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

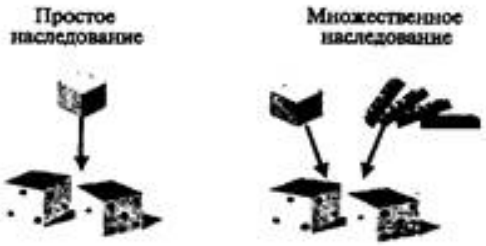
те ж об'єкт «продавець магазину», у нього може не виявитися відповідного методу для вирішення поставленого завдання. Якщо припустити, що об'єкт «продавець магазину» взагалі сприйме цей запит, він «видасть» належне повідомлення про помилку. На відміну від програм, люди працюють не за алгоритмами, а за евроритмами. 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'єкт*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;

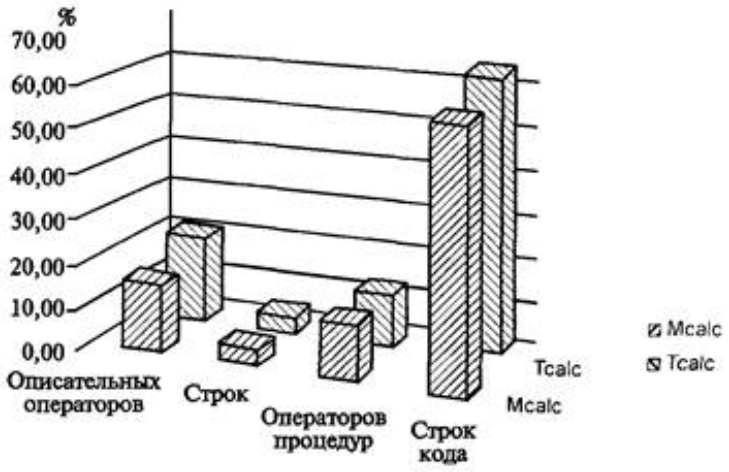
mcmvsmem.asm - асемблерний файл підпрограм запам'ятовування в оперативній пам'яті інформації екрана, а також відновлення раніше збереженої інформації екрана.
Усі файли закодовані з дотриманням стандартів оформлення.
Хоча фірма Borland Inc. займається розробкою компіляторів, файл mcparser.pas також є 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 QOA 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 B немає A, то B не варто виробляти від A.
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. Пославшись 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-вивода);
• 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 А, повинна обслуговуватися об'єктом класу В. 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

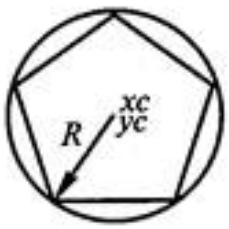
Намагаючись провести класифікацію деяких нових об'єктів, ставимо такі питання: У чому схожість цього об'єкта з іншими об'єктами загального класу? У чому його розходження? Кожен клас має набір поведінок та характеристик, що його визначають. Почнемо з верхівки фамільного дерева зразка і спускатимемося 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, x-координата, y-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, x-координата, y-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*)

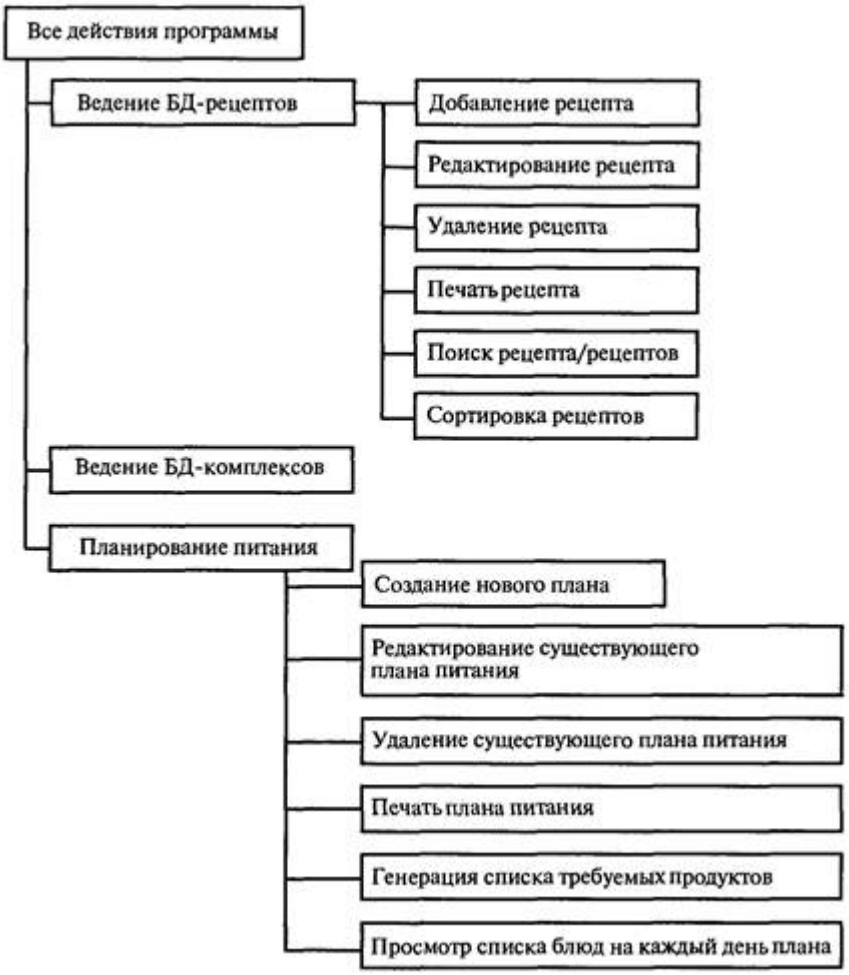
*Мал. 8.6.*Зразок CRC-картки

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.

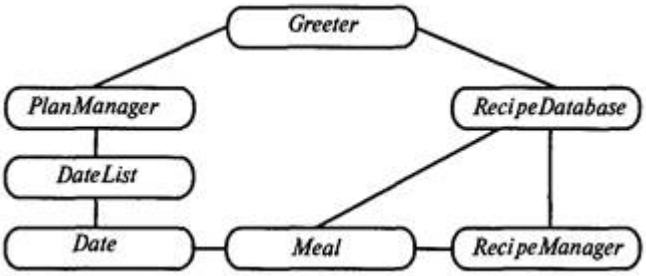


*Мал. 8.7.*Детальний опис функцій програми

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.



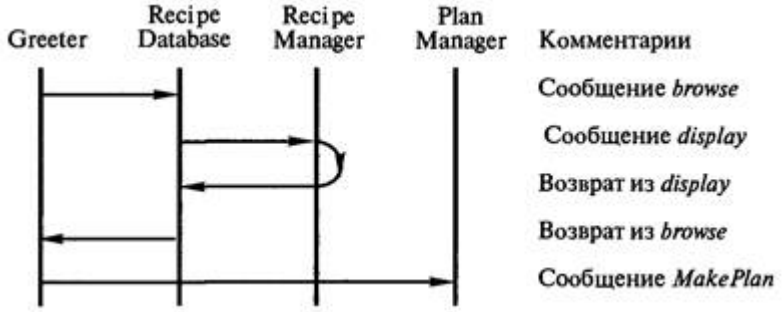
*Мал. 8.8.*Схема статичних зв'язків між об'єктами програми РКП

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

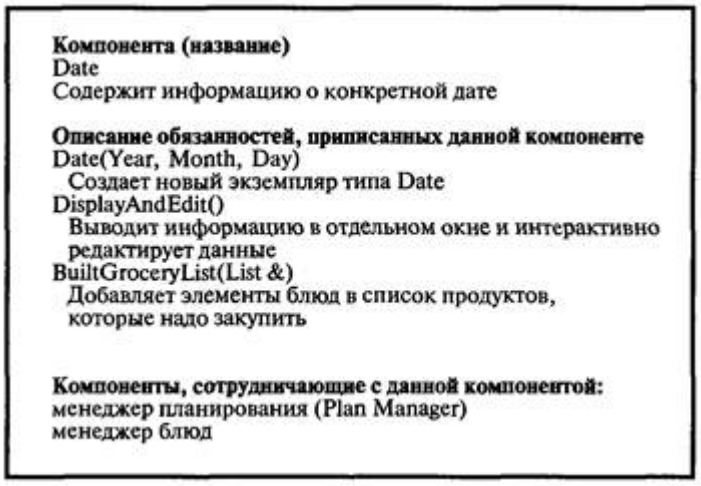
Діаграма станів пов'язує події та стани. При прийомі події наступний стан системи залежить від її поточного стану, і від події (рис. 8.10). Зміна стану називається переходом. Діаграма станів - це граф, вузли якого представляють стани, а спрямовані дуги, позначені іменами відповідних подій - переходи. 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).

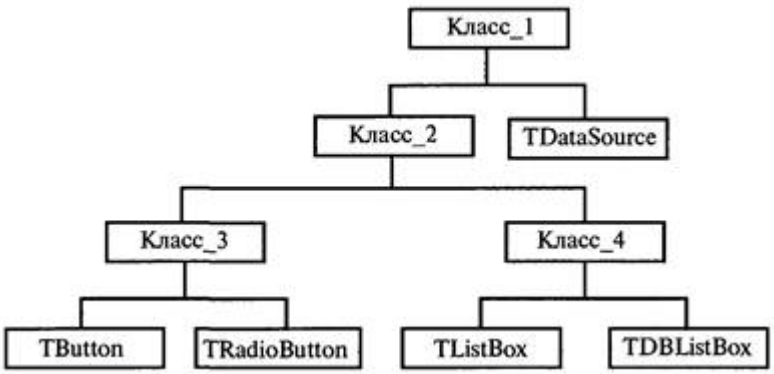
В результаті отримана ієрархія класів (рис. 8.12), яка забезпечує виключення надмірності коду (функції, що здійснюють виконання об'єктами різних класів однакових обов'язків, кодуються приблизно, а іноді абсолютно однаково), підвищує доступність для огляду коду програми, а отже, потенційно скорочує час на її налагодження .

Тепер розглянемо рис. 8.13 фрагмент схеми ієрархії класів для перерахованих елементів (до кореневого суперкласу).

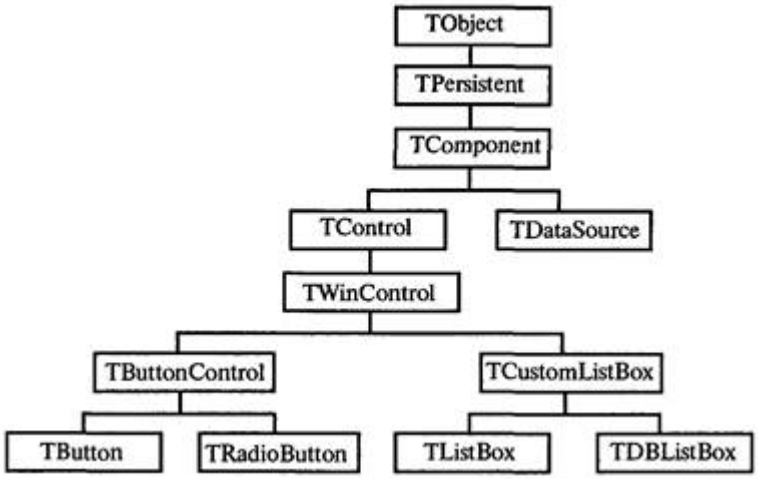
Порівнюючи малюнки 8.12 та 8.13, можна помітити:

1. Клас_1 у нашій ієрархії відповідає гілки TObject → TPersistent → TComponent;
2. Клас_2 - TControl -> TWinControl;
3. Клас_3 - TbuttonControl;
4. Клас_4 - TCustomListBox.

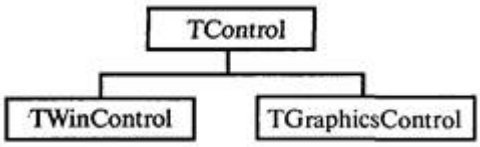
Якщо з двома останніми класами все зрозуміло, виникає питання: чому перші два класи нашої ієрархії відповідають не одному, а цілим ланцюжкам класів C++ Builder? Справа в тому, що в цьому прикладі описані не всі класи C++ Builder. Тому ті з них, які призводять до розгалуження двох перших ланцюжків, тут 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.

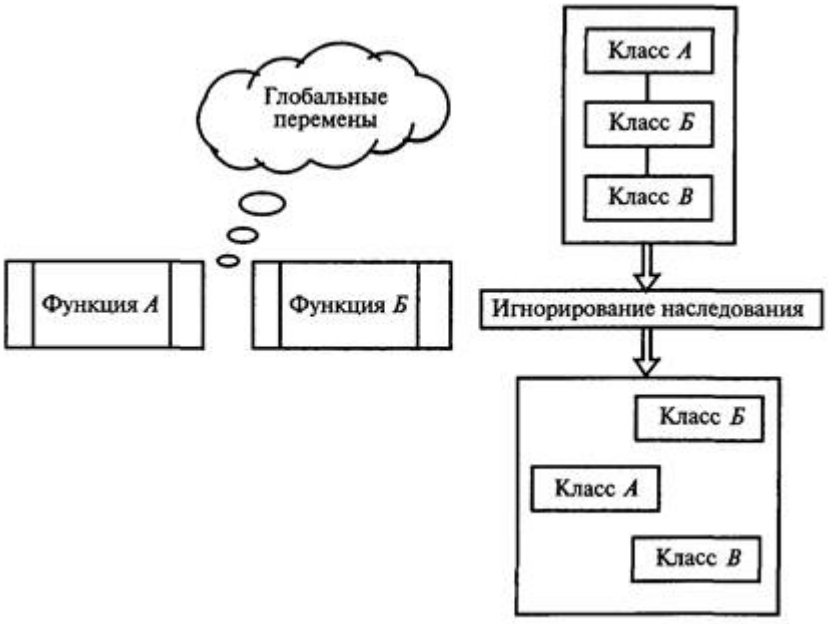
Розробка системи реального часу аналогічна до розробки системи з інтерактивним інтерфейсом.

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?