити кількість модулів системи.
4.2. Інверсія прийому.
4.3. Замінити глобальну змінну фактичним параметром, що передається модулю як аргумент. Цим прийомом виключається можливість непередбачених змін світових змінних.
4.4. Інверсія прийому.
4.5. Замінити глобальні змінні локальними змінними.
4.6. Інверсія прийому.
4.7. Виконати декомпозицію модуля на кілька. Даний прийом дозволяє розподілити функції, що виконуються між окремими функціями.

4.8. Поєднати кілька модулів в один. Цей прийом дає можливість заощадити час виробництва обчислень; дає особливий ефект, коли дозволяє виключити дублювання тих самих процесів у різних модулях.
4.9. Оформити модулі, пов'язані між собою єдиною логікою, бібліотеку.
4.10. Використовувати у проектуванні системи стандартні модулі системи програмування.
4.11. Використовувати бібліотечні модулі, розроблені іншими програмістами.

5. ПЕРЕТВОРЕННЯ СТРУКТУРИ МОДУЛЯ

5.1. Замінити лінійну структуру команд циклічною. (Підвищує компактність коду програми.)
5.2. Інверсія прийому.
5.3. Замінити гілку структуру циклічної.
5.4. Інверсія прийому.
5.5. Замінити розгалужену структуру if - then - else варіантом оператора case.
5.6. Замінити розгалужену структуру case ланцюжком операторів if - then.
5.7. Інверсія прийому.
5.8. Замінити цикл repeat - until циклом while.
5.9. Інверсія прийому.
5.10. Замінити цикл repeat-until циклом for.
5.11. Інверсія прийому.
5.12. Замінити цикл при циклі for.
5.13. Інверсія прийому.
5.14. Виділити тіло циклу окрему підпрограму. Цей прийом підвищує читабельність програми, але його слід використовувати лише тоді, коли це не порушує внутрішньої логіки циклу.
5.15. Використовувати рекурсію.
5.16. Замінити підпрограму-процедуру підпрограмою-функцією. Даний прийом дозволяє отримати додатковий параметр, який видається підпрограмою (наприклад, код помилки).
5.17. Інверсія прийому. Дозволяє уникнути резервування місця під змінну, яка сприймає значення підпрограми-функції.
5.18. Цілком виключити або мінімізувати використання оператора goto. Покращує структуру програми, її читабельність та логіку.
5.19. Використовувати оператор goto для швидкої передачі керування. Дозволяє швидко без залучення додаткових засобів передавати керування іншим процесом. Слід застосовувати лише у випадках, коли перехід є найбільш лаконічним, простим і ясним засобом.
5.20. Використовувати процедуру exit для виходу із підпрограми. Дозволяє обходитися без оператора goto та без ускладнення логіки підпрограми.
5.21. Використовувати директиву компілятора для безболісного використання процедур як функцій і як процедур.
5.22. Використання процедурного типу даних.
5.23. Використовувати покажчики на процедури та функції.
5.24. Збільшити розмірність масиву.
5.25. Інверсія прийому.
5.26. Використовувати тип даних безліч set замість масивів.
5.27. Інверсія прийому.
5.28. Заміна запису фіксованої довжини записом із варіантом.
5.29. Інверсія прийому.
5.30. Замінити звичайні рядки (тип String) рядками із нульовим закінченням.
5.31. Інверсія прийому.
5.32. Використовувати оператор with для спрощення роботи із записами.
5.33. Використовувати перетворення типів даних.
5.34. Використовувати типізовані константи.

5.35. Давати змінним, константам та типам даних змістовні позначення.
5.36. Широко використовувати коментарі пояснення обчислювальних алгоритмів.

6. ОРГАНІЗАЦІЯ І ЗБЕРІГАННЯ ДАНИХ

6.1. Замінити типізований файл на нетипізований файл.
6.2. Інверсія прийому.
6.3. Замінити типізований файл текстовим файлом.
6.4. Інверсія прийому.
6.5. Замінити нетипізований файл текстовим файлом.
6.6. Інверсія прийому.
6.7. Замінити носій даних.
6.8. Проводити сортування даних з метою полегшення пошуку.
6.9. Використовувати індексовані масиви даних для організації пошуку за вторинними ключами.
6.10. Виключити надмірність даних.
6.11. Декомпозувати дані на кілька файлів.
6.12. Об'єднати дані в один файл даних.

7. ЕКОНОМІЯ РЕСУРСІВ ПРОГРАМИ

7.1. Використовувати inline-процедури та inline-директиви. Дозволяє економити пам'ять комп'ютера та збільшує швидкодію алгоритму, оскільки реалізація такого ж алгоритму за допомогою операторів мови високого рівня після компіляції призводить до збільшення об'єктного коду та ускладнення алгоритму за рахунок додавання різних операторів контролю кордонів тощо. У процедурах inline здійснюється безпосереднє введення тексту у машинних кодах, і вся відповідальність щодо організації процесу лежить на програмісті.
7.2. Використовувати директиви вбудованого асемблера.
7.3. Використовувати абсолютну адресацію даних через директиву absolute та стандартні масиви Mem, MemW, MemL.
7.4. Використовувати безпосереднє звернення до портів через стандартні масиви Port, PortW, PortL.
7.5. Використовувати систему переривань через функції модуля DOS - Intr та MS DOS.
7.6. Використовувати профіль коду програм за допомогою програм-профілювальників.
7.7. Замінити статичні змінні та масиви динамічними.
7.8. Використовувати оверлейну організацію програм.
7.9. Об'єднати оверлейні файли в один файл типу *.EXE.
7.10. Розбити програму на резидентну частину (TSR) і частини, що підвантажуються.
7.11. Використовуйте додаткову пам'ять комп'ютера (expanded memory).
7.12. використовувати розширену пам'ять комп'ютера (extended memory).
7.13. Використовувати захищений режим процесора (protected mode).
7.14. Використання режиму віртуального процесора 8086.

8. ОФОРМЛЕННЯ ВАРІАНТУ (ВЕРСІЇ) ПРОГРАМИ

8.1. Розмноження околиці (копіювання старого варіанту окремий файл). Вкрай неефективний метод через захаращення дискового простору.
8.2. Заміна виклику старої процедури на виклик нової також неефективна, оскільки старі процедури також підключаються до об'єктного коду програми, що призводить до захаращування програми.
8.3. Використання оператора вибору. Ті самі обмеження.
8.4. Коментування зміненого коду програми.

8.5. Використання директив компілятора {$IFDEF <умова>} та {$IFOPT <опція>}.

9. ТЕСТУВАННЯ ПРОГРАМ

9.1. Замінити висхідне проектування тестів низхідним.
9.2. Інверсія прийому.
9.3. Використовувати метод великого стрибка.
9.4. Використовувати метод «сандвіч».
9.5. Організувати вхідні дані для тестування у зовнішньому файлі. Це виключить повторне введення даних при кожному тестуванні, що дозволить заощадити час.
9.6. Використовуйте генератор вхідних даних.

10. НАЛАДКА ПРОГРАМ

10.1. Використовувати вбудований налагоджувач системи (трасування програми).
10.2. Використовувати директиви компілятора {$D} та {$L} при компіляції модулів з метою мати безпосередній доступ до змінних та процедур модуля.
10.3. Використовувати налагоджувальний друк. Виводити значення окремих ключових змінних і масивів безпосередньо на екран або зовнішній файл на диску.
10.4. Вставити «заглушки» на ті модулі програми, які зараз не налагоджуються.
10.5. Використовувати процедуру halt у разі виняткової ситуації.
10.6. Використовувати повернення функцією або процедурою спеціального значення у разі виняткової ситуації.
10.7. Використовувати код повернення у вигляді окремої глобальної змінної.

11. ОРГАНІЗАЦІЯ ДІАЛОГУ З КОРИСТУВАЧЕМ

11.1. Замінити горизонтальне меню вертикальним меню.
11.2. Інверсія прийому.
11.3. Використовувати скролінг меню.
11.4. Замінити спливаюче меню, що випадає.
11.5. Інверсія прийому.
11.6. Організувати меню, яке активується за гарячими клавішами.
11.7. Використовувати кнопки та панелі діалогу.
11.8. Організовувати громіздкі екранні форми як багатосторінкових форм.
11.9. Використовувати скролінг екранних форм.
11.10. Використовувати спливаючі екранні форми.
11.11. Використовувати гіпертекстову систему як систему допомоги.

Додаток 4 ЕЛЕМЕНТИ МОВИ OBJECT PASCAL

1. МОДУЛЬ В OBJECT PASCAL

Мова об'єктно-орієнтованого програмування Object Pascal застосовується під час роботи у середовищі візуального програмування Delphi і Lazarus. Мова Object Pascal в основному включає "стару" мову Borland Pascal.
Програми мовою Object Pascal складаються з кількох файлів: файлу проекту (Delphi Project) з розширенням *.dpr, одного або кількох файлів модулів (Unit) з розширенням *.pas та файлів дизайнера екранних форм із розширенням *.dfm.
Файл проекту містить текст основної програми Program, з якої починається виконання всієї програми. Тексти підпрограм і об'єктів, що використовуються, знаходяться у файлах модулів.
Розглянемо організацію вихідного тексту модуля:

```
unitMyUnit1;

interface

uses
Unit1, Unit2, Unit3;

const
Pi =3.14;
type
MyType = . . .;
var
var1: MyType;

procedureProc1;
functionFunc: MyType;

implementation

uses
Unit4, Unit5, Unit6;

const
...;
type
...;
var
...;

procedureProc1;
begin
{ Операторы}
...
end;

functionFunc: MyType;
begin
{Операторы}
...
end;

initialization
{Операторы}
...
```

```
finalization
{Оператори}
...
end.
```

Модуль починається з описового оператора заголовка модуля:

```
unitMyUnit1;
```

Імена файлів MyUnit1.pas, MyUnit1.dfm повинні збігатися з ім'ям, описаним у заголовку модуля MyUnit1. Наявність файлу MyUnit1.dfm не є обов'язковою.

Між зарезервованими словами interface та implementation знаходяться описові оператори секції інтерфейсу. В інтерфейсній частині оголошуються константи, типи, змінні, прототипи процедур і функцій (тільки оператор заголовка без операторів), які повинні бути доступні для використання в інших модулях. Описи підключень інших модулів здійснюються за допомогою оператора uses, який може розташовуватися за оператором interface. Імена модулів, що підключаються, повинні бути розташовані в такому порядку, щоб забезпечити послідовний опис всіх потрібних типів в даній інтерфейсній секції і інтерфейсних секцій модулів, що підключаються.

За зарезервованим словом implementation знаходяться описові оператори секції реалізації. На відміну від описів секції інтерфейсу, описи з секції реалізації недоступні для інших модулів, але доступ до них можливий з даного модуля. Як і в секції інтерфейсу, може слідувати оператор uses, а за ним оголошення констант, типів, змінних. На відміну від секції інтерфейсу процедури та функції описуються разом з їх операторами. У секції повинні бути визначені як процедури та функції із секції реалізації, так і процедури та функції, прототипи яких були оголошені у секції інтерфейсу. Код процедур та функцій модуля, описаний у секції implementation, підключається (лінкується або зв'язується) до основної програми або точок виклику підпрограм та функцій в інших модулях (через механізм лінкера, робота якого визначається uses).

У необов'язковому розділі initialization розміщуються оператори, які виконуються відразу після запуску програми.

Розділ finalization не є обов'язковим, більше того, він може бути присутнім тільки в тому випадку, якщо в модулі була секція initialization. У секції розміщуються оператори, які виконуються безпосередньо до завершення програми.

2. ОБ'ЄКТИ І КЛАСИ В МОВІ OBJECT PASCAL

У звичайній мові Pascal існує тип-запис:

```
type
TmyRecord = record
MyField1: String;
MyField2: Integer;
end;
```

Тип-запис дозволяє описувати структуровані змінні, які містять кілька значень як одного, і різних типів. У наведеному прикладі запис TmyRecord містить поля MyField1 та MyField2, які відповідно мають типи String та Integer.

Тип-клас в Object Pascal на вигляд близький до запису, але відрізняється від запису можливістю успадкування від інших класів, а також можливістю опису методів класу.

Класом в Object Pascal називається особливий тип запису, який може мати у своєму складі поля, методи та властивості. Такий тип також називатимемо об'єктним типом:

```
type
TMyObject = class(TObject)
MyField: Integer;
ProcedureMyMethod1 (X: Real; var Y: Real);
functionMyMethod2: Integer;
end;
```

У прикладі описаний клас TMyObject, який успадковується від класу Tobject. Поняття «спадкування класів» та поняття «властивості» будуть детально розглянуті далі. Поки можна визначити поняття властивості як поле, яке доступне не безпосередньо, а через надсилання повідомлень особливим методам.

Клас TMyObject має поле MyField та методи MyMethod1 та MyMethod2. Потрібно загострити увагу, що класи можуть бути описані або в секції інтерфейсу модуля Interface (під модулем тут розуміється файл з вихідним кодом виду Unit), або на верхньому рівні вкладеності секції реалізації Implementation. Не допускається опис класів усередині процедур та інших блоків коду.

Якщо клас включає поле з типом іншого класу, дозволено випереджаюче оголошення класу як у наступному прикладі:

```
type
TFirstObject = class;
TSecondObject = class (TObject)
Fist: TFirstObject;
{...}
end;
TFirstObject = class(TObject)
{...}
end;
```

Код методів описується нижче за текстом об'яв класів, наприклад:

```
ProcedureTMyObject.MyMethod1(X: Real; var Y: Real);
begin
y := 5.0 * sin(X);
end;
functionTMyObject.MyMethod2: Integer;
begin
{...}
MyMethod2: = MyField + 3;
end;
```

Для того щоб використовувати новий тип у програмі, потрібно, як мінімум, оголосити змінну цього типу, яка називається або змінною об'єктного типу, або екземпляром класу, або об'єктом:

```
var
AMyObject: TMyObject;
```

Відповідно до звичайної мови Borland Pascal, змінна AMyObject повинна містити в собі весь екземпляр об'єкта типу TMyObject (код та дані разом) – статичний об'єкт. Але в Delphi всі об'єкти динамічні, тому, не вдаючись до подробиць, виконаємо оператор:

```
{Дія зі створення екземпляра об'єкта}
AMyObject := TMyObject.Create;
```

Тепер інші об'єкти програми можуть надсилати повідомлення даному об'єкту. Надсилання повідомлень полягає у викликі методів потрібного об'єкта, наприклад:

```
var
До: Integer;
{...}
AMyObject.MyMethod1(2.3, Z);
До: = 6 + AMyObject.MyMethod2;
```

Методи- Це процедури та функції, описані всередині класу. Як видно, надсилання повідомлень у Object Pascal близьке до виклику процедур мови Pascal, але імені процедури, що викликається, або процедури-функції передує ім'я конкретного об'єкта, наприклад: AMyObject.

Опишемо два об'єкти AMyObject, BMyObject одного класу TMyObject:

```
var
AMyObject,
BMyObject: TmyObject;
```

При проектуванні програмісти вважають, кожен об'єкт (примірник класу) має власний внутрішній код методів і індивідуальну пам'ять, де розміщуються поля.

Насправді методи різних об'єктів одного класу загальні. Інакше кажучи, вони реалізуються загальним кодом, розташованим лише одному місці пам'яті. Це заощаджує пам'ять. У наведеному прикладі виклик методів:

```
AMyObject.MyMethod1(2.3, Z);
BMyObject.MyMethod1(0.7, Q)
```

насправді призведе до виконання одного й того ж коду при видимості, що моделюється для програміста, приналежності свого індивідуального коду різним об'єктам. Це зближує методи з процедурами та процедурами-функціями мови Pascal. Нагадаємо, що покажчик - це змінна, що містить у пам'яті адресу (номер осередку) іншої змінної, процедури або об'єкта. До складу класу входить покажчик на спеціальну таблицю, де міститься вся інформація, необхідна виклику методів. Від звичайних процедур та функцій методи відрізняються тим, що при виклику передається (неявно) покажчик на той об'єкт, який їх викликав. Усередині методів він доступний під зарезервованим Self.

Засилання та вилучення значень у поля, відповідно до прямого доступу, що не рекомендується в Object Pascal, практично не відрізняється від використання полів запису звичайної мови Pascal:

```
AMyObject.MyField := 3;
I := AMyObject.MyField + 5.
```

На відміну від методів поля об'єкта, це дані, унікальні для кожного об'єкта, що є екземпляром навіть одного класу. Поля AMyObject.MyField та BMyObject.MyField. є зовсім різними полями, оскільки вони розміщуються у різних об'єктах.

3. ОБЛАСТИ ВИДІМОСТІ

При описі нового класу важливим є розумний компроміс. З одного боку, потрібно приховати методи і поля, що є внутрішнім пристроєм класу. Незначні деталі на інших рівнях будуть марними і лише завадять цілісності сприйняття. Доступ до важливих деталей необхідно організувати через систему перевірок.

У мові Object Pascal введено механізм доступу до складових частин об'єкта, визначальний області, де можна використовувати (тобто області видимості). Поля та методи можуть належати до чотирьох груп, що відрізняються областями приховування інформації:

• *public* -загальні;
• *private* -особисті;
• *protected*-захищені;
• *published*- Опубліковані.

Розподіл на складові працює на рівні файлів модулів (Unit у сенсі мови Pascal). Якщо ви потребуєте спеціального захисту об'єкта або його частини, то для цього необхідно помістити його в окремий модуль, в якому є власні секції interface та implementation.

Поля, властивості та методи, що знаходяться у секції public, не мають обмежень на видимість. Вони доступні з інших функцій і методів об'єктів як в даному модулі, так і в інших, що посилаються на нього. Зазвичай методи цієї секції утворюють інтерфейс між об'єктним обміном повідомленнями під час виконання програми (run-time).

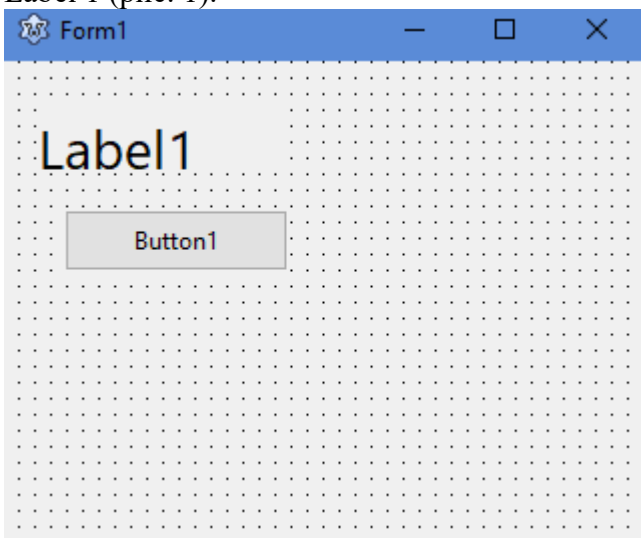
Поля, властивості та методи, що знаходяться в секції private, доступні тільки в методах класу та функціях, що містяться в тому ж модулі, що і клас, що описується. Така директива дозволяє приховати деталі внутрішньої реалізації класу всіх. Елементи з секції private можна змінювати, і це не позначатиметься на програмах, що працюють із об'єктами цього класу. Єдиний спосіб для когось іншого – звернутися до них – переписати заново створений вами модуль.

Розділ protected комбінує функціональне навантаження розділів private і public таким чином, що якщо ви хочете приховати внутрішні механізми вашого об'єкта від кінцевого користувача, цей користувач не зможе в run-time використовувати жодне з об'яв об'єкта з його protected-області. Але це не завадить розробнику нових компонентів використовувати ці

механізми в інших успадкованих класах, тобто protected-оголошення доступні у будь-якого зі спадкоємців вашого класу.
Розділ published виявився необхідним при введенні в Object Pascal можливості встановлення властивостей та поведінки компонентів ще на етапі конструювання форм і самої програми (design-time) у середовищі візуальної розробки програм Delphi. Саме published-оголошення доступні через Object Inspector, чи це посилання на властивості або обробники подій. Під час виконання програми розділ об'єкта published повністю аналогічний public.
Слід зазначити той факт, що при породженні нового класу шляхом успадкування можливе перенесення оголошень з одного розділу в інший з єдиним обмеженням: якщо ви робите приховування оголошення за рахунок його перенесення в private, в подальшому його «витягування» у спадкоємця в більш доступний розділ в іншому модулі буде вже неможливо. Таке обмеження, на щастя, не поширюється на динамічні методи обробки повідомлень Windows.
Приклад опису класу із заданими областями видимості наведено нижче.

4. ІНКАПСУЛЯЦІЯ

Класичне правило об'єктно-орієнтованого програмування стверджує, що для забезпечення надійності небажаний прямий доступ з інших об'єктів до полів об'єкта: читання та оновлення їхнього вмісту повинно здійснюватися через виклик відповідних методів. Це і називається інкапсуляцією. Досі ідея інкапсуляції впроваджувалась у програмування лише за допомогою закликів та прикладів у документації, але в мові ж Object Pascal з'явилася відповідна конструкція. В об'єктах Object Pascal користувач об'єкта може бути повністю відгороджений з його полів за допомогою властивостей.
Роботу з властивостями розглянемо на прикладі. Нехай ми створили за допомогою дизайнера форм Delphi екранну форму Forml із двома елементами візуальних компонентів: Button 1 та Label 1 (рис. 1).



*Мал. 1.*Екранна форма прикладу

Натисніть кнопку Button1 і відредагуємо вихідний текст модуля до наступного тексту:

```
unittestir;
interface
uses
Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs,
StdCtrls;
type
TForm1 = class(TForm)
Button1: TButton;
Label1: TLabel;
procedureButton1Click(Sender: TObject);
```

```
private
{Private declarations}
public
{Public declarations}
end;
type
TSomeType = String;
type
TAnObject = class(TObject)
private
FValue: TSomeType;
functionGetAPproperty: TSomeType;
procedureSetAPproperty (ANewValue: TSomeType);
public
propertyAProperty: TSomeType
readGetAProperty
writeSetAProperty;
end;
var
Form1: TForm1;
AnObject: TAnObject;
implementation
{$R *.DFM}
procedureTForm1.ButtonlClick(Sender: TObject);
begin
AnObject := TAnObject.Create;
AnObject.AProperty := 'Привіт!';
Label1.Caption := AnObject.AProperty;
end;
procedureTAnObject.SetAProperty(
ANewValue: TSomeType);
begin
FValue: = ANewValue; {Засилання значення у полі}
end;
functionTAnObject.GetAProperty: TSomeType;
begin
GetAProperty: = FValue; {Читання значення з поля}
end;
end.
```

Збережемо проект (Save Project As). При збереженні проекту вкажемо нове ім'я модуля – testir та нове ім'я проекту – PrTestir. Розглянемо текст файлу проекту, що вийшов (пункти меню View і далі Project Source):

```
programPrTestir;
uses
Forms,
testir in 'testir.pas' {Form1};
{$R *.RES}
begin
Application.Initialize;
Application.CreateForm(TForm1, Form1);
Application.Run;
end.
```

Цей файл містить текст основної програми PrTestir. До основної програми підключаються модуль Forms (для роботи з формою) та вихідний код модуля testir. Три оператори основної програми, що виконуються, послідовно організують новий обчислювальний процес виконання написаної програми PrTestir, створять об'єкт форми Form1, здійснять запуск програми на виконання (Run).

У прикладі текст модуля містить згенерований текст оголошення об'єктного типу Tform1. Тип містить вказівки на агрегацію в даному класі об'єкта кнопки Button1: Tbutton і об'єкта Label1: Tlabel. Завдяки агрегації екземпляри об'єктів кнопки та написи будуть створюватися одночасно зі створенням екземпляра об'єкта форми, в результаті чого як би вийде об'єкт

форми, що спільно працює з кнопкою і написом. У типі Tform1 Delphi по подвійному клацанню миші по кнопці <Button1> згенерувала прототип виклику методу:

```
procedureButton1Click(Sender: TObject).
```

Також Delphi автоматично згенерувала змінну об'єктного типу

```
var
Form1: TForm1;
```

і в секції реалізації вставила текст "порожній" процедури Button1Click відпрацювання дій із натискання кнопки Button1.

Розглянемо елементи, додані нами у текст модуля реалізації інкапсуляції. Отже, нами було набрано текст опису класу TAnObject та змінна даного об'єктного типу AnObject:

```
type
TAnObject = class(TObject) private
FValue: TSomeType;
functionGetAPproperty: TSomeType;
procedureSetAPproperty (ANewValue: TSomeType);
public
propertyAProperty: TSomeType
readGetAProperty
writeSetAProperty;
end;
var
AnObject: TanObject.
```

Зазвичай властивість (property) визначається трьома своїми елементами: полем та двома методами, які здійснюють його запис/читання:

```
private
FValue: TSomeType;
functionGetAPproperty: TSomeType;
procedureSetAPproperty (ANewValue: TSomeType);
public
propertyAProperty: TSomeType
readGetAProperty
writeSetAProperty;
...
procedureTAnObject.SetAProperty(ANewValue: TSomeType);
begin
FValue: = ANewValue; {Засилання значення у полі}
end;
functionTAnObject.GetAProperty: TSomeType;
begin
GetAProperty: = FValue; {Читання значення з поля}
end;
```

В даному прикладі властивість Aproperty виконує таку саму функцію, яку в попередньому прикладі виконувало поле MyField. Доступ до значення властивості Aproperty здійснюється через виклики методів GetAProperty та SetAProperty. Однак у зверненні до цих методів у явному вигляді немає необхідності (у прикладі вони навіть захищені — protected), достатньо написати:

```
AnObject.AProperty: = ...;
... := AnObject.Aproperty;
```

та компілятор відтранслює ці оператори на виклики методів.

Розглянемо три оператори, вписані в текст "порожній" процедури (метод) Button1Click:

```
AnObject := TAnObject.Create;
AnObject.AProperty := 'Привіт!';
Label1.Caption := AnObject.AProperty;
```

Перший оператор створює екземпляр об'єкта. Другий оператор у недоступне за інтерфейсом (protected) поле Fvalue посилає за допомогою недоступної за інтерфейсом (protected) процедури AnObject.SetAProperty текст «Привіт!». Третій оператор за допомогою недоступної за інтерфейсом (protected) процедури АпОЬject.GetAProperty зчитує дані з недоступного за інтерфейсом поля Fvalue і посилає їх як Caption об'єкта написи Label1.

Сама властивість AProperty описано як загальнодоступне за інтерфейсом (public), тобто зовні властивість виглядає в точності, як доступ до звичайного поля, але за будь-яким зверненням до якості можуть стояти необхідні програмісту дії. Наприклад, якщо у нас є об'єкт, що є квадратом на екрані, і ми його властивості «колір» присвоюємо значення «білий», то відбудеться негайне перемальовування, що приводить реальний колір на екрані у відповідність до значення властивості.
У методах, що входять до складу властивостей, може здійснюватися перевірка величини, що встановлюється, на потрапляння в допустимий діапазон значень і виклик інших процедур, що залежать від внесених змін. Якщо ж потреби у спеціальних процедурах читання та/або запису немає, то, можливо, замість імен методів застосовувати імена полів.
Розглянемо таку конструкцію:

```
type
TPropObject = class(TObject)
FValue: TSomeType;
procedureDoSomething;
procedureCorrect(AValue: Integer);
procedureSetValue(NewValue: Integer);
procedureAValue: Integer read Fvalue
writeSetValue;
end;
...
procedureTPropObject.SetValue(NewValue: Integer);
begin
if(NewValue <> FValue) and Correct (NewValue)
then
FValue: = NewValue; {Засилання значення у полі}
DoSomething;
end;
```

У цьому прикладі читання значення властивості AValue означає просто читання поля FValue. Зате при присвоєння йому значення всередині методу SetValue викликається одразу два методи.
Якщо властивість повинна тільки читатися або записуватися, то в його описі може бути тільки відповідний метод:

```
type
TAnObject = class(TObject)
propertyAProperty: TSomeType read GetValue;
end;
```

У цьому прикладі поза об'єктом значення якості можна лише прочитати. Спроба встановити значення Aproperty викликає помилку компіляції.

5. ОБ'ЄКТИ ТА ЇХ ЖИТТЄВИЙ ЦИКЛ

Як створюються та знищуються об'єкти? У Object Pascal екземпляри об'єктів можуть бути лише динамічними! Це означає, що у наведеному на початку розділу фрагменті змінна AMyObject хоч і виглядає як статична змінна мови Pascal, насправді є вказівником, що містить адресу об'єкта. Будь-яка змінна об'єктного типу є покажчик (покажчик — змінна, що містить значення адреси оперативної пам'яті).
Пам'ять під конкретні об'єкти (динамічні екземпляри класів) виділяється диспетчером пам'яті в період виконання програми в особливій heap-області, де має бути вільне місце для розміщення нових об'єктів. Heap - "купа сміття". Диспетчер пам'яті може розміщувати в heap-області нові об'єкти, так і видаляти вже непотрібні, звільняючи пам'ять під всі нові об'єкти. Саме при розміщенні об'єкта в пам'яті ініціалізується значення змінної типу об'єкта об'єкта. При видаленні об'єкта змінна об'єктного типу ініціалізується значенням nil (порожній покажчик) - немає об'єкта пам'яті. При розміщенні нового об'єкта в пам'яті може виявитися, що загальна вільна пам'ять має великий обсяг, але будь-який з ділянок пам'яті, що звільнилися, займаних вже віддаленими об'єктами, виявляється менше обсягу нового

великого об'єкта. Для уникнення такої ситуації при використанні об'єктних програм встановлюють в ЕОМ оперативну пам'ять свідомо великого обсягу. Новий екземпляр об'єкта створюється особливим методом – конструктором, а знищується спеціальним методом – деструктором:

```
AMyObject := TMyObject.Create; {дії зі створеним об'єктом}
...
AMyObject.Destroy; {знищення об'єкта}
```

У Object Pascal конструкторів у класу може бути кілька. Загальноприйнято називати конструктор Create. Типова назва деструктора - Destroy. Рекомендується використовувати для знищення екземпляра об'єкта метод Free, який спочатку перевіряє покажчик (чи не дорівнює він nil) і потім викликають Destroy.

Для того щоб правильно проініціалізувати в об'єкті поля, що створюється, відносяться до класу-предка, потрібно відразу ж при вході в конструктор викликати конструктор предка:

```
constructorTmyObject.Create;
begin
inheritedCreate;
...
end;
```

Метод створення об'єкта MyObject типу TMyObject успадкований від класу предка TObject. Взявши будь-який із прикладів, що постачаються разом із Delphi, ми виявимо, що там майже немає викликів конструкторів та деструкторів. Справа в тому, що будь-який компонент, що потрапив при візуальному проектуванні в програму з палітри компонентів, включається до певної ієрархії. Ієрархія ця замикається на формі (TForm): всім її складових частин конструктори і деструктори викликаються автоматично, незримо для програміста.

Хто створює та знищує форми? Це робить програму (глобальний об'єкт з ім'ям Application). У файлі проекту (.DPR) можна побачити виклик функції CreateForm, призначений для цієї мети. Що ж до об'єктів, створюваних динамічно (під час виконання докладання), тут потрібен явний виклик конструктора.

6. СПАДЧИНА

Другим «стовпом» ОВП, крім інкапсуляції, є успадкування. Цей простий принцип означає, що якщо потрібно створити новий клас, який лише трохи відрізняється від старого, то немає потреби в переписуванні заново вже існуючих полів і методів. Новий клас

```
TNewObject = class(TOldObject);
```

є нащадком, чи дочірнім класом старого класу, званого предком, чи батьківським класом. Додаються до нього лише нові поля, методи та властивості.

У Object Pascal усі класи є нащадками класу TObject. Тому, якщо будується дочірній клас прямо від TObject, то у визначенні TOject можна не згадувати. Наступні два вирази однаково вірні:

```
TMyObject = class(TObject);
TMyObject = class;
```

Використання останнього висловлювання виправдане, якщо розробник хоче показати, що, за його задумом, клас, що проектується, як би не має предків.

Наведемо оголошення базового всім об'єктних типів класу TObject:

```
TObject = class
constructorCreate;
destructorDestroy; virtual;
procedureFree;
class functionNewInstance: TObject; virtual;
procedureFreeInstance; virtual;
class procedureInitInstance(Instance: Pointer):
TObject;
functionClassType: TClass;
class functionClassName: string;
class functionClassParent: TClass;
```

```
class functionClassInfo: Pointer;
class functionInstanceSize: Word;
class functionInheritsFrom(AClass: TClass):
Boolean;
procedureDefaultHandler(var Message); virtual;
procedureDispatch (var Message);
class functionMethodAddress(const Name: string):
Pointer;
class functionMethodName(Address: Pointer):
string;
функція FieldAddress(const Name: string):
Pointer;
end;
```

Така архітектура можлива лише за наявності механізму підтримки інформації про типи – RTTI (RunTime Type Information). Основою такого механізму є внутрішня структура класів і, зокрема, можливість доступу до неї за рахунок використання методів класів, що описуються конструкцією class function...

Успадковані від предка поля та методи доступні у дочірньому класі; якщо має місце збіг імен методів, ці методи перекриваються.

По тому, які дії відбуваються під час виклику, методи поділяються на групи:

• статичні (static);
• віртуальні (virtual);
• динамічні (dynamic);
• абстрактні (abstract).

Статичні методи, а також поля в об'єктах-нащадках поводяться однаково: можна без обмежень перекривати старі імена і змінювати тип методів:

```
type
T1stObj = class
I: Real;
procedureSetData(Avalue: Real);
end;
T2ndObj = class(T1stObj)
I: Integer;
procedureSetData(Avalue: Integer);
end;
...
procedureT1stObj.SetData;
begin
i: = v;
end;
procedureT2nd0bj.SetData;
begin
i := 0;
inheritedSetData(0.99);
end;
```

У цьому прикладі різні методи з ім'ям SetData надають значення різним полям з ім'ям і. Перекрите поле предка недоступне нащадку. На відміну від поля всередині інших методів, перекритий метод доступний при вказівці зарезервованого слова bherited. Методи об'єктів за умовчанням статичні — їх адреса визначається ще на стадії компіляції проекту. Вони викликаються найшвидше.

Мова C++ дозволяє так зване множинне спадкування. У цьому випадку новий клас може успадковувати частину своїх елементів від одного батьківського класу, а частина від іншого, це поряд із зручностями часто призводить до проблем.

У Object Pascal поняття множинного спадкування відсутнє. Якщо необхідно, щоб новий клас поєднував властивості кількох, можна породити предки один від одного або включити в клас кілька полів, що відповідають цим бажаним класам.

Принципово відрізняються від статичних методів віртуальні та динамічні методи. Вони можуть бути оголошені шляхом додавання відповідної директиви virtual чи dynamic. Адреса

таких методів визначається під час виконання програми за спеціальною таблицею. З погляду наслідування методи цих двох видів однакові: вони можуть бути перекриті в дочірньому класі лише однойменними методами, що мають той самий тип.
Спільним їм є те, що з їх виклику адреса визначається під час компіляції, а під час виконання шляхом пошуку у спеціальних таблицях. Такий спосіб ще називається пізнім зв'язуванням. Різниця між методами полягає особливо у пошуку адреси.
Коли компілятор зустрічає звернення до віртуального методу, він підставляє замість звернення до конкретної адреси код, який звертається до спеціальної таблиці та витягує звідти потрібну адресу. Ця таблиця називається таблицею віртуальних методів (Virtual Method Table, VMT) і є для кожного об'єктного типу. У ній зберігаються адреси всіх віртуальних методів класу незалежно від того, чи вони успадковані від предка чи перекриті. Звідси і переваги, і недоліки віртуальних методів: вони викликаються порівняно швидко (але повільніше статичних), проте для зберігання покажчиків на них потрібна велика кількість пам'яті.
Динамічні методи викликаються повільніше, але дозволяють більш економно витрачати пам'ять. Кожному динамічному методу системою надається унікальний індекс. У таблиці динамічних методів (Dynamic Method Table, DMT) класу зберігаються індекси та адреси тільки тих динамічних методів, які описані в даному класі (базова інформація з обробки динамічних методів міститься в модулі x:\delphi\source\rtl\sys\dmth.asm). Під час виклику динамічного методу відбувається пошук у цій таблиці. У разі невдачі проглядаються всі класи-предки в порядку ієрархії і, нарешті, TObject, де є стандартний обробник виклику динамічних методів. Економія пам'яті очевидна.
Для перекриття і віртуальних, і динамічних методів служить нова директива override, за допомогою якої (і тільки з нею!) можна перевизначати обидва типи методів:

```
type
TFirstClass = class
FMyField1: Integer;
FMyField2: Longlnt;
procedureStatMethod1;
procedureVirtMethod1; virtual;
procedureVirtMethod2; virtual;
procedureDynaMethod1; dynamic;
procedureDynaMethod2; dynamic;
end;
TSecondClass = class(TFirstClass)
procedureStatMethod;
procedureVirtMethod; override;
procedureStatMethod; override;
end;
var
Obj2: TFirstClass;
Obj1: TSecondClass;
```

Перший із методів у прикладі створюється заново, решта — перекривається. Спроба застосувати override до статичного методу викликає помилку компіляції.
У Object Pascal абстрактними називаються методи, які визначені у класі, але не містять жодних дій, ніколи не викликаються і мають бути перевизначені у нащадках класу. Абстрактними можуть бути лише віртуальні та динамічні методи. Для цього використовується директива abstract, що вказується при описі методу:

```
procedureNeverCallMe; virtual; abstract;
```

При цьому жодного коду для цього писати не потрібно. Як і раніше, виклик методу NeverCallMe призведе до помилки часу виконання.

7. ПОЛІМОРФІЗМ

Розглянемо приклад, який пояснює, навіщо потрібне використання абстрактних методів. Нехай у нас є якесь узагальнене поле для зберігання даних — клас TField і три його нащадки

— для зберігання рядків, цілих та дійсних чисел. У разі клас Tfield не використовується сам собою; його основне призначення - бути родоначальником ієрархії конкретних класів - «полів» і дати можливість абстрагуватися від частковостей. Хоча параметр у ShowData описаний як TField, але якщо помістити туди об'єкт цього класу, відбудеться помилка виклику абстрактного методу:

```
type
TField = class
functionGetData: string; virtual; abstract;
end;
TStringField = class(TField)
Data: string;
functionGetData: string; override;
end;
TIntegerField = class(TField)
Data: Integer;
functionGetData: string; override;
end;
TExtendedField = class(TField)
Data: Integer;
function GetData: string; override;
end;
functionTStringField.GetData;
begin
GetData: = Data;
end;
functionTIntegerField.GetData;
begin
GetData := IntToStr(Data);
end;
functionTExtendedField.GetData;
begin
GetData := IntToStrF(Data, ffFixed, 7, 2);
end;
...
procedureShowData(AField: TField);
begin
Form1.Label1.Caption := AField.GetData;
end;
```

У цьому прикладі класи містять різнотипні дані, які «вміють» лише повідомити про значення цих даних текстовим рядком (за допомогою методу GetData). Зовнішня щодо них процедура ShowData отримує об'єкт як параметр і показує цей рядок.

Правила контролю відповідності типів (typecasting) мови Pascal свідчать, що об'єкту як покажчику на екземпляр об'єктного типу може бути присвоєно адресу будь-якого екземпляра будь-якого з дочірніх типів. У процедурі ShowData параметр описаний як TField. Це означає, що в неї можна передавати об'єкти класів і TStringField, і TIntegerField, і TExtendedField, і будь-якого іншого нащадка TField.

Але який (точніше, чий) метод GetData буде викликаний? Той, що відповідає класу фактично переданого об'єкта. Цей принцип називається поліморфізмом, і він, мабуть, є найбільш важливим «козирем» ОВП. Припустимо, є справа з деякою сукупністю явищ чи процесів. Щоб змоделювати їх засобами ОВП, необхідно виділити їх найзагальніші, типові риси. Ті з них, які не змінюють свого змісту, мають бути реалізовані як статичні методи. Ті ж, які варіюють при переході від загального до приватного, краще надати форму віртуальних методів. Основні «родові» риси (методи) потрібно описати в класі-предку і потім перекрити їх у класах-нащадках. У попередньому прикладі програмісту, який пише процедуру на кшталт ShowData, важливим є лише одне: те, що будь-який об'єкт, переданий у неї, є нащадком TField, і він вміє повідомити про значення своїх даних (виконавши метод GetData).

Якщо таку процедуру скомпілювати та помістити в динамічну бібліотеку, то цю бібліотеку можна буде раз і назавжди використовувати без змін, хоча з'являтимуться і нові, невідомі в момент її створення класи-нащадки TField!
Наочний приклад використання поліморфізму дає сама Delphi. У ній є клас TComponent, лише на рівні якого зосереджені певні «правила» того, як взаємодіяти з середовищем розробки та іншими компонентами. Дотримуючись цих правил, можна породжувати від TComponent свої компоненти, налаштовуючи Delphi рішення спеціальних завдань.

8. ОБРОБКА ПОВІДОМЛЕНЬ

Потреба динамічних методів особливо відчутна розробки об'єктів, відповідних елементам інтерфейсу Windows, коли кожен із великої ієрархії об'єктів містить обробники десятків різноманітних повідомлень.
Методи, призначені спеціально для дня обробки повідомлень Windows, становлять підмножину динамічних методів і оголошуються директивою message, за якою слідує індекс — ідентифікатор повідомлення. Вони повинні бути обов'язково описані як процедури, що мають один var-параметр, який може бути описаний довільно, наприклад:

```
type
TMyControl = class(TWinControl)
procedureWMSize (var Message: TWMSize);
messageWM_SIZE;
end;
type
TMyOtherControl = class(TMyControl)
procedureResize(var Info);
messageWM_SIZE;
end;
```

Для перекриття методів обробки повідомлень директива override не використовується. У цьому випадку слід зберегти в описі директиву message з індексом методу.
Необхідності винаходити власні структури, які по-своєму інтерпретують зміст того чи іншого повідомлення, немає: для більшості повідомлень Windows типи вже описані в модулі MESSAGES.
В обробниках повідомлень (і тільки в них) можна викликати метод-предок, просто вказавши ключове слово inherited без вказівки його імені та перетворення типу параметрів: предок буде знайдений за індексом. Слід нагадати, що система обробки повідомлень вбудована в Object Pascal на рівні моделі об'єктів, і найзагальніший обробник метод DefaultHandler описаний в класі TObject.

9. ПОДІЇ І ДЕЛЕГУВАННЯ

Працювати з великою кількістю повідомлень, навіть маючи під рукою довідник, нелегко, тому одним із великих досягнень Delphi є те, що програміст позбавлений необхідності працювати з повідомленнями Windows (хоча така можливість у нього є). Стандартних подій у Delphi не більше двох десятків, і всі вони мають просту інтерпретацію, яка не потребує глибоких знань середовища.
Розглянемо, як реалізовані події лише на рівні мови Object Pascal. Події - це властивості процедурного типу, призначені для створення користувальницької реакції на ті чи інші вхідні дії:

```
propertyOnMyEvent : TMyEvent read FonMyEvent
writeFonMyEvent;
```

Привласнити таку властивість значення — це означає вказати об'єкту адресу методу, який буде викликатись у момент настання події. Такі методи назвемо обробниками подій.
Наприклад:

```
Application.OnActive := MyActivatingMethod;
```

Це означає, що при кожній активізації Application (так називається об'єкт, що відповідає працюючому додатку) буде викликаний метод-обробник MyActivatingMethod.
Всередині бібліотеки часу виконання Delphi виклики обробників подій перебувають у методах, що обробляють повідомлення Windows. Виконавши принципово необхідні дії, цей метод перевіряє, чи відома адреса оброблювача, і якщо це так, викликає його:

```
ifAssigned(FonMyEvent)
then
FonMyEvent(Self);
```

Залежно від походження та призначення події мають різні типи. Спільним для всіх є параметр Sender, що вказує на об'єкт-джерело події. Найпростіший тип - TNotifyEvent - не має інших параметрів:

```
TNotifyEvent = procedure(Sender: TObject) of object;
```

Тип методу, призначений для сповіщення про натискання кнопки, передбачає передачу програмісту коду цієї кнопки, а пересування миші — її координат тощо.
Всі події в Delphi прийнято називати з On: OnCreate, OnMouseMove, OnPaint і т. д.
Клацнувши в Інспекторі об'єктів на сторінці Events в полі будь-якої події, автоматично виходить заготівля методу потрібного типу. При цьому його ім'я складатиметься з імені поточного компонента та імені події (без On), а відноситися він буде до поточної форми. Нехай, наприклад, у формі Form1 є текст Label1. Тоді для обробки клацання мишею (подія OnClick) буде створено метод Tform1. Label1Click.
Оскільки події є властивостями об'єкта, їх значення можна змінювати під час виконання програми. Така чудова нагода називається делегуванням. Можна в будь-який момент взяти способи реакції на події одного об'єкта і делегувати їх іншому:

```
Object.OnMouseMove := Object2.OnMouseMove;
```

Але який механізм дозволяє замінювати обробники, адже це не просто процедури, а методи? Тут до речі доводиться введене в Object Pascal поняття покажчика на метод. Крім явно описаних параметрів методу передається ще й покажчик на екземпляр (Self), що його викликав. Ви можете описати тип процедури, яка буде сумісна з присвоєння методом (тобто передбачати отримання Self). Для цього до її опису потрібно додати зарезервовані слова of Object. Покажчик на метод – це покажчик на таку процедуру:

```
type
TmyEvent = procedure(Sender: TObject;
varAvalue: Integer) of object;
T1stObject = class;
FOnMyEvent: TMyEvent;
propertyOnMyEvent: TMyEvent read FonMyEvent
writeFonMyEvent;
end;
T2ndObject = class;
procedureSetValue1(Sender: TObject;
varAvalue: Integer);
procedureSetValue2(Sender: TObject;
varAvalue: Integer);
end;
...
var
Obj1: T1stObject;
Obj2: T2ndObject;
begin
Obj1: = T1stObject.Create;
Obj2: = T2ndtObject.Create;
Obj1.OnMyEvent := Obj2.SetValue1;
Obj2.OnMyEvent := Obj2.SetValue2;
...
end;
```

Як у цьому прикладі, так і всюди Delphi за властивостями-подіями стоять поля, що є покажчиками на метод. Таким чином, при делегуванні можна надавати методи інших класів.

Тут обробником події OnMyEvent об'єкта Obj1 по черзі виступають методи SetValue1 та SetValue2 об'єкта Obj2.

10. ФУНКЦІЇ КЛАСУ

Object Pascal має можливість визначення полів процедурного типу. Очевидно, що в тілі функцій, що прив'язуються до цих полів, розробнику необхідний доступ до інших полів об'єкта, методів і т.п. . Такі функції називаються функціями класів. Для оголошення функцій класів необхідно використовувати спеціальну конструкцію function... of object.

11. ПРИВЕДЕННЯ ТИПІВ

На операції зі змінною певного типу компілятор зазвичай накладає обмеження, дозволяючи виконання лише операцій, характерні для зазначеного типу даних. Іноді компілятор здійснює автоматичне наведення типу, наприклад, при присвоєнні цілого значення дійсної змінної.
У мові Pascal є механізм очевидного приведення типів.
В операції is визначається, чи належить цей об'єкт зазначеному типу чи одному з його нащадків.
Вираз, наведений у наступному прикладі, повертає True, якщо змінна AnObject посилається на зразок об'єктного типу TMyClass або одного з його нащадків.

```
AnObject is TmyClass
```

Сама собою операція is не є операцією завдання типу. У ній лише перевіряється сумісність об'єктних типів. Для коректного приведення типу об'єкта застосовується операція as:

```
WithAnObject як TmyClass do ...
```

Можливий такий спосіб приведення типу без явного вказівки as.

```
WithTMyClass(AnObject)do ...
```

У програмах перед операцією as перевіряють сумісність типів з допомогою операції is. Якщо типи несумісні, запускається обробник виключної ситуації EinvalidCast.
Таким чином, у конструкції as операція явного приведення типу виявляється укладеною в безпечну оболонку:

```
IfAnObject is TobjectType then
withTobjectType(AnObject) do ...
else
raiseEinvalidCast.Create('Неправильне приведення типу');
```

12. ОБ'ЄКТНЕ ПОСИЛАННЯ

Delphiдозволяє створити спеціальний описник об'єктного типу (саме тип, а чи не на екземпляр!), відомий як object reference — об'єктна посилання.
Об'єктні посилання використовуються у таких випадках:
- Тип створюваного об'єкта не відомий на етапі компіляції;
- Необхідний виклик методу класу, чий тип не відомий на етапі компіляції;
- В якості правого операнда в операціях перевірки та приведення типів з використанням is і as.

Об'єктне посилання визначається з використанням конструкції class of... . Наведемо приклад оголошення та використання class reference:

```
type
TMyObject = class (TObject)
MyField:TMyObject;
constructorCreate;
end;
TObjectRef = class of TObject;
...
var
ObjectRef:TObjectRef;
```

```
s:string;
begin
ObjectRef:=TMyObject; {Привласнюємо тип, а не екземпляр!}
s:=ObjectRef.ClassName; {рядок s містить 'TMyObject'}
end;
```

Таким чином, Delphi визначено спеціальне посилання TClass, сумісна з присвоєння з будь-яким спадкоємцем TObject. Аналогічно оголошені класи: TPersistentClass і TCotponentClass.

13. СТРУКТУРНА ОБРОБКА ВИКЛЮЧНИХ СИТУАЦІЙ

Під винятковою ситуацією (raise) тут розуміється ситуація, яка дозволяє без особливих додаткових заходів продовжити виконання програми, наприклад розподілу на нуль, переповнення розрядної сітки, вилучення квадратного кореня з негативного числа тощо. При традиційній обробці помилок, помилки, виявлені в процедурі, зазвичай передаються назовні (в викликав процедуру) у вигляді значення функції, параметрів або глобальних змінних (прапорів), що повертається. Кожна процедура, що викликає, повинна перевіряти результат виклику на наявність помилки і виконувати відповідні дії. Часто це просто вихід у більш верхню процедуру, що викликає, і т.д.

Структурна обробка виняткових ситуацій — це програмний механізм, що дозволяє програмісту у разі виникнення помилки (виключної ситуації — exception) зв'язатися з кодом програми, підготовленим для обробки такої помилки. У Delphi система називається структурною, оскільки обробка помилок визначається областю «захищеного» коду. Такі області можуть бути вкладені. Виконання програми не може перейти на довільну ділянку коду. Виконання програми може перейти лише на обробник виняткової ситуації активної програми.

Модель виняткових ситуацій в Object Pascal є невідновлюваною (non-resumable). При виникненні виняткової ситуації ви вже не зможете повернутися в точку, де вона виникла, для продовження виконання програми (це дозволяє зробити лише відновлювану модель).

Для обробки виняткових ситуацій до мови Object Pascal додано нове ключове слово «try», яке використовується для позначення першої частини захищеної ділянки коду. Існують два типи захищених ділянок:

1) try..except;
2) try..finally.

Перший тип використовується для обробки виняткових ситуацій. Його синтаксис:

```
try
Statement1;
Statement2;
...
except
on Exception1 do Statement;
on Exception2 do Statement;
...
else
Statements; {default exception-handler}
end;
```

Для впевненості в тому, що ресурси, зайняті вашою програмою, звільняться в будь-якому випадку, можете використовувати конструкцію другого типу. Код, розташований у частині finally, виконується у будь-якому разі, навіть якщо виникає виняткова ситуація. Відповідний синтаксис:

```
try
Statement1;
Statement2;
finally
Statements; {These statements always execute}
end;
```

Додаток 5 ОСНОВНІ ТЕРМІНИ І ВИЗНАЧЕННЯ

Абстрагування від проблеми ігнорування ряду подробиць для того, щоб звести завдання до більш простого завдання.

Анотація машина Дейкстри— застосовується у проектуванні архітектури системи, нижній рівень абстракції — це рівень апаратури. Кожен рівень реалізує абстрактну машину з дедалі більшими можливостями.

Абстрактний батьківський клас- Батьківський клас, що не має екземплярів об'єктів.

Абстракція- Уявне відволікання, відокремлення від тих чи інших сторін, властивостей або зв'язків предметів і явищ для виявлення суттєвих їх ознак.

Абстракція сутності довільна абстракція. Об'єкт є корисною модель якоїсь сутності в предметній області.

Автоматизована система (АС)- Організаційно-технічна система, що забезпечує вироблення рішень на основі автоматизації інформаційних процесів у різних сферах діяльності (управління, проектування, виробництво тощо) або їх поєднаннях, система, що складається з персоналу та комплексу засобів автоматизації його діяльності, що реалізує інформаційну технологію виконання встановлених функцій.

Автономне тестування (тестування модуля) (module testing) контроль окремого модуля в ізольованому середовищі (наприклад, за допомогою провідної програми); інспекція тексту модуля на сесії програмістів, яка іноді доповнюється математичним доказом правильності модуля.

Агрегований об'єкт об'єкт, складений із подобъектов. Подіб'єкти називаються частинами агрегату, і агрегат відповідає за них.

Алгоритм суворо однозначно визначена для виконавця послідовність дій, що призводять до вирішення задачі.

*Альфа-тестування (системне тестування, лабораторні випробування)*фаза тестування, що виконується розробниками для підтвердження, що всі фрагменти правильно інтегровані в систему, а система працює надійно.

Аналіз (Від грец. Analysis - розкладання, розчленування) - прийом розумової діяльності, пов'язаний з уявним (або реальним) розчленуванням на частини предмета, явища або процесу. Теоретично проектування аналіз — це процес визначення функціонування за заданим описом системи.

Артефакт реалізації -щось, що не можна виявити у постановці розв'язуваного завдання, але необхідне складання програми.

Архітектура системи структура об'єднання кількох програмних засобів на одне ціле.

АС -див. автоматизована система.

Атестація (certification) -авторитетне підтвердження правильності програми.

Бета-тестування це фаза загального тестування, коли програмний виріб поставляється обмеженому колу кінцевих користувачів більш жорсткого тестування.

Блочно-ієрархічний підхід приватний евроритм системного підходу, у якому процес проектування і ставлення до об'єкті розчленовується на ієрархічні рівні. На найвищому рівні використовується найменш детальне уявлення, що відображає найзагальніші риси та особливості проектованої системи. На кожному новому послідовному рівні розробки ступінь подробиці розгляду зростає, причому система розглядається не в цілому, а окремими блоками.

Візуальне моделювання- Процес графічного представлення моделі за допомогою деякого стандартного набору графічних елементів.

Впровадження -стадія, по завершенні якої програмна документація розмножена у потрібній кількості, програма встановлена та супроводжується, користувачі навчені.

Відновлюваність програмного забезпечення- властивість, що характеризує можливість пристосовуватися до виявлення помилок та їх усунення.

Генетичний аналіз дослідження об'єкта з його відповідність законам розвитку програмних систем. У процесі аналізу вивчається історія розвитку (генеза) досліджуваного об'єкта:

конструкції аналогів та можливих частин, технології виготовлення, обсяги тиражування, мови програмування тощо.
ГОСТ-державний стандарт.
Деструктор -особливий спосіб самого об'єкта, який би знищення даного объекта.
Діаграма варіантів використання діаграма, яка відображає взаємодію між варіантами використання, що представляють функції системи, та дійовими особами, які представляють людей або системи, які отримують або передають інформацію до цієї системи.
Діаграма класів- Діаграма, що відображає взаємодію між класами системи.
Діаграма компонент діаграма, що показує, як виглядає модель фізично. На ній зображуються компоненти (файли) програми та зв'язки між ними.
Діаграма кооперативна діаграма, що відображає ту саму інформацію, що і діаграми послідовності, але зв'язок з часом відсутня.
Діаграма послідовності діаграма, що відображає потік подій, що відбуваються у рамках варіанта використання.
*Діаграма потоків даних (ДПД)*діаграма, що описує порядок зміни даних від їх джерел через процеси, що їх перетворюють, до їх споживачів.
Діаграма розміщення- Діаграма, що показує фізичне розташування різних компонентів програмної системи в мережі.
Діаграма станів діаграма, призначена для моделювання різних станів, у яких може бути об'єкт. Коли діаграма класів показує статичну картину класів та його зв'язків, діаграма станів застосовується в описах динаміки поведінки системи.
Динамічна змінна це як би статична змінна, але розміщується в особливій області пам'яті поза кодом програми. У будь-який момент часу пам'ять для розміщення динамічних змінних може виділятися, так і звільнятися.
*Динамічні структури даних*є зв'язковими. Зв'язок - особливий продуманий логічний пристрій збереження цілісності структури даних, елементи якої можуть перебувати в довільних, несуміжних, неконтрольованих за адресацією ділянках пам'яті, що динамічно розподіляється, поза кодом програми.
Динамічне зв'язування— асоціація запиту з об'єктом та однією з його операцій під час виконання.
*Доказ (proof)*спроби знайти у програмі помилки шляхом доказів на основі математичних теорем про правильність програми, безвідносно до зовнішнього програмного середовища.
Документ -документ, виконаний за заданою формою, в якому подано якесь проектне рішення.
ДПД -Діаграма потоків даних.
*Єдина система програмної документації (ЄСПД)*комплекс державних стандартів, що встановлює взаємопов'язані правила розробки, оформлення та звернення програм та програмної документації.
ЕСПД -Єдина система програмної документації.
*Життєвий цикл*с укупність взаємозалежних процесів створення та послідовної зміни стану продукції від формування до неї вихідних вимог до закінчення її експлуатації чи споживання.
Заглушка макет ще не реалізованого модуля, необхідний при низхідній реалізації, є найпростішою підпрограмою або без дій, або з діями виведення вхідних даних, або повертає у вищестоящі модулі тестові дані (які зазвичай присвоюються всередині заглушки), або містить комбінацію цих дій.
Ієрархія підпорядкованість.
Мінливість структури даних -зміна числа елементів та (або) зв'язків між елементами структури. У визначенні мінливості структури не відображено факт зміни значень елементів даних, оскільки у разі всі структури даних мали властивість мінливості. За ознакою мінливості розрізняють структури: статичні структури даних і динамічні структури даних.

Інженер(Від лат. ingenium - природний розум, винахідливість) - спеціаліст зі створення штучних систем.
Інженерія програмування(англ. software engineering, у термінах автоматизованих систем – розробка програмного забезпечення) – інженерна справа, творча технічна діяльність. Інженерія спирається на специфічні методи та методики, у тому числі евристичні. Інженерія вивчає різні методи та інструментальні засоби з погляду певних цілей, тобто має очевидну практичну спрямованість. Основна ідея інженерії програмування полягає в тому, що розробка програмного забезпечення є формальним процесом, який можна вивчати, виражати в методиках та вдосконалювати. Головна різниця між технологією програмування та програмною інженерією полягає у способі розгляду та систематизації матеріалу. У технології програмування акцент робиться на вивченні процесів розробки програм (технологічних процесів) у порядку їх проходження – методи та інструментальні засоби розробки програм використовуються у цих процесах (їх застосування та утворюють технологічні процеси). У програмній інженерії вивчаються, насамперед, методи та інструментальні засоби розробки програм з погляду досягнення певних цілей — вони можуть використовуватися у різних технологічних процесах (і різних технологіях програмування). Як ці методи та засоби утворюють технологічні процеси — питання другорядне.
Інженерний технологічний підхід визначається специфікою комбінації стадій розробки, етапів та видів робіт, орієнтованої на різні класи програмного забезпечення та на особливості колективу розробників.
Інженер-програміст найменування посади згідно з кваліфікаційним довідником посад керівників, спеціалістів та інших службовців, спеціаліст зі створення та експлуатації програм.
Інженер-системотехнік найменування посади згідно з кваліфікаційним довідником посад керівників, спеціалістів та інших службовців, інженер інженерів, спеціаліст з вирішення проектних завдань створення таких особливо складних штучних систем, як автоматизовані системи.
Інкапсуляція -це механізм суміщення у класі полів даних із методами, які маніпулюють захищеними полями даних.
Інтегрування структури даних -структури даних, складовими яких є інші структури даних — прості чи своє чергу інтегровані. Інтегровані структури даних конструюються програмістом з допомогою засобів інтеграції даних, наданих мовами програмування.
Інтерфейс -це набір форматів допустимих повідомлень. Для унеможливлення можливих, але неприпустимих повідомлень використовується механізм приховування інформації.
Випробування (validation)- Спроба знайти помилки, виконуючи програму в заданому програмному середовищі.
*Каркасні інженерні підходи*є каркас для видів робіт і включають їх величезну кількість. Яскравим представником каркасного підходу є раціональний уніфікований підхід до виконання (rational unified process). Вагомою перевагою даного підходу є створення інструментарію його автоматизованої підтримки — програмного продукту Rational Rose фірми Rational Software Corporation.
*Каскадні інженерні технологічні підходи*задають деяку послідовність виконання видів робіт, що зазвичай зображується у вигляді каскаду. Ці підходи іноді називають підходами на основі моделі водоспаду.
Кодування-виправлення (code and fix)- Інженерно-технологічний підхід, спрощено може бути описаний наступним чином. Розробник починає кодування системи з самого першого дня, не займаючись серйозним проектуванням.
Кодувальник програм- Програміст, що пише і автономно тестує код компонент програм.
Комплексне тестування (system testing)- Контроль та/або випробування системи по відношенню до вихідних цілей. Є процесом контролю, якщо воно виконується в моделюється середовищі, і процесом випробування при виконанні в реальному середовищі.

Композиція об'єктів- Це реалізація складового об'єкта, що складається з декількох спільно працюючих об'єктів і утворюють єдине ціле з новою, складнішою функціональністю.
Компонентний аналіз- Розгляд об'єкта, що включає в себе складові елементи і входить, у свою чергу, в систему вищого рангу.
Конструктор -особливий метод класу, призначений до створення екземпляра об'єкта.
Контейнер-менеджер, або контейнер, - клас, що дозволяє об'єднувати (агрегувати) у собі різні класи об'єктів, у тому числі й інші контейнери.
Контроль (verification) -спроба знайти помилки, виконуючи програму в тестовому чи моделюваному середовищі.
*Коректність програмного забезпечення*властивість безпомилкової реалізації необхідного алгоритму за відсутності таких факторів, що заважають, як помилки вхідних даних, помилки операторів ЕОМ (людей), збоїв і відмов ЕОМ.
Критерій- Показник якості.
Логічна структура даних- Розгляд структури даних без урахування її подання в машинній пам'яті.
ЛПР -особа, яка приймає рішення.
Метод -спосіб практичного здійснення чогось.
Методика -сукупність методів практичного виконання чогось.
Методологія (Від грец. metnhodos і logos - слово, вчення про методи) - система принципів і способів організації та побудови теоретичної та практичної діяльності, а також вчення про цю систему.
Методологія програмування вивчає методи з погляду основ побудови. Це об'єднана єдиним філософським підходом сукупність методів, що застосовуються у процесі розробки програмних продуктів. Будь-яка методологія створюється на основі вже накопичених у предметній галузі емпіричних фактів та практичних результатів.
Метод мозкового штурму метод синтезу варіантів систем, що використовує взаємну стимуляцію мислення групи.
Метод морфологічних таблиць— згідно з цим методом, для об'єкта, що цікавить нас, формується набір відмітних ознак: найбільш характерних підсистем, властивостей або функцій. Потім кожного з них визначаються альтернативні варіанти реалізації. Комбінуючи альтернативні варіанти, можна отримати багато різних рішень. Аналізуючи їх, виділяють кращі варіанти.
Метод проб і помилок метод синтезу варіантів систем, заснований на послідовному висуванні та розгляді ідей.
Метод евристичних прийомів метод синтезу варіантів систем, що базується на виділенні базових прийомів, знайдених при аналізі кращих програмних виробів.
Методи об'єкта (methods, member functions) -підпрограми, що реалізують дії (виконання алгоритмів) у відповідь на їхній виклик у вигляді переданого повідомлення.
Механізм приховування інформації— механізм, який використовується для унеможливлення можливих, але неприпустимих повідомлень об'єктам.
Множинне успадкування класів успадкування, у якому кожен клас може, у принципі, породжуватися від однієї чи відразу від кількох батьківських класів, наслідуючи поведінка всіх своїх предків.
Модель один об'єкт чи система може у ролі моделі іншого об'єкта чи системи, якщо з-поміж них встановлено схожість у сенсі.
Модуль -фундаментальне поняття та функціональний елемент технології структурного програмування, підпрограма, але оформлена відповідно до особливих правил.
Модуль -у технології об'єктно-орієнтованого програмування це файл (unit) із описами родинних класів.
Модульність програм- Основний принцип технології структурного програмування, характеризується тим, що вся програма складається з модулів.

Спадкування визначення класу і потім використання його для побудови ієрархії класів-нащадків, причому кожен нащадок успадковує доступ до коду та даних всіх своїх класів прабатьків.
*Науково-дослідна робота (НДР)*самостійний етап, проведений виявлення останніх наукових досягнень з метою їх використання у проекті, перевірки реалізованості вироби та уточнення окремих його характеристик.
НДР -науково-дослідницька робота.
Східне проектування один із головних принципів технології структурного програмування, згідно з яким при розробці ієрархії модулів програм виділяються спочатку модулі найвищого рівня ієрархії, а потім підлеглі модулі.
Низхідна реалізація програми- у технології структурного програмування первинна реалізація групи модулів верхніх рівнів, які називаються ядром програми, і далі поступово, відповідно до плану, реалізуються модулі нижніх рівнів. Необхідні для лінкування програми, відсутні модулі імітуються заглушками.
Узагальнення -виявлення групи класів загальних властивостей і винесення в загальний базовий клас.
Об'єкт логічна одиниця, що містить всю інформацію про деякий фізичний предмет або поняття, що реалізується в програмі, структурована змінна типу клас, яка містить поля даних і методи з кодом алгоритму.
Об'єктна модель- Модель, що описує структуру об'єктів, що складають систему, їх атрибути, операції, взаємозв'язки з іншими об'єктами. В об'єктній моделі повинні бути відображені ті поняття та об'єкти реального світу, які важливі для системи, що розробляється.
*Об'єктно-орієнтоване програмування (ООПр) (object-oriented programming)*це процес реалізації програм, заснований на представленні програми у вигляді сукупності об'єктів.
Об'єктно-орієнтоване проектування (ООР) (object-oriented design, OOD)— методологія проектування, що поєднує в собі процес об'єктної декомпозиції та прийоми подання логічної та фізичної, а також статичної та динамічної моделей проектованої системи.
Об'єктно-орієнтований аналіз (ООО) (object-oriented analysis) -методологія, за якої вимоги до системи сприймаються з погляду класів та об'єктів, прагматично виявлених у предметній галузі.
*Операції над структурами даних*над усіма структурами даних може виконуватися п'ять операцій: створення, знищення, вибір (доступ), оновлення, копіювання.
Операційний підхід до складання алгоритмів- згідно з цим підходом, операції (алгоритмічні дії) виділяються послідовно по ходу шляху обчислень при якихось наборах даних.
Оптимізація розробки програм— знаходження розумного компромісу між метою, що досягається, і витраченими на це ресурсами.
Організованість даних- Продуманий пристрій з метою раціонального використання за призначенням.
ОС -операційна система.
*Налагодження (debugging)*перестав бути різновидом тестування, а є засобом встановлення точної природи помилок.
Параметричний аналіз встановлення якісних меж розвитку об'єкта: фізичних, економічних, екологічних та інших. Що стосується програм параметрами може бути: час виконання якого-небудь алгоритму, розмір пам'яті і т.д.
Паспорт модуля внутрішній документ проекту, який зазвичай є конвертом з ім'ям модуля. Усередині конверта містяться описи прототипу виклику самого модуля та модулів, що викликаються даним модулем; розшифровка вхідних та вихідних змінних модуля; опис функції, що виконується модулем; принципи реалізації алгоритму модуля із описом основних структур даних.
Паттерн проектування це зразок, типове вирішення будь-якого механізму об'єктно-орієнтованої програми.

Планування на всіх стадіях проекту Основний принцип проектування дозволяє спочатку спланувати як склад стадій, так і тривалість всіх етапів робіт. Таке планування дозволяє завершити розробку в заданий термін за заданих витрат на розробку. Далі планується порядок і час інтеграції модулів у ядро, що все розширюється. Плануються заходи щодо тестування програми від ранніх до заключних етапів.
ПЗ -програмне забезпечення автоматизованих систем
Повторне використання -це використання в програмі класу для створення екземплярів або як базовий для створення нового класу, що успадковує частину або всі характеристики батька. Повторне використання скорочує обсяг коду, який необхідно написати та відтестувати під час реалізації програми, що скорочує обсяги праці.
*Підпрограма*деяка послідовність інструкцій, яка може викликатись у кількох місцях програми; програмна одиниця, компилируемая незалежно з інших елементів програми. В об'єктно-орієнтованому програмуванні відповідає методу.
Пізніше зв'язування— зв'язки між об'єктами визначаються динамічно під час виконання програми, процес зв'язування полягає у заміні адрес пам'яті віртуальних функцій.
*Показники якості (критерії)*величини, властивості, поняття, що характеризують систему з погляду суб'єкта, що дозволяють оцінити рівень задоволення його потреб.
*Поле об'єкту (data members)*порція даних об'єкта, значення яких визначають стан об'єкта.
*Поліморфізм*це засіб надання різних значень одному й тому події залежно від типу оброблюваних даних, т. е. поліморфізм визначає різні форми реалізації однойменного действия.
Порт -програмний механізм накопичення та верифікації як вхідних, так і вихідних даних у відповідних чергах.
*Нащадок*клас, використовуваний характеристики іншого класу у вигляді успадкування.
Предок- Це клас, що надає свої можливості та характеристики іншим класам.
*Програмна документація*єдина система програмної документації (ЄСПД).
Програмний документ -документ, що містить відомості, необхідні для розробки, виготовлення, експлуатації та супроводу програмного виробу.
*Програмний продукт*програма, яку можна запускати, тестувати, виправляти та розвивати. Така програма має бути написана в єдиному стилі, ретельно відтестована до необхідного рівня надійності, супроводжена докладною документацією та підготовлена для тиражування. Стандартний термін – програмний виріб.
Програмний виріб- Програма на носії даних, що є продуктом промислового виробництва.
Програмне забезпечення автоматизованих систем (ПЗ)- Сукупність програм на носіях даних та програмних документів, призначена для налагодження, функціонування та перевірки працездатності автоматизованих систем.
Проект(Від лат. Projectus - кинутий вперед) - сукупність проектних документів відповідно до встановленого переліку, що представляє результат проектування.
Проектування— це розробка проекту, процес створення специфікації, яка потрібна на побудови в заданих умовах ще неіснуючого об'єкта з урахуванням первинного описи цього об'єкта. Результатом проектування є проектне рішення чи сукупність проектних рішень, які відповідають заданим вимогам. Задані вимоги обов'язково мають включати форму подання рішення.
Проектне завдання(англ. engineering Task) - характеризується невизначеністю апріорі інформації: що потрібно отримати, що поставлено. Більш того, спосіб розв'язання задачі є об'єктом проектування. І нарешті, рішення проектної завдання має бути знайдено у межах обмежень довкілля: доступних коштів, заздалегідь заданих термінів, можливостями технічних засобів та інструментарію програмування, наукових знань програмних заділів тощо.
Проектна операція- Дія або формалізована сукупність дій, що становлять частину проектної процедури, алгоритм яких залишається незмінним для низки проектних процедур, наприклад креслення схеми, диференціювання функції.

*Проектна процедура (методика), складання функціональних описів.*методика розробки описів функціонування систем, що відрізняється використанням особливого структурування. Інструкція використання будь-яким пристроєм, інструкція взагалі або алгоритм програми є описами функціонування.
Проектна ситуація реальність (ситуація), у якій ведеться проектування.
Проектне рішення -опис у заданій формі об'єкта проектування або його частини, необхідний та достатній для визначення подальшого напряму проектування.
Проектний документ -документ, виконаний за заданою формою, в якому подано якесь проектне рішення. У програмуванні проектні рішення оформлюються як програмної документації. Розрізняють зовнішню програмну документацію, яка узгоджується із замовником, та внутрішню проміжну документацію проекту, яка необхідна самим програмістам для їхньої роботи.
Просте успадкування класів успадкування, при якому певний користувачем клас має лише одного з батьків. Схема ієрархії класів у цьому випадку є рядом одиночних дерев (hierarchical classification).
*Прості структури даних*не можуть бути розчленовані на складові, більші, ніж біти і байти. З погляду фізичної структури важливим є та обставина, що у цій машинної архітектурі, у цій системі програмування ми можемо заздалегідь сказати, який буде розмір даного простого типу і яка структура його розміщення у пам'яті. З логічного погляду прості дані є неподільними одиницями. У мовах програмування прості структури описуються простими (базовими) типами. Прості структури даних є основою для побудови складніших інтегрованих структур.
Професійний програміст це фахівець, який має інтегральну особистісну характеристику людини: домагається майстерності у програмуванні, слідує професійній ціннісній орієнтації, дотримується професійної етики, володіє мистецтвом спілкування з людьми, прагне і вміє викликати інтерес суспільства до результатів своєї професійної діяльності.
Робочий проект (РП)- Найменування стадії та програмний документ, що містить опис реалізованого виробу.
*Раннє зв'язування*зв'язки між об'єктами визначаються статично під час компіляції.
Резидентна програма не видалена ОС програма, постійно що у оперативної пам'яті ЕОМ.
Батько- Безпосередній клас-предок, що стоїть біля кореня схеми ієрархії, і від якого народжуються перші нащадки, а від нащадків ще нащадки.
Батьківський клас початковий клас, від якого успадковуються класи-нащадки.
РП- Робочий проект.
САПР -система автоматизованого проектування
Властивості (property)- це особливим чином оформлені методи, призначені як для читання та контрольованої зміни внутрішніх даних об'єкта (полів), так і для виконання дій, пов'язаних з поведінкою об'єкта.
Сесія програмістів- зустріч кодувальників для проведення взаємної інспекції текстів програм та набору використаних тестів.
Синтез (Від грецьк. synthesis - поєднання, поєднання, складання) - метод наукового дослідження явищ дійсності в їх єдності та цілісності, у взаємодії їх частин, узагальнення, зведення в єдине ціле. Теоретично проектування синтез — це процес побудови описи системи по заданому функціонуванню.
Система -безліч елементів, що у відносинах і зв'язках друг з одним, що утворює певну цілісність, єдність.
Системний аналітик— програміст, який розробляє проект від вимог до внутрішньої структури програми та бере участь у тестуванні як із інтеграції компонентів у ядро, і у комплексному тестуванні ПЗ.
Системний підхід- загальнонауковий узагальнений евроритм, що передбачає всебічне дослідження складного об'єкта з використанням компонентного, структурного, функціонального, параметричного та генетичного видів аналізу.

Наскрізний структурний контроль- Використання на багатьох етапах проекту контролю коректності специфікації зв'язків частин програми.
Злиття- Об'єднання кількох невеликих, але тісно взаємодіючих класів в один.
Супровід- діяльність з надання послуг, необхідних для забезпечення сталого функціонування або розвитку програмного виробу, включає аналіз функціонування, розвиток та вдосконалення програми, а також внесення змін до неї з метою усунення помилок.
Специфікація— у сфері проектної діяльності це якийсь опис у точних термінах.
Стадія проекту- Одна з частин процесу створення програми, встановлена нормативними документами і закінчується випуском проектної документації, що містить опис повної, в рамках заданих вимог моделі програми на заданому для даної стадії рівні, або виготовлення програм. Після досягнення стадії замовник має можливість розглянути стан проекту та прийняти рішення щодо подальшого продовження проектних робіт.
Стратегія (від грец. Stratos - військо і ago - веду) - наука, мистецтво генерації найбільш істотних загальних довгострокових цілей і найбільш загального плану досягнення переваги, курсу дій та розподілу ресурсів ще до виконання реальних дій. Стратегія охоплює теорію і практику підготовки до виконання проекту, а також загальне планування тактик ведення проектів. Стратегія визначає, куди, в якому напрямку рухатись, куди тримати курс ще до початку проекту. А тактика визначає, як, яким способом рухатися, які конкретні дії робити при труднощі під час виконання проекту.
Структура програми -штучно виділені програмістом взаємодіючі частини програми.
Структура даних програми- безліч елементів даних, безліч зв'язків між ними, а також характер їхньої організованості.
Структурне кодування модулів програм- Основний принцип технології структурного програмування, сприйнятий технологією об'єктно-орієнтованого програмування, який полягає в особливому оформленні текстів модулів (методів). У модуля має бути легко помітний заголовок з коментарем, що пояснює функціональне призначення модуля. Імена змінних мають бути мнемонічними. Суть змінних та порядок розміщення в них інформації мають бути пояснені коментарями, а код закодований з використанням типових алгоритмічних структур.
Структурний аналіз- Виявлення елементів об'єкта і зв'язків між ними.
Структурний підхід набір принципів, що характеризує технологію структурного програмування: модульність програм; структурне кодування модулів програм; низхідне проектування раціональної ієрархії модулів програм; низхідна реалізація програми з використанням заглушок; здійснення планування на всіх стадіях проекту; наскрізний структурний контроль програмних комплексів загалом і модулів, що їх складають.
СУБД -система керування базами даних.
Схема ієрархії програми використовується в технології структурного програмування, відображає лише підпорядкованість модулів (підпрограм), але не порядок їхнього виклику або функціонування програми.
Сценарій послідовність подій, яка може бути при конкретному виконанні системи.
Сценарій діалогу програми- Послідовність введення та виведення інформації в діалоговому режимі роботи програми.
Тактика (від грецьк. taktika — мистецтво упорядковувати) — фіксована у своїй послідовності сукупність засобів і прийомів задля досягнення наміченої мети і мистецтво її застосування, способи дії, зорієнтовані досягнення конкретних цілей, є ланками реалізації стратегічних цілей. Метою застосування способу дії є здійснення оптимальних дій у заздалегідь не передбачених стратегічним планом ситуаціях у процесі виконання реальних дій.
*Тестування (testing)*процес виконання програми із наміром знайти помилки; може здійснюватися як з ЕОМ, і без ЕОМ.
Тестування прийнятності (acceptance testing) -перевірка відповідності програми до вимог користувача.

Тестування пар (integration testing)- Контроль сполучень між частинами системи як між компонентами в комплексі, так і між модулями окремого компонента (наприклад, у заглушки).
Тестувальник— програміст, який готує набори тестів для налагодження програмного виробу, що розробляється.
Технічний проект (ТП)- Комплект проектних документів на програму, що розробляється на стадії «Технічний проект», затверджений в установленому порядку, що містить основні проектні рішення за програмою в цілому, її функцій і достатній для розробки робочого проекту.
Технічне завдання (ТЗ)— документ, оформлений в установленому порядку та визначальний мети створення програми, вимоги до програми та основні вихідні дані, необхідні для її розробки, а також план-графік створення програми.
Технологія (від грец. techne - мистецтво, майстерність, уміння і logos - слово, вчення) - сукупність виробничих процесів у певній галузі виробництва, а також науковий опис способів виробництва, сукупність прийомів, що застосовуються у будь-якій справі, майстерності, мистецтві. Сучасна методологія проектування дозволила довести методи проектування до технологій із набором методик.
Технологія візуального програмування— популярна інженерія програмування, яка полягає у автоматизованій розробці програм із використанням особливої діалогової оболонки.
Технологія об'єктно-орієнтованого програмування— технологія, орієнтована отримання програм, які з об'єктів.
Технологія програмування як наука вивчає технологічні процеси та порядок їх проходження (з використанням знань, методів та засобів). p align="justify"> Технологічний процес - послідовність спрямованих на створення заданого об'єкта дій (технологічних процедур і операцій), кожна з яких заснована на будь-яких природних процесах і людської діяльності. Звернемо увагу на те, що знання, методи та засоби можуть використовуватись у різних процесах і, отже, у технологіях. Технологія програмування для інженера - це наукова та практично апробована стратегія створення програм, що містить опис сукупності методів та засобів розробки програм, а також порядок застосування цих методів та засобів.
Технологія програмування Дейкстри, заснована на абстракції даних- У цій технології на чолі ставляться дані; спочатку дуже ретельно специфікується вихід, вхід, проміжні дані; велика увага приділяється типізації даних з використанням структур для об'єднання близької за змістом інформації у єдині дані.
Технологія структурного програмування- Технологія, заснована на структурному підході.
Типізація -це спосіб захиститися від використання об'єктів одного класу замість іншого або принаймні керувати таким використанням. Типізація змушує нас висловлювати наші абстракції так, щоб мова програмування, що використовується в реалізації, підтримувала дотримання прийнятих проектних рішень.
ТП- Технічний проект.
Управління розробкою програмних систем (software management) - діяльність, спрямована на забезпечення необхідних умов для роботи колективу розробників програмного забезпечення, планування та контроль діяльності цього колективу з метою забезпечення необхідної якості програмного забезпечення, виконання термінів та бюджету розробки програмного забезпечення.
Стійкість програмного забезпечення властивість здійснювати необхідне перетворення інформації за збереження вихідних рішень програми у межах допусків, встановлених специфікацією під впливом програми таких чинників нестійкості, як помилки операторів ЕОМ, і навіть не виявлених помилок програми.
Фізична структура даних спосіб фізичного представлення даних у пам'яті машини і називається ще структурою зберігання, внутрішньою структурою, структурою пам'яті або дампом.
Форма- Візуальний компонент, що володіє властивістю вікна Windows.

Функція системи- Сукупність дій системи, спрямована на досягнення певної мети.
Функціональний аналіз -розгляд об'єкта як комплексу виконуваних ним функцій.
Евристика— наука, яка розкриває природу розумових операцій людини під час вирішення конкретних завдань незалежно від своїх конкретного змісту. У вужчому сенсі, евристика — це припущення, засновані на досвіді розв'язання родинних завдань.
Євроритм -порядок дії людини під час виконання якоїсь діяльності. На відміну від алгоритму може змінюватися у процесі виконання завдяки розумності виконавця.
Примірник класу- Об'єкт.
Експлуатаційна документація- частина робочої документації на програму, призначена для використання при експлуатації програми та визначає правила дії персоналу та користувачів програми при її функціонуванні, перевірці та забезпеченні її працездатності.
Екстремальне програмування (extreme programming)(XP) - адаптивний інженерний підхід, раціональне об'єднання відомих методів та їх сукупне використання дає суттєві результати та успішно виконані проекти при розробці невеликих систем, вимоги до яких чітко не визначені та цілком можуть змінитися.
*Ескізний проект (ЕЛ)*комплект проектних документів на програму, що розробляється на стадії «Ескізний проект», затверджений у встановленому порядку, що містить опис кількох альтернативних варіантів реалізації майбутнього виробу та уточнені вимоги на основі їх аналізу. Ступінь опрацювання при цьому має бути достатньою лише для досягнення можливості порівняння варіантів.
ЕТ -електронної таблиці.
Етап проекту- Частина стадії проекту, виділена з міркувань єдності характеру робіт і (або) завершального результату або спеціалізації виконавців.
Ядро -всезростаюча вже реалізована частина програми.
CASE-кошти— це програмні засоби, що підтримують процеси створення та супроводу програмних продуктів, включаючи аналіз та формулювання вимог, проектування продукту та баз даних, генерацію коду, тестування, документування, забезпечення якості, конфігураційне управління та управління проектом, а також інші процеси. CASE-засоби разом із системним програмним забезпеченням та технічними засобами утворюють повне середовище розробки програмних систем.
CASE-технологія (Computer Aided Software Engineering)— технологія, що є методологією проектування АС, а також набір інструментальних засобів, що дозволяють у наочній формі моделювати предметну область, аналізувати цю модель на всіх етапах розробки та супроводу програмних систем та розробляти програми відповідно до інформаційних потреб користувачів.
COM -Component Object Model.
Component Object Model(модель компонентних об'єктів) - специфікація методу створення компонентів та побудови з них програм.
CRC-картка (Component, Responsibility, Collaborator – об'єкт, обов'язки, співробітники) – проміжний документ проекту, необхідний специфікації об'єктів.
DFD - ДПД (Data Flow diagramm).
RDD-проектування (Responsibility-Driven-Design) – технологія проектування на основі обов'язків, запропонована Т. Бадтом. Ця технологія за способом мислення аналогічна розробці структури служб якоїсь організації: директора, заступників директора, служб та підрозділів.

ЛІТЕРАТУРА

1. Авраменко В. С., Авраменко А. С. Проектування інформаційних систем : навч. посіб. Черкаси : Черкаський нац. ун-т ім. Б. Хмельницького, 2017. 434 с. URL: http://files.znu.edu.ua/files/Bibliobooks/Inshi72/0053479.pdf.
2. Алексенко О. В. Технології програмування та створення програмних продуктів : конспект лекцій. Суми : Сумський держ. ун-т, 2013. 133 с. URL: http://files.znu.edu.ua/files/Bibliobooks/Inshi78/0058335.pdf.
3. Барішенко О. М. Об'єктно-орієнтоване програмування : навч.-метод. посіб. Запоріжжя : ЗДІА, 2011. 110 с. URL: http://ebooks.znu.edu.ua/files/ZII/metodychky/do2018/f348226.doc.
4. Безменов М. І. Основи програмування у середовищі Delphi : навч. посіб. Харків : НТУ "ХПІ", 2010. 608 с. URL: http://files.znu.edu.ua/files/Bibliobooks/Inshi78/0058610.pdf.
5. Бичков О. С., Іванов Є. В. Об'єктно-орієнтоване програмування мовою C# : підручник. Київ : Київський університет, 2018. 208 с.
6. Бородкіна І. Л., Бородкін Г. О. Інженерія програмного забезпечення : навч. посіб. Київ : Центр учбової літератури, 2019. 206 с.
7. Бородкіна І. Л., Бородкін Г. О. Інженерія програмного забезпечення : посібник. Київ, 2018. 251 с. URL: http://files.znu.edu.ua/files/Bibliobooks/Inshi78/0058333.pdf.
8. Бублик В. В. Об'єктно-орієнтоване програмування : підручник. Київ : ІТ-книга, 2015. 640 с. URL: http://files.znu.edu.ua/files/Bibliobooks/Inshi73/0053670.pdf.
9. Буч Г. Объектно-ориентированный анализ и проектирование с примерами приложений на C++ / пер. с англ., под ред. И. Романовского и Ф. Андреева. 2-е изд. Калифорния : Rational Санта-Клара. 359 с. URL: http://ebooks.znu.edu.ua/files/Bibliobooks/Inshi40/0031552.pdf.
10. Вакалюк Т. А., Шевчук Л. Д., Постова С. А. Структурне та візуальне програмування : навч. посіб. Переяслав-Хмельницький : ПХДПУ, 2019. 318 с. URL: http://files.znu.edu.ua/files/Bibliobooks/Inshi78/0057946.pdf.
11. Гришанович Т. О., Глинчук Л. Я. Основи об'єктно-орієнтованого програмування : навч. посіб. Луцьк : ВНУ імені Лесі Українки, 2022. 120 с. URL: http://files.znu.edu.ua/files/Bibliobooks/Inshi70/0051132.pdf.
12. Дібрівний О. А., Гребенюк В. В. Вступ до об'єктно орієнтованого програмування C# : навч. посіб. Київ : Державний університет телекомунікацій, 2018. 190 с. URL: http://files.znu.edu.ua/files/Bibliobooks/Inshi73/0053671.pdf.
13. Довгаль В. В. Технології об'єктно-орієнтованого візуального програмування : конспект лекцій. Запоріжжя : ЗДІА, 2017. 54 с. URL: http://ebooks.znu.edu.ua/files/ZII/metodychky/do2018/f358174.pdf.
14. Дудзяний І. М. Об'єктно-орієнтоване моделювання програмних систем : навч. посіб. Львів : Видавничий центр ЛНУ імені Івана Франка, 2007. 108 с. URL: http://files.znu.edu.ua/files/Bibliobooks/Inshi78/0058606.pdf.
15. Казимир В. В. Об'єктно-орієнтоване програмування : навч. посіб. Київ : Слово, 2008. 186 с.
16. Карпенко М. Ю., Манакова Н. О., Гавриленко І. О. Технології створення програмних продуктів та інформаційних систем : навч. посіб. Харків : ХНУМГ ім. О. М. Бекетова, 2017. 93 с. URL: http://files.znu.edu.ua/files/Bibliobooks/Inshi78/0058287.pdf.

17. Коновалов В. С., Радоуцький К. Є. Сучасні принципи і методи проектування програмного забезпечення : конспект лекцій з дисципліни " Системне програмування". Ч. 2. Харків : УкрДАЗТ, 2015. 109 с. URL: http://files.znu.edu.ua/files/Bibliobooks/Inshi78/0058622.pdf.

18. Коцовський В. М. Технологія програмування та створення програмних продуктів : метод. посіб. Ужгород : Говерла, 2016. 83 с. URL: http://files.znu.edu.ua/files/Bibliobooks/Inshi80/0059902.pdf.

19. Кравець П. О. Об'єктно-орієнтоване програмування : навч. посіб. / Нац. ун-т "Львів. політехніка". Львів : Вид-во Львів. політехніки, 2012. 622 с.

20. Кривцова О. П. Програмування мовою С++. Технологія візуального програмування : навч. посіб. Полтава : ПНПУ ім. В. Г. Короленка, 2020. 144 с. URL: http://files.znu.edu.ua/files/Bibliobooks/Inshi78/0057947.pdf.

21. Куліков В. М., Рябцев В. В., Паршуков С. С. Об'єктно-орієнтоване програмування для фахівців з кібербезпеки : навч. посіб. Київ : КПІ ім. Ігоря Сікорського, 2023. 365 с. URL: http://files.znu.edu.ua/files/Bibliobooks/Inshi78/0058602.pdf.

22. Кучеров Д. П., Артамонов Є. Б. Інженерія програмного забезпечення : навч. посіб. Київ : НАУ, 2017. 386 с. URL: http://files.znu.edu.ua/files/Bibliobooks/Inshi78/0058620.pdf.

23. Лавріщева К. М. Програмна інженерія : підручник. Київ : [б. в.], 2008. 319 с. URL: http://files.znu.edu.ua/files/Bibliobooks/Inshi70/0051182.pdf.

24. Лістровий С. В., Мірошник М. А., Клименко Л. А. Теорія автоматичного керування, штучний інтелект і автоматизація процесу прийняття рішення : навч. посіб. Харків : УкрДУЗТ, 2019. 120 с. URL: http://files.znu.edu.ua/files/Bibliobooks/Inshi78/0058597.pdf.

25. Львов М. С., Співаковський О. В. Вступ до об'єктно-орієнтованого програмування. Херсон : ХГПУ, 2000. 238 с. URL: http://files.znu.edu.ua/files/Bibliobooks/Inshi70/0051141.pdf.

26. Міхайлуца О. М., Скрипник І. А. Аналіз вимог до програмного забезпечення : навч.-метод. посіб. Запоріжжя : ЗНУ, 2022. 175 с. URL: http://files.znu.edu.ua/files/ZII/metodychky/2022/0052057.doc.

27. Муляр В. П. Об'єктно-орієнтоване програмування : конспект лекцій. Луцьк : Вежа-друк, 2022. 122 с. URL: http://files.znu.edu.ua/files/Bibliobooks/Inshi71/0052125.pdf.

28. Навчальний посібник з дисципліни "Технології розробки програмного забезпечення" для студентів спеціальності 123 "Комп'ютерна інженерія" / уклад.: Л. М. Дегтярьова, П. М. Гроза, С. В. Сомов. Полтава : ПолтНТУ, 2017. 218 с. URL: http://files.znu.edu.ua/files/Bibliobooks/Inshi78/0058363.pdf.

29. Настенко Д. В., Нестерко А. Б. Об'єктно-орієнтоване програмування : навч. посіб. Ч. 1 : Основи об'єктно-орієнтованого програмування на мові C# / НТУУ «КПІ». Київ : НТУУ «КПІ», 2016. 76 с. URL: http://files.znu.edu.ua/files/Bibliobooks/Inshi70/0051144.pdf.

30. Об'єктно орієнтоване програмування на Java : конспект лекцій з дисципліни «Об'єктно орієнтоване програмування» / уклад. П. Г. Бивойно. Чернігів : ЧНТУ, 2019. 136 с. URL: http://files.znu.edu.ua/files/Bibliobooks/Inshi78/0058609.pdf.

31. Омельчук Л. Л. Об'єктно-орієнтоване програмування. Лабораторний практикум : навч. посіб. Київ : Київський нац. ун-т імені Тараса Шевченка, 2021. 265 с. URL: http://files.znu.edu.ua/files/Bibliobooks/Inshi73/0053666.pdf.

32. Попівщий В. І. Об'єктно-орієнтоване програмування : навч.-метод. посіб. Ч. 1. Запоріжжя : ЗДІА, 2017. 192 с. URL: http://ebooks.znu.edu.ua/files/ZII/metodychky/do2018/f357875.pdf.

33. Попівщий В. І., Безверхий А. І., Лимаренко Ю. О. Об'єктно-орієнтоване програмування : навч.-метод. посіб. Ч. 2. Запоріжжя : ЗНУ, 2019. 178 с. URL: http://ebooks.znu.edu.ua/files/ZII/metodychky/2019/0044094.doc.

34. Порєв В. М. Об'єктно-орієнтоване програмування: конспект лекцій : навч. посіб. Київ : КПІ ім. Ігоря Сікорського, 2022. 271 с. URL: http://files.znu.edu.ua/files/Bibliobooks/Inshi78/0058099.pdf.

35. Решевська К. С., Лісняк А. О., Борю С. Ю. Об'єктно-орієнтоване програмування : навч. посіб. для здобувачів ступеня вищ. освіти бакалавра спец. "Комп'ютерні науки" освіт.-проф. програми "Комп'ютерні науки". Запоріжжя : ЗНУ, 2020. 94 с. URL: http://ebooks.znu.edu.ua/files/metodychky/2020/06/0045039.doc.

36. Рудий Т. В., Паранчук Я. С., Сеник В. В. Алгоритмізація та програмування : навч. посіб. Ч. 2 : Модульне програмування. Львів : ЛьвДУВС, 2024. 176 с. URL: http://files.znu.edu.ua/files/Bibliobooks/Inshi79/0059145.pdf.

37. Рудий Т. В., Паранчук Я. С., Сеник В. В. Алгоритмізація та програмування : навч. посіб. Ч. 1 : Структурне програмування. Львів : ЛьвДУВС, 2023. 240 с. URL: http://files.znu.edu.ua/files/Bibliobooks/Inshi75/0055629.pdf.

38. Савченко А. С., Синельніков О. О. Методи та системи штучного інтелекту : навч. посіб. для студентів напряму підготовки 6.050101 "Комп'ютерні науки" / НАУ. Київ : НАУ, 2017. 190 с. URL: http://files.znu.edu.ua/files/Bibliobooks/Inshi70/0051184.pdf.

39. Скалозуб В. В., Ільман В. М., Івченко Ю. М., Андрющенко В. О. Дискретні та алгоритмічні структури в інструментарії програмної інженерії : навч. посіб. / Дніпропетр. нац. ун-т залізн. трансп. ім. акад. В. Лазаряна. Дніпропетровськ : [б. в.], 2016. 254 с. URL: http://files.znu.edu.ua/files/Bibliobooks/Inshi70/0051135.pdf.

40. Табунщик Г. В., Каплієнко Т. І., Петрова О. А. Проектування та моделювання програмного забезпечення сучасних інформаційних систем : навч. посіб. Запоріжжя : Дике Поле, 2016. 250 с. URL: http://files.znu.edu.ua/files/Bibliobooks/Inshi72/0053480.pdf.

41. Цибульник С. О., Барандич К. С. Технології розроблення програмного забезпечення : підручник. Ч. 1 : Життєвий цикл програмного забезпечення. Київ : КПІ ім. Ігоря Сікорського, 2022. 270 с. URL: http://files.znu.edu.ua/files/Bibliobooks/Inshi72/0053481.pdf.

42. Щербаков О. В., Парфьонов Ю. В., Федорченко В. М. Основи об'єктно-орієнтованого програмування : навч. посіб. Харків : ХНЕУ ім. С. Кузнеця, 2019. 237 с. URL: http://files.znu.edu.ua/files/Bibliobooks/Inshi70/0051149.pdf.

43. Яковлєва С. О. Аналіз вимог до програмного забезпечення : навч.-метод. посіб. Запоріжжя : ЗНУ, 2014. 84 с. URL: http://files.znu.edu.ua/files/ZII/metodychky/do2018/0053885.doc.

44. Bocharov B. P. Scripting programming languages : tutorial. Kharkiv : O. M. Beketov NUUE, 2021. 109 p. URL: http://ebooks.znu.edu.ua/files/Bibliobooks/Inshi67/0049248.pdf.

45. Lee K. D. Foundations of Programming Languages. Cham : Springer, 2017. 370 p. URL: http://ebooks.znu.edu.ua/files/Bibliobooks/Inshi66/0048328.pdf.

46. Sundnes J. Introduction to Scientific Programming with Python. Cham : Springer, 2020. 148 p. URL: http://ebooks.znu.edu.ua/files/Bibliobooks/Inshi61/0045557.pdf.

47. Wazlawick R. S. Object-Oriented Analysis and Design for Information Systems : Agile Modeling with BPMN, OCL, IFML, and Python. 2nd ed. Cambridge : Morgan Kaufmann, 2024. 391 p. URL: http://files.znu.edu.ua/files/Bibliobooks/Inshi78/0057855/.