МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
Національний авіаційний університет

Д. П. Кучеров, Є. Б. Артамонов

ІНЖЕНЕРІЯ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

Навчальний посібник

Київ 2017

УДК 004.415/.416(075)
ББК 3973.20-018.2я7
К959

Рецензенти: *С. В. Лапицький* – д-р техн. наук, професор (Центральний науково-дослідний інститут озброєння та військової техніки Збройних сил України);
С. О. Лук'яненко – д-р техн. наук, професор (Національний технічний університет України «Київський політехнічний університет»);
Ю. М. Тесля – д-р техн. наук, професор (Київський національний університет імені Тараса Шевченка)

Рекомендовано вченою радою Національного авіаційного університету (протокол № від р.).

Кучеров Д. П.
К959 **Інженерія програмного забезпечення:** навч. посібник / Д. П. Кучеров, Є. Б. Артамонов. – К. : НАУ, 2017. – 388 с.

ISBN 978-966-932-

Містить теоретичні та практичні відомості про інженерію програмного забезпечення, які включають процес розроблення програмного забезпечення, типові підходи до архітектурного проектування, принципи побудови об'єктно-орієнтованих програмних систем, основи візуального проектування і тестування програмних систем. Розглянуто підходи до оцінювання якості об'єктно-програмних систем на основі відомих метричних показників.
Для студентів

УДК 004.415/.416(075)
ББК 3973.20-018.2я7

ISBN 978-966-932-

ЗМІСТ

ВСТУП

Сучасна теорія розроблення та створення програмних систем більш як за 40 років свого розвитку завдяки зусиллям іноземних та вітчизняних авторів перетворилася у гармонійну науку зі своїми законами. Велика кількість створюваних програмних засобів лише підкреслює важливість цього наукового напряму. Розроблення програмних систем припинило бути завданням окремих програмістів. Вони збираються у великі колективи, які, своєю чергою, поділяються на групи, що потребують узгодженої взаємодії. Основний наголос при цьому ставиться на компетентність та організаційні здібності менеджера чи керівника проекту, завдання якого полягає у забезпеченні термінів виконання та якості кінцевого продукту.

Основними завданнями теоретичної частини посібника є навчання студентів схематизувати процес розроблення програмного забезпечення (ПЗ). При цьому увага приділяється питанням обґрунтованості доцільності розроблення, грамотному формулюванню технічного завдання на розроблення, оволодінню навичками застосування сучасними засобами менеджера і проектувальника, використанню наявних програмних засобів. Розгляд цих питань неможливий без ознайомлення студентів з основами аналізу правильності роботи програм та їх тестування. Теоретичні положення посібника супроводжуються прикладами використання допоміжних засобів проектувальника таких, як Microsoft Project, IBM Rational Rose, мова проектування C++ та ін.

Обмежений обсяг посібника, визначений передусім часом, що відводиться навчальною програмою, а також необхідністю раціонального розподілу часу між теоретичною та практичними частинами курсу, не

дозволяє розглянути зазначені питання ґрунтовно. Деякі з них згадуються лише в межах формулювання постановки завдання.

Незважаючи на це, у посібнику приділяється увага не лише формуванню правильних поглядів на сучасний процес розроблення програмних систем, але і методичній підготовці студентів як майбутніх керівників зі створення програмних систем.

Зміст посібника умовно поділено на чотири частини. Перша частина включає вступ та чотири розділи і містить системоутворювальні поняття програмної інженерії, вимоги до програмних продуктів та систем, порядок складання завдання на розроблення програмної системи, її життєвий цикл, основні положення відповідно до цілей, що ставляться професійними об'єднаннями провідних фахівців світу *SWEBOK*, а також інструменти менеджера щодо керування розробленням програмного продукту.

Друга частина (розділи 5–7) містить блок питань зі структурного проектування програмної системи. На прикладах наводяться підходи до моделювання структури системи відповідно до відомих моделей керування. Ці моделі ґрунтуються на принципах централізованого керування та розподілених систем. Завершуються розділи питаннями проектування інтерфейсу користувача.

У третій частині (розділи 8–10) подається інформація про реалізацію об'єктно-орієнтованих програмних систем. Стисло викладаються основні принципи об'єктно-орієнтованого програмування: абстрагування, інкапсуляція, наслідування та поліморфізм. Ці принципи розглядаються в контексті таких базових понять, як об'єкт, клас та їх взаємовплив, які надалі використовуються як інструмент візуального проектування розробника – діаграми *UML*. Матеріал охоплює статичне, динамічне подання реалізації програмної системи, що стає зрозумілим не лише проектувальнику, але і замовнику, і використовуються для подальшої генерації коду.

У четвертій частині (розділи 12–14) міститься інформація про принципи та методи тестування ПЗ. Через вимірюваність кількісних та визначеність якісних показників ПЗ забезпечується зв'язок з розділом 1, у якому наводиться інформація про вимоги до програмної системи, а також указуються метрики, здатні визначати якість ПЗ та вплинути на його якість ще на етапі розроблення системи.

Такий підхід до створення програмних систем набуває комплексного характеру та дозволяє майбутнім інженерам набути навичок командної роботи.

І. МОДЕЛІ ЖИТТЄВОГО ЦИКЛУ ДЛЯ РОЗРОБЛЕННЯ ПРОГРАМНИХ СИСТЕМ

1. ПРОГРАМНА ІНЖЕНЕРІЯ ЯК ФАХ. БАЗОВІ ПОНЯТТЯ ПРОГРАМНОЇ ІНЖЕНЕРІЇ

1.1. Загальні відомості про дисципліну

Інженерія програмного забезпечення – дисципліна, що охоплює аспекти створення ПЗ від початкової стадії розроблення системних вимог до його використання.

Суб'єктами цієї діяльності є інженери, фахівці, які виконують практичну роботу. Вони застосовують теоретичні конструкції, методи і засоби там, де це потрібно, завжди намагаються вирішити завдання, навіть якщо придатної теорії або методів вирішення не існує. Інженер розуміє, що він повинен працювати в організаційних і фінансових межах укладених контрактів, тобто вирішувати поставлене завдання з урахуванням визначених умов.

У дисципліні не розглядаються технічні аспекти створення ПЗ. Основна увага приділяється питанням керування проектом створення ПЗ та розроблення засобів, методів і теорій, необхідних для створення програмних систем.

Фахівці з ПЗ застосовують існуючі методи інженерії ПЗ для вирішення завдань, знаходять найбільш ефективні способи створення високоякісних програмних продуктів. Інженерія ПЗ надає необхідну інформацію для вибору найбільш придатного методу для вирішення практичних завдань. Це не виключає творчого, неформального підходу, який у певних обставинах також може бути більш ефективним. Наприклад, розроблення сучасних програмних систем електронної комерції в мережі Інтернет потребує неформального підходу на основі поєднання графічного ескізного проектування і відповідного ПЗ.

Слід розрізнювати інженерію ПЗ та інші комп'ютерні науки. Істотна відмінність полягає в тому, що комп'ютерна наука охоплює теорію і методи побудови обчислювальних та програмних систем, тоді як інженерія ПЗ акцентує увагу на практичних проблемах роз-

роблення ПЗ. Знання комп'ютерної науки необхідне фахівцям із ПЗ так само, як знання фізики інженерам-електронникам.

Натепер інженерія ПЗ охоплює такі наукові напрями, які безпосередньо пов'язані з розробленням комп'ютерних систем, серед яких електроніка та мікроелектроніка, архітектура комп'ютерів і комп'ютерних мереж. Дисципліна безпосередньо базується на знаннях з програмування та системного ПЗ, а також на знаннях дискретної математики (рис. 1.1).

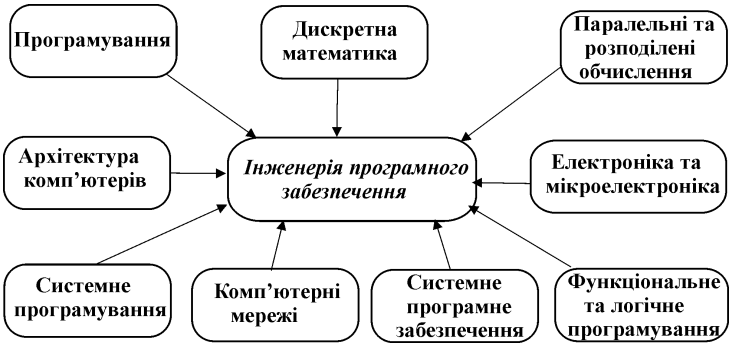


Рис. 1.1. Інженерні дисципліни, що належать до інженерії ПЗ

Фахівці з ПЗ розробляють програмні продукти, які потрібні споживачу. Розрізнюють два типи програмних продуктів.

1. *Програмні продукти загального призначення* або «*коробкове ПЗ*», до якого належать автономні програмні системи, які продаються на ринку програмних продуктів. Прикладами цього типу програмних продуктів є системи керування базами даних, текстові процесори, графічні пакети і засоби керування проектами.

2. *Програмні продукти, створені на замовлення.* Цей тип програмних продуктів включає системи керування електронними пристроями, системи підтримання певних виробничих або бізнес-процесів, системи керування повітряним транспортом тощо.

Відмінність між ними полягає в тому, що для створення програмних продуктів загального призначення специфікацію вимог до них розробляє компанія-виробник. Для замовлених програмних продуктів специфікацію розробляє організація, яка купує цей продукт. Специфікація необхідна розробникам ПЗ для створення будь-якого програмного продукту.

Вирішення завдань без збільшення вартості апаратних засобів призводить до виникнення програмних помилок унаслідок модернізації ПЗ відповідно до зміни вимог, пропонованих до створюваної системи. Для зменшення кількості таких вимог необхідно знати властивості будь-якої системи.

1.2. Система та її властивості

Створення систем охоплює процеси підготовки специфікацій, проектування, розроблення, тестування, упровадження і супроводження систем як єдиного цілого. Фахівець, що займається розробленням обчислювальних систем, не зосереджений лише на ПЗ, він приділяє однакову увагу ПЗ, апаратним засобам і засобам взаємодії з користувачами та системному оточенню. Він визначає функції, які буде виконувати система і для яких вона будується, а також взаємодію системи з її оточенням. Фахівець зі створення ПЗ повинен розуміти завдання системотехніки, оскільки виникаючі проблеми часто є результатом рішень, прийнятих системотехніками. Існує багато найрізноманітніших визначень поняття «система» – від абстрактних до конкретних.

Система – це сукупність взаємодійних компонентів, що працюють спільно для досягнення певних цілей.

Визначальною ознакою системи є те, що взаємодія властивостей і поводження системних компонентів має надто складний і суперечливий характер. Коректне функціонування кожного системного компонента залежить від роботи багатьох інших компонентів. Так, операційне забезпечення призначено для функцій, які виконують лише процесор і відповідні апаратні засоби. Процесор виконує обчислення, якщо програмні засоби, що задають ці обчислення, коректно інстальовані.

Системи складаються з компонентів та підсистем. Наприклад, комп'ютерна система поліції містить геоінформаційну систему, що пропонує інформацію про місце, де трапилася або може трапитися яка-небудь подія.

Підсистемами називаються компоненти інших систем. Визначальна властивість підсистем полягає в їх здатності функціонувати самостійно, незалежно від тих систем, до яких вони входять. Тому цю інформаційну систему можна використовувати також і в транспортних засобах. Разом з тим її поводження в складі конкретного об'єкта залежить від взаємодії з іншими підсистемами.

Хибність взаємодії між системними компонентами означає, що система не зводиться лише до суми її складових частин. Вона має певні властивості, які специфічні для неї як цілісної системи. Такі *інтеграційні властивості* не можуть бути характеристиками якої-небудь окремої частини системи. Більше того, вони проявляються тоді, коли система розглядається як єдине ціле. Деякі з них можна знайти в аналогічних окремих підсистемах, але частіше вони є результатом комплексної взаємодії всіх підсистем і їх неможливо оцінити, виходячи з аналізу окремих системних компонентів.

Показники інтеграційних властивостей:

1) *сумарний розмір системи* – обчислюється відповідно до властивостей окремих компонентів;

2) *безвідмовність системи* – залежить від безвідмовності складових компонентів і взаємозв'язків між ними;

3) *зручність експлуатації системи* – залежить не лише від ПЗ та апаратних засобів системи, але і від оточення, у якому експлуатується система, і від системних операторів.

Фахівець із ПЗ повинен знати системотехніку обчислювальних систем, оскільки тут програмний компонент відіграє надважливу роль. Технології інженерії ПЗ часто є критичним фактором під час розроблення складних обчислювальних систем.

Інтеграційними властивостями системи є функціональні та нефункціональні властивості.

Функціональні властивості проявляються тільки тоді, коли система працює як одне ціле. Наприклад, велосипед має функціональні властивості транспортного засобу лише тоді, коли складений з його компонентів.

Нефункціональні властивості – безвідмовність, продуктивність, безпека і захищеність (обмеження несанкціонованого доступу до системи) – залежать від поводження системи в операційному оточенні. Такі властивості часто критичні для обчислювальних систем, і якщо вони не досягають певного оптимального рівня, то система не буде працездатною. Деякі функції і можливості системи можуть бути не затребувані всіма користувачами, тому система може бути працездатною і без них. Ненадійна або неефективна система за окремими функціями вважається бракованою.

Щоб проілюструвати складність визначення інтеграційних властивостей, розглянемо показник безвідмовності системи. Це ком-

плексний показник, що завжди варто розглядати на рівні системи, а не її окремих компонентів. Компоненти в системі взаємозалежні, тому збій одного компонента може поширитися на всю систему і викликати відповідну реакцію всіх інших. Проектувальники систем часто не можуть вгадати послідовність збоїв у системі, тому важко оцінити безвідмовність системи лише на підставі даних про безвідмовність її окремих компонентів.

На загальну безвідмовність системи впливають три тісно пов'язані між собою фактори:

1. *Безвідмовність апаратних засобів* – визначається ймовірністю виходу з ладу окремих апаратних компонентів і часом, потрібним на їх заміну.

2. *Безвідмовність ПЗ.* Програмні помилки зазвичай не впливають на апаратні засоби системи. Тому система може продовжувати функціонувати навіть тоді, коли ПЗ видає некоректні результати.

3. *Помилки операторів* – допускаються людьми, які експлуатують систему.

Безвідмовність системи також залежить від оточення, у якому вона експлуатується. Передбачити системне оточення, у якому буде експлуатуватися система, дуже важко. Інакше кажучи, складно описати оточення у вигляді обмежень, які потрібно враховувати у процесі розроблення системи. Підсистеми, що становлять цілісну систему, можуть по-різному реагувати на зміни в системному оточенні, тим самим впливаючи на загальну безвідмовність системи непередбаченим чином. Унаслідок цього, навіть якщо система є єдиним цілим, буває важко або зовсім неможливо виміряти рівень її безвідмовності.

Інші інтеграційні характеристики (такі, як продуктивність і зручність експлуатації) також важко визначати, але їх можна оцінити в процесі експлуатації системи. Оцінювання інших властивостей, наприклад безпеки системи та її захищеності, створює більші складності. Ці властивості відображають невидимі характеристики. Наприклад, розробляючи заходи щодо виключення несанкціонованого доступу до даних, порівняно легко визначити всі можливі режими доступу до даних і заборонити небажані. Тому оцінити рівень захищеності можна лише через характеристики системи, властиві їй за замовчуванням. Більше того, система буде вважатися захищеною доти, поки не будуть зламані її засоби захисту.

Система і її оточення. Будь-яка система залежить від сигналів або іншої інформації, що надходить на її входи; інакше кажучи, система функціонує в певному оточенні, що впливає на її функціонування і продуктивність. Іноді оточення розглядається як самостійна система, що складається з множини підсистем, які впливають одна на одну.

Декілька систем, об'єднаних у систему життєзабезпечення офісного будинку, показано на рис. 1.2. Системи опалення, електроживлення, освітлення, водозабезпечення і каналізації та безпеки є підсистемами будівлі, що, своєю чергою, можна розглядати як систему. Будинок, розташований на вулиці, також є системою вищого рівня, вулиця – підсистемою системи «місто» т. ін. Таким чином, оточення будь-якої системи є системою більш високого рівня. У загальному випадку оточення будь-якої системи – це композиція її локального оточення й оточення системи вищого рівня.

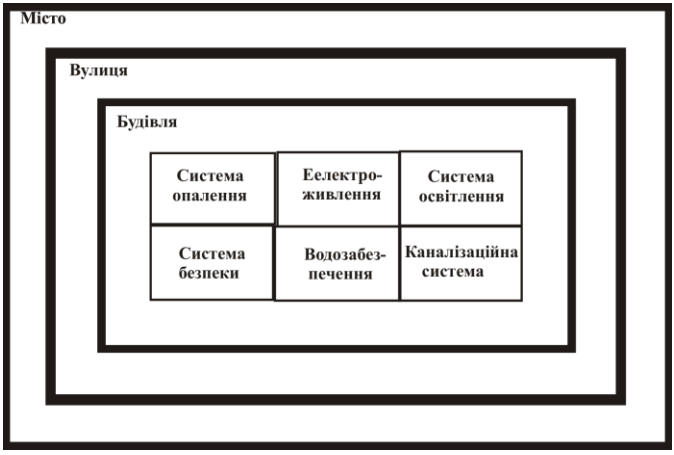


Рис. 1.2. Оточення системи життєзабезпечення офіса

Система життєзабезпечення будинку (рис. 1.2) складається з інших систем цього будинку. Оточення системи також є системами, що не входять до системи «будинок», але не належать до систем «вулиця і місто», включаючи природні системи, у тому числі погодну (тобто вплив погодних факторів на систему безпеки).

Потреба враховувати оточення систем під час її розроблення зумовлена декількома причинами. У багатьох випадках система

призначається саме для реагування на зміну певних параметрів оточення. Так, система «опалення» реагує на зміни в навколишньому середовищі, підвищуючи або знижуючи температуру своїх опалювальних приладів. Правильне функціонування такої системи полягає в реагуванні на зміну параметрів оточення. Але якість її функціонування залежить від параметрів оточення. Так, система «електропостачання» залежить безпосередньо від вуличного оточення будинку. Наприклад, під час робіт з благоустрою вулиці через недогляд може бути пошкоджений силовий кабель, що може вивести з ладу електропостачання будинку.

На розроблення систем впливають як людські, так і організаційні фактори (експлуатаційні, організаційні, фактор персоналу).

1. *Експлуатаційний фактор.* Якщо система потребує внесення змін у робочий процес експлуатації внаслідок змін параметрів оточення, то необхідним стає навчання персоналу, який експлуатує цю систему. Якщо навчання тривале або втрачається заробіток персоналу, то існує ймовірність, що така система не буде вибрана користувачами.

2. *Фактор персоналу.* Якщо впровадження системи може призводити до зниження вимог до кваліфікації персоналу або докорінно змінити способи його роботи, тоді персонал може спробувати протидіяти впровадженню системи в їх організацію, наприклад на рівні менеджерів, що керують проектами, якщо вони відчують, що їх статус в організації знизиться після її впровадження.

3. *Організаційний фактор.* Іноді впровадження системи може змінити структуру владних повноважень в організації. Наприклад, значущості у владній структурі організації можуть набути оператори цієї системи.

Зазначені фактори часто виявляються критичними під час прийняття рішення про те, буде чи не буде впроваджуватися система. Передбачити їх дію дуже складно, особливо якщо розробники системи не мають певного досвіду. Відомості про системне оточення включають у специфікацію системи для врахування їх розробниками під час її проектування.

1.3. Апаратне та програмне забезпечення обчислювальних систем

Під час моделювання використовують графічне зображення структури системи, тобто її компоненти і взаємозв’язки між ними.

Архітектуру системи зазвичай подають блок-схемою, де блоки відповідають основним підсистемам, а зв'язки між ними позначають лініями зі стрілками, що з'єднують окремі блоки діаграми. Зв'язки можуть відповідати потокам даних, послідовності включення підсистем у роботу або будь-які інші типи залежності. Блок-схему основних компонентів системи сигналізації попередження про несанкціонований доступ до приміщення та короткий опис підсистем показано на рис. 1.3. Функціональні підсистеми системи сигналізації подано в табл. 1.1.

Таблиця 1.1

Функціональні підсистеми системи сигналізації

Підсистема	Опис
Датчики руху	Реагують на рух у приміщенні
Дверні датчики	Виявляють, чи відкриті двері будинку
Контролер	Керує діями системи
Сирена	Видає потужний звуковий сигнал
Синтезатор голосу	Видає голосове попередження
Телефонна лінія	Робить виклик службі безпеки для припинення несанкціонованого проникнення

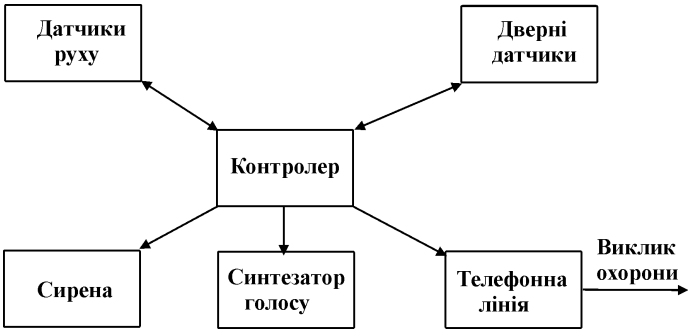


Рис. 1.3. Система сигналізації

На цьому рівні деталізації система поділяється на окремі підсистеми. Кожну підсистему, своєю чергою, можна подати декомпозицією її функціональних компонентів. Декомпозицію системи та її підсистем можна проводити і за іншими ознаками, наприклад конструктивними або технологічними.

Історично склалося так, що модель системної архітектури використовується для виділення апаратних і програмних компонентів системи, які зазвичай розробляються одночасно. Разом з тим протиставлення «апаратні засоби – програмне забезпечення» у сучасній практиці недоречне і неістотне, оскільки всі системні компоненти мають певні обчислювальні можливості.

На рівні системної архітектури більш раціонально класифікувати підсистеми відповідно до виконуваних функцій, не акцентуючи увагу на тому, чи вони є апаратними чи програмними компонентами. Чи ці функції будуть реалізовані апаратно або програмно, часто визначають на підставі нетехнічних факторів (наприклад, потрібний час для створення компонента) або за наявності на ринку придатних готових промислових пристроїв.

Кілька основних підсистем системи керування польотами, які є великими системами, зображено на рис. 1.4. Інформаційні потоки між підсистемами показано сполучними лініями зі стрілками.

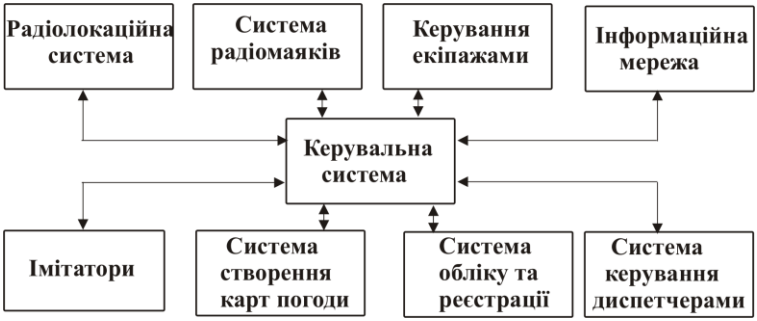


Рис. 1.4. Архітектура системи керування польотом

Системна архітектура описується в термінах функціональних підсистем, незалежно від того, чи ці підсистеми є апаратними, чи програмними. За функціональними ознаками в системі завжди можна виокремити шість основних компонентів.

1. *Сенсорні* – збирають інформацію про системне оточення. Наприклад радіолокатори в системі керування польотами, датчик розміщення паперу в принтері або термопара в радіаторі двигуна.

2. *Виконавчі* – виконують деякі дії. Як приклад можна навести регулювальний клапан, що зачиняє або відчиняє заслінку в трубопроводі для зменшення чи збільшення швидкості потоку рідини, за-

крилки крил літака, які керують кутом нахилу літака, механізм обертання антен.

3. *Комунікаційні* – надають можливість іншим системним компонентам обмінюватися інформацією, наприклад, плати мережевого інтерфейсу комп'ютерів, що об'єднані в локальну мережу.

4. *Координуючі* – узгоджують роботу інших компонентів, що вказує на процес. Прикладом є планувальник завдань у системах реального часу, який вказує на процес, що обробляється процесором.

5. *Обчислювальні* – виконують розрахунки з даними, що надходять до них. Прикладом обчислювального компонента є математичний співпроцесор, що виконує обчислення із числами в експонентному форматі.

6. *Інтерфейсні* – перетворюють одну систему подання в іншу. Наприклад, аналогово-цифровий перетворювач перетворює аналоговий сигнал у цифровий код.

Наведена класифікація компонентів допомагає проектувати систему. Більшість систем містять компоненти всіх типів, і завдання розробника полягає в точному визначенні типу компонента виходячи зі специфікації системи. Якщо компоненти містять ознаки різних типів, це може зумовити певні проблеми під час проектування системи.

1.4. Інженерія складних систем

Складною будемо називати систему, що утворюється множиною взаємодійних складових (підсистем), унаслідок чого вона набуває нових властивостей, яких немає на підсистемному рівні і які не можуть бути звичайним поєднанням властивостей підсистем [1].

Найпростішим прикладом складної системи є персональний комп'ютер. Його основні складові: материнська плата, монітор, клавіатура, миша, внутрішня та зовнішня пам'ять, *DVD*-привід. Кожен із цих компонентів можна, своєю чергою, розкласти на більш дрібні. Наприклад, материнська плата складається з первинної пам'яті, процесора і шини, до якої приєднані периферійні пристрої. Процесор можна далі розкласти на регістри і схеми логіки, які, своєю чергою, складаються із ще простіших деталей: вентилів, інверторів і т. ін.

Цей приклад демонструє *ієрархічну природу* побудови складної системи. Нормальна робота персонального комп'ютера забезпечу-

ється не лише взаємодією всіх його складових частин. Разом узяті окремі частини утворюють логічне ціле. Щоб зрозуміти, як працює комп'ютер, необхідно розкласти його на компоненти і розглянути їх окремо. Отже, роботу процесорної частини, монітора або жорсткого диска можна вивчати незалежно.

Ієрархія дозволяє абстрагуватись від неістотного. Так, компоненти персонального комп'ютера містять поняття, що утворюють певні рівні абстракції. Кожному із цих рівнів відповідає сукупність пристроїв, взаємодія яких забезпечує функціонування компонентів більш високого рівня. Конкретний рівень абстракції можна вибирати виходячи з певних потреб. Наприклад, для дослідження синхронізації первинної пам'яті доцільно розглянути комп'ютер на рівні вентилів. Проте цей рівень абстракції неприйнятний для роботи з електронними таблицями. Між різними рівнями абстракції завжди є чітка межа.

Складна система не містить централізованих частин, що безпосередньо координують діяльність компонентів, які належать до нижчих рівнів. Попри це вона містить окремі частини, що діють як незалежні агенти, кожен з яких характеризується досить складним поводженням, яке є частиною функцій більш високого рівня. На цьому рівні функціонування складної системи забезпечується лише завдяки цілеспрямованій взаємодії незалежних агентів. У теорії складності це явище називають емерджентністю: поводження цілого складніше, ніж поводження суми його складових [1].

Прикладом надскладного фізичного об'єкта є матерія. Дослідження в астрономії і ядерній фізиці надають багато інших прикладів надскладних систем. Астрономи вивчають галактики, які об'єднані в скупчення. Своєю чергою, галактики складаються із зірок, планет та інших небесних тіл. Фахівці з ядерної фізики зіштовхуються зі структурною ієрархією фізичних тіл більшого масштабу. У цих складних ієрархіях знову виявляються універсальні механізми. Незважаючи на те, що у Всесвіті діють усього чотири типи сил: гравітаційна, електромагнітна, сильна і слабка взаємодії, існують закони фізики, що мають універсальний характер і поширюються як на галактики, так і на кварки. Це закони, що стосуються взаємодії об'єктів, наприклад, закон збереження енергії та імпульсу.

Не всі системи ПЗ є складними. Існує клас програм, що проектуються, розроблюються, супроводжуються і використовуються в інтересах однієї людини. Такі системи, як правило, дуже обмежено

застосовуються і нетривалий час використовуються. Їх можна замінити новими програмами, ніж намагатися повторно використовувати, переробляти або вдосконалювати. Інтерес становлять проблеми розроблення промислового ПЗ. Це програми:

– керування засобами за обмежених ресурсів часу і пам'яті;

– підтримання цілісності баз даних, що містять сотні тисяч записів і забезпечують одночасне відновлення і запити;

– системи керування і контролю за реальними об'єктами, такими як повітряний або наземний транспорт.

Такі системи зазвичай використовуються тривалий час, і згодом від їх нормального функціонування починають залежати багато користувачів. У галузі промислового програмування є також засоби, що спрощують створення додатків у конкретних предметних галузях, а також програми, що імітують деякі аспекти людського інтелекту.

Найважливішою особливістю промислової програми є її надскладність. Одному програмісту не під силу вирішити всі проблеми, пов'язані із проектуванням такої системи. Складність промислових програм перевищує інтелектуальні здібності окремої людини. Термін «істотна» означає, що зі складністю промислових програм можна впоратися, але ігнорувати її не можна. Причини складності ПЗ:

– слабке уявлення про предметну галузь;

– організаційні труднощі розроблення ПЗ;

– необхідність забезпечення гнучкості програм;

– складність опису функціонування дискретних систем.

Розглянемо вимоги до електронної системи багатомоторного літака, комутатора стільникового телефону чи автономного робота. Вони мають складні для розуміння механізми функціонування, а якщо додати до цього вимоги щодо зручності, продуктивності, вартості, стійкості, надійності, то завдання стає надскладним.

Ця зовнішня складність породжується непорозумінням, що існує між користувачами системи і її розробниками, оскільки користувачам важко сформулювати свої потреби у формі, зрозумілій розробнику. У цьому не можна дорікати користувачів або розробників; кожна з цих груп не має необхідних знань у предметній галузі іншої групи. Користувачі і розробники часто по-різному бачать сутність проблеми і пропонують різні способи її вирішення. Насправді, навіть якщо користувач точно знає, що йому потрібно, його вимоги дуже важко формалізувати. Як правило, вони описуються в

багатотомних документах, проілюстрованих рисунками. Такі документи важко зрозуміти, вони припускають неоднозначну інтерпретацію і дуже часто містять інформацію, що стосується швидше проектування, а не виражає основні вимоги замовника.

Вимоги, пропоновані до системи ПЗ, у процесі розроблення змінюються. Це ще більше підвищує її складність. Як правило, технічне завдання коригується, оскільки в процесі проектування системи воно поступово уточняється. Ознайомлюючись з першими результатами, описаними в проектній документації і реалізованими в прототипах, а також використовуючи систему після її інсталяції, користувачі починають краще розуміти і чіткіше формулювати свої реальні потреби. Цей процес дозволяє розробникам вникнути в предметну галузь і ставити більш точні питання для виявлення «темних» місць проектованої системи.

Велика програмна система ПЗ завжди потребує інвестування, тому не можна відкидати існуючу систему з кожною зміною технічного завдання. Системи згодом еволюціонують. Іноді цей процес називають супроводженням ПЗ.

Основне завдання розробника ПЗ – створити ілюзію простоти, щоб захистити користувачів від великої і часто довільної складності проекту. Очевидно, що велика ємність системи ПЗ не є її основною проблемою. Для зменшення розмірів програм відшукуються потужні методи, що створюють ілюзію простоти і дозволяють повторно використовувати існуючі коди та проектні рішення. Проте вимоги, пропоновані до системи, часто змушують проектувальників або створювати заново велику кількість програм, або використати існуючий код, адаптуючи його до нових умов. Такий обсяг робіт під силу лише команді розробників, склад якої в ідеалі має бути мінімальним. Незалежно від кількості розробників постійно виникають складні проблеми, зумовлені колективним проектуванням, які вони повинні вирішувати. Чим більше розробників, тим складніші зв’язки між ними і тим тісніша координація їх взаємодії, особливо якщо учасники робіт часто географічно віддалені один від одного. Таким чином, колективне розроблення ПЗ потребує забезпечення єдності і одного задуму проекту та узгодження дій розробників.

Програмування дає змогу проектувальнику виражати абстракції будь-якого рівня. Ця гнучкість приваблива, оскільки вона спонукає розробника самостійно створювати базові конструкції, з яких

формуються абстракції високих рівнів. Стандарти з якості ПЗ розробляються дуже повільно. Як результат – проектування ПЗ є трудомістким.

Програма виконується на комп'ютерах, тому є системою з дискретними станами. Стан прикладної програми в кожен момент часу описується сукупністю змінних, їх поточних значень, адрес і стеків виклику для кожного процесу. Аналогові системи є безперервними. Дискретні системи мають кінцеву кількість можливих станів. У великих системах спостерігається так званий комбінаторний вибух, що робить цю кількість дуже великою. Проектуючи систему, розробники поділяють їх на компоненти так, щоб одна частина якомога менше впливала на іншу. Однак переходи між дискретними станами неможливо моделювати за допомогою неперервних функцій. Кожна подія, зовнішня відносно системи, може надати їй нового стану; перехід з одного стану в інший не завжди є детермінованим. У найгіршому випадку зовнішня подія може пошкодити систему, оскільки її творці не передбачили всі можливі варіанти. Якщо у ПЗ руху космічного корабля відбудеться переповнення пам'яті, спричинене некоректними вхідними даними, наслідки можуть виявитися непередбачуваними. Ще недавно в ПЗ систем керування метрополітеном, автомобілями, супутниками, повітряним рухом, складами тощо спостерігалося різке зростання кількості збоїв. Таке поводження безперервних систем було б малоймовірним, але в дискретних системах будь-яка зовнішня подія може вплинути на будь-яку частину внутрішнього стану системи. Очевидно, це є основним стимулом для інтенсивного тестування систем ПЗ. Натепер немає ні математичних інструментів, ні інтелектуальних можливостей для повноцінного моделювання поводження великих дискретних систем. Виходом із цієї ситуації є прийнятний рівень упевненості їх правильної роботи.

Можна виокремити такі ознаки складної системи, характерні для програмних систем.

Ієрархічність структури. Складні системи складаються із взаємозалежних підсистем, що мають, у своєю чергою, власні підсистеми, аж до найнижчого рівня, утвореного елементарними компонентами. Архітектура складної системи залежить як від її компонентів, так і від їх ієрархічності.

Відносність вибору елементарних компонентів. Спостерігач довільно вирішує, які компоненти у складній системі вважати еле-

ментарними. Обраний компонент з погляду одного спостерігача може виявитися на набагато вищому рівні абстракції з погляду іншого.

Зв'язки всередині компонентів. Вони тісніші, ніж зв'язки між компонентами. Ця обставина дозволяє відокремити «високочастотну» динаміку компонентів у їх внутрішній структурі, від «низькочастотної» динаміки щодо взаємодії між компонентами. Ця відмінність між внутрішніми і міжкомпонентними взаємодіями дозволяє поділити функції між частинами системи і вивчати їх окремо.

Складна система, що розроблена «з нуля», ніколи не працює як треба, і жодна «латка» не змусить її працювати правильно. Проектування варто починати із простої працездатної системи.

У процесі еволюції системи об'єкти, що спочатку вважалися складними, стають елементарними компонентами, з яких створюються ще більш складні системи. Правильні елементарні об'єкти неможливо створити відразу: спочатку необхідно вивчити реальне поводження системи і лише потім її удосконалювати.

1.5. Проектування та конструювання програмного забезпечення

Створення ПЗ – це сукупність процесів, що приводять до появи нового програмного продукту.

Чотири фундаментальні процеси, які властиві будь-якому проекту створення ПЗ:

– розроблення специфікації вимог на програмне забезпечення;
– створення ПЗ;
– атестація ПЗ;
– удосконалення (модернізація) ПЗ.

Для реалізації різноманітних програмних проектів ці процеси можуть бути організовані різними способами і описані на різних рівнях деталізації. Тривалість реалізації цих процесів також далеко не завжди однакова. Загалом різні організації, що займаються виробництвом ПЗ, найчастіше використовують різні процеси для створення програмних продуктів навіть одного типу. Проте певні процеси більше підходять для створення програмних продуктів одного типу і менше для іншого типу програмних додатків. Якщо використати невідповідний процес, то це може призвести до зниження якості та функціональності розроблюваного програмного продукту.

Така модель являє собою спрощений опис процесу створення ПЗ – послідовності практичних етапів, необхідних для розроблення створюваного програмного продукту. Подібні моделі, незважаючи на їх різноманітність, являють собою абстрактне подання реального процесу створення ПЗ. Моделі можуть відображати процеси, які є частиною технологічного процесу створення ПЗ, компонентами програмних продуктів і дій людей, що беруть участь у створенні ПЗ. Вони такі:

Модель послідовності робіт. Показує послідовність етапів, виконуваних у процесі створення ПЗ, включаючи початок і завершення кожного етапу, а також залежність між виконанням етапів. Етапи в цій моделі відповідають певним роботам, виконуваним розробниками ПЗ.

Моделі потоків даних і процесів. Процес створення ПЗ подається множиною активності (процесів), що здійснює перетворення певних даних. Наприклад, якщо створюється специфікація ПЗ, то на вході отримують загальні дані про роботу системи, а на виході дані, які надходять на вхід процесу проектування ПЗ. Перетворювати дані при реалізації активностей можуть як розробники ПЗ, так і комп'ютери.

Рольова модель. Її складовими є люди або дії, виконувані ними.

Розглянемо основні відмінності процесу створення систем і процесу розроблення ПЗ. Етапи процесу створення системи показано на рис. 1.5.

Рис. 1.5. Процес створення системи

Після прийняття рішень у процесі розроблення систем (наприклад, про установлення певних типів радіолокаторів у систему керування польотами) внесення змін у систему може виявитися до-

сить дорогим. Перепроектувати систему часто взагалі неможливо. Це одна із причин широкого використання ПЗ для створення різних систем – гнучкість процесу розроблення, що дозволяє вносити зміни в розроблювану систему у відповідь на нові вимоги, пропоновані до неї.

У команду розробників систем неминуче залучаються фахівці різних профілів, тому команда розробників повинна мати відповідні знання, щоб усебічно розглянути системні можливості під час прийняття будь-яких рішень. Розглянемо систему керування польотом (СКП), у якій використовується радіолокаційна або будь-яка інша система для визначення координат літаків (див. рис. 1.3).

Фахівці різних профілів можуть пропонувати різні варіанти структурної моделі системи, які міститимуть різні функціональні компоненти. Вибір певної моделі не обов'язково ґрунтується лише на технічних аргументах. Нехай, наприклад, однією з альтернатив розроблення СКП є установлення нової радіолокаційної системи замість модернізації існуючої. Якщо в команду розробників входять будівельники, то вони можуть наполягати саме на цьому варіанті створення СКП, оскільки він забезпечить роботою і їх, і будівельні підрозділи. При цьому для обґрунтування потрібного варіанта можна вдаватися до технічних аргументів.

Визначення системних вимог. На етапі визначення системних вимог формуються і формалізуються вимоги до системи, розглядуваної як єдине ціле. На цьому етапі формуються вимоги трьох типів.

Загальні функціональні вимоги. Основні функції системи визначаються на найвищому (абстрактному) рівні подання системи. Функціональні вимоги деталізуються вже на рівні підсистем.

Системні властивості. Це інтегровані властивості системи, які впливають на вимоги для підсистем. До них належать продуктивність, безвідмовність, захищеність та ін.

Властивості, яких не повинно бути в системі. Іноді набагато легше вказати, чого система не повинна робити, ніж те, що вона має виконувати. Наприклад, від СКП необхідно очікувати операторам лише необхідну інформацію, яка не відволікає їх увагу.

Розглянемо систему захисту від несанкціонованого вторгнення, призначену для установлення в офісному будинку, та сформулюємо її завдання.

Варіант 1. Система повинна попереджувати про несанкціоноване проникнення в будинок.

Це завдання точно описує призначення системи і її функції. Таке формулювання підходить для системи безпеки, що вже існує і обмежує можливості проектування системи. Відповідно до нього від несанкціонованих проникнень можна застосувати будь-які засоби, навіть без перевірки працездатності самої системи сигналізації. Якщо систему потрібно модернізувати, формулювання має включати більш «широку» ціль.

Варіант 2. Система повинна гарантувати відсутність істотних порушень нормального функціонування й експлуатації будинку внаслідок незаконних вторгнень.

Основні труднощі у визначенні системних вимог полягають у тому, що система будується для вирішення проблем, що мають взаємозалежні вхідні впливи, які неможливо точно описати. Дійсна природа таких проблем може виявитися лише в процесі її вирішення. Наприклад, завдання прогнозування землетрусів. Дотепер не існує точних способів прогнозування ні епіцентру землетрусу, ні його часу, ні сили, ні впливу на навколишнє середовище. Тому неможливо заздалегідь повністю спланувати всі дії на випадок великого землетрусу – це можна зробити лише тоді, коли він відбудеться.

Проектування систем. Проектування системи (рис. 1.6) полягає у визначенні системних компонентів на підставі функціональних вимог до системи. Процес проектування складається з декількох етапів.

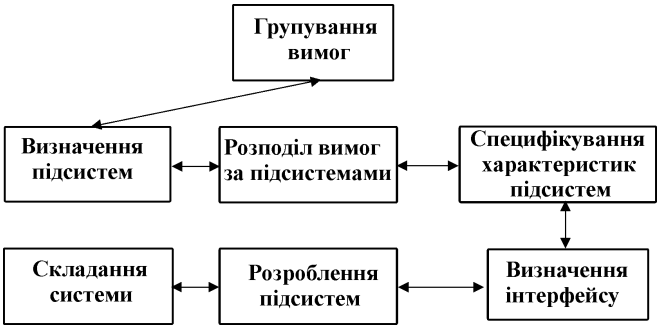


Рис. 1.6. Етапи проектування системи

Розподіл вимог на групи. Вимоги аналізують і поділяють на окремі групи. Звичайну множину вимог можна поділити на групи декількома способами, але на цьому етапі необхідно зберегти «життєздатні» розподіли.

Визначення підсистем. Визначаються підсистеми, які окремо або спільно реалізують системні вимоги. Група вимог зазвичай проектується на кілька підсистем, тому можна об'єднати кілька вимог в одну.

Розподіл вимог за підсистемами. Вимоги ставляться відповідно до підсистем, що будуть їх реалізовувати.

Специфікація функціональних характеристик підсистем. Визначаються функціональні характеристики кожної підсистеми. Цей етап є частиною створення специфікації для підсистем. На цьому етапі також формалізується взаємодія між підсистемами.

Визначення інтерфейсів підсистем. Для кожної підсистеми визначається інтерфейс. Лише після цього можна почати розроблення самих підсистем.

Наявність зворотного зв'язку між етапами і можливість вертатися до попереднього етапу в процесі проектування системи показано на рис. 1.6 двосторонніми стрілками. Це потрібно для вирішення проблем, що виникли на цьому етапі.

Для більшості систем можна розробити декілька проектів. Це припускає широкий діапазон можливих рішень, що складаються з різних комбінацій апаратних і програмних компонентів і людського фактора. Для подальшого розроблення вибирається рішення, яке найбільш повно задовольняє системні вимоги.

Розроблення підсистем. На цьому етапі реалізуються підсистеми, визначені на етапі проектування системи. Якщо підсистема є програмною, етап розроблення включатиме процеси формалізації вимог, проектування, створення та ін.

Натепер рідко розроблення всіх підсистем починають «з нуля». Інколи дешевше купити готовий виріб, ніж розробляти власну підсистему. На цьому етапі може виникнути потреба повернутися до етапу проектування для того, щоб на рівні вимог «пригнати» куплений виріб до системи. Цей виріб може не задовольняти всі вимоги, пропоновані до компонента, який він заміщає, але якщо асортименти комерційних продуктів досить широкий, витрати на повторне проектування невеликі.

Усі підсистеми, як правило, розробляються одночасно. Якщо виникають внутрішні проблеми, що переривають процес розроблення підсистем, можна модифікувати всю систему. Коли система складається здебільшого з апаратних компонентів, проведення мо-

дифікації системи після початку виробництва її компонентів може виявитися досить витратним. Доводиться шукати якесь «обхідне» рішення для виходу з подібних ситуацій. Часто таким рішенням є включення в систему програмних компонентів, оскільки вони досить гнучкі і порівняно легко модифікуються. Своєю чергою, це веде до зміни вимог, пропонованих до ПЗ, і до змін у проекті ПЗ.

Складання системи. Цей процес являє собою інтеграцію незалежно розроблених підсистем у єдину закінчену систему. У процесі складання можна використовувати метод, відповідно до якого всі підсистеми інтегруються одночасно. Але з технічних і організаційних причин набагато важливішим є послідовне складання, коли окремі підсистеми інтегруються в систему по черзі, одна за одною.

Процес послідовного складання вважається більш придатним для системної інтеграції, оскільки, по-перше, неможливо скласти такий графік робіт, згідно з яким етап розроблення всіх підсистем закінчується одночасно; по-друге послідовне складання зменшує кількість помилок, зумовлених неправильною інтеграцією системи. Якщо одночасно інтегрується кілька підсистем, то причиною виявленої в процесі тестування помилки може бути кожна з них. Якщо інтегрується одна підсистема у вже працюючу систему, то причиною виявленої помилки, найімовірніше, буде остання інтегрована підсистема або знову встановлені зв'язки між нею та існуючими підсистемами.

Помилки і дефекти окремих підсистем і системи в цілому часто проявляються саме на етапі складання. Це може викликати дискусії і конфлікти між розробниками різних підсистем через ту підсистему, яку визнано «винною». Проте вирішення виниклих проблем потребуватиме кілька тижнів, а то і місяців роботи.

Інсталяція системи. У процесі інсталяції система «поринає» у те оточення, у якому вона повинна працювати. Інсталяція складних систем може зумовити різні проблеми, вирішення яких інколи потребує кілька місяців, а то і років. Серед них можуть бути такі.

1. Оточення, у якому інсталюється система, не відповідає тому, для якого вона спроектована.

2. Потенційні користувачі не сприймають упровадження цієї системи у своїй організації.

3. Нова система може співіснувати зі старою доти, доки в організації, де вона інсталюється, не переконаються, що нова система працює так, як потрібно. При цьому випробувати нову систему можна тільки тоді, коли стара система не функціонує.

4. Під час інсталяції можливі суто фізичні проблеми. Це властиво для достатньої кількості приміщень із каналами для мережевих кабелів; можуть знадобитися додаткові повітряні кондиціонери або інші вбудовані в будинок прилади тощо.

Уведення системи в експлуатацію. Після інсталяції системи, її вводять в експлуатацію, для чого проводять навчання системних операторів і змінюють їх звичайний робочий процес для ефективного використання нової системи. На цьому етапі можуть виникати непередбачені проблеми, якщо системна специфікація містить помилки або прорахунки. Поки система функціонує відповідно до специфікації, ці дефекти можуть не виявитися, і тому розробники не передбачають відповідних режимів експлуатації системи.

Наприклад, проблемою, що може виявитися лише після введення системи в експлуатацію, є сумісність нової та існуючої систем. Це може бути суто фізична проблема сумісності. Можливі також проблеми, що виникають під час передавання даних від однієї системи до іншої. Більш складною проблемою може стати розходження інтерфейсів різних систем. Тоді введення в експлуатацію системи може призвести до зростання помилок системних операторів унаслідок неправильного використання команд інтерфейсу нової системи.

Методи інженерії ПЗ. Ці методи ґрунтуються на структурному підході до створення ПЗ, що сприяє виробництву високоякісного програмного продукту ефективним в економічному аспекті способом. Структурний аналіз уперше був проведений ще в 1970-х роках. До цих методів додалися об'єктно-орієнтовані методи, запропоновані Бучем і Рамбо. Ці методи тепер інтегровані в єдиний уніфікований метод, побудований на основі уніфікованої мови моделювання *UML* (*Unified Modeling Language*). Згадані методи ґрунтуються на ідеї створення моделей системи, які можна зобразити графічно.

Об'єктно-орієнтованими методами створюють інтерактивні (діалогові) програмні системи, але майже не використовуються під час розроблення систем, що працюють у режимі реального часу.

В інженерній практиці можуть застосовуватися *CASE*-засоби (*Computer-Aided Software Engineering* – засоби автоматизованого розроблення ПЗ). Це широкий спектр програм, застосовуваних для підтримання і супроводження різних етапів створення ПЗ: аналізу системних вимог, моделювання системи, її налагодження і тестування тощо. Вони включають: редактори описів, застосовуваних для опису моделей, модулі аналізу, що перевіряють відповідність мо-

делі правилам методу, генератори звітів, що допомагають створенню документації на розроблюване ПЗ. Крім того, *CASE*-засоби можуть містити генератор коду, що автоматично генерує вихідний код програм на основі моделі системи, а також посібник користувача.

CASE-засоби, призначені для аналізу специфікацій і проектування ПЗ, іноді називають *CASE*-засобами верхнього рівня, оскільки їх застосовують на початковій стадії розроблення програмних систем. Водночас *CASE*-засоби, призначені для підтримання розроблення і тестування ПЗ, тобто налагоджувачі, системи аналізу програм, генератори тестів і редактори програм, належать до *CASE*-засобів нижнього рівня.

Крім функціональних можливостей, властивих програмним продуктам за визначенням, ці продукти мають і інші показники, що характеризують їх якість. Ці показники не випливають безпосередньо з того, які дії може виконувати програмний продукт. Вони оцінюють поводження програми під час виконання алгоритмічних дій, структуру і організацію вихідного коду програми, її документованість. Як приклад таких показників (нефункціональних) можна навести час очікування користувачем відповіді на його запит або зрозумілість програмного коду.

Зазвичай множина цих показників або характеристик залежить від типу програмної системи. Наприклад, банківська система повинна бути захищеною, інтерактивна гра – чутливою до дій користувача-гравця, а система телефонних перемикань надійною і т. ін. Але ці специфічні показники, як і множину інших подібних характеристик, можна узагальнити у вигляді показників якісних програмних систем (табл. 1.2).

Таблиця 1.2

Основні показники якісного ПЗ

Показник	Опис
Зручність супроводження	ПЗ має бути придатним до удосконалення у відповідь на зміну вимог замовника або користувача
Надійність	Можливі збої в роботі системи не призведуть до фізичних або економічних збитків
Ефективність	Робота не повинна спричиняти великі витрати системних ресурсів
Зручність використання	ПЗ має бути зручним в експлуатації і не потребувати надмірних зусиль користувача того рівня, на який його розраховано

1.6. Супроводження, керування інженерією програмного забезпечення

Великі і складні системи мають дуже тривалий термін життя, протягом якого вони вдосконалюються через виправлення помилок у вихідних системних вимогах, а також урахування нових вимог. Обчислювальні компоненти систем замінюються новими, більш продуктивними компонентами. Організації, що експлуатують систему, можуть бути реорганізовані і, отже, використовувати систему іншим чином, ніж передбачалося спочатку. Може змінитися зовнішнє оточення системи, що також потребує внесення в нього змін.

Необхідність еволюції систем, як і ПЗ, зумовлено такими факторами:

– «моральним старінням» існуючого ПЗ;

– потребою внесення змін майже у всі підсистеми у разі внесення змін тільки в одну підсистему, що обов'язково впливає на продуктивність або поводження інших підсистем;

– неефективністю рішень, прийнятих на початковому етапі проектування, а отже, змінами самої системи;

– зростанням витрат на модифікацію у міру збільшення «віку» системи.

Унаслідок зростаючої залежності суспільства від систем різних типів значно більше зусиль спрямовується для вдосконалення існуючих систем, ніж для розроблення нових.

Виведення з експлуатації означає вилучення системи з її оточення після закінчення терміну експлуатації. Часто це не призводить до ускладнень. Але деякі системи можуть містити матеріали, потенційно небезпечні для навколишнього середовища. Системотехніки повинні передбачити процедуру виведення такої системи з експлуатації ще на етапі її проектування. Наприклад, використовувані в системі токсичні хімічні сполуки укладають у герметичні контейнери, кожний з яких можна видаляти із системи як єдиний елемент для наступної утилізації.

Деінсталяція ПЗ не створює фізичних проблем. Разом з тим деякі програмні компоненти можуть бути інкорпорованими в системи, потрібні для деінсталяції ПЗ; наприклад, якщо ПЗ використовувалося для моніторингу стану інших системних компонентів.

Після виведення системи з експлуатації деякі її компоненти можуть використовуватися в інших системах**.** Якщо їх деінстальовано

і вони повернулися в організацію, то їх можна застосовувати у будь-яких інших системах. Ці дані часто мають велику вартість.

Замовниками складних обчислювальних систем зазвичай є великі організації, наприклад уряд, міністерства оборони, внутрішніх справ, надзвичайних ситуацій. Такі системи можна купити як єдине ціле, як окремі частини, які потім інтегруються в створювану систему, можна спроектувати систему і розробити за окремим замовленням. Для великих систем процес вибору одного із цих варіантів може тривати кілька місяців або навіть років. Придбання системи – це найбільш оптимальний для організації шляху її купівлі і вибору її найкращого постачальника.

Процес придбання системи повністю підлягає процесу системотехніки. Завжди дешевше купити систему, ніж її розробити. Архітектура системи необхідна для того, щоб визначити, які її підсистеми можна купити, а які необхідно розробляти.

Великі системи зазвичай складаються із придбаних компонентів і компонентів, спеціально створених для цієї системи. Це одна з передумов, що потребує включення програмних компонентів до складу систем – «склеювання» її у єдине ціле (причому ефективно працююче) окремо існуючих апаратних компонентів. У потребі розроблення такого «клею» криється причина того, що економія від застосування придбаних компонентів не така велика, як очікується.

Етапи процесу придбання як готових систем, так і розроблюваних за замовленням показано на рис. 1.7.

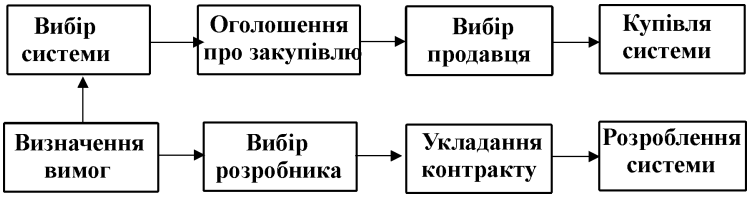


Рис. 1.7. Етапи процесу придбання системи

Наведемо властивості процесу.

1. Розроблені компоненти не задовольняють точно всі системні вимоги, унаслідок чого їх потрібно адаптувати відповідно до цих компонентів. Більш того, зазвичай виникає потреба вибору між системними вимогами і властивостям придбаної системи. Най-

частіше нехтують системними вимогами, що впливають на інші підсистеми.

2. Якщо система розробляється на замовлення, специфікація вимог є основою контракту на систему. Таким чином, специфікація має таку саму правову силу, як і інша технічна документація.

3. Після вибору розробника системи в контракті з ним необхідно обумовити можливості внесення змін у вимоги, хоча це може змінити вартість системи.

Більшість апаратних підсистем і багато програмних підсистем (таких, як системи керування базами даних) не розробляються спеціально для включення до складу великих систем. Часто в них убудовуються вже готові системи.

Далеко не всі організації мають можливості для проектування, виробництва і тестування всіх компонентів складних більших систем. Організація-розробник системи, яку називають провідною або головним підрядником, може містити контракти на розроблення окремих підсистем з іншими субпідрядниками (рис. 1.8).

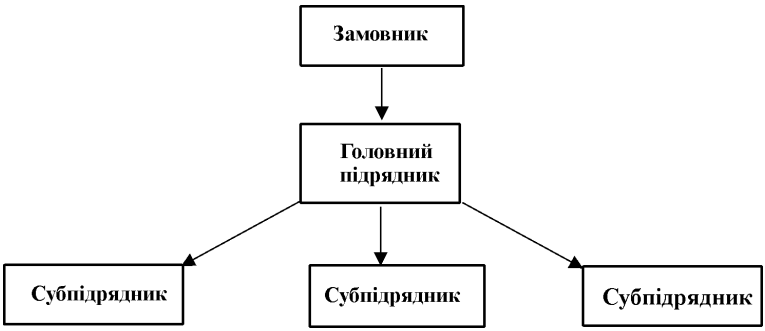


Рис. 1.8. Модель підпорядкованості підрядник–субпідрядник

Модель «підрядник–субпідрядник» мінімізує кількість організацій, що беруть участь у реалізації контракту. Субпідрядники розробляють і приводять частини системи у відповідність зі специфікацією, наданою провідним підрядником.

Після завершення робіт субпідрядниками система складається з окремих частин провідним підрядником. Готова система поставляється замовнику (покупцю). Залежно від умов контракту замовник може надати головному підряднику вільний вибір субпідрядників

або рекомендувати обирати субпідрядників зі заздалегідь підготовленого списку.

Структура витрат на створення ПЗ. Точна структура витрат на створення ПЗ істотно залежить від процесів, використовуваних для розроблення ПЗ, а також від типу розроблюваного програмного продукту. Якщо припустити загальну вартість створення ПЗ за 100 одиниць, то розподіл вартостей окремих етапів виробництва може мати такий вигляд, як на рис. 1.9–1.11.

Рис. 1.9. Розподіл вартостей окремих етапів створення ПЗ

Рис. 1.10. Витрати у разі еволюційного підходу до розроблення ПЗ

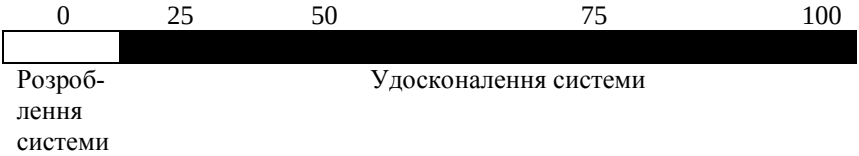


Рис. 1.11. Витрати на розроблення та удосконалення ПЗ

До вартості створюваного ПЗ також можуть включатися витрати на його модернізацію після початку експлуатації програмного продукту. Для багатьох програмних систем витрати на вдосконалення системи можуть перевищувати вартість розроблення у три-чотири рази.

Відзначимо, що часто вартість етапу складання і тестування перевищує вартість етапу безпосередньо розроблення ПЗ. Наприклад, на рис. 1.9, 1.10 зображено структуру витрат, за якою на тестування програмної системи тратиться приблизно 40 % від загальної вартості витрат. Для деяких критичних систем ця стаття витрат може перевищувати 50 %.

За еволюційного підходу до розроблення ПЗ майже неможливо чітко розмежувати етапи створення специфікації, проектування і розроблення ПЗ. Тут залишається окремий етап розроблення специфікації, оскільки загальна специфікація вищого рівня розробляється ще до початку створення програмного продукту. Створення специфікації нижнього рівня, проектування, реалізація, складання і тестування ПЗ за такого підходу виконуються одночасно на етапі розроблення програмної системи. Утім цей підхід потребує виконання окремого етапу тестування системи після закінчення початкового етапу її розроблення.

Структура витрат на створення замовленого ПЗ (тобто коли вимоги до системи встановлюються замовником і розроблення ПЗ виконується за контрактом) приблизно така сама, як показано на рис. 1.9, але вартість різних етапів створення програмного продукту може значно розрізнятися. Це стосується, зокрема, програм, розроблюваних для персональних комп'ютерів. Як правило, таке ПЗ створюється на підставі еволюційного підходу з використанням уже готового ескізу специфікації. Тому вартість розроблення вимог до ПЗ відносно низька. Такі програмні продукти призначені для роботи на різних комп'ютерних платформах, що істотно підвищує витрати на тестування систем. Типову структуру витрат на створення такого ПЗ показано на рис. 1.12.

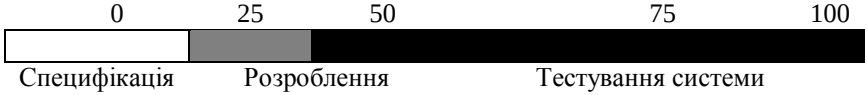


Рис. 1.12. Структура витрат на створення замовленого ПЗ

Вартість модернізації загальних програмних продуктів (тобто тих, які продаються на відкритому ринку програм) підлягає оцінюванню. У багатьох випадках проводиться невелика формальна модернізація. Зазвичай з початком реалізації створеного програмного продукту починається робота з його наступною версією. Проте виходячи з вимог маркетингу важливіше подати нову версію як новий (але сумісний зі старою версією) програмний продукт, а не як модифіковану версію того продукту, який користувач уже купив. Тому вартість модернізації ПЗ не розраховується окремо, як це робиться у разі модернізації замовлених програмних продуктів, а входить у вартість розроблення наступної версії програмної системи.

Структура витрат на створення систем для електронної комерції в Інтернет зазвичай відрізняється від витрат на проектування. У таких системах замість створення програмних модулів, що керують інформацією, використовують готове ПЗ, а основні витрати припадають на розроблення інтерфейсів користувача.

Проблеми, що найчастіше потребують вирішення:

– спадкування раніше створеного ПЗ;

– зростаюча різнорідність програмних систем;

– зменшення часу на створення ПЗ.

Ці проблеми взаємопов'язані. Наприклад, для їх вирішення в ситуації, коли необхідно швидко розробити на основі існуючої системи її проміжний варіант, потрібні нові засоби і технології, що ґрунтуються на кращих методах сучасної інженерії ПЗ.

Як і будь-які інші професіонали, фахівці із ПЗ повинні погодитися, що до них ставляться більше вимог, ніж необхідність мати той або інший професійний рівень. Вони працюють у певному правовому і соціальному оточенні. Галузь інженерії ПЗ, як і будь-яка інша сфера людської діяльності, має обмеження у вигляді місцевого, національного і міжнародного законодавства. Тому фахівці із ПЗ повинні брати на себе певні етичні і моральні зобов'язання, щоб стати дійсними професіоналами.

Не потребує зайвих пояснень твердження, що фахівці повинні бути чесними і благородними людьми. Вони не повинні використовувати професійні навички і можливості для діяльності, що дискредитує професію фахівця із ПЗ. Утім вимоги до фахівців не обмежуються лише моральними або юридичними приписами; вони охоплюють значно більше професійних зобов'язань. Зокрема:

– конфіденційність – недопущення розголошення відомостей, що складають «комерційний інтерес»;

– компетентність – рівень компетенції фахівця має відповідати роботі, що йому доручається;

– порушення прав інтелектуальної власності роботодавця не є припустимим;

– зловживання комп'ютером – фахівець не повинен використовувати свій професійний рівень для пошкодження комп'ютерів інших людей.

У розробленні подібних етичних зобов'язань велика роль належить професійним суспільствам та інститутам. Такі організації, як

Асоціація з обчислювальної техніки (*Association for Computing Machinery, ACM*), Інститут інженерів з електротехніки і електроніки (*Institute of Electrical and Electronics Engineers, IEEE*) і Британське комп'ютерне співтовариство (*British Computer Society*) опублікували кодекс професійного поводження, або етичний кодекс. Члени цих організацій беруть на себе зобов'язання додержуватися кодексу. Правила поведінки із цього кодексу ґрунтуються на загальнолюдських етичних нормах.

Роль обчислювальної техніки зростає у діловій сфері, промисловості, медицині, освіти, сфері розваг і суспільстві в цілому. Інженерія ПЗ безпосередньо або за допомогою технологій робить внесок в аналіз і створення специфікації, проектування, розроблення, сертифікацію, підтримання і тестування ПС. Відповідно до своєї ролі у створенні ПС фахівці із ПЗ повинні дотримуватися кодексу етики в професійній діяльності.

Кодекс містить вісім принципів поводження і прийняття рішень фахівцями з ПЗ, пов'язаних із практичною діяльністю, самонавчанням, керуванням, керівництвом, а також навчанням. Принципи визначають етику відносин між окремими особистостями, а також між групами і організаціями, що поділяють ці принципи. Статті кожного принципу описують конкретні зобов'язання, що визначають ці відношення. Джерела цих зобов'язань – людські якості фахівців із ПЗ, особлива моральна педантичність, властива людям, що займаються інженерією ПЗ, і є унікальною складовою практичної діяльності фахівців, що реалізують методи і технології інженерії ПЗ. Кодекс пропонує зобов'язання як приписи, що припускають неодмінне виконання, і як високі цілі, до яких повинен прагнути фахівець із ПЗ.

Контрольні запитання і завдання

1. Як співвідносяться ПЗ і системотехніка?
2. Чи слід розуміти під інженерією ПЗ написання програм мовами високого рівня?
3. Яка відмінність між розробленням програмної системи та розробленням технічної?
4. Подайте у вигляді складної системи принтер, *DVD*-привід, моноплату комп'ютера.
5. Із чого починається розроблення програмного продукту?
6. Які складові має технологічний процес створення ПЗ?

7. Які складові має технологічний процес купівлі ПЗ?
8. Яка структура витрат на створення ПЗ?
9. Назвіть методи інженерії ПЗ.
10. Що розуміють під *CASE*-засобами? Які засоби вам вже відомі, якими користувалися?
11. Назвіть ознаки якісного ПЗ.
12. Що розуміють, коли кажуть «зручний», «простий», «ефективний» програмний продукт?
13. Які основні проблеми виникають перед фахівцями з ПЗ?
14. Як програміст повинен поводитися, якщо його погляди не відповідають поглядам керівництва на розроблення нового продукту?
15. У яких випадках вартість розроблення може знижуватися?

2. ЗАГАЛЬНІ ВИМОГИ ДО ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ. КЕРУВАННЯ ВИМОГАМИ

2.1. Загальне уявлення про вимоги до програмного забезпечення

Опис функціональних можливостей і обмежень, що накладаються на програмну систему, називається вимогами до цієї системи, а сам процес формування, аналізу, документування і перевірки цих функціональних можливостей і обмежень – розробленням вимог.

Термін *вимоги* до програмної системи трактується по-різному. У деяких випадках під вимогами розуміють високорівневі узагальнені твердження про функціональні можливості й обмеження системи. Інший крайній випадок – деталізований математичний формальний опис системних функцій.

Проблеми, що виникають у процесі розроблення вимог, спричиняються відсутністю чіткого розуміння розбіжності між різними рівнями вимог. Щоб розрізнити вимоги різних рівнів, використовують терміни *користувацькі вимоги* для позначення високорівневих узагальнених вимог і *системні вимоги* для деталізованого опису виконуваних системою функцій. Крім вимог цих двох рівнів, застосовується більш деталізований опис системи – проектна специфікація на систему, яка є «містком» між етапами розроблення вимог і проектування системи. Наведемо такі визначення.

1. *Користувацькі вимоги* – опис природною мовою функцій, виконуваних системою, і обмежень, що накладаються на неї.

2. *Системні вимоги* – деталізований опис системних функцій і обмежень, що іноді називають функціональною специфікацією.

3. *Проектна системна специфікація* – узагальнений опис структури програмної системи, що є основою для більш деталізованого проектування системи і наступної реалізації. Вона доповнює і деталізує специфікацію системних вимог.

Розбіжність між користувацькими і системними вимогами наведено в табл. 2.1. Користувацькі вимоги формулюють для замовника ПЗ та особи, що задає розроблення ПС, причому вони не містять детальних технічних знань із розроблюваної системи (табл. 2.2).

Таблиця 2.1

Відмінність між записами вимог у специфікації

Користувацькі вимоги	Системні вимоги
ПЗ повинне надавати доступ до зовнішніх файлів, створених в інших програмах	Користувач повинен мати можливість визначати тип зовнішніх файлів
	Зовнішні файли мають відповідати засобу, застосованому до цього типу файлів
	Зовнішні файли потрібно позначати відповідною піктограмою на дисплеї користувача
	Надати користувачу можливість самому визначати піктограму для зовнішнього файла
	Вибирати піктограми для зовнішнього файла відповідно до асоційованого з ним ПЗ

Таблиця 2.2

Користувачі з різними формами вимог

Користувацькі вимоги	Системні вимоги	Специфікація
Менеджери організації-замовника. Кінцеві користувачі системи. Спеціалісти організації-замовника. Менеджери субпідрядних організацій. Розробники системної архітектури	Кінцеві користувачі системи. Спеціалісти організації-замовника. Розробники системної апаратури. Розробники системи	Спеціалісти організації-замовника. Розробники системної апаратної частини. Розробники програмної системи

Специфікацією системних вимог користуються керівний технічний склад організації-розробника та менеджери проекту. Вона потрібна також замовнику, розробникам ПЗ і субпідрядникам з розроблення. Документ також призначений для кінцевих користувачів ПС.

2.2. Типи вимог

Вимоги до програмної системи часто класифікуються як функціональні, нефункціональні та вимоги предметної галузі.

1. *Функціональні вимоги* включають перелік сервісів, які має виконувати система, де зазначається: як система реагує на ті або інші вхідні дані, як вона поводиться в певних ситуаціях, що система не повинна робити та ін.

2. *Нефункціональні вимоги* описують характеристики системи і її оточення, а не поводження системи. Тут також може бути наведений перелік обмежень, що накладаються на дії функції, виконувані системою: часові обмеження, обмеження процесу розроблення системи, стандарти та ін.

3. *Вимоги предметної галузі* характеризують сферу експлуатації системи. Ці вимоги можуть бути функціональними і нефункціональними.

Чіткої межі між цими типами вимог не існує. Користувацькі вимоги, що стосуються безпеки системи, можна вважати нефункціональними. Однак, якщо включити в систему засоби авторизації користувача, тоді вимоги є функціональними.

Функціональні вимоги для програмних систем можна описати різними способами.

Приклад 2.1. Функціональні вимоги до бібліотечної системи, призначеної для замовлення книг і документів з бібліотек, можна визначити у такий спосіб.

1. Користувач повинен мати можливість проводити пошук необхідних йому книг та документів за всіма доступними каталожними базами даних.

2. Система повинна надавати користувачу придатний засіб перегляду бібліотечних документів.

3. Кожне замовлення повинно супроводжуватися унікальним ідентифікатором (ORDER_ID), що копіюється у формуляр користувача для постійного зберігання.

Функціональні вимоги визначають властивості, які повинна мати система. Вони описані з різним рівнем деталізації.

Багато проблем, що виникають у процесі розроблення систем, пов'язані з неточністю і «розмитістю» специфікації вимог. Розробники можуть інтерпретувати вимоги так, щоб систему було простіше реалізувати. Таке тлумачення може не збігатися з очікуваннями замовника, що спонукає до розроблення нових вимог і внесення змін у систему, що, своєю чергою, призводить до затримання здачі готової системи та її удорожчання.

У вимозі до бібліотечної системи звернемо увагу на вираз «придатний засіб перегляду документів». Бібліотечна система повинна надавати документи в будь-якому форматі. Але оскільки ця умова чітко не прописана, розробники у випадку дефіциту часу можуть пропонувати прості засоби для перегляду текстових документів і наполягати на тому, що саме таке рішення потрібне для реалізації цієї вимоги.

Специфікація функціональних вимог повинна бути комплексною і несуперечливою. Комплексність – це опис (визначення) всіх системних сервісів. Несуперечність означає відсутність несумісних і взаємовиключних визначень сервісів. На практиці для великих і складних систем украй важко розробити комплексну і несуперечливу специфікацію функціональних вимог. Причина криється частково в складності самої розроблюваної системи, а частково – у неузгоджених поглядах на те, що повинна робити система. Ця неузгодженість може не виявитися на етапі початкового формулювання вимог, для її виявлення потрібен більш глибокий аналіз специфікації. Неузгодженість системних функцій на будь-якому етапі життєвого циклу програми потребує внесення відповідних змін у специфікацію системи.

Нефункціональні вимоги безпосередньо не стосуються функцій, виконуваних системою. Вони пов'язані з інтеграційними властивостями системи, до яких належать надійність, час відповіді або розмір системи. Крім того, нефункціональні вимоги можуть визначати обмеження на систему, наприклад на пропускну здатність пристроїв уведення-виведення, або формати даних, використовуваних у системному інтерфейсі. Багато нефункціональних вимог стосуються системи в цілому, а не окремих її засобів. Це означає, що вони більш значущі і критичні, ніж окремі функціональні вимоги. Поми-

лка, допущена у функціональній вимозі, може знизити якість системи, а помилка в нефункціональних вимогах – зробити систему непрацездатною.

Нефункціональні вимоги відображають користувацькі потреби; при цьому вони ґрунтуються на бюджетних обмеженнях, ураховують організаційні можливості компанії-розробника та можливість взаємодії розроблюваної системи з іншими програмними і обчислювальними системами, а також такі зовнішні фактори, як правила техніки безпеки, законодавство про захист інтелектуальної власності та ін. Класифікацію нефункціональних вимог показано на рис. 2.1.

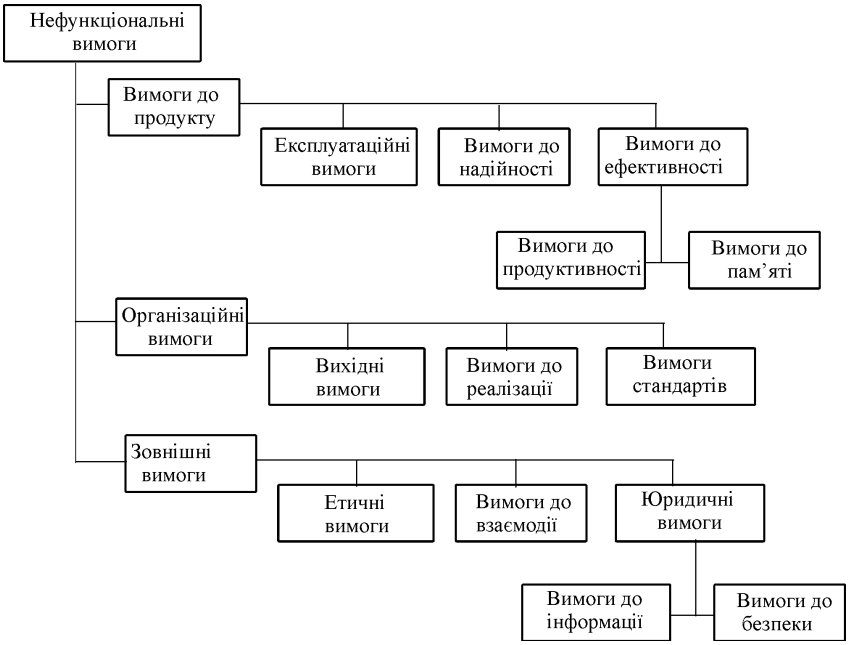


Рис. 2.1. Класифікація нефункціональних вимог

Нефункціональні вимоги, показані на рис. 2.1, поділено на три великі групи.

1. *Вимоги до продукту* – описують експлуатаційні властивості програмного продукту. Сюди належать вимоги до продуктивності системи, ємності необхідної пам'яті, надійності, підтримувані апаратні платформи та зручність експлуатації.

2. *Організаційні вимоги* – відображають політику й організаційні процедури замовника та розробника ПЗ. Вони включають стандарти розроблення програмного продукту, вимоги до реалізації ПЗ (тобто до мови програмування та методів проектування), вихідні вимоги, які визначають терміни виготовлення програмного продукту, і супутню документацію.

3. *Зовнішні вимоги* – ураховують фактори, що визначають взаємодію проектованої системи з іншими системами, юридичні вимоги, включення яких гарантує, що система буде розроблятись і функціонувати в межах існуючого законодавства, а також етичні вимоги. Останні повинні забезпечувати прийнятність системи для користувачів або замовника.

Приклад 2.2. Запис нефункціональних вимог.

Вимоги до продукту: система повинна включати підсистеми, що здійснюють обмін інформацією та оброблення заявок на безготівкові, готівкові, гривневі і валютні платежі бухгалтерією і фінансовою службою організації, а також забезпечують підсистеми, що виконують завдання з підтримання спільної роботи всіх її складових.

Організаційні вимоги: адміністратори програмної системи повинні мати кваліфікацію «інженер» і обов'язкові навички адміністрування мережі на основі операційної системи *Microsoft Windows* як мінімум версії 7.

Зовнішні вимоги: система повинна забезпечувати збереження інформації у разі таких подій:

– відмови обладнання робочої станції у випадку зберігання даних на сервері;

– вимкнення живлення на сервері баз даних;

– відмови ліній зв'язку;

– відмови апаратури сервера (процесора, нагромаджувача на жорсткому диску).

2.3. Системні вимоги

Основна проблема відпрацювання нефункціональних вимог полягає в тому, що їх виконання важко перевірити. Це простота експлуатації, можливість відновлення після збоїв або швидка відповідь на запити користувача. Їх реалізація може виявитися складною для системних розробників, оскільки вони нечітко сформульовані і відкривають простір для різних тлумачень.

Приклад 2.3. Системна мета, яку не можна перевірити.

Система повинна бути простою в експлуатації для досвідченого оператора і кількість його помилок зводити до мінімуму.

Приклад 2.4. Запис нефункціональної вимоги, який можна перевірити.

Досвідченому оператору повинні бути доступні всі системні функції після двох годин навчання роботи з проектованою системою. Після навчання середня кількість помилок оператора не повинна бути більшою, ніж дві за робочий день.

Нефункціональні вимоги необхідно виражати через кількісні показники, які можна об'єктивно вимірювати. Показники, за допомогою яких можна специфікувати нефункціональні системні властивості, наведено в табл. 2.3.

Таблиця 2.3

Кількісні показники для нефункціональних вимог

Показник	Одиниці вимірювання
Швидкість	Кількість виконаних транзакцій за секунду, час реакції на дії користувача, час відновлення екрана
Розмір	Кілобайти, кількість модулів пам'яті
Простота експлуатації	Час навчання персоналу, кількість статей у довідковій системі
Надійність	Середня тривалість часу між двома послідовними проявами помилок у системі, імовірність виходу системи з ладу, коефіцієнт готовності системи
Стійкість до збоїв	Час відновлення системи після збою, відсоток подій, що призводять до збоїв, імовірність псування даних у разі збоїв
Сумісність	Відсоток машиннозалежних операторів, кількість машиннозалежних підсистем

На практиці виразити нефункціональні вимоги за допомогою кількісних показників дуже важко. Часто замовник ПЗ не може оформити своє бачення майбутньої системи за допомогою вимог, виражених кількісними показниками. Деякі системні вимоги, наприклад зручність супроводження, взагалі не можна виразити через кількісні показники. Крім того, витрати на об'єктивне вимі-

рювання кількісних нефункціональних вимог можуть виявитися вкрай високими. Тому часто документ, що специфікує вимоги до системи, має містити опис системних цілей разом із чітко сформульованими вимогами.

Нефункціональні вимоги часто конфліктують з іншими вимогами. Функціональні і нефункціональні вимоги в документі, що описують вимоги до системи, поміщують у різні розділи. Якщо нефункціональні вимоги помістити окремо від функціональних, буде важко відстежити взаємозв'язки між ними. Якщо зібрати всі вимоги в єдиний список, буде складно аналізувати функціональні і нефункціональні вимоги окремо і визначати вимоги, що ставляться до системи в цілому. Вид подання вимог в одному документі також істотно залежить від типу системи. Вимоги, що описують інтеграційні властивості системи, мають бути виділені. Для цього їх поміщають в окремий розділ або будь-яким іншим способом відокремлюють.

Вимоги предметної галузі відображають умови, у яких буде експлуатуватися програмна система. Їх можна подати у вигляді:

- нових функціональних вимог;
- обмежень уже сформульованих функціональних вимог;
- указівок до виконання обчислень системою.

Ці вимоги дуже важливі, оскільки відображають ту предметну галузь, де буде використовуватися система, що проектується. Невиконання вимог предметної галузі може призвести до виходу системи з ладу.

Приклад 2.5. Вимоги до електронної бібліотеки.

1. Користувацький інтерфейс, що надає доступ до всіх бібліотечних баз даних, повинен бути уніфікований та ґрунтуватися на стандарті Z39.85-2001 (USA).

2. *Для забезпечення авторських прав деякі документи мають бути отримані із системи відразу після їх замовлення. Для цього, залежно від бажання користувача, ці документи можуть бути роздруковані або на локальному системному сервері, або на мережевому принтері.*

Перша вимога – обмеження системної функціональної вимоги. Вона вказує, що користувацький інтерфейс до бази даних повинен бути реалізований згідно з бібліотечним стандартом. Друга вимога є зовнішньою і спрямована на виконання закону про авторські пра-

ва, застосовуваного до бібліотечних матеріалів. Із неї випливає, що система повинна мати принтер, застосовуваний автоматично для деяких типів бібліотечних документів.

Вимоги цього типу використовують мову і позначення, властиві предметній галузі, що ускладнює їх розуміння розробниками ПЗ. Унаслідок цього вимоги предметної галузі не завжди виконуються так, як мають на увазі замовники програмної системи.

Користувацькі вимоги до системи повинні описувати функціональні і нефункціональні системні вимоги так, щоб вони були зрозумілі навіть користувачу, який не має спеціальних технічних знань. Ці вимоги повинні визначати лише зовнішнє поводження системи, уникаючи визначення структурних характеристик системи. Їх пишуть природною мовою з використанням наочних і зрозумілих таблиць та діаграм.

Під час опису вимог природною мовою можуть виникати різні проблеми.

1. *Нечіткість викладу:* багатослівний і складний для розуміння текст.

2. *Змішання вимог.* Користувацькі вимоги не мають чіткого поділу на функціональні та нефункціональні, на системні цілі та проектну інформацію.

3. *Об'єднання вимог.* Декілька різних вимог до системи можуть описуватися єдиною користувацькою вимогою.

Приклад 2.6. Вимога до бази даних для середовища програмування.

База даних повинна підтримувати генерацію і керування конфігурацією об'єктів; згруповані об'єкти можуть мати вигляд окремих об'єктів. Засіб керування конфігурацією має надавати можливість доступу до об'єктів, що входять до складу груп, за допомогою їх неповних імен.

У документі, що містить вимоги до системи, бажано відокремлювати користувацькі вимоги від системних вимог. Змішування вимог різних типів демонструє приклад 2.7.

Приклад 2.7. Формулювання користувацької вимоги.

Точне позиціонування структурних елементів схеми здійснюється за допомогою сітки, параметри якої можуть задаватися у сантиметрах або дюймах на панелі керування з використанням спеціальної опції. За замовчуванням сітка не відображається. Сітка може бути виведена на екран або схована в будь-який мо-

мент сесії редагування, також у будь-який момент часу є можливість переведення із сантиметрів у дюйми і навпаки. Крок сітки має бути регульованим, щоб відповідати розміру схеми.

Ця вимога містить не менше, ніж три різні вимоги.

1. *Концептуальна функціональна вимога* – система редагування повинна мати можливість відображати сітку. Це основна причина появи цієї вимоги.

2. *Нефункціональна вимога* – дає детальну інформацію про те, у яких одиницях вимірюватиметься крок сітки (у сантиметрах чи дюймах).

3. *Нефункціональна користувацька вимога* – ставиться до інтерфейсу: як користувач може відобразити або сховати сітку.

Описана вимога містить і іншу інформацію, зокрема необхідну для ініціалізації системи. У вимозі зазначається, що за замовчуванням сітку вимкнено. Проте не йдеться, які одиниці вимірювання обрано за замовчуванням. Користувач може вибирати розмір сітки або в сантиметрах, або в дюймах, але не вказується, що він може змінювати крок сітки.

Коли користувацька вимога містить так багато інформації, то розуміння ускладнюється й обмежується вільний пошук рішення поставленого користувачу завдання. Користувацькі вимоги повинні описувати основні можливості системи.

Приклад 2.8. Опис засобу відображення сітки.

Редактор повинен мати засіб виведення на екран сітки, що складається з паралельних горизонтальних і вертикальних ліній і відображатися у вигляді фону на екрані редактора. Сітка – пасивний елемент, що полегшує вирівнювання користувачем структур елементів схем.

***Обґрунтування**. Сітка має допомагати користувачу створювати акуратну схему із правильно розміщеними елементами. Хоча активна сітка (така, у якій елементи «прив'язуються» до вузлів сітки) також може бути корисною, вона не виконує точного позиціонування елементів. Користувач повинен мати можливість самостійно визначати положення елементів схеми.*

У цьому прикладі показано переписану користувацьку вимогу, у якій сфокусовано увагу лише на самому засобі відображення сітки без деталізації його властивостей.

Обґрунтування вимоги допомагає розробникам зрозуміти, чому цю вимогу включено до специфікації і якою мірою вона надалі

може змінитися. Наприклад, в обґрунтуванні вимоги на засіб відображення сітки зазначається, що також може бути корисною активна сітка, де елементи схеми автоматично «прив'язуються» до вузлів сітки. Однак тут перевагу свідомо віддано пасивній сітці. Якщо надалі виникне потреба змінити дану вимогу, то із цього обґрунтування буде видно, що варіант пасивної сітки обрано навмисно, а не він з'явився на етапі реалізації системи.

Рекомендовані правила щодо написання користувацьких вимог.

1. Розробити стандартну форму для запису вимог і неухильно її дотримуватися. Стандартна форма запису зменшує нечіткість у формулюванні вимог і дозволяє легко їх перевірити.

2. Розрізнювати обов'язкові й описові вимоги. Тут обов'язковим є додавання нових структурних елементів, описовим – опис послідовності дій користувача. Описова вимога не є абсолютно
необхідною для реалізації конкретної вимоги і в разі потреби може бути змінена.

3. Використовувати різний шрифт для виокремлення ключових частин вимоги.

4. Уникати комп'ютерного жаргону. Це не забороняє використання технічних термінів тієї предметної галузі, для якої розробляється ПЗ.

2.4. Документування системних вимог

Системні вимоги визначають функції системи, не показуючи механізму їх реалізації. Проте для повного опису системи потрібна деталізована інформація про неї, яка по можливості повинна включати всю інформацію про системну архітектуру. На те існує низка причин.

1. Це допомагає структурувати специфікацію вимог. Системні вимоги повинні описувати підсистеми, з яких складається розроблювана система.

2. Розроблювана система повинна взаємодіяти з уже існуючими системами.

3. Зовнішні системні вимоги можуть бути умовами використання спеціальної архітектури для розроблюваної системи.

Специфікація системних вимог теж пишеться природною мовою. Її застосування означає однаковість розуміння тексту спе-

цифікації всіма учасниками процесу розроблення. Для цього використовують методи опису, які структурують специфікацію і зменшують розмитість визначень. Ці методи наведено в табл. 2.4.

Таблиця 2.4

Способи оформлення специфікацій вимог

Подання	Опис
Природною мовою	Використання стандартних форм і шаблонів
Спеціальними мовами	Використання спеціальних структурованих мов, подібних до мов програмування, коли специфікація вимог будується на основі обраної операційної моделі системи
Графічне	Подання функціональних вимог діаграмами і блок-схемами з текстовими поясненнями
Математичне	Опис, що ґрунтується на математичних концепціях, таких, як теорія кінцевих автоматів або теорія множин. Це формалізований однозначно і позбавлений двозначності опис системних вимог

Структурована мова специфікацій – це скорочена форма написання природною мовою специфікації вимог.

Перевагою підходу до написання специфікацій є збережуваність виразності і зрозумілості природної мови і водночас формалізація опису вимог.

Структурованість мови проявляється у використанні спеціальної термінології, а також шаблонів для опису системних вимог. Цей стиль опису може включати конструкції, запозичені з мов програмування.

Опис системних вимог часто розробляється за допомогою спеціальних форм і шаблонів. Вони повинні враховувати основу побудови специфікації: об’єкти, керовані системою, функції, виконувані системою, або події, оброблювані системою. Форму для специфікації показано в прикладі 2.9.

Приклад 2.9. Специфікація системної вимоги з використанням стандартної форми.

Функція. Додавання структурних елементів в існуючу схему.

Опис. Користувач вибирає тип структурного елемента і його місце розташування. Після вставлення у схему структурний елемент стає виділеним (поточним структурним елементом). Користу-

вач визначає місце розташування елемента переміщенням курсора по схемі.

Вихідні дані. Тип елемента, позиція елемента, ідентифікатор схеми.

Джерела вхідних даних. Тип елемента і позиція елемента задаються користувачем, ідентифікатор схеми отримується з бази даних проекту.

Вихідні дані. Ідентифікатор схеми.

Призначення. База даних проекту. Ідентифікатор схеми повертається в базу даних проекту по завершенні виконання функції.

Виконання функції. Для її виконання потрібна схема, що відповідає вхідним ідентифікаторам схеми.

Передумова. Схема відкрита і відображається на екрані користувача.

Пост-умова. Схема змінюється лише за умови вставлення нового структурного елемента.

Побічні ефекти. Немає.

Стандартні форми, використовувані для специфікування функціональних вимог, повинні містити таку інформацію:

1) опис функції або об'єкта;

2) опис вхідних даних та їх джерел;

3) опис вихідних даних із вказівкою пункту їх призначення;

4) указівка, потрібна для виконання функції;

5) опис попередніх умов (передумов), які мають виконуватися перед викликом функції, і опис заключної умови (пост-умов), що виконуються після завершення функції;

6) опис побічних ефектів (якщо такі є).

Використання структурованої мови нівелює деякі проблеми, властиві специфікаціям, написаним природною мовою, оскільки знижує «варіабельність» специфікації і більш ефективно її структурує. Разом з тим деяка «розмитість» визначень і описів у специфікації залишається. Альтернативою використанню структурованої природної мови може бути спеціальна мова опису, що вирішує проблему нечіткості опису вимог. Утім для неспеціаліста така специфікація виявиться важкою для читання і розуміння.

Специфікація системних вимог – це офіційний документ для розробників програмної системи, який містить користувацькі вимоги і деталізований опис системних вимог. Може бути реалізована у вигляді однорідного документа. Якщо загальна кількість вимог велика, то деталізовані системні вимоги можуть бути викладені в

окремих томах основного документа. Категорії співробітників, які мають доступ до специфікації, наведено в табл. 2.5.

Таблиця 2.5

Призначення специфікації за групами користувачів

Замовник	Керівництво розробника	Розробник	Тестувальники	Інженери з експлуатації
Визначає вимоги. Перевіряє специфіковані вимоги на відповідність вимогам замовленої системи. Уносить зміни в специфікацію	Використовує специфікацію для розрахунку системи. Планує процес розроблення системи	Користується в процесі розроблення системи	Використовують під час розроблення тестів	Організовують правильну експлуатацію системи. Виконують поточний ремонт

Умови, яким повинна відповідати специфікація програмної системи:

– описувати лише зовнішнє поводження системи;

– указувати обмеження, що накладаються на процес реалізації системи;

– передбачати можливість унесення змін у специфікацію;

– бути довідковим засобом у процесі супроводження системи;

– відображати весь життєвий цикл системи;

– передбачати реакцію системи і групи супроводження на непередбачені (позаштатні) ситуації.

Цей процес затверджується стандартом, де припускається така структура специфікації.

1. Вступ.

1.1. Мета документа.

1.2. Призначення програмного продукту.

1.3. Визначення, акроніми і абревіатури.

1.4. Огляд специфікації.

2. Загальний опис.

2.1. Опис програмного продукту.
2.2. Функції програмного продукту.
2.3. Користувацькі характеристики.
2.4. Загальні обмеження.
2.5. Обґрунтування, припущення і допущення.

3. Специфікація вимог охоплює функціональні, нефункціональні та інтерфейсні вимоги. Тут можуть бути документовані зовнішні інтерфейси, описані функціональні можливості системи, наведені вимоги, що визначають логічну структуру баз даних, обмеження, що накладаються на структуру системи, описані інтеграційні властивості системи або її якісні характеристики.

4. Додатки.

5. Покажчики.

Для написання специфікації необхідно враховувати стандарти, прийняті в організації-розробнику ПЗ.

Якщо розроблюється ПЗ, яке є частиною великої системи, що складається із взаємодійних апаратних і програмних систем, тоді виклад вимог має бути детальним. У цьому випадку специфікація може мати значний обсяг і містити більше розділів, ніж передбачено стандартом.

Розроблення вимог – це процес, що включає заходи, необхідні для створення і затвердження документа, що містить специфікацію системних вимог. Розрізняють чотири основні етапи процесу розроблення вимог: аналіз технічної реалізовності створення системи, формування і аналіз вимог, специфікування вимог і створення відповідної документації, а також атестація цих вимог (рис. 2.2).

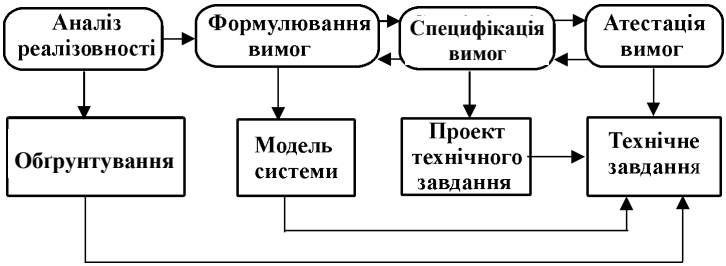


Рис. 2.2. Процес розроблення вимог

Оскільки в процесі розроблення системи через різноманітні причини вимоги можуть змінюватися, то процес керування змінами системних вимог є необхідною складовою частиною їх розроблення.

2.5. Аналіз реалізовності вимог

Для нових програмних систем процес розроблення вимог повинен починатися з аналізу реалізовності. Початком такого аналізу є загальний опис системи і її призначення, а результатом аналізу – звіт, який повинен містити чітку рекомендацію щодо продовження чи завершення процесу розроблення вимог проектованої системи. Інакше кажучи, аналіз реалізовності має висвітлити:

1) чи відповідає система загальним, а також бізнес-цілям організаціям замовника і розробника;

2) чи можна реалізувати систему, використовуючи існуючі на даний момент технології і не виходити за межі заданої вартості;

3) як проектована система узгоджується із системами, які вже експлуатуються.

Якщо система не відповідає бізнес-цілям, то її розроблення недоцільне. Є організації, які розробляють системи, що не відповідають їх цілям, або вони не розуміють ці цілі, або на них впливають політичні чи суспільні фактори.

Виконання аналізу реалізовності включає збір і аналіз інформації про майбутню систему і написання відповідного звіту. Після оброблення зібраної інформації готується звіт про аналіз реалізовності створення системи, у якому мають бути надані рекомендації щодо продовження розроблення системи. За результатом звіту можуть бути запропоновані зміни бюджету і графіка робіт зі створення системи або поставлені більш жорсткі вимоги до системи.

2.6. Формування та аналіз вимог

Після виконання аналізу реалізовності наступним етапом процесу розроблення вимог є формування (визначення) і аналіз вимог. На цьому етапі команда розробників ПЗ працює із замовником і кінцевими користувачами системи для з'ясування галузі застосування, опису системних сервісів, визначення режимів роботи системи і її характеристик виконання, апаратних обмежень і т. ін.

Етапи процесу формування й аналізу вимог:

1. *Предметна галузь* – вивчається аналітиками, які досліджують сферу експлуатації системи.

2. *Формулювання вимог* – здійснюється особами, що формують вимоги шляхом взаємодії із замовником.

3. *Класифікація вимог* – передбачає перетворення безформного набору вимог у логічно зв'язані групи.

4. *Вирішення суперечностей* – етап, на якому визначаються і роз'вязуються суперечності формулювання різних вимог.

5. *Визначення пріоритетів* – передбачає визначення найбільш важливих вимог.

6. *Перевірка вимог* – визначає їх повноту, послідовність і несуперечність.

Як показано на рис. 2.3, процес формування й аналізу вимог циклічний.

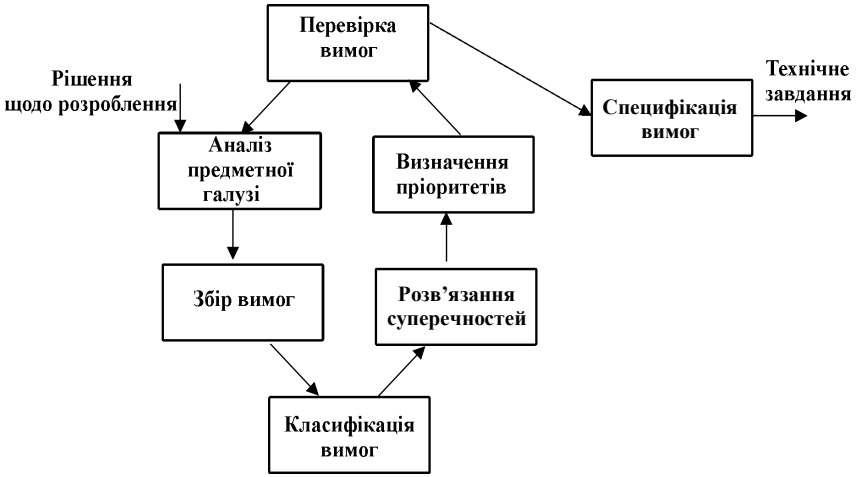


Рис. 2.3. Процес формування вимог

Цикл починається з аналізу предметної галузі і закінчується перевіркою вимог. Розуміння вимог предметної галузі збільшується на кожному циклі процесу формування вимог.

На практиці застосовують три найбільш поширені методи до формування вимог: метод опорних точок зору, сценарії й етнографічний метод. У процесі розроблення вимог можуть використовуватися також методи структурного аналізу і методи прототипування.

Універсального підходу до формування й аналізу вимог не існує. Зазвичай для розроблення вимог одночасно використовується кілька підходів.

2.7. Атестація й огляд вимог

Система ПЗ орієнтована на різні типи кінцевих користувачів. Багато осіб, що беруть участь у формуванні вимог, виражають у них власні інтереси. Наприклад, у процесі формування вимог для системи банкоматів можуть брати участь такі особи:

– клієнти банку, які користуються послугами банкоматів;

– представники інших банків, що мають взаємні домовленості з цим банком про спільне використання банкоматів;

– менеджери філій банку, що отримують інформацію із системи керування банкоматами;

– співробітники філій банку, залучені в повсякденну роботу системи банкоматів, що обробляють рекламації клієнтів;

– адміністратори баз даних, які відповідають за зв'язок банкоматів з базою даних клієнтів;

– керівники служби безпеки банку, що забезпечують захист системи банкоматів;

– відділ маркетингу банку, що використовує систему банкоматів як засіб маркетингу;

– розробники апаратних і програмних засобів, які відповідають за супровід і модернізацію апаратних та програмних засобів.

Цей перелік показує, що навіть для відносно простої системи існує багато різних точок зору, які потрібно розглянути. Різні сприйняття проблеми дозволяють розглянути її всебічно. Однак ці погляди не є повністю незалежними і зазвичай перекривають один одного, а тому можуть бути основою загальних вимог.

Підхід з використанням різних *опорних* точок зору до розроблення вимог визнає і використовує їх як основу побудови й організації як процесу формування вимог, так і безпосередньо самих вимог. Аналіз, орієнтований на різні опорні точки зору, полягає в тому, що він визнає множину поглядів і забезпечує основу для виявлення суперечності у вимогах, запропонованих різними особами.

Точки зору можна трактувати з позицій:

– *джерела інформації про системні дані.* На основі опорних точок зору будується модель створення і використання даних у системі. На цій підставі визначаються дані, які будуть створені або використані під час роботи системи, і способи оброблення цих даних;

– *структури подання*. Точки зору розглядають як особливу частину моделі системи. Наприклад, на основі різних точок зору можуть розроблятися моделі «сутність – зв'язок», моделі кінцевого автомата;

– *отримувача системних сервісів.* Точки зору є зовнішніми (відносно системи) отримувачами системних сервісів. Цей спосіб допомагає визначити дані, необхідні для виконання системних сервісів або їх керування.

На основі цього підходу розроблено метод *VORD* (*Viewpoint Oriented Requirements Definition* – визначення вимог на основі точок зору) для формування та аналізу вимог [203, 204]. Основні етапи методу *VORD* показано на рис. 2.4.

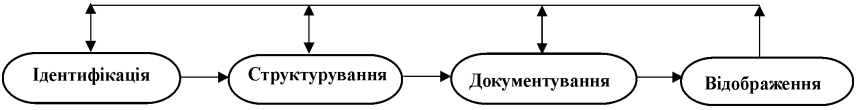


Рис. 2.4. Метод *VORD*

Ідентифікація точок зору передбачає виявлення та ідентифікацію системних сервісів. *Структурування* точок зору включає групування точок зору за ієрархією. Загальносистемні сервіси відповідають більш високим рівням ієрархії і успадковуються на нижчому рівні. Ідентифіковані точки зору і сервіси підлягають *документуванню*. Точки зору *відображуються* у системні об'єкти.

Розглянемо використання методу *VORD* на перших трьох кроках аналізу вимог для системи керування банкоматами. Банкомат має вбудоване ПЗ для керування апаратними засобами і для зв'язку із центральною базою даних банківських рахунків. Пристрій приймає запити клієнта, видає готівку, інформацію про рахунок, змінює дані в базі даних банку та ін. Банкомати одного банку можуть дозволити клієнтам інших банків використовувати якусь частину своїх сервісів (зазвичай це зняття з рахунка готівки і запит про поточний баланс рахунка).

Ідентифікація опорних точок зору проводиться методом «мозкового штурму», коли визначаються потенційні системні сервіси і

організації, які взаємодіють із системою. Знайдені точки зору мають вид діаграми з ряду кругових ділянок, що відображають можливі точки зору (рис. 2.5). Додатковими джерелами інформації для створення початкового уявлення про систему можуть бути документи, що описують призначення системи, або консультації з інженерами-програмістами, які мають досвід з попередніх проектів або з клієнтами банку.

Рис. 2.5. Діаграма ідентифікації точок зору

Наступним кроком процесу формування вимог є групування виявлених опорних точок зору (світлі кругові ділянки, рис. 2.5) за сервісами (затінені ділянки). Сервіси повинні відповідати опорним точкам зору. Але можуть бути сервіси, яким не знайдена така відповідність. Це означає, що на початковому етапі «мозкового штурму» деякі опорні точки зору не були ідентифіковані. Наприклад, для сервісів «Віддалене відновлення ПЗ» і «Контроль стану» (рис. 2.5) необхідно мати точку зору про обслуговування ПЗ. Такі самі сервіси можна співвідносити з декількома точками зору.

Точки зору також визначають вхідні дані і керувальну інформацію для сервісів. Наприклад, банкомат повинен визначати залишок грошей після видачі готівки.

На ранніх етапах процесу формування вимог ці дані та керувальна інформація ідентифікуються безпосередньо за ім’ям. Керувальну

інформацію про точки зору власника рахунка наведено в табл. 2.6, а її сервіси – у табл. 2.7. Керувальна інформація вводиться кнопками панелі банкомата, дані – зчитуються з клієнтської картки або клавіатури банкомата.

Інформація на основі точок зору використовується для заповнення форм шаблонів точок зору і організації точок зору в ієрархію спадкування.

Таблиця 2.6

Керувальна інформація

Вихідна інформація	Дані
Вибір сервісу в меню. Початок транзакції. Відміна транзакції. Кінець транзакції	Пін-код. Сума готівки. інформування

Таблиця 2.7

Сервіси власника рахунка

Клієнт	Клієнт без картки	Працівник банку
Запит пін-коду. Запит балансу. Видача готівки. Друк квитанції. *SMS*-інформування	Запит балансу. Видача готівки	Інтернет-розрахунки. Обслуговування апаратури. Платіж за кордон. Журнал транзакцій

Це дозволяє бачити загальні точки зору і повторно використовувати інформацію в ієрархії спадкування. Частину ієрархії точок зору системи банкоматів, яка включає лише сервіси, пов’язані із двома точками зору клієнта, показано на рис. 2.6.

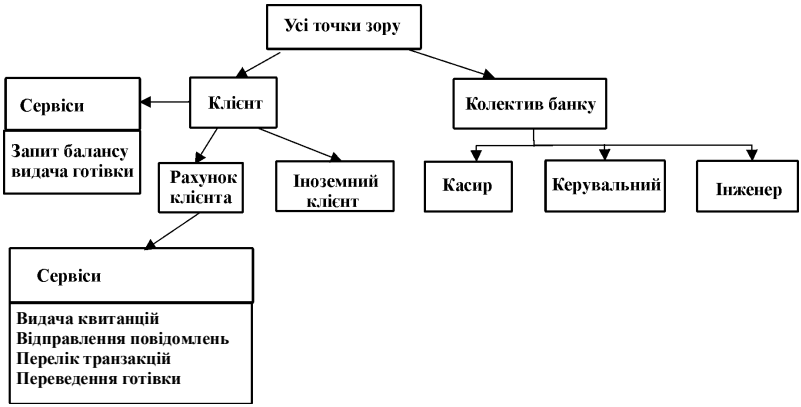


Рис. 2.6. Ієрархія точок зору

Останній крок процесу формування вимог – отримання більш детальної інформації про сервіси, використовувані сервісами даних, і керувальних даних. Ця інформація ґрунтується на міркуваннях осіб, що формують вимоги. Для цього використовується сценарний підхід.

Сценарій починається із загального опису, поступово деталізується для створення повного опису взаємодії користувача із системою. Сценарій здебільшого містить:

– опис стану системи на початку сценарію;

– опис нормального перебігу подій;

– опис виняткових ситуацій і способів їх оброблення;

– інформацію про інші дії, які можна проводити під час виконання сценарію;

– опис стану системи після завершення сценарію.

Первинний опис сценарію може бути виконаний неформально в процесі опитування осіб, що формує вимоги, альтернативою – сценарний підхід або варіанти використання.

Сценарії подій використовуються в методі *VORD* для документування поводження системи, поданого певними подіями. Кожна подія, наприклад уставлення картки в банкомат або вибір сервісу, документально підтверджується окремим сценарієм. Сценарії включають опис потоків даних, системних операцій і виняткових ситуацій, які можуть виникнути. Приклад сценарію події «Початок транзакції», що ініціюється клієнтом при вставлянні картки в банкомат, показано на рис. 2.7. Коли картка вставлена, запитується персональний ідентифікаційний номер клієнта (пін-код). Якщо картка

дійсна, вона може оброблятися банкоматом, тоді керування переходить до наступної стадії сценарію.

Рис. 2.7. Сценарій події «Початок транзакції»

На першій стадії сценарію можливі три виняткові ситуації.

– *перевищення ліміту часу очікування.* Клієнт може не встигнути ввести пін-код у відведений для введення час. Картка вертається;

– *недійсна картка.* Картка не зорієнтується і вертається;

– *утримання картки.* Картка утримується банкоматом.

Кожну виняткову ситуацію можна визначити більш детально, побудувавши окремі діаграми потоків даних і керування. Загальна діаграма сценарію також може бути постачена коментарями з додатковою інформацією, що містить опис дій, які мають початися у разі виникнення виняткової ситуації.

Варіанти використання – це методика формування вимог, що ґрунтується на сценаріях. Вони стали основою нотацій у мові моделювання *UML* для опису об’єктних моделей систем. У найпростішій формі у варіанті використання *визначено діючі особи* (в *UML* вони називаються *акторами*), тобто користувачі, залучені у взаємодію, і імена типів взаємодії. Найпростіші варіанти використання взаємодії читачів і бібліотеки ілюструє рис. 2.8, на якому показано основні елементи позначень. Діючі особи (актори) зображено у вигляді стилізованих фігурок людей, а кожний клас взаємодії подано

іменованим еліпсом. Множини варіантів використання охоплює усі можливі взаємодії, які відобразяться в системних вимогах.

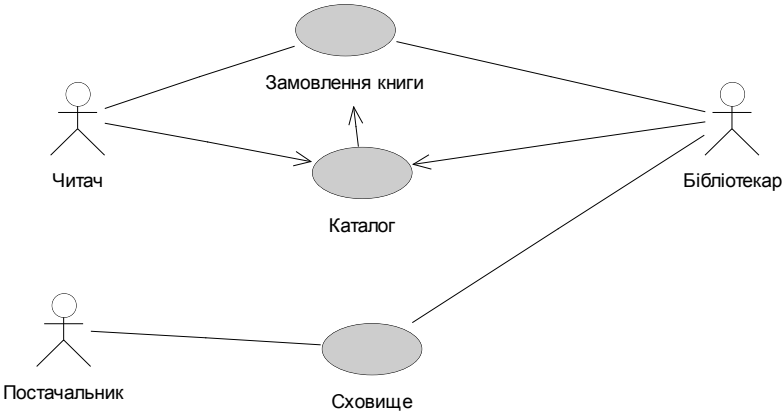


Рис. 2.8. Варіанти використання для бібліотеки

Мова *UML* передбачає спільне використання діаграм послідовностей (рис. 2.9) і варіантів використання (див. рис. 2.8), що істотно розширює інформаційні можливості графічного зображення варіантів використання. Ці діаграми показують акторів, залучених у взаємодію, системні об’єкти, з якими вони взаємодіють, і дії, які пов’язані із цими об’єктами. Взаємодії під час купівлі книг і створення каталогу бібліотеки показано на рис. 2.9.

Рис. 2.9. Схема послідовності керування каталогом

Етнографічний підхід. Відповідно до цього підходу розробник вимог заглиблюється в робітниче середовище, у якому використовуватиметься система. Його щоденна робота пов'язана зі спостереженням і протоколюванням реальних дій, виконуваних користувачами системи. Значущість етнографічного підходу полягає в тому, що він допомагає виявляти неявні вимоги до системи, які відображають реальні аспекти її експлуатації, а не формальні процеси.

Звичайним людям важко чітко описати всі аспекти виконуваної роботи, оскільки спосіб виконання часто визначається їх характером і практичним досвідом. Вони розуміють роботу, яку вони виконують, але не можуть пояснити її взаємозв'язок з іншими видами робіт, виконуваних в організації. Соціальні й організаційні фактори, які впливають на роботу, не є очевидними, але можуть бути виявлені сторонніми спостерігачами.

Етнографічний підхід до формування вимог можна об'єднати із прототипуванням (рис. 2.10). Етнографічний підхід дозволяє отримати вимоги, які враховуються в розроблюваному прототипі. Крім того, етнографічний підхід використовується для вирішення конкретних проблем прототипування й оцінювання створеного прототипу [1–3]. Етнографічний підхід дозволяє деталізувати вимоги для

критичних систем, чого не завжди можна домогтися іншими методами розроблення вимог.

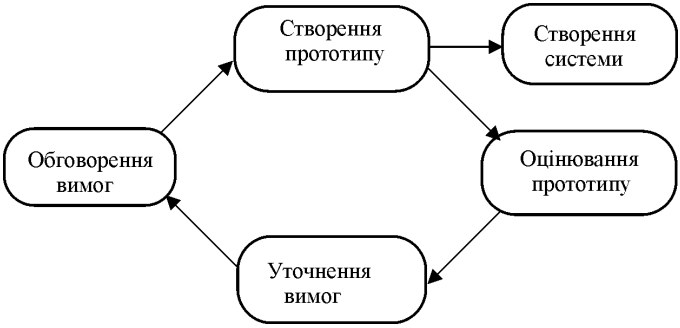


Рис. 2.10. Використання етнографічного підходу і прототипування для формування вимог

Однак, оскільки цей метод орієнтований на кінцевого користувача, він не може охопити всі вимоги предметної галузі та вимоги організаційного характеру. Тому він не є всеохопним підходом до формування вимог і має використовуватися разом з такими підходами, як аналіз варіантів використання.

Атестація повинна продемонструвати, що вимоги дійсно визначають ту систему, яку хоче мати замовник. Перевірка вимог важлива, оскільки помилки в специфікації вимог можуть призвести до перероблення системи та більших витрат, якщо будуть виявлені під час процесу розроблення системи або після введення її в експлуатацію. У процесі атестації мають виконуватися різні типи перевірок документації вимог: правильності вимог, на несуперечність, на повноту та на виконуваність.

Існує ряд методів атестації вимог, які можна використовувати спільно або кожну окремо. Наприклад, такі:

1. *Рецензування вимог.*
2. *Прототипування.* На цьому етапі прототип системи демонструється кінцевим користувачам і замовнику для того, щоб переконатися у відповідності їх потребам.
3. *Генерація тестових сценаріїв.* Якщо тести складні або їх неможливо розробити, то це означає, що вимоги важко виконати і тому їх необхідно переглянути.
4. *Автоматизований аналіз несуперечності.* Використання ін-

струментальних *CASE*-засобів перевірки несуперечності ґрунтується на побудові бази даних вимог, а потім виконується їх автоматизована перевірка. Цей процес показано на рис. 2.11. Аналізатор вимог готує звіт про виявлені суперечності.

Рис. 2.11. Автоматизований аналіз несуперечності вимог

Огляд вимог – це процес перегляду системної специфікації для знаходження неточних описів і помилок. До цього процесу залучається велика кількість осіб з боку як замовника, так і розробників. Огляд може бути неформальним і формальним. Неформальний огляд – це просте обговорення вимог з великою кількістю осіб, що беруть участь у їх формуванні. За формального огляду група розробників доводить замовнику необхідність включення кожної вимоги.

Виявлені під час огляду суперечності, помилки і недогляди у вимогах повинні бути зафіксовані документально. Потім ці документи передаються замовнику та розробникам системи для вживання відповідних заходів.

2.8. Керування вимогами

Вимоги до великих систем ПЗ змінюються у процесі їх розроблення. Оскільки під час процесу створення ПЗ розуміння розробниками поставлених перед ними завдань змінюється, що зумовлює потребу повертатися до вимог.

Керування вимогами – це процес внесення змін до специфікації вимог. Усі вимоги можна поділити на два класи.

1. *Постійні вимоги*, що стосуються безпосередньо предметної галузі, де буде експлуатуватися система.

2. *Змінювані вимоги*, які відображають зміни під час розроблення системи або після уведення її в експлуатацію.

Класифікацію змінюваних вимог наведено в табл. 2.8. Планування є першим етапом процесу керування вимогами.

Таблиця 2.8

Класифікація змінюваних вимог

Тип вимог	Опис
Змінні	Вимоги, що змінюються через зміни оточення

Неочікувані	Вимоги, що ставляться у процесі розроблення
Другорядні	Вимоги, що залежать від особливостей системи

У процесі керування потрібно відслідковувати питання, що стосуються розроблення вимог.

1. *Ідентифікація вимог.* Передбачає запобігати перетинанню вимог, що виявляється за допомогою оперативного контролю.

2. *Керування процесом унесення змін*, за якого оцінюється вплив на систему внесених змін та їх вартість.

3. *Стратегія оперативного контролю.* Визначає відношення між вимогами, а також між вимогами і проектуванням системи.

4. *Підтримання CASE-засобів.* Керування вимогами припускає оброблення великого обсягу інформації про вимоги. У цьому процесі можуть використовуватися різні інструментальні засоби, наприклад електронні таблиці або прості системи баз даних.

Оперативний контроль дозволяє виявляти пов'язані вимоги і відстежувати їх взаємовплив.

В оперативному контролі використовуються три типи інформації.

1. *Інформація про джерело вимоги* – відповідає вимогам осіб, які їх запропонували, з логічним обґрунтуванням цих вимог.

2. *Інформація про вимоги* – відповідає вимогам усередині специфікації. Ця інформація використовується для оцінювання кількості вимог, які стосуються запропонованих змін.

3. *Інформація про структуру системи* – пов'язує вимоги із системними модулями, які реалізують вимоги.

Інформацію для оперативного контролю часто подають у вигляді спеціальних матриць. Приклад простої матриці залежності вимог наведено в табл. 2.9. Символ *U* (від *use* – використання) на перетині рядка і стовпця показує, що вимога в рядку використовує засоби, визначені у вимогах, розміщених у стовпці. Символ *R* (від *re1ation* – зв'язок, залежність) означає, що існує деякий взаємозв'язок між вимогами.

Матриці оперативного контролю використовуються для керування невеликим набором вимог, вони стають громіздкими і незручними для великих систем з багатьма вимогами. Для таких систем інформацію оперативного контролю раціонально тримати в базі даних вимог, де кожна вимога залежить від інших вимог. Великі системи потребують спеціалізованих засобів підтримання.

Таблиця 2.9

Матриця оперативного контролю

Вимоги	1.1	1.2	1.3	2.1	2.2	2.3	3.1	3.2
1.1		*U*	*R*			*U*		*U*
1.2			*U*			*R*		
1.3	*R*			*R*				*U*
2.1			*R*		*U*			*U*
2.2								*U*
2.3		*R*		*U*				
3.1								*R*
3.2							*R*	

Перевага використання формального процесу керування змінами полягає в тому, що всі запропоновані зміни обробляються послідовно, при цьому можна керувати і відслідковувати внесення змін у системну специфікацію.

Процес керування змінами складається із трьох основних етапів:

1) *аналізу проблем зміни специфікації*;

2) *аналізу змін і розрахунку їх вартості*, яка оцінюється вартістю внесення зміни в специфікацію та вартістю внесення змін у структуру системи і безпосередньо в програмний код;

3) *реалізації змін* у системній специфікації, структурі системи та програмному коді.

2.9. Моделі зрілості в керуванні вимогами

Модель зрілості можливостей створення ПЗ – це еволюційна модель розвитку здатності компанії розробляти ПЗ.

У листопаді 1986 р. американський інститут *Software Engineering Institute* (*SEI*) спільно з *Mitre Corporation* почали аналізувати зрілість розроблення ПЗ. Розроблення такого аналізу викликано запитом американського федерального уряду на надання методу оцінювання субпідрядників для розроблення ПЗ. Реальна ж проблема полягала в нездатності керувати великими проектами. У багатьох компаніях проекти виконувалися зі значним запізненням і з перевищенням запланованого бюджету. Необхідно було знайти вирішення цієї проблеми.

У вересні 1987 р. *SEI* випустив короткий аналіз процесів розроблення ПЗ з описом їх рівнів зрілості, а також опитувальник, що призначався для виявлення напрямів діяльності компанії, які потребують істотних змін. Однак більшість компаній розглядали такий опитувальник як готову модель, унаслідок чого через чотири роки його перетворили в реальну модель зрілості для ПЗ (*CMM*). Першу версію *CMM* (1991 р.) у 1992 р. переглянули учасники робочої зустрічі, у якій брали участь близько 200 фахівців у галузі ПЗ.

2.9.1. Рівні зрілості в процесах розроблення програмного забезпечення

Рівень зрілості 1 (початковий). Найпримітивніший статус організації, яка здатна розробляти ПЗ. Організація не має чітко усвідомленої послідовності процесу розроблення ПЗ, і якість продукту цілком визначається індивідуальними здібностями розробників. Успіх одного проекту не гарантує успіх іншого. По завершенні проекту не фіксуються дані щодо трудовитрат і якість. Процеси першого рівня зрілості характеризуються хаотичністю, реактивністю, непередбачуваністю. Хоча організації на цьому етапі розвитку виробляють досить якісні продукти.

Рівень зрілості 2 (керований рівень). На цьому рівні описано основні процеси, їх можна використовувати неодноразово. Тобто проекти, що виконуються організацією, відповідають вимогам. Процеси керовані, вони плануються, виконуються, вимірюються і контролюються. На рівні 2 контролюються вимоги замовників і проміжні продукти, а також установлені основні практики керування проектом. Ці засоби дозволяють керувати проектом, проте дають фрагментарне уявлення про нього. Виробничий процес можна уявити послідовністю «чорних ящиків» і реальне бачення проекту можливе лише на проміжних етапах.

Рівень зрілості 3 (рівень визначеності). На цьому рівні визначено процеси, стандарти установлено в межах організації, процеси описано не на рівні окремого проекту, а на рівні всієї організації, детально описано зв’язки і залежності, що дозволяє поліпшити керування. На цьому рівні зрілості стає видимою внутрішня частина «чорних ящиків». Ця внутрішня структура відображає спосіб, застосування стандартного виробничого процесу організації.

Рівень зрілості 4 (кількісно-керований). На цьому рівні досягнуто всі цілі рівнів 1–3. Обрано субпрактики, які з використанням статистичних методів та інших кількісних технік дозволяють контролювати якість виконання процесів. Головна відмінність цього рівня від попереднього полягає в передбачуваності ефективності процесів і можливості нею (ефективністю) керувати. На рівні 4 певні процеси кількісно контролюються за допомогою відповідних засобів і технік.

Рівень зрілості 5 (постійне поліпшення, оптимізація процесів). На цьому рівні наявні точні характеристики оцінювання ефективності бізнес-процесів, що дозволяє постійно й ефективно їх поліпшувати через розвиток існуючих методів і технік та впровадження нових.

2.9.2. Розвиток моделі зрілості в керуванні процесами

CMMI – це набір моделей (методологій) удосконалення процесів в організаціях різних розмірів і видів діяльності. Вона містить набір рекомендацій у вигляді практик, реалізація яких, на думку розробників моделі, дозволяє реалізувати цілі, необхідні для повної реалізації певних сфер діяльності.

CMMI стала результатом послідовного розвитку первинної моделі (*CMM*), що поглинула ряд інших. У середині 1980-х років перед міністерством оборони США постала проблема підвищення якості розроблюваного на їх замовлення ПЗ. Оскільки кошти були бюджетні, а бюджет зазвичай плановий, крім якості розроблення, виконавець повинен виконувати замовлення в призначений термін і в межах установленого бюджету.

Проблему вирішено за допомогою створеної моделі, на відповідність якій оцінювалися всі потенційні виконавці замовлення міністерства оборони. Розроблення моделі було покладено на *Software Engineering Institute*, створений на базі *Carnegie Mellon University* (Піттсбург, штату Пенсільванія).

Для створення цієї моделі були проаналізовані ключові активності, які виконуються при розробленні ПЗ, і пов'язані з ними ризики. Аналізувалися як *best practices* – практики, які дозволили успішно уникнути або пом'якшити той чи інший ризик, так і *worst practices* – типові помилки, які призводять до зриву термінів та перевищення бюджетів. Для кожної ключової активності (або цілі) модель пропонує ряд практик, які дозволяють усунути або істотно

зменшити відповідні проектні ризики. Активності були згруповані в так звані процесні області. У 1987 р. з'явився прообраз майбутньої моделі (анкета), яка містила всього 85 процесних і 16 технологічних питань, відповіді на які і визначали оцінювану компанію до одного з п'яти рівнів зрілості. Згодом змінювалися лише кількість і зміст процесних галузей.

2.9.3. Процесні галузі

Процесні галузі – це те, з чого складається вся модель. *CMMI* визначає стандартні процесні галузі, для кожної з яких існує ряд цілей, які повинні бути досягнуті з упровадженням *CMMI* в цій процесній галузі. Деякі цілі є унікальними – їх називають спеціальними. Загальні цілі застосовують до кількох процесних галузей. Цілі досягаються за допомогою виконання практик; так само, як і цілі, практики поділяють на спеціальні та загальні.

Перелік основних процесних галузей:

1. Менеджмент вимог (*Requirements Management*) – керування вимогами до продуктів проекту або компонентів продукту з метою виявлення невідповідності між вимогами і планами проекту.
2. Планування проекту (*Project Planning*) – розроблення і підтримання планів визначають розвиток проекту.
3. Моніторинг і контроль проекту (*Project Monitoring and Control*) – забезпечення розуміння стадії розроблення проекту з упровадження коригувальних дій у разі істотного відхилення від плану.
4. Менеджмент договорів з постачальниками (*Supplier Agreement Management*) – керування придбанням товарів і послуг від зовнішніх постачальників, з якими укладено договори.
5. Вимірювання і аналіз (*Measurement and Analysis*) – розроблення і підтримання можливості вимірювання, що використовується для підтримання потреб інформаційного менеджменту.
6. Оцінювання (гарантування) якості товарів і процесів (*Process and Product Quality Assurance*) – забезпечення підтримання та керування відповідно до цілей процесів і пов'язаних з ними продуктів роботи.
7. Конфігураційний менеджмент (*Configuration Management*) – передавання даних та відновлення цілісності продуктів роботи у результаті використання ідентифікації конфігурацій, конфігураційного контролю та конфігураційного аудита.

8. Розроблення вимог (*Requirements Development*) – збирання й аналіз вимог споживачів до продуктів і компонентів продуктів.

9. Технічне рішення (*Technical Solution*) – розроблення, дизайн та впровадження рішень щодо відповідних вимог. Рішення, дизайн і впровадження виражені продуктами, компонентами продуктів і пов'язаними із цими продуктами процесами.

10. Інтеграція продукту (*Product Integration*) – складання (монтування) продукту з його складових, перевірка якості інтеграції, її функціональність і випуск продукту.

11. Верифікація (*Verification*) – гарантування того, що обрані продукти роботи відповідають пропонованим вимогам.

12. Валідація (*Validation*) – демонстрація того, що продукт і його компоненти відповідають їх передбаченому використанню в передбачуваному середовищі.

2.9.4. Переваги використання CMMI

Використання *CMMI* дозволяє організації оцінити ефективність процесів, установити пріоритетні напрями їх удосконалення, а також упровадити дані вдосконалення.

Упровадження *CMM/CMMI* дозволяє поліпшити структуру й якість процесів (основні проблеми в програмних розробленнях – це проблеми керування, а не технічні проблеми), забезпечити стабільно високу якість розробок і освоїти процеси, які можуть стати основою для підвищення конкурентної спроможності та подальшого розвитку і розширення компанії.

В основу *CMM/CMMI* покладено поняття процесу, що допомагає уникнути природної для багатьох організацій тенденції звинувачувати в невдачах людей. Звільнення співробітників – не є вирішенням проблеми. За останні десятиліття відбулися революційні зміни в технології, однак проблеми успішного виконання проекту залишилися. У цьому аспекті технологія також не вирішує проблему. Цінність процесу полягає в тому, що він допомагає виявляти і використовувати найвищі досягнення в майбутніх проектах. Саме на цій передумові і базується *CMMI*.

CMM та *CMMI* – це моделі, які містять важливі елементи процесів, що забезпечують різні аспекти діяльності, і можуть бути використані як керівництво для розроблення і поліпшення виробничих процесів. В офіційних виданнях моделі наголошується, що вона не

містить процесів або їх опису. Реальні процеси в будь-якій організації залежать від безлічі факторів, включаючи специфіку бізнесу, структуру і розмір організації.

2.10. Шкала рівнів зрілості процесу керування вимогами

Використовуючи підхід до керування вимогами, необхідно мати уявлення про критерії оцінювання адекватності вимог до проектів. Одним з методів оцінювання є шкала зрілості керування вимогами (*Requirements Management Maturity*), яку вперше описав працівник *Rational Software* Джим Хьюман у 2003 р.

Термін «зрілість» визначається як повний та закінчений розвиток тієї чи іншої системи. Під час роботи з проектами і формування вимог до них необхідно уникати випадків експлуатації незакінчених систем або систем, що перебувають у стані модифікації. У випадку роботи з гнучкою методологією проектування або безупинного модифікування систем бажаними є проміжні вимоги, які описуватимуть кінцевий стан системи на цьому етапі.

Проектуючи будь-які системи, треба розуміти, що розвиток потребує витрат; у контексті бізнесу це означає, що організація приймає рішення про обсяги інвестицій у власний розвиток. Організація повинна чітко усвідомлювати, які вигоди вона при цьому отримує і чи покривають очікувані вигоди заплановані витрати.

Нижче наведено шкалу рівнів зрілості процесу керування вимогами, побудовану за аналогією з моделлю *CMMI*.

Рівень 0. Відсутність вимог.

Рівень 1. Документування вимог.

Рівень 2. Організація вимог.

Рівень 3. Структурування вимог.

Рівень 4. Трасування вимог.

Рівень 5. Інтеграція вимог.

Ці рівні ніяк не пов’язані між собою, але перетинаються. Так, досягнення рівня 5 (інтеграція вимог) зрілості процесу керування вимогами дозволить отримати як мінімум рівень 3 (процеси визначені на рівні всієї організації) за моделлю *CMMI*. Однак це не є безпосереднім наслідком, оскільки досягнення високого рівня зрілості в одному процесі не гарантує загального підвищення зрілості організації.

2.11. Сучасні системи керування вимогами

Відповідно до глосарію термінів програмної інженерії *IEEE*, що є загальноприйнятим міжнародним стандартним глосарієм, вимога – це:

– умови або можливості, необхідні користувачу для вирішення проблем або досягнення цілей;

– умови або можливості, які повинна мати система або системні компоненти, щоб виконати контракт або задовольняти стандарти, специфікації або інші формальні документи;

– документоване подання умов або можливостей для рівнів 1 і 2.

Відповідно до стандарту розроблення вимог *ISO*/*IEC* 29148 вимога – це твердження, яке ідентифікує експлуатаційні, функціональні параметри, характеристики чи обмеження проектування продукту або процесу.

Вимога повинна мати такі характеристики:

– одиничність;

– завершеність;

– послідовність;

– атомарність;

– спостережуваність;

– актуальність;

– здійсненість;

– недвозначність;

– обов'язковість;

– верифікованість.

Відповідно до *ITILv*3 усі вимоги в проекті можна поділити на такі групи:

– функціональні (*Functional*) – реалізують саму бізнес-функцію;

– керувальні (*Manageability*) – вимоги до доступних і безпечних сервісів; ставляться до розміщення системи, адміністрування і безпеки;

– ергономічні (*Usability*) – забезпечують зручність роботи кінцевих користувачів;

– архітектурні (*Architectural*) – вимоги до архітектури системи;

– взаємодії (*Interface*) – забезпечують взаємозв'язки між існуючими додатками і програмними засобами та новим додатком;

– сервісного рівня (*Service Level*) – описують поведінку сервісу, якість його вихідних даних та інші якісні аспекти, вимірювані замовником.

Програмне забезпечення для керування вимогами:

1. *IBM Rational Requisite Pro*. Програмне забезпечення *Rational* – це кращі практичні методи визначення вимог і керування ними, які забезпечують економію часу та коштів, допомагають вирішувати такі завдання:

– скорочення обсягу доробок і пришвидшення виходу на ринок завдяки спільній роботі із зацікавленими особами;

– підвищення продуктивності праці шляхом контролю над змінами у вимогах і керуванні ними;

– мінімізація витрат і ризиків через оцінювання впливу змін;

– демонстрація відповідності вимог завдяки їх повному відстеженню.

2. *IBM Rational / Telelogic DOORS* – група рішень для керування вимогами і створення складних наукоємних виробів (авіа-, суднобудування, ракети, автомобілі тощо).

Спочатку *DOORS* розроблялося лише як засіб керування вимогами в процесі розроблення ПЗ. Однак ідеї, закладені в *DOORS*, виявилися успішними і зараз система використовується навіть у компаніях, які не займаються розробленням ПЗ, але змушені контролювати великий обсяг взаємозалежної інформації, наприклад, під час розроблення інженерних систем.

3. Інше ПЗ. Досить відомі системи керування вимогами:

– *Sybase PowerDesigner*;

– *OpenSource Requirements Management Tool*;

– *RequirementsWin* та ін.

2.12. Керування проектами в умовах постійної зміни вимог

Оскільки в програмних проектах беруть участь менеджери і програмісти, то основні зусилля спрямовуються як на організацію самих процесів розроблення, так і на реалізацію засобів автоматизації цих процесів.

Одним з поширених прийомів керування програмними проектами є метод під назвою «базові тікети». Перед початком розроблення складається повний перелік завдань проекту. Завдання масштабуються від 4 до 16 годин розроблення. Якщо буде кілька сотень завдань, їх можна об’єднати.

Основна ідея методу полягає в тому, що на кожне завдання з *WBS* створюється тікет у трекері з повною постановкою на конт-

роль цього завдання (також цю дію можна реалізовувати в будь-якій системі, де можна вести облік завдань за принципом: 1 задача = 1 стрічка). Підхід ґрунтується на прецедентах (*use case*, *user story*). Тікети мають індикатор, який визначає на чиєму боці в даний момент знаходиться завдання. У міру розроблення завдання (тікети) здаються замовнику і на початку процесу розроблення отримується зворотний зв'язок від замовника.

Керівнику такий підхід дозволяє оцінити темпи та якість на початку розроблення (наприклад, якщо замовнику передаватимуть неякісний результат, то керівник швидко це виявить, і зможе скоригувати ситуацію й отримати новий відгук).

Головним у «базових тікетах» є відстеження змін вимог. Зміни вимог – це одна з актуальних проблем *IT*-проектів, і саме тому в них не можливо використовувати принцип «водоспаду» (написавши технічне завдання – зробив). Інакше кажучи, керування *IT*-проектом і є керуванням вимогами. Причому вимоги змінюють всі – і великі компанії, і внутрішні замовники і навіть невеликі фірми. Тому, маючи базові тікети, завжди легко подивитися первинну постановку.

Таким чином, отримуємо актуальні знання про стан проекту і вимоги до нього без окремого ведення документів або *wiki* з актуальними вимогами. Зазвичай окреме ведення історії зміни вимог не буває актуальним або потребує неадекватних зусиль керівництва на її підтримання. Також цілком реально відслідковувати стан проекту, адже завдання приймає замовник (тобто замість абстрактних «у нас майже все готово», скільки завдань прийнято замовником, скільки відправлено на доопрацювання, які ще не передавалися і т. ін.).

У складних проектах дуже важливо спочатку отримати стабільну версію, а вже потім вносити в неї зміни.

Що стосується стратегії, то потрібно прагнути якомога швидше випускати першу версію, оскільки після початку експлуатації вимоги можуть кардинально змінитися.

Контрольні запитання і завдання

1. Сформулюйте поняття вимоги. Наведіть класифікацію вимог.
2. Чим розрізняються користувацькі та системні вимоги?
3. Поясніть призначення проектної системної специфікації.
4. Яка відмінність між функціональними і нефункціональними вимогами?

5. Укажіть показники для вимог з надійності, зручності, простоти використання ПЗ.
6. Сформулюйте вимоги до зручності використання графічним редактором.
7. Які існують правила написання користувацьких вимог?
8. Назвіть та прокоментуйте способи записування специфікацій вимог.
9. Поясніть застосування структурованої мови специфікацій.
10. Поясніть призначення аналізу реалізовності вимог.
11. За допомогою методу *VORD* поясніть процес отримання грошей з банкомата.
12. На основі методу сценаріїв покажіть порядок замовлення книги в електронній бібліотеці.
13. Поясніть формування вимог на підставі етнографічного підходу.
14. Які етапи передують етапу реалізації вимоги до керування вимогами?
15. Складіть вимоги для системи керування польотами. Проаналізуйте їх за допомогою матриці оперативного контролю.

3. ПРОЦЕСИ ЖИТТЄВОГО ЦИКЛУ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

3.1. Життєвий цикл програмного забезпечення

Організації-розробники використовують різні підходи до створення ПЗ. Незважаючи на те, що спостерігається величезне різноманіття підходів, методів і технологій створення ПЗ, існують фундаментальні базові процеси, без реалізації яких не може обійтися жодна технологія розроблення програмних продуктів. Вони такі:

– розроблення специфікації ПЗ;
– проектування і розроблення ПЗ;
– атестація ПЗ;
– еволюція ПЗ.

Під моделлю процесу створення ПЗ розуміють послідовність етапів, виконання яких приводить до розроблення ПЗ, абстрагуючись від дрібних деталей окремих складових його етапів. Вони не містять точного опису всіх стадій процесу створення ПЗ. Але вони є корисними абстракціями, що надають різні підходи і технології процесу розроблення. Розрізнюють такі моделі створення ПЗ:

– каскадну;
– еволюційну;
– формальну;

– інкрементну;
– ітераційні моделі.

Найбільшого поширення на практиці набули каскадна і еволюційна моделі розроблення. Використання готових системних компонентів практикується повсюдно, але більшість організацій не дотримуються точності моделі розроблення ПЗ на підставі раніше створених компонентів.

3.2. Каскадна модель

Каскадна (водоспадна) модель (рис. 3.1) відповідає моделі життєвого циклу ПЗ. Основні принципові етапи цієї моделі відображають усі базові види діяльності, необхідні для створення ПЗ. Її основна ідея полягає в тому, що кожний наступний етап починається лише після повного закінчення попереднього.

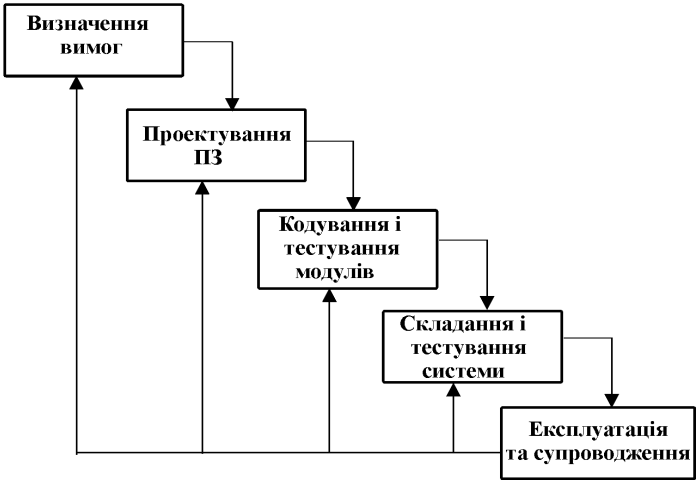


Рис. 3.1. Життєвий цикл ПЗ

Життєвий цикл ПЗ – це сукупність процесів, що перебігають у період від моменту прийняття рішення про створення ПЗ до його повного виведення з експлуатації.

Таким чином, життєвий цикл ПЗ є більш широким поняттям, ніж модель процесу створення ПЗ. Каскадну модель можна розглядати як одну з моделей життєвого циклу ПЗ. Її складові такі.

1. *Аналіз і формування вимог.* Через консультацій із замовником ПЗ визначаються функціональні можливості, обмеження і цілі створюваної програмної системи.

2. *Проектування системи і ПЗ.* У процесі проектування системні вимоги поділяють на групи вимог до апаратних засобів і ПЗ системи. Розробляється загальна архітектура системи. Визначаються основні програмні компоненти та їх взаємозв'язки.

3. *Кодування і тестування програмних модулів.* На цій стадії архітектура ПЗ реалізується у вигляді множини програм або програмних модулів. Тестування кожного модуля включає перевірку його відповідності вимогам до цього модуля.

4. *Складання і тестування системи.* Окремі програми і програмні модулі інтегруються і тестуються у вигляді цілісної системи. Перевіряється їх відповідність специфікації.

5. *Експлуатація і супроводження системи.* Це найбільш тривала фаза життєвого циклу ПЗ. Система інсталюється і починається період її експлуатації. Супроводження системи включає виправлення помилок, які не були виявлені на більш ранніх етапах життєвого циклу, удосконалення системних компонентів і пристосування функціональних можливостей системи до нових вимог.

Результат кожного етапу затверджується документально. Хоча наступний етап не може початися до завершення попереднього, проте на практиці етапи можуть перекриватися з постійним перетіканням інформації від одного етапу до іншого. Наприклад, на етапі проектування може виникнути необхідність уточнити системні вимоги або на етапі кодування виявиться проблеми, які можна вирішити лише на етапі проектування, і т.ін. Процес створення ПЗ не описується простою лінійною моделлю, оскільки він неминуче містить послідовність повторюваних процесів.

Оскільки на кожному етапі проводяться певні роботи і оформлюється супутня документація, повторення етапів зумовить повторні роботи і значні витрати. Тому після невеликої кількості повторень частина етапів створення ПЗ «заморожується», наприклад етап визначення вимог, але триває виконання наступних етапів. Виникаючі проблеми, вирішення яких потребує повернення до «заморожених» етапів, ігноруються або робляться спроби вирішити їх програмно. «Заморожування» етапу визначення вимог може призвести до того, що розроблена система не буде задовольняти всі

вимоги замовника. Якщо недоліки етапу проектування виправляються лише за рахунок здібностей програмістів, то це може призвести до погано структурованої системи.

Етап експлуатації і супроводження життєвого циклу ПЗ – це інтенсивне використання ПС. На цьому етапі можуть виявитися помилки, допущені, наприклад, на етапі формування вимог, помилки проектування і кодування, що може потребувати визначення нових функціональних можливостей системи, попри те, що система постійно має залишатися працездатною. Потрібні зміни ПС можуть зумовити повтор деяких або навіть всіх етапів процесу створення ПЗ.

Переваги каскадної моделі:

– добре відображає практику створення ПЗ;

– набула поширення, зокрема для розроблення великих систем.

Недолік: відсутність гнучкості поділу процесу створення ПЗ на окремі фіксовані етапи.

3.3. Еволюційна модель

Згідно із цією моделлю розробляється перша версія програмного продукту, що передається на випробування користувачам, потім її доопрацьовують за пропозиціями користувачів. Випускають наступну версію продукту, що також проходить «випробування користувачем», знову доопрацьовують і так кілька разів, поки не отримають необхідний програмний продукт (рис. 3.2). Відмітною ознакою цієї моделі є те, що деякі процеси специфікування, розроблення і атестації ПЗ виконуються на паралельній основі з постійним обміном інформацією між ними.

Розрізняють такі підходи до реалізації еволюційного методу розроблення.

Пробні розроблення. У межах цього підходу спочатку розробляють ті частини системи, які є очевидними чи добре специфікованими. Система доопрацьовується (еволюціонує) шляхом додавання нових засобів за пропозиціями замовника.

Прототипування. На цьому етапі малими зусиллями створюється працююча система (можливо неефективно, з помилками і не повною мірою). Потім обов’язково переглядаються етапи архітектури системи, розроблення, реалізації і тестування кінцевого продукту.

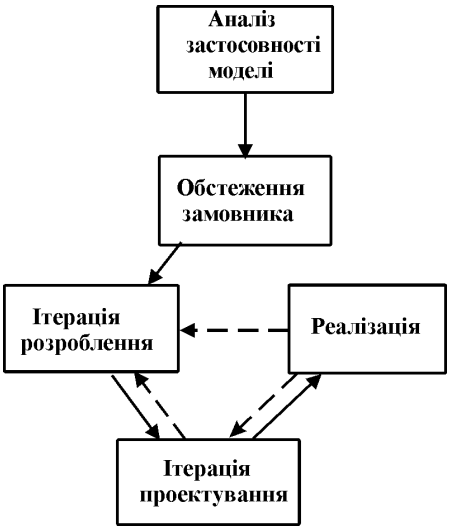


Рис. 3.2. Еволюційна модель розроблення
на основі прототипування

Еволюційний підхід часто є ефективнішим, ніж каскадний, особливо якщо вимоги замовника змінюються у процесі розроблення системи. Перевагою підходу є те, що тут специфікацію можна розробляти поступово, у міру того, як замовники (або користувачі) усвідомлюють і формулюють ті пункти завдання, які має вирішувати ПЗ. Разом із тим підхід має низку недоліків:

1) недокументованість деяких етапів процесу створення ПЗ;

2) незадовільна структурованість системи;

3) потреба в спеціальних засобах і технологіях розроблення ПЗ.

Еволюційний підхід найбільш прийнятний для розроблення невеликих ПС (до 100 000 рядків коду) і систем середнього розміру (до 500 000 рядків коду) з нетривалим часом життя.

У змішаному підході для з'ясування «темних місць» у системній специфікації можна використовувати прототипування. Системні компоненти, для яких повністю визначені вимоги, можна створювати за каскадною моделлю. Інші системні компоненти, які важко специфікувати, наприклад користувацький інтерфейс, можна розробляти з використанням прототипування.

3.4. Формальна модель

Цей підхід до створення ПЗ побудовано на підставі формальних математичних перетворень системної специфікації у виконувану програму. Порядок створення ПЗ відповідно до цього підходу показано на рис. 3.3, на якому для спрощення зворотні зв'язки між етапами процесу не зазначені.

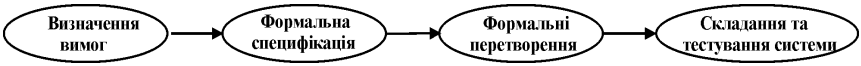


Рис. 3.3. Модель формального розроблення ПЗ

Модель формального розроблення відрізняється від каскадної за такими ознаками:

– специфікація має вид спеціальної математичної нотації;

– процеси проектування, написання програмного коду і тестування системних модулів шляхом послідовних формальних перетворень легко трансформуються у програму, що виконується.

Цей процес показано на рис. 3.4, на якому Т1–*Т*4 – чергові трансформації, а *V* 1–*V* 3 – варіанти виконуваного коду.

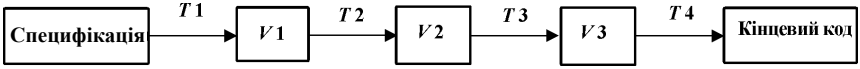


Рис. 3.4. Процес формальних перетворень

У процесі перетворення формальне математичне подання системи послідовно і математично коректно трансформується в програмний код, що поступово дедалі більше деталізується. Ці перетворення виконуються доти, доки всі позиції формальної специфікації не будуть трансформовані в еквівалентну програму. Якщо перетворення виконуються математично коректно, то проблеми перевірки відповідності специфікації і програми не існує. Найбільш відомим прикладом методу формальних перетворень є метод «чистої кімнати» (*Cleanroom*), розроблений компанією *IBM*. Цей метод припускає покрокове розроблення ПЗ, коли на кожному кроці застосовуються формальні перетворення. Це дозволяє відмовитися від тестування окремих програмних модулів, а вся система тестується після її складання. Методи формальних перетворень зазвичай застосовують для розроблення систем, які повинні задовольняти

жорсткі вимоги надійності, безвідмовності і безпеки, оскільки вони гарантують відповідність створених систем їх специфікаціям.

Перевага методу полягає в тому, що дистанція між послідовними перетвореннями на кожному кроці трансформації зменшується.

Недоліки методу:

– доведення коректності програмного коду для великих масштабованих систем потребує тривалого часу, а часто і неможливо;

– вибір для застосування відповідних формальних перетворень складний і неочевидний;

– метод потребує спеціальних знань та досвіду використання;

– істотного виграшу у вартості або якості порівняно з іншими методами розроблення ПЗ метод не дає.

Функціонування більшості систем складно піддається опису формальної специфікації, тому для створення більшості програмних систем значні зусилля розробників спрямовуються саме на створення специфікацій.

3.5. Інкрементна модель

Сучасний підхід до створення програмних проектів ґрунтується на повторному використанні деяких програмних модулів. Це відбувається тоді, коли розробники проекту мають відомості про вже створені програмні продукти, у складі яких є компоненти, що приблизно задовольняють вимоги розроблюваних компонентів. Ці компоненти модифікуються відповідно до нових вимог і потім включаються до складу нової системи. В еволюційній моделі розроблення для пришвидшення процесу створення досить часто повторно використовується ПЗ уже створених компонентів.

Неформальне рішення щодо повторного використання раніше створених програмних компонентів зазвичай приймається незалежно від загального процесу створення ПЗ. Цей підхід ґрунтується на наявності великої бази існуючих програмних компонентів, які можна інтегрувати у створювану нову систему. Такими компонентами можуть бути вільні програмні продукти, які виконують певні спеціальні функції, такі як форматування тексту, числові обчислення тощо.

У цьому підході початковий етап специфікування вимог і етап атестації такі саме, як і в інших моделях процесу створення ПЗ. Інші послідовні етапи (рис. 3.5) розміщені між ними. Їх сенс такий:

1. Аналіз та пошук компонентів, які задовольняють вимоги.

2. Модифікація вимог – здійснюється таким чином, щоб максимально використовувати можливості відібраних компонентів. Якщо зміна вимог неможлива, то модифікація виконується повторно для знаходження будь-якого альтернативного рішення.

3. Проектування системи з урахуванням відібраних програмних компонентів і побудова структури відповідно до їх функціональних можливостей. Якщо деякі готові програмні компоненти недоступні, проектується нове ПЗ.

4. Складання системи.

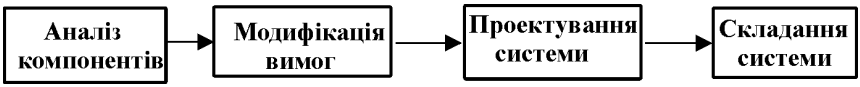


Рис. 3.5. Розроблення ПЗ з повторним використанням раніше створених компонентів

Основні переваги описуваної моделі процесу розроблення ПЗ з повторним використанням раніше створених компонентів полягають у тому, що скорочується кількість безпосередньо розроблюваних компонентів і зменшується загальна вартість створюваної системи.

Недоліки:

– система не буде задовольняти всі вимоги замовника, що потребує неминучих компромісів для визначених вимог;

– для проведення модернізації системи (тобто створення її нової версії) немає можливості впливати на появу нових версій компонентів, використовуваних у системі.

3.6. Ітераційні моделі

Описані моделі процесу розроблення ПЗ мають свої переваги і недоліки. Великі системи не створюються за конкретною моделлю, а застосовуються й інші моделі. Тому важливою є можливість виконувати окремі процеси розроблення підсистем і весь процес створення ПЗ є ітераційним, коли у відповідь на зміни вимог повторно виконуються певні етапи створення системи (найчастіше етапи проектування і кодування). Тому доцільно розглянути гібридні ітераційні моделі, що поєднують кілька різних підходів до розроблення ПЗ і розроблені спеціально для підтримання ітераційного

способу створення ПЗ. Це покрокове розроблення і спіральна модель.

Покрокове розроблення. Згідно із цією моделлю процеси специфікування вимог, проектування і написання коду поділяють на послідовність невеликих кроків, які ведуть до створення ПЗ.

Спіральна модель розроблення. Процес створення ПЗ від початкового ескізу системи до її кінцевої реалізації розгортається по спіралі.

3.6.1. Модель покрокового розроблення

Модель покрокового розроблення (рис. 3.6) запропонував Міллс [3] як спробу зменшити кількість повторно виконуваних робіт у процесі створення ПЗ і збільшити час замовнику для остаточного прийняття рішення про системні вимоги. Як і в каскадній моделі створення ПЗ вимоги визначаються через тісну співпрацю із замовником до початку проектування системи і має зрозумілий процес керування створенням ПЗ. Як і в еволюційній моделі тут можна відтермінувати прийняття остаточних рішень про специфікацію і структуру системи.

Рис. 3.6. Модель покрокового розроблення

У процесі покрокового розроблення замовник спочатку в цілому визначає сервіси (функціональні можливості), які мають бути у створюваній системі. При цьому встановлюються пріоритети сервісів. Також визначається кількість кроків розроблення, причому на кожному кроці отримується системний компонент, що реалізує певну підмножину системних функцій. Розподіл реалізації системних сервісів по кроках розроблення залежить від їх пріоритетів. Сервіси з більш високими пріоритетами реалізуються першими.

Послідовність кроків розроблення визначається заздалегідь до початку їх виконання. На перших кроках деталізуються вимоги для

сервісів, потім застосовується один із придатних способів розроблення ПЗ. У ході їх реалізації аналізуються і деталізуються вимоги для компонентів, які розроблятимуться пізніше, причому вимоги для тих компонентів, які перебувають у процесі розроблення, не змінюються.

Після завершення кроку розроблення отриманий програмний компонент передається замовнику для інтегрування в підсистему, яка реалізує певний системний сервіс. Замовник може експериментувати з готовими підсистемами і компонентами для уточнення вимог, пропонованих до наступних версій уже готових компонентів або до компонентів, розроблюваних на наступних кроках. По завершенні чергового кроку розроблення отриманий компонент інтегрується з уже виконаними компонентами; таким чином, після кожного кроку розроблення система здобуває дедалі більшу функціональну завершеність. Загальносистемні функції в цьому процесі можуть реалізуватися відразу або поступово в міру розроблення необхідних компонентів.

В описуваній моделі не передбачається, що на кожному кроці використовується один і той самий підхід до процесу розроблення компонентів. Якщо створюваний компонент має добре розроблену специфікацію, то для його створення можна застосувати каскадну модель. Якщо ж вимоги визначені нечітко, використовують еволюційну модель розроблення.

Процес покрокового розроблення має ряд переваг.

1. Замовнику немає потреби чекати повного завершення розроблення системи, щоб уявити її. Компоненти, отримані на перших кроках розроблення, задовольняють найбільш критичні вимоги (найбільший пріоритет) і їх можна оцінити на ранній стадії створення системи.

2. Замовник може використовувати компоненти, отримані на перших кроках розроблення, як прототипи і провести з ними експерименти для уточнення вимог до тих компонентів, які розроблятимуться пізніше.

3. Цей підхід зменшує ризик загальносистемних помилок. Хоча під час розроблення окремих компонентів можливі помилки, але ці компоненти мають пройти відповідне тестування й атестацію, перш ніж їх передадуть замовнику.

4. Оскільки системні сервіси з високим пріоритетом розробляються першими, а всі наступні компоненти інтегруються з ними, то найбільш важливі підсистеми підлягають більш ретельному всебі-

чному тестуванню і перевірці. Це знижує ймовірність програмних помилок в особливо важливих частинах системи.

Під час реалізації покрокового розроблення можуть виникнути певні проблеми. Компоненти, отримувані на кожному кроці розроблення невеликого розміру (не більше, як 20 000 рядків коду), але вони мають реалізувати яку-небудь системну функцію. Відобразити множину системних вимог до компонентів потрібного розміру досить складно. Більше того, багато систем повинні мати набір базових системних властивостей, які реалізовуються спільно різними частинами системи. Оскільки вимоги детально не визначаються доти, доки не будуть розроблені всі компоненти, досить складно розподілити загальносистемні функції по компонентах.

3.6.2. Спіральна модель

Спіральна модель процесу створення ПЗ (рис. 3.7) є найбільш поширеною.

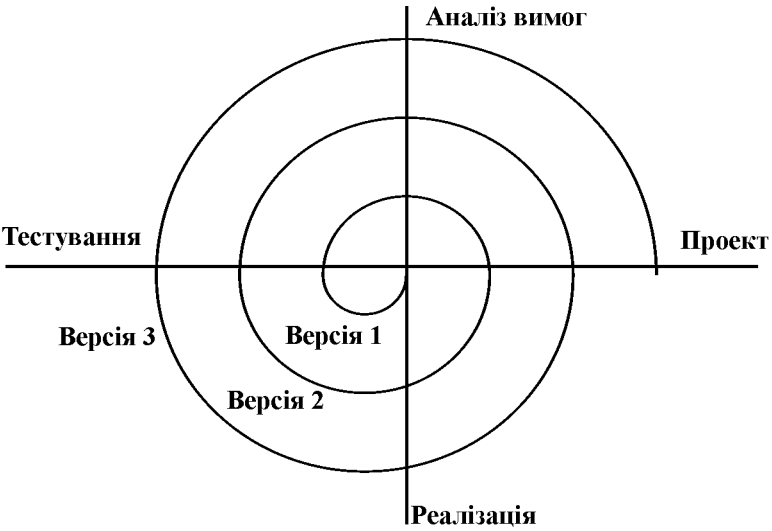


Рис. 3.7. Спіральна модель

На відміну від розглянутих моделей, де процес створення ПЗ являє собою послідовність окремих процесів з можливим зворотним зв’язком між ними, тут процес розроблення має вигляд спіралі.

Кожний виток спіралі відповідає одній стадії (ітерації) процесу створення ПЗ. Так, внутрішній виток спіралі відповідає стадії прийняття рішення про створення ПЗ, на наступному витку визначаються системні вимоги, далі – стадія (виток спіралі) проектування системи і т. ін.

Кожний виток спіралі розбитий на чотири сектори:

1) визначення цілей кожної ітерації проекту;

2) оцінювання і вирішення ризиків;

3) розроблення і тестування;

4) планування.

Істотна відмінність спіральної моделі від інших моделей процесу створення ПЗ полягає в точному визначенні й оцінюванні ризиків. Неформальне визначення ризику – це відхилення від технічного завдання в процесі розроблення системи. Наприклад, якщо для написання програмного коду використовується нова мова програмування, то ризик може полягати в тому, що компілятор цієї мови може виявитися ненадійним або що результуючий код не досить ефективний. Ризики можуть також спричинятися перевищенням термінів або вартості проекту.

Перша ітерація створення ПЗ у спіральній моделі починається з ретельного опрацювання системних показників (цілей системи), таких як експлуатаційні показники і функціональні можливості системи. Зазвичай альтернативних шляхів досягнення цих показників або цілей можна сформувати нескінченно багато. Але кожна альтернатива повинна оцінювати вартість досягнення кожної сформульованої мети. Результати аналізу можливих альтернатив є джерелом оцінювання проектного ризику. Це відбувається на наступній стадії спіральної моделі, у якій для оцінювання ризиків використовують більш детальний аналіз альтернатив, прототипування, імітаційне моделювання і т. ін. З урахуванням отриманих оцінок ризиків обирають той або інший підхід до розроблення системних компонентів, далі його реалізують і планують наступний етап процесу створення ПЗ.

Спіральна модель не містить фіксованих етапів, у ній можуть бути будь-які інші моделі розроблення систем. Наприклад, на одному витку спіралі може використовуватися прототипування для більш чіткого визначення вимоги (і, отже, для зменшення відповідних ризиків), а на наступному – каскадна модель. Якщо вимоги

чітко сформульовані, то використовується метод формальних перетворень.

Стандартизація моделей життєвого циклу. Стандарт *ISO* / *IEC* 12207 установлює загальну структуру процесів життєвого циклу програмній засобів, на яку можна орієнтуватися в програмної індустрії. Стандарт визначає процеси, види діяльності і завдання, які використовуються для придбання програмного продукту або послуги, а також для поставлення, розроблення, застосування за призначенням, супроводження та припинення застосування програмних продуктів. Відповідно до стандарту *ISO* / *IEC* 12207 життєвий цикл ПЗ охоплює:

– угоду;
– організаційне забезпечення проекту;
– проект;
– технічні процеси;
– реалізацію програмних засобів;
– процеси підтримання програмних засобів;
– процеси повторного застосування програмних засобів.

Цей стандарт опубліковано 1 серпня 1995 р. як перший міжнародний стандарт, що містив представницький набір процесів життєвого циклу, дій і завдань щодо ПЗ, яке розглядалося як частина більшої системи, а також щодо програмних продуктів і послуг.

У листопаді 2002 р. випущений стандарт *ISO* / *IEC* 15288 для процесів життєвого циклу систем. Широке застосування ПЗ зумовило неможливість розгляду ПЗ у відриві від систем, за винятком складової частини системи і процесу її створення. У Додатки до стандарту *ISO* / *IEC* 12207 уведено мету процесу і визначено його еталонну модель відповідно до вимог стандарту *ISO* / *IEC* 15504-2.

Міжнародний стандарт *ISO* / *IEC* 12207: 2008 являє собою перероблені і виправлені доповнення до стандарту *ISO* / *IEC* 12207 і є першим кроком у стратегії *SC*7 з гармонізації специфікацій, що має на меті створення повністю інтегрованого набору процесів ЖЦ систем і програмних засобів та керівництва з їх застосування.

Комітет *Software Engineering Coordinating Committee* разом з *IEEE Computer Society* підготував документ, що має назву *Software Engineering Body of Knowledge* (*SWEBOK*). Призначення *SWEBOK* – об'єднання знань з інженерії ПЗ.

Стандарти *ISO* / *IEC* 12207 та *SWEBOK* мають забезпечити:

– необхідний набір знань та рекомендовані практики;
– етичні та професійні стандарти;

– якість навчання студентів, аспірантів і тих, хто бажає підвищити кваліфікацію.

Ядро знань *SWEBOK* містить 10 галузей знань, але базовими вважаються:

1) розроблення вимог;
2) проектування;
3) конструювання;
4) тестування;
5) супроводження.

Ці процеси задають послідовність завдань та дій для розроблення різних типів ПЗ із застосуванням методів та засобів, які подано в ядрі знань *SWEBOK*. Усі процеси та галузі знань збігаються за змістом і назвою, але зміст дій визначається методами та засобами п'яти наведених вище галузей.

3.7. Характеристика галузей знань *SWEBOK* та його співвідношення зі стандартом *ISO* / *IEC* 12207

Порівняльний перелік основних галузей *SWEBOK*, їх завдань і відповідно до завдань життєвого циклу стандарту наведено в табл. 3.1. При цьому процеси придбання та постачання зі складу основних процесів вилучені, оскільки вони не належать до процесів розроблення ПЗ.

Таблиця 3.1

Завдання основних галузей знань *SWEBOK* і процесів життєвого циклу

Етапи життєвого циклу ПЗ	Завдання основних і додаткових процесів	Завдання процесів життєвого циклу стандарту *ISO* / *IEC* 12207
Розроблення вимог	*Інженерія вимог*: виявлення, аналіз, специфікація, перевірка, керування	Підготовка замовлення. Виявлення вимог. Аналіз вимог до системи. Аналіз вимог до ПЗ. Опис документа
Проектування	*Розроблення архітектури ПЗ*: нотація, аналіз якості проектування,	Проектування архітектури системи та ПЗ. Кодування ПЗ. Тестування ПЗ

	стратегії та методи проектування	

Закінчення табл. 3.1

Етапи життєвого циклу ПЗ	Завдання основних і додаткових процесів	Завдання процесів життєвого циклу стандарту *ISO* / *IEC* 12207
Конструювання	*Зменшення складності*: запобігання відхиленням від стилю, структурування системи, використання зовнішніх стандартів	Конструювання структури системи. Кодування елементів структури і ПЗ. Інтеграція елементів. Застосування стандартів програмної інженерії
Тестування	Тестування елементів, специфікацій, структури і системи за наборами даних. Метричне вимірювання тестування. Планування та оцінювання якості	Тестування ПЗ. Інтеграційне тестування. Кваліфікаційне тестування. Інтеграція системи. Системне тестування. Установлення и приймання ПЗ
Супроводження	Запуск ПЗ Знаходження помилок, планування виправлень Унесення змін	Інсталяція ПЗ Аналіз проблем і модифікація Реалізація модифікацій Аналіз супроводження Міграція, видалення ПЗ
Експлуатація системи	Методи забезпечення експлуатації системи	Упровадження процесу Функціональне тестування Експлуатація системи Підтримання користувача

Інші п’ять галузей ядра *SWEBOK* належать до процесів забезпечення і керування розроблення проекту, які передбачають верифікацію, збирання даних для оцінювання якості. Попри те, що галузі ядра знань явно не містять назв процесів життєвого циклу, функціонально та змістовно вони належать до процесів основних, допоміжних і організаційних категорій.

Допоміжним і організаційним діям життєвого циклу відповідають такі галузі знань *SWEBOK*:

– керування конфігурацією;

– керування інженерією ПЗ (або керування проектом);

– процес інженерії ПЗ (інфраструктура процесу розроблення);
– методи та засоби інженерії;
– інженерія якості (керування якістю).

Ці галузі знань включають методи та засоби розроблення ПЗ, а також керування проектом, ризиками, конфігурацією, якістю створюваного продукту.

Перелік галузей ядра *SWEBOK* та відповідні завдання допоміжних (організаційних і додаткових) процесів життєвого циклу стандарту *ISO* / *IEC* 12207 наведено в табл. 3.2.

Таблиця 3.2

Завдання галузей *SWEBOK* і додаткових процесів життєвого циклу

Етапи життєвого циклу ПЗ	Завдання додаткових процесів *SWEBOK*	Завдання процесів стандарту *ISO* / *IEC* 12207
Керування конфігурацією	Процес керування конфігурацією. Ідентифікація елементів. Урахування статусу, аудит. Контроль конфігурації. Керування версіями	Визначення та контроль конфігурації. Облік стану та оцінювання конфігурації. Керування реалізацією і поставкою версії
Керування проектом	Організаційне керування. Планування проектом. Керування процесами і проектом. Інженерія вимірювань ПЗ. Керування ризиками	Ініціація і визначення галузі застосування. Планування. Виконання та контроль. Аналіз керування проектом: технічний аналіз; аудит (ревізія)
Керування якістю	Концепція якості ПЗ. Визначення та планування якості. Верифікація та валідація. Вимірювання під час аналізу якості ПЗ	Упровадження процесу. Забезпечення виробництва та якості. Процес верифікації та валідації. Аналіз та оцінювання якості
Розроблення методів та засобів інженерії	Методи інженерії. Інструменти інженерії	Процес удосконалення: визначення процесу, оцінювання процесу, поліпшення процесу
Процес інженерії ПЗ	Інфраструктура процесу. Визначення процесу. Вимірювання процесу.	Створення інфраструктури. Упровадження інфраструктури.

	Аналіз проекту. Виконання змін. Оцінювання вартості і затрат	Упровадження процесу. Завершення

Перелік галузей ядра *SWEBOK* та відповідні завдання допоміжних (організаційних і додаткових) процесів життєвого циклу стандарту *ISO* / *IEC* 12207.

Зіставлення концепцій, методів і засобів *SWEBOK* із завданнями процесів життєвого циклу дозволяє регламентувати пошук, виявлення помилок та внесення змін у вимоги до системи.

3.8. *Agile*-методологія розроблення програмного забезпечення

Гнучка методологія розроблення (*Agile Software Development*, *agile*) – серія підходів до розроблення ПЗ, орієнтованих на використання ітеративного розроблення і динамічного формування вимог та забезпечення їх реалізації в результаті постійної взаємодії всередині самоорганізованих робочих груп, що складаються з фахівців різного профілю. Існує кілька методик, що належать до класу гнучких методологій розроблення, зокрема, відомі як гнучкі методики екстремального програмування, *DSDM*, *Scrum*.

Більшість цих методологій націлені на мінімізацію ризиків через зведення розроблення до серії коротких циклів, названих ітераціями, які зазвичай тривають два-три тижні. Кожна ітерація сама по собі виглядає як програмний проект у мініатюрі і включає всі завдання, необхідні для видачі міні-приросту за функціональністю: планування, аналіз вимог, проектування, кодування, тестування і документування. Хоча окрема ітерація недостатня для випуску нової версії продукту, утім гнучкий програмний проект готовий до випуску по завершенні кожної ітерації. При цьому команда переоцінює пріоритети процесу розроблення.

Agile-методи націлені на безпосереднє спілкування із замовником. Більшість *agile*-команд розміщені в одному офісі, який іноді називають *bullpen*. Щонайменше *agile*-команда включає замовника або його повноважного представника, який визначає вимоги до продукту. Цю роль може виконувати менеджер проекту, бізнес-

аналітик або клієнт. Офіс може також включати тестувальників, дизайнерів інтерфейсу, технічних письменників і менеджерів.

Основною метрикою *agile*-методів є робочий продукт. Віддаючи перевагу безпосередньому спілкуванню, *agile*-методи зменшують обсяг письмової документації порівняно з іншими методами. Це викликало критику цих методів.

3.8.1. Принципи agile-методології

Agile – група процесів, а не єдиний підхід до розроблення ПЗ, що визначається *Agile Manifesto*. *Agile* не містить практик, а визначає цінності і принципи, якими керуються успішні команди.

Agile Manifesto розроблений і прийнятий 11–13 лютого 2001 р. на лижному курорті *The Lodge at Snowbird* у горах Юти. Маніфест підписали представники методологій *Extreme programming*, *Scrum*, *DSDM*, *Adaptive software development*, *Crystal Clear*, *Feature-driven development*, *Pragmatic Programming*. Група *Agile Manifesto* містить 4 основні ідеї і 12 принципів. Примітно, що *Agile Manifesto* не містить практичних порад.

Основні ідеї:

– особистості та їх взаємодія важливіші, ніж процеси та інструменти;

– діюче ПЗ більш значуще, ніж повна документація;

– співпраця із замовником важливіша, ніж контрактні зобов'язання;

– реагування на зміни більш важливе, ніж дотримання плану.

Принципи, які роз'яснює *Agile Manifesto*:

– задоволення клієнта за рахунок раннього і безперебійного постачання цінного ПЗ;

– прийнятність змін вимог, навіть по завершенні процесів розроблення (це може підвищити конкурентоспроможність отриманого продукту);

– частість постачання робочого ПЗ (щомісяця, щотижня або навіть частіше);

– тісне щоденне спілкування замовника з розробниками протягом усього проекту;

– забезпеченість проекту належними умовами роботи, підтримкою і довірою;

– узгодженість вимог безпосередньо із замовником;

– працездатність ПЗ – кращий вимірник прогресу;

– можливість спонсорів, розробників і користувачів підтримувати постійний темп протягом невизначеного терміну;

– підвищення технічної майстерності і зручність дизайну;

– адаптованість до мінливих обставин.

3.8.2. Недоліки agile-методологій

За *agile*-підходу часто нехтують створенням «дорожньої карти» розвитку продукту, так само, як і керуванням вимогами, у процесі якого і формується така «дорожня карта». Гнучкий підхід до керування вимогами не має на увазі далекосяжних планів (керування вимогами не відображено в цій методології), а передбачає можливість замовника раптом і несподівано наприкінці кожної ітерації ставити нові вимоги, що часто суперечать архітектурі вже створеного і поставленого товару. Це іноді призводить до катастрофічних «авралів» з масовим рефакторингом і переробленням майже на кожній черговій ітерації.

Крім того, вважається, що робота в *agile* мотивує розробників вирішувати всі завдання найпростішим і найшвидшим можливим способом, часто не звертаючи уваги на правильність коду щодо вимог нижньої платформи (підхід – «працює і добре», при цьому не враховується, що може перестати працювати у разі найменшої зміни або ж спричинити складні для відтворення дефекти після реального розгортання у клієнта). Це призводить до зниження якості продукту і нагромадження дефектів.

3.8.3. Методології, які підтримують agile-підхід

Існують методології, у яких дотримуються цінності і принципи, заявлені в *Agile Manifesto*. Деякі з них такі.

Agile Modeling – набір понять, принципів і прийомів (практик), що дозволяють швидко і просто виконувати моделювання та документування в проектах розроблення ПЗ, а також не містить у собі детальної інструкції з проектування та описів побудови діаграм на

UML. Основна мета – ефективне моделювання і документування, але не охоплює програмування і тестування, не включає питання керування проектом, розгортання і супроводження системи. Однак передбачає перевірку моделі кодом [3].

Agile Unified Process (*AUP*) – спрощена версія *IBM Rational Unified Process* (*RUP*), розроблена Скоттом Амблер, яка описує просте і зрозуміле наближення (модель) для створення ПЗ для бізнес-додатків.

Agile Data Method – група ітеративних методів розроблення ПЗ, у яких вимоги і вирішення досягаються в межах співпраці різних крос-функціональних команд.

DSDM ґрунтується на концепції швидкого розроблення додатків (*Rapid Application Development – RAD*) і являє собою ітеративний і інкрементний підхід, який надає особливої значущості тривалій участі в процесі користувача (споживача).

Essential Unified Process (*EssUP*).

Екстремальне програмування (*Extreme Programming*, *XP*).

Feature driven development (*FDD*) – функціонально-орієнтоване розроблення. Поняття функції або властивості в *FDD* досить близьке до поняття прецеденту використання, застосовуваному в *RUP*, істотна відмінність – це додаткове обмеження: «кожна функція повинна допускати реалізацію не більше, ніж за два тижні». Тобто, якщо сценарій використання досить малий, його можна вважати функцією, а якщо ж великий, то його треба розбити на кілька відносно незалежних функцій.

Getting Real – ітеративний підхід без функціональних специфікацій, що використовується для *web*-додатків. У цьому методі спочатку розробляється інтерфейс програми, а потім її функціональна частина.

OpenUP – це ітеративно-інкрементальний метод розроблення ПЗ. Позиціонується як легкий і гнучкий варіант *RUP*. Метод *OpenUP* ділить життєвий цикл проекту на чотири фази: початкову, уточнення, конструювання та передавання. Життєвий цикл проекту забезпечує надання зацікавленим особам та членам колективу можливості ознайомлення і прийняття рішень протягом усього проекту. Це дозволяє ефективно контролювати ситуацію і вчасно прий-

мати рішення про прийнятність результатів. План проекту визначає життєвий цикл, а кінцевим результатом є остаточний додаток.

Scrum – установлює правила керування процесом розроблення і дозволяє використовувати вже існуючі практики кодування, коригуючи вимоги або вносячи тактичні зміни. Використання цієї методології дає змогу виявляти й усувати відхилення від бажаного результату на більш ранніх етапах розроблення програмного продукту.

Ощадливе розроблення ПЗ (*Lean software development*) – використовує підходи з концепції бережливого виробництва.

Контрольні запитання і завдання

1. Чи є причиною обмеженого застосування автоматизованих засобів відсутність відомостей про них?
2. Чи належать до фундаментальних складових базового процесу розроблення програмних продуктів виправлення помилок коду?
3. Запропонуйте підхід до створення ПЗ для операційних систем.
4. Проаналізуйте способи вдосконалення ПЗ.
5. Що означає зворотний зв'язок у моделі життєвого циклу ПЗ?
6. У чому проявляються недоліки процесу створення ПЗ за каскадною моделлю?
7. Поясніть відмінності процесу створення ПЗ за еволюційною і каскадною моделями розроблення ПЗ.
8. Поясніть відмінність процесу створення ПЗ між формальним розробленням систем і каскадною моделлю.
9. Наведіть приклади застосування формального розроблення ПЗ.
10. У чому сутність методу формальних перетворень?
11. Проаналізуйте метод розроблення ПЗ на основі створених компонентів.
12. Поясніть сутність моделі покрокового розроблення ПЗ.
13. Поясніть сутність методу спіральної моделі розроблення ПЗ.
14. Які ризики розроблення ПЗ існують? Наведіть приклади.
15. Проаналізуйте метод спіральної моделі розроблення ПЗ.

4. КЕРУВАННЯ ПРОЕКТАМИ

4.1. Процеси керування

Необхідність керування програмними проектами вперше виникла в 60–70-х роках минулого століття, коли почали розроблятися проекти великих програмних продуктів. Було зафіксовано затримання створення ПЗ як ненадійного, витрати на розроблення в кілька разів перевищували первинні оцінки, розроблені програмні системи мали низькі показники продуктивності. Провали програмних проектів зумовлювалися не стільки некомпетентністю керівників і програмістів, скільки підходами, які використовувалися в керуванні проектами. Застосовувані методики керування технічними проектами виявилися неефективними для розроблення ПЗ.

Потреба керувати програмними проектами випливає з того факту, що процес створення професійного ПЗ завжди є суб'єктом бюджетної політики організації, у якій його розробляють, і має часові обмеження. Робота керівника програмного проекту полягає в гарантуванні виконання цих бюджетних і часових обмежень з урахуванням бізнес-цілей організації щодо розроблюваного ПЗ.

Менеджери проектів покликані спланувати всі етапи розроблення програмного продукту. Вони також повинні контролювати хід виконання робіт і дотримання всіх необхідних стандартів. Постійний контроль за ходом виконання робіт необхідний для того, щоб процес розроблення не виходив за часові і бюджетні обмеження. Успішне керування не гарантує успішного завершення проекту, утім незадовільне керування обов'язково призведе до його провалу. Це може статися через затримання термінів здачі готового ПЗ, перевищення кошторису проекту та невідповідність готового ПЗ специфікації вимог.

Керівники програмних проектів виконують таку саму роботу, що і керівники технічних проектів. Проте процес розроблення ПЗ істотно відрізняється від процесів реалізації технічних проектів, що породжує певні складності в керуванні програмними проектами. Наведемо деякі з цих відмінностей.

1. *Програмний продукт нематеріальний.* Менеджер проекту не може бачити результати його виконання, «росту» розроблюваного ПЗ, його не можна побачити або доторкнутися до нього. Менеджер

покладається лише на документацію, що фіксує процес її розроблення.

2. *Не існує стандартних процесів розроблення ПЗ.* Натепер немає чіткої залежності між процесом створення ПЗ і типом створюваного продукту. Якщо процеси розроблення технічних систем добре вивчені, то під час розроблення ПЗ не можна визначити етап, на якому виникнуть проблеми, що загрожують усьому проекту.

3. *Великі програмні проекти – це часто одноразові проекти.* Вони, як правило, значно відрізняються від проектів, реалізованих раніше. Щоб зменшити невизначеність у плануванні проекту, керівники проектів повинні мати дуже великий практичний досвід. Але розвиток технологій та нове устаткування знецінюють попередній досвід. Знання і навички, нагромаджений досвід можуть не затребуватися в новому проекті.

Це може призвести до того, що реалізація проекту не відповідатиме часовому графіку або перевищить бюджетні асигнування. Програмні системи найчастіше виявляються новинками, як в ідейному, так і в технічному сенсі. Інноваційні технічні проекти (наприклад, нова транспортна система) також часто порушують часові графіки робіт. Передбачаючи можливі проблеми в реалізації програмного проекту, варто знати, що багато з них виходять за межі часових і бюджетних обмежень.

Керування програмними проектами – тема досить велика, тому розглядатимемо лише основні процеси, а саме:

– розроблення пропозицій зі створення ПЗ;
– планування і складання графіка робіт;
– оцінювання вартості проекту;
– контроль за ходом виконання робіт;
– підбір персоналу;
– написання звітів і донесень.

Перша стадія програмного проекту може складатися з розроблення пропозицій щодо реалізації цього проекту. Пропозиції повинні містити опис цілей проектів і способи їх досягнення. Вони зазвичай містять оцінки фінансових і часових витрат на виконання проекту. У разі потреби тут може наводитись обґрунтування для передання проекту для виконання сторонньою організацією або командою розробників.

На етапі планування проекту визначаються процеси, етапи і отримані на кожному з них результати, які мають забезпечити виконання проекту. Реалізація цього плану означає досягнення цілей проекту. Визначення вартості проекту безпосередньо пов'язане з його плануванням, оскільки тут оцінюються ресурси, потрібні для виконання плану.

Контроль за ходом виконання робіт – це безперервний процес, що триває протягом усього терміну реалізації проекту. Менеджер повинен постійно відслідковувати хід реалізації проекту і порівнювати фактичні та планові показники виконання робіт з їх вартістю. Хоча багато організацій мають механізми формального моніторингу робіт, досвідчений менеджер може чітко уявити стадію розвитку проекту через неформальне спілкування з розробниками.

Неформальний моніторинг часто допомагає виявляти потенційні проблеми, які в явному вигляді можуть виявитися пізніше. Наприклад, щоденне обговорення ходу виконання робіт може виявити окремі недоробки у створюваному програмному продукті. Замість очікування звітів, у яких буде відображений факт затримання графіка робіт, менеджер може обговорити з фахівцями існуючі проблеми і запобігти зриву графіка робіт.

Протягом реалізації проекту зазвичай проводиться кілька формальних контрольних перевірок ходу виконання робіт зі створення ПЗ. Такі перевірки повинні дати загальну картину ходу реалізації проекту і показати, наскільки вже розроблена частина ПЗ відповідає цілям проекту.

Час виконання великих програмних проектів може займати кілька років. Протягом цього часу цілі і наміри організації, що замовила програмний проект, можуть істотно змінитися. Може виявитися, що розроблюваний програмний продукт став уже непотрібним або вихідні вимоги до створюваного ПЗ морально застаріли і їх необхідно кардинально змінювати. У такій ситуації керівництву організації-розробника треба прийняти рішення щодо припинення розроблення ПЗ або про зміну проекту в цілому для того, щоб урахувати цілі, що змінилися, і наміри організації-замовника.

Керівники – менеджери проектів зазвичай самі підбирають виконавців для своїх проектів. В ідеальному випадку професійний рівень виконавців повинен відповідати тій роботі, яку вони виконуватимуть у ході реалізації проекту. Однак у багатьох випадках менеджерам доводиться покладатися на команду розробників, що

далека від ідеальної. Така ситуація може бути зумовлена такими причинами:

– бюджет проекту не дозволяє залучити висококваліфікований персонал;

– організація хоче підвищити професійний рівень своїх працівників. Для цього вона може залучити до участі в проекті недостатньо кваліфікованих працівників, щоб вони здобули потрібний досвід у більш досвідчених фахівців.

Таким чином, майже завжди підбір фахівців для виконання проекту має певні обмеження і не є довільним. Разом з тим необхідно, щоб хоча б кілька членів групи розробників мали кваліфікацію і досвід, достатні для роботи над проектом. У противному випадку неможливо уникнути помилок у процесі розроблення ПЗ.

Менеджер проекту зобов'язаний надсилати звіти про хід його виконання як замовнику, так і підрядним організаціям. Це невеликі за обсягом документи, що ґрунтуються на інформації, взятій з докладних звітів про проект. Ці звіти містять інформацію, яка дозволяє оцінити ступінь готовності створюваного програмного продукту.

4.2. Планування проекту

Ефективне керування програмним проектом залежить безпосередньо від правильного планування робіт, необхідних для його виконання. План допомагає менеджеру передбачати проблеми, які можуть виникнути на будь-яких етапах створення ПЗ, і розробити превентивні заходи для їх попередження або вирішення. План, розроблений на початковому етапі проекту, розглядається всіма його учасниками як керівний документ, виконання якого передбачає успішне завершення проекту. Цей початковий план повинен максимально детально описувати всі етапи реалізації проекту.

Окрім розроблення плану проекту, менеджер зобов'язаний розробляти інші плани (табл. 4.1).

Таблиця 4.1

Види планів

План	Опис
Якості	Стандартів і заходів щодо підтримання якості розроблюваного ПЗ
Атестації	Способів, ресурсів і переліку робіт, необхідних для атестації програмної системи

Керування	Структури і процесів зміни конфігурації
Супроводження ПЗ	Плану заходів для супроводження ПЗ у процесі його експлуатації; розрахунок вартості супроводження і необхідні для цього ресурси
Керування персоналом	Заходи, спрямовані на підвищення кваліфікації членів команди розробників

Процес планування починається з визначення проектних обмежень (часових, можливості наявного персоналу, бюджету і т. ін.). Ці обмеження потрібно визначати одночасно з оцінюванням проектних параметрів (таких, як структура та розмір проекту) та розподілом функцій серед виконавців. Потім визначаються етапи розроблення і результати (документація, прототипи, підсистеми або версії програмного продукту), які отримують по закінченні цих етапів. Далі починається циклічна частина планування. Спочатку розробляється графік робіт з виконання проекту або надається дозвіл на продовження використання раніше створеного графіка. Після цього (приблизно через 2–3 тижні) здійснюється контроль виконання робіт і вказуються розбіжності між реальним і плановим ходом робіт.

У міру надходження нової інформації про хід виконання проекту можливий перегляд первинних оцінок параметрів проекту. Це, своєю чергою, може призвести до зміни графіка робіт. Якщо в результаті цих змін порушуються терміни завершення проекту, необхідно переглянути і узгодити із замовником ПЗ проектні обмеження.

Зазвичай більшість менеджерів проектів не сподіваються, що їх проекти реалізуються без будь-яких проблем. Бажано описати ймовірні проблеми ще до того, як вони виявляться в ході виконання проекту. Тому доцільно встановлювати «песимістичні» графіки робіт, ніж «оптимістичні». Оскільки неможливо побудувати план, у якому враховувалися б усі, у тому числі випадкові, проблеми і затримання виконання проекту, тому виникає потреба у періодичному перегляді проектних обмежень та етапів створення програмного продукту.

У плані проекту потрібно чітко показати ресурси, необхідні для реалізації проекту, поділ робіт на етапи і часовий графік виконання цих етапів. У деяких організаціях план проекту складають як єдиний документ, що містить усі види планів (табл. 4.1). В інших випадках план проекту описує лише технологічний процес створення

ПЗ. У такому плані обов'язково наявні посилання на плани інших видів, які розробляються окремо від плану проекту.

Деталізація планів проектів відрізняється залежно від типу розроблюваного програмного продукту і організації-розробника. Складаючи пояснювальну записку до плану, варто дотримувати такі розділи.

1. *Вступ* – містить короткий опис цілей проекту і проектних обмежень (бюджетних, часових, бюджетних), важливих для керування проектом.

2. *Організація виконання проекту* – вказується опис досвіду команди розробників.

3. *Аналіз ризиків* – включає опис можливих проектних ризиків, імовірності їх прояву і стратегій, спрямованих на їх зменшення.

4. *Апаратні та програмні ресурси*, потрібні для реалізації проекту. Перелік апаратних засобів і ПЗ, необхідного для розроблення програмного продукту. Якщо апаратні засоби потрібно закуповувати, наводиться їх вартість разом із графіком закупівлі і постачання.

5. *Поділ робіт на етапи.* Процес реалізації проекту поділяють на окремі процеси, визначають етапи виконання проекту, наводять опис результатів кожного етапу і контрольні оцінки.

6. *Графік робіт*. У ньому відображаються залежності між окремими процесами (етапами) розроблення ПЗ, оцінки часу їх виконання і розподіл членів команди розробників по окремих етапах.

7. *Механізми моніторингу і контролю за ходом виконання проекту* – описується зміст надаваних менеджером звітів про хід виконання робіт, терміни їх надання, а також механізми моніторингу всього проекту.

План потрібно регулярно переглядати в процесі реалізації проекту. Одні частини плану, наприклад, графік робіт, змінюються часто, інші більш стабільні. Унесення змін у план потребує спеціальної організації документообігу, що дозволяє відслідковувати ці зміни.

Менеджеру для організації процесу створення ПЗ і керування ним необхідна певна інформація. Ця інформація може бути отримана лише у вигляді документів, що відображають виконання чергового етапу розроблення програмного продукту. Без неї не можна оцінити ступінь готовності створюваного продукту, визначити витрати або змінити графік робіт.

Під час планування процесу визначають контрольні результати етапів робіт. Для кожного контрольного оцінювання розробля-

ють звіт, який надається керівництву проекту. Цей звіт не може бути великого обсягу; він має містити короткі підсумки окремого завершеного етапу проекту. Результатом етапу не може бути твердження типу «написання 80 % коду програм», оскільки неможливо перевірити його завершення; подібна інформація є беззмістовною для керування, оскільки тут не відображається зв'язок цього етапу з іншими етапами створення ПЗ.

По завершенні основних етапів (розроблення специфікації, проектування та ін.) замовнику ПЗ надаються звітні проектні елементи. Це може бути документація, прототип програмного продукту, закінчені підсистеми ПЗ тощо. Звітні документи, надавані замовнику ПЗ, можуть збігатися з контрольними результатами виконання будь-якого етапу.

Для визначення контрольних оцінок весь процес створення ПЗ потрібно поділити на окремі етапи із зазначеним «виходом» (результатом) кожного етапу. Етапи розроблення специфікації вимог у випадку, коли для її перевірки використовується прототип системи, а також вихідні результати (контрольні оцінки) кожного етапу показано на рис. 4.1. Тут контрольними проектними елементами є вимоги і специфікація вимог.

Рис. 4.1. Етапи процесу розроблення специфікації

4.3. Графік робіт

Складання графіка – одна з найбільш відповідальних робіт, виконуваних менеджером проекту. Тут менеджер оцінює тривалість проекту, визначає ресурси, необхідні для реалізації окремих етапів робіт, і подає їх у вигляді узгодженої послідовності робіт. Якщо цей проект подібний до раніше реалізованого, то графік робіт останнього проекту береться за основу цього проекту. У цей проект вносяться зміни, що відповідають створюваному проекту.

Якщо проект є інноваційним, первинні оцінки тривалості і необхідних ресурсів напевно будуть надто оптимістичними, навіть якщо менеджер спробує передбачити всі можливі несподівані ситуації. Із цього погляду проекти створення ПЗ не відрізняються від великих інноваційних технічних проектів. Нові аеропорти, мости і навіть нові автомобілі, як правило, з'являються пізніше ніж оголошені терміни їх здачі або надходження на ринок через непередбачені виниклі проблеми і труднощі. Саме тому графіки робіт необхідно постійно обновлювати в міру надходження нової інформації про хід виконання проекту.

У процесі складання графіка (рис. 4.2) весь обсяг робіт, необхідних для реалізації проекту, поділяють на окремі етапи і оцінюють час, потрібний для виконання кожного етапу. Зазвичай багато етапів виконуються одночасно. У графіку робіт потрібно це передбачати і оптимально розподіляти виробничі ресурси між ними. Нестача ресурсів для виконання будь-якого критичного етапу – часта причина затримання виконання всього проекту.

Рис. 4.2. Процес складання графіка робіт

Тривалість етапів має бути не меншою як тиждень. Якщо вона буде меншою, то виявиться низька точність часових оцінок етапів, що може зумовити частий перегляд графіка робіт. Доцільно установити максимальну тривалість етапів, що не перевищує 10 тижнів. Якщо є етапи більшої тривалості, їх варто поділити на етапи меншої тривалості.

Розраховуючи тривалість етапів, менеджер повинен брати до уваги, що виконання будь-якого етапу не обійдеться без великих або малих проблем і затримань. Розробники можуть допускати помилки або затримувати роботу, техніка може вийти з ладу, а апаратні або програмні засоби підтримання процесу розроблення можуть надходити із запізненням. Якщо проект інноваційний і технічно складний, це стає додат-

ковим фактором виникнення непередбачених проблем і збільшення тривалості реалізації проекту порівняно з первинними оцінками.

Крім часових витрат, менеджер повинен розраховувати інші ресурси, необхідні для успішного виконання кожного етапу. Особливий вид ресурсу – команда розробників, залучена до виконання проекту. Інші види ресурсів:

– вільний дисковий простір на сервері;

– час використання спеціального устаткування;

– бюджетні витрати на відрядження персоналу, що працює над проектом.

Є таке емпіричне правило: оцінювати часові витрати так, начебто нічого непередбаченого і «поганого» не може трапитися, потім збільшити ці оцінки для врахування можливих проблем. Можливі, але важко прогнозовані проблеми істотно залежать від типу і параметрів проекту, а також від кваліфікації та досвіду команди розробників. До вихідних розрахункових оцінок доцільно додавати 30 % на можливі проблеми і ще 20 % на непередбачувані.

Графік робіт проекту являє собою набір діаграм і графіків, у яких відображають поділ проектних робіт на етапи, залежності між етапами і розподіл розробників по етапах. Часова діаграма показує час початку та закінчення кожного етапу і його тривалість. Мережева діаграма відображає залежності між різними етапами проекту. Ці діаграми можна створювати автоматично за допомогою програмних засобів підтримання керування на підставі інформації, уведеної в базу даних проекту.

Із проекту (табл. 4.2) видно, що етап Т3 залежить від етапу Т1. Це означає, що етап Т1 має завершитися раніше, ніж почнеться етап Т3. Наприклад, на етапі Т1 виконується компонентний аналіз створюваного програмного продукту, а на етапі Т3 – проектування системи.

Таблиця 4.2

Етапи проекту

Етап	Тривалість, дні	Залежність
Т1	8	–
Т2	15	–
Т3	15	Т1 (М1)
Т4	10	–
Т5	10	Т2, Т4 (М2)

Т6	5	Т1.Т2 (М3)
Т7	20	Т1(М1)
Т8	25	Т4(М5)
Т9	15	Т3, Т6(М4)
Т10	15	Т5, Т7 (М7)
Т11	7	Т9 (М6)
Т12	10	Т11 (М8)

На підставі наведених у табл. 4.2 тривалостей етапів і залежності між ними будується мережевий графік послідовності етапів (рис. 4.3).

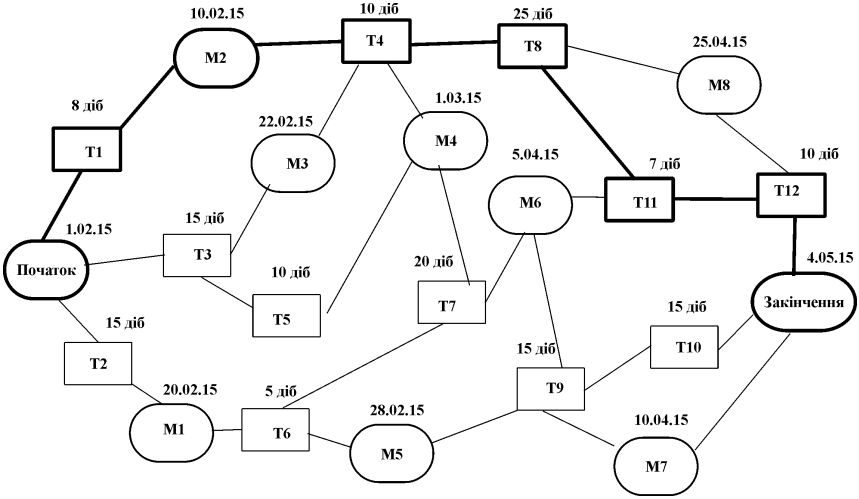


Рис. 4.3. Мережева діаграма етапів

Графік показує, які роботи можна виконувати одночасно, а які послідовно. Етапи позначено прямокутниками. Контрольні оцінки та звітні документи показано у вигляді овалів і позначено (як і в табл. 4.2) літерою М з відповідним номером. Дати на діаграмі відповідають початку виконання етапів. Мережеву діаграму варто читати зліва направо і зверху донизу.

Якщо для побудови мережевої діаграми використовуються ПЗ підтримання керування проектом, кожний етап потрібно закінчувати звітом. Черговий етап може початися тільки після отримання оцінки за результатом звіту, яка залежить від декількох попередніх етапів. Тому в третьому стовпчику табл. 4.2 наведено контрольні оцінки; вони

будуть досягнуті лише тоді, коли буде завершено етап, у рядку якого поміщено відповідну контрольну оцінку.

Будь-який етап не може початися доти, доки не виконані всі етапи по всіх шляхах, що ведуть від початку проекту до даного етапу. Наприклад, етап Т7 не може початися, доки не завершаться етапи Т4, Т5, Т6. Досягнення контрольної оцінки М4 свідчить про завершення попередніх етапів.

Мінімальний час виконання всього проекту можна розрахувати, якщо підсумувати у мережевій діаграмі тривалості етапів на найдовшому шляху від початку проекту до його закінчення (***критичний шлях***). Тривалість проекту (рис. 4.3) становить 12 тижнів або 60 робочих днів. Критичний шлях показано товстішими лініями, ніж інші шляхи. Таким чином, загальна тривалість реалізації проекту залежить від етапів робіт, що перебувають на критичному шляху. Затримання завершення будь-якого етапу критичного шляху призведе до затримання виконання всього проекту.

Затримання завершення етапів, що не входять у критичний шлях, не впливає на тривалість усього проекту доти, доки сумарна тривалість цих етапів (з урахуванням затримань) на якому-небудь шляху не перевищить тривалості робіт на критичному шляху. Наприклад, якщо етап Т10 затримується на термін, менший за 20 днів, це не вплине на загальну тривалість проекту.

Мережева діаграма дозволяє побачити в залежності етапів значущість того або іншого етапу для реалізації всього проекту. Увага до етапів критичного шляху часто дає змогу знайти способи їх зміни для скорочення тривалості усього проекту. Менеджери використовують мережеву діаграму для розподілу робіт.

Інший графік робіт показано на рис. 4.4. Це часова діаграма Ганта, яка будується за допомогою *Microsoft Project* – програмного засобу підтримання процесу керування. Вона здатна показувати тривалість виконання кожного етапу і можливі їх затримання, показані затіненими прямокутниками, а також дати початку та закінчення кожного етапу. Якщо етапи критичного шляху не мають затінених прямокутників, це означає, що затримання завершення цих етапів призведе до збільшення тривалості всього проекту.

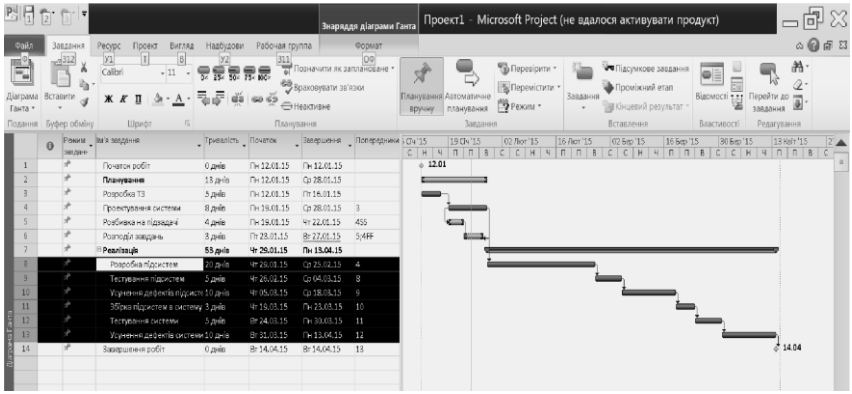


Рис. 4.4. Діаграма Ганта

Подібно до розподілу часу виконання етапів менеджер повинен розрахувати розподіл ресурсів за етапами, зокрема призначити виконавців на кожний етап. У табл. 4.3 наведено розподіл розробників на кожний етап, показаний на рис. 4.4.

Таблиця 4.3

Розподіл виконавців за етапами

Етап	Виконавець	Етап	Виконавець
Т1	Денис	Т7	Данил
Т2	Ганна	Т8	Федір
Т3	Денис	Т9	Денис
Т4	Федір	Т10	Ганна
Т5	Марія	Т11	Федір
Т6	Ганна	Т12	Федір

Таблиця 4.3 може бути використана ПЗ підтримання процесу керування для побудови часової діаграми зайнятості співробітників на певних етапах робіт. Протягом цього періоду співробітники можуть бути у відпустці, працювати над іншими проектами, навчатися і т. ін.

У великих організаціях працює багато фахівців, які залучаються до виконання проекту в міру потреби. Такий підхід може створити певні проблеми для менеджерів проектів. Наприклад, якщо фахівець зайнятий у проекті, що затримується, це може становити складності для інших проектів, де він також повинен брати участь.

Початковий графік робіт неминуче містить помилки або недоробки. У міру реалізації проекту розраховані оцінки тривалості виконання етапів робіт повинні відповідати реальним термінам вико-

нання цих етапів. Результати порівняння належить використовувати як основу для перегляду графіка робіт ще нереалізованих етапів проекту, зокрема для того, щоб спробувати зменшити тривалість етапів критичного шляху.

4.4. Керування ризиками

Важливою частиною роботи менеджера проекту є оцінювання ризиків, які можуть вплинути на графік робіт або на якість створюваного програмного продукту, і розроблення заходів щодо запобігання ризикам. Результати аналізу ризиків мають бути відображені в плані проекту. Визначення ризиків і розроблення заходів щодо зменшення їх впливу на хід виконання проекту називається керуванням ризиками.

Спрощено ризик розуміють як імовірність прояву будь-яких несприятливих обставин, що негативно впливає на реалізацію проекту. Ризики загрожують проекту в цілому, створюваному програмному продукту або організації-розробнику. Можна виокремити три типи ризиків:

1) *ризики для проекту*, що впливають на графік робіт або ресурси, необхідні для виконання проекту;

2) *ризики для розроблюваного продукту*, що впливають на якість або продуктивність розроблюваного програмного продукту;

3) *бізнес-ризики*, що стосуються організації-розробника або постачальників.

Ці типи ризиків можуть перетинатися. Наприклад, якщо досвідчений програміст залишає проект, це буде ризиком для проекту, оскільки затримується термін здачі готового продукту, ризиком для продукту, тому що новий програміст може виявитися не надто досвідченим і робити помилки в програмі, і бізнес-ризиком бо затримання проекту може негативно вплинути на майбутні ділові контакти між замовником і організацією-розробником.

Конкретні типи ризиків, які можуть вплинути на проект, залежать від виду створюваного програмного продукту та організаційного оточення, де реалізується програмний проект. Разом з тим багато типів ризиків здатні вплинути на будь-які програмні проекти (табл. 4.4).

Таблиця 4.4

Можливі ризики програмних проектів

Ризик	Тип

Плинність кадрів	Ризик для проекту
Заміна керівництва організації	Ризик для проекту
Неготовність апаратних засобів	Ризик для проекту
Зміна вимог	Ризик для проекту та розроблюваного продукту
Затримання розроблення специфікації	Ризик для проекту та розроблюваного продукту
Недооцінювання розміру розроблюваної системи	Ризик для проекту та розроблюваного продукту
Неефективність *CASE*-засобів	Ризик для розроблюваного продукту
Зміни в технології розроблення ПЗ	Бізнес-ризик
Поява конкуруючого програмного продукту	Бізнес-ризик

Схему процесу керування ризиками показано на рис. 4.5. Цей процес складається із чотирьох стадій.

1. *Визначення ризиків* для проекту, розроблюваного продукту і бізнес-ризиків.

2. *Аналіз ризиків* за допомогою оцінювання ймовірності та послідовності появи ризикових ситуацій.

3. *Планування ризиків* для реалізації заходів щодо запобігання їм або мінімізації їх впливу на проект.

4. *Моніторинг ризиків* шляхом постійного оцінювання ймовірностей ризиків і виконання заходів щодо пом'якшення наслідків прояву ризикових ситуацій.

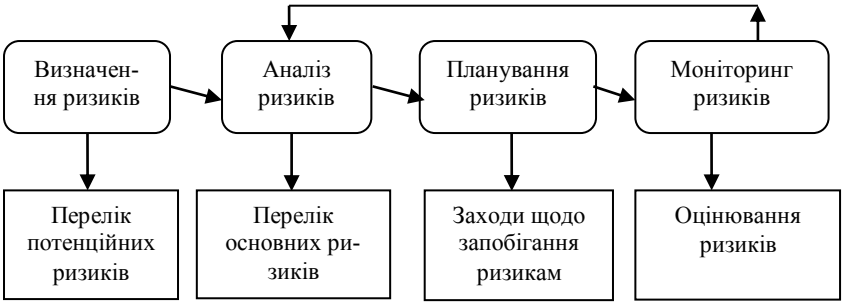


Рис. 4.5. Процес керування ризиками

Процес керування ризиками, як і інші процеси, є ітераційним, виконуваним протягом усього терміну реалізації проекту. Спочатку

розробляють плани керування ризиками, потім постійно відслідковують ситуацію навколо реалізації проекту. Із надходженням нової інформації про можливі ризики заново аналізують ризики і першорядну увагу приділяють новим ризикам. У міру надходження нової інформації також змінюються плани заходів щодо запобігання і пом'якшення ризиків.

Результати процесу керування ризиками документують у вигляді планів керування ризиками. Вони містять опис можливих проектних ризиків, їх аналіз і перелік заходів, необхідних для керування ризиками.

Визначення ризиків – перша стадія процесу керування ризиками. На цій стадії описуються ризики, які можуть виявитися під час реалізації проекту, і не оцінюється ймовірність та значущість ризиків, але на практиці малоймовірні ризики з незначними наслідками зазвичай до уваги не беруть.

Ризики можуть визначатися в режимі командної роботи з використанням підходу «мозковий штурм» або ґрунтуватися на досвіді менеджера. Для визначення ризиків можна використати такі можливі категорії ризиків:

1) *технологічні ризики* пов'язані із програмними і апаратними технологіями, відповідно до яких розробляється система;

2) *ризики, пов'язані з персоналом;*

3) *організаційні ризики;*

4) *інструментальні ризики;*

5) *ризики, пов'язані із системними вимогами;*

6) *ризики оцінювання.*

Деякі приклади ризиків з можливих категорій ризиків наведено у табл. 4.5. Результатом етапу визначення ризиків буде множина можливих ризиків, які можуть вплинути на розроблюваний програмний продукт, проект або організацію-розробника.

Таблиця 4.5

Типові ризики

Категорія	Приклади
Технологічні	База даних, що використовується в програмній системі, не забезпечує оброблення очікуваного обсягу транзакцій. Програмні компоненти, які використовуються повторно, мають дефекти, що обмежують їх функціональні можливості

Пов'язані з персоналом	Неможливо підібрати працівників з належним професійним рівнем. Провідний розробник занедужав у критичний час. Неможливо організувати необхідне навчання персоналу
Організаційні	В організації, що розробляє ПЗ, відбулася реорганізація, у результаті чого змінилися пріоритети в керуванні проектом. Фінансові утруднення в організації призвели до зменшення бюджету проекту
Інструментальні	Програмний код, що генерується *CASE*-засобами, не ефективний. *CASE*-засоби неможливо інтегрувати з іншими засобами підтримання проекту

Закінчення табл. 4.5

Категорія	Приклади
Ризики, пов'язані із системними вимогами	Зміни вимог приводять до повторних робіт із проектування системи. Первинне нечітке формулювання користувацьких вимог зумовило значні зміни системних вимог, що виявилися на пізніх стадіях розроблення проекту
Ризики оцінювання	Недооцінювання часу виконання проекту. Швидкість виявлення дефектів у системі менша від раніше запланованої. Розмір системи значно перевищує розрахований на початковому етапі

Під час аналізу для кожного ризику розраховують імовірність його прояву і збиток, якого він може завдати. Не існує простих методів виконання аналізу ризиків – значною мірою він ґрунтується на інтуїції та досвіді менеджера. Можна навести таку шкалу ймовірностей ризиків і їх наслідків.

1. Імовірність ризику вважається дуже низькою, якщо вона має значення менше за 10 %; низькою, якщо її значення становить 10–25 %; середньою за значення 25–50 %; високою, якщо значення – 50–75 %; дуже високою, якщо значення перевищує 75 %.

2. Можливий збиток від ризикових ситуацій можна поділити на катастрофічний, істотний, терпимий і незначний.

Результати аналізу ризиків мають бути подані у вигляді таблиці ризиків, упорядкованих за ступенем можливого збитку. У табл. 4.6

наведено упорядкований перелік ризиків, описаних у табл. 4.5; там же зазначені ймовірності цих ризиків. Тут імовірності ризиків і ступінь збитку від них зазначені довільно. На практиці для їх визначення необхідна докладна інформація про проект, технологію створення ПЗ, команду розробників і саму організацію.

Таблиця 4.6

Перелік ризиків після проведення їх аналізу

Ризик	Імовірність	Ступінь збитку
Фінансові ускладнення в організації	Низька	Катастрофічний
Низький професійний рівень працівників	Висока	Катастрофічний
Хвороби розробників	Середня	Істотний
Дефектні програмні компоненти	Те саме	Те саме

Закінчення табл. 4.6

Ризик	Імовірність	Ступінь збитку
Зміни вимог	–‖–	–‖–
Зміни пріоритетів у керуванні проектом	Висока	–‖–
База даних, що використовується в програмній системі, не забезпечує оброблення очікуваного обсягу транзакцій	Середня	–‖–
Недооцінювання часу виконання проекту	Висока	–‖–
Неможливість інтеграції *CASE*-засобів з іншими засобами підтримання проекту	Висока	Прийнятний
Первинне нечітке формулювання користувацьких вимог	Середня	Те саме
Неможливість організації навчання персоналу	Те саме	–‖–
Швидкість виявлення дефектів у системі менша від планової	–‖–	–‖–
Розмір системи значно перевищує початковий	Висока	–‖–
Неефективність програмного коду, що генерується *CASE*-засобами	Середня	Незначний

Як імовірність ризиків, так і можливий збиток від них потрібно переглядати з надходженням додаткової інформації про ці ризики і в міру реалізації заходів щодо керування ними. Тому подібні

таблиці ризиків необхідно опрацьовувати на кожній ітерації процесу керування ризиками.

Після аналізування ризиків визначаються найбільш значущі ризики, які потім відслідковуються протягом усього терміну виконання проекту. Визначення цих значущих ризиків залежить від їх імовірностей і можливого збитку. У загальному випадку завжди відслідковуються ризики з катастрофічними наслідками, а також ризики з істотним збитком, значення ймовірності яких вище за середнє.

Кількість ризиків, які необхідно відслідковувати, залежить від конкретного проекту. Це може бути п'ять ризиків, а може – п'ятнадцять. Але кількість ризиків, за якою проводиться моніторинг, має бути доступною для розгляду. Велика кількість ризиків, що відслідковуються, потребує величезного обсягу зібраної інформації. Із переліку ризиків, наведених у табл. 4.6, для моніторингу варто відібрати всі вісім ризиків, які можуть призвести до катастрофічних та істотних наслідків.

Планування полягає у визначенні стратегії керування кожним значущим ризиком, відібраним для моніторингу після аналізу ризиків. Тут також не існує загальноприйнятих підходів для розроблення таких стратегій – багато аспектів ґрунтується на «чутті» і досвіді менеджера проекту. У табл. 4.7 наведено можливі стратегії керування основними ризиками (див. табл. 4.6). Є три категорії стратегій керування ризиками:

1) *стратегії запобігання ризикам*, які ґрунтуються на заходах, що знижують імовірність прояву ризиків;

2) *стратегії мінімізації*, які спрямовані на зменшення можливого збитку від ризиків;

3) *планування «аварійних» ситуацій.*

Таблиця 4.7

Стратегії керування ризиками

Ризик	Стратегія
Фінансові проблеми організації	Підготувати документ для керівництва організації, що показує важливість проекту для досягнення фінансових цілей організації
Проблеми некваліфікованого персоналу	Попередити замовника про потенційні труднощі та можливе затримання проекту, розглянути питання купівлі компонентів системи
Хвороби	Визначити обов'язки і роботу членів команди так,

персоналу	щоб вони перекривались
Дефектні системні компоненти	Замінити потенційно дефектні системні компоненти покупними компонентами, що гарантують якість роботи
Зміни вимог	Визначити найбільш імовірні вимоги, які підлягають змінам; у структурі системи не відображати детальну інформацію
Реорганізація компанії-розробника	Підготувати документ керівництву компанії, у якому відображено значущість проекту для досягнення фінансових цілей компанії
Недостатня продуктивність бази даних	Розглянути можливість купівлі більш продуктивної бази даних
Недооцінювання часу виконання проекту	Розглянути питання про купівлю системних компонентів, дослідити можливість використання генератора програмного коду

Моніторинг ризиків полягає в регулярному перегляді ймовірностей ризиків і збитку, яких вони можуть завдати. Для цього необхідно постійно відслідковувати фактори, які впливають на ймовірність ризиків і можливий збиток. Ці фактори залежать від типів ризику. Ознаки, які допомагають визначити тип ризику, наведено в табл. 4.8.

Таблиця 4.8

Ознаки ризиків

Тип ризику	Ознаки
Технологічні ризики	Затримання постачання устаткування або програмних засобів підтримання процесу створення ПЗ, численні документовані технологічні проблеми
Ризики, пов’язані з персоналом	Низький моральний стан персоналу, напружені стосунки між членами команди розробників, низька якість виконаної роботи
Організаційні ризики	Розмови серед персоналу про пасивність і недостатню компетентність вищого керівництва організації
Інструментальні ризики	Небажання розробників використовувати програмні засоби підтримання, несхвальні відгуки про *CASE*-засоби, запити нових інструментальних засобів
Ризики, пов’язані із системними	Необхідність перегляду багатьох системних вимог, незадоволення замовника ПЗ

вимогами	
Ризики оцінювання	Зміни графіка робіт, численні звіти про порушення графіка робіт

Моніторинг ризиків має бути безперервним процесом, що відслідковує хід виконання заходів щодо керування ризиками, при цьому кожний основний ризик необхідно розглядати окремо.

4.5. Керування фінансовими потоками

Важливим завданням менеджера проекту є керування фінансовими розрахунками із замовниками та підрядниками. Основні функції *Microsoft Project* забезпечує достатнє керування вартістю проекту, яке необхідне для зручного відстеження платежів. Основна операція проводиться через вкладку «Звіт» (рис. 4.6).

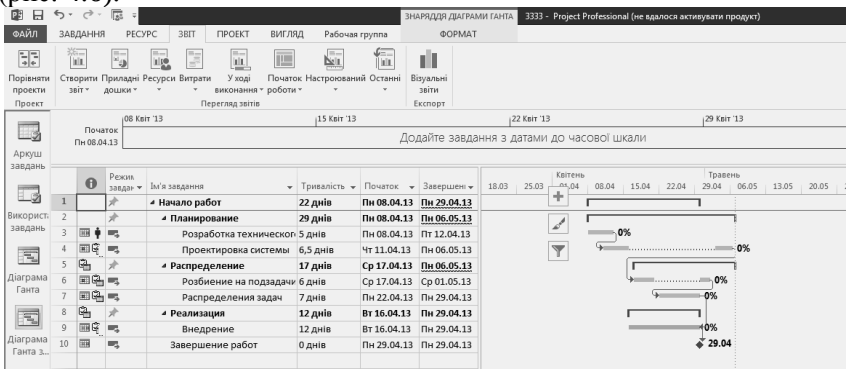


Рис. 4.6. Вікно створення нового звіту

Проблема звітності у *MS Project* полягає в неможливості створити стандартний звіт (графік) з даних за ресурсами і рухом грошей. Проте така функція є, хоча і прихована в розширених налаштуваннях (рис. 4.7).

Рис. 4.7. Вікно створення звіту за часом

Як видно з рис. 4.8, неможливо об'єднати ці дані і показати більш точну інформацію про рух грошей. Цю проблему усувають створенням звіту в *Excel*. Для цього спочатку обирається «Звіт про рух грошових коштів» і натискається кнопка «Новий шаблон», після цього обирається «Звіт про плановані витрати» (рис. 4.9).

Далі обирають дані для звіту, які розміщують у новому документі *MS Excel*. Інформація для рядків і / або стовпців (час і ресурси) обирається перетягуванням з приладової панелі (права частина екрана) до рядка або стовпчика (рис. 4.10). На основі отриманих даних створюється діаграма.

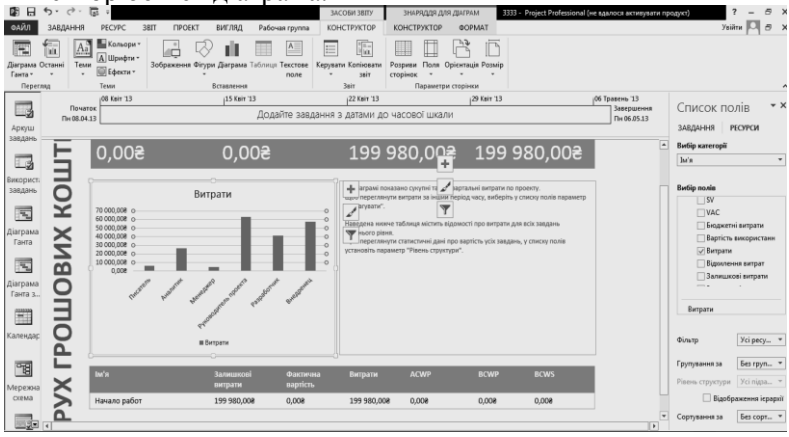


Рис. 4.8. Вікно створення звіту за ресурсами

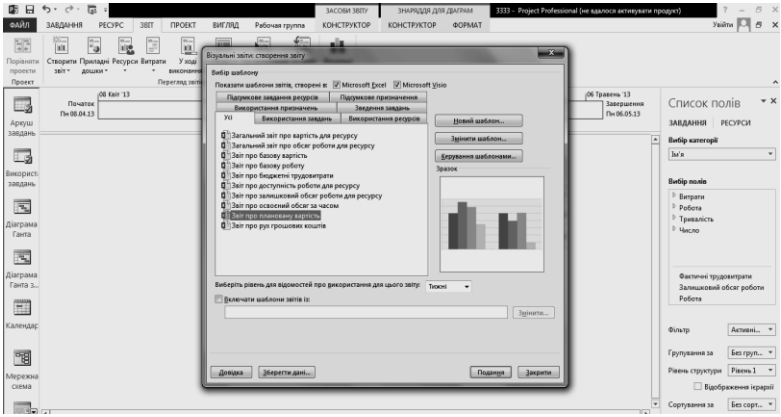


Рис. 4.9. Вікно вибору типу звіту

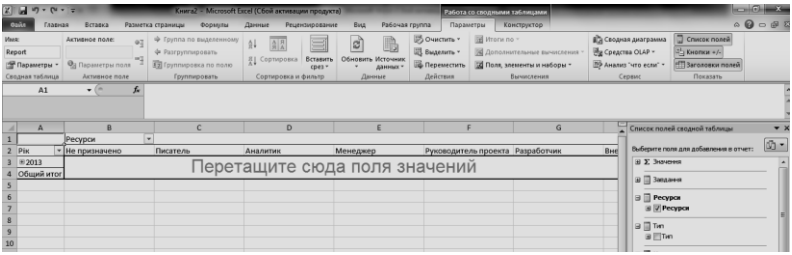


Рис. 4.10. Вікно вибору даних для відображення у звіті

Для цього необхідно виділити всі таблиці, натиснути на кнопку «Склеювання», вибрати формат діаграми *Gistaram* (рис. 4.11). Отримують діаграму, яка відображає інформацію про рух грошей як у часі, так і для кожного ресурсу.

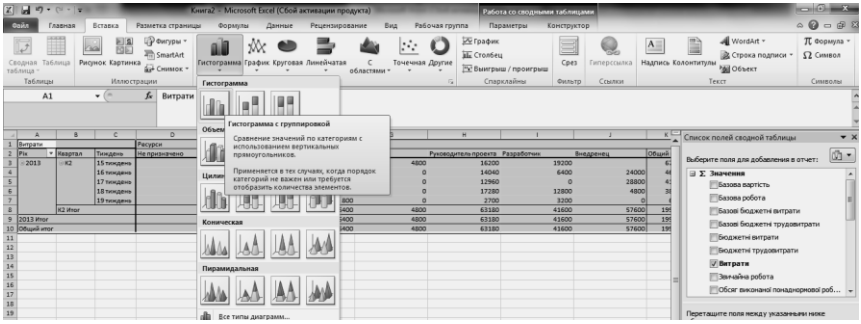


Рис. 4.11. Вікно вибору формату відображення звіту

Контрольні запитання і завдання

1. Поясніть проблеми керування програмними проектами.
2. Укажіть та поясніть відмінності процесу розроблення ПЗ і технічних проектів.
3. Назвіть та поясніть складові процесу керування.
4. Як досягається неформальний моніторинг за проектами?
5. Поясніть основні проблеми розробників, що виникають під час розроблення проектів.
6. Поясніть основні завдання планування проекту.
7. Поясніть основні види планів.
8. Наведіть та поясніть основні складові процесу планування.
9. Наведіть основні складові етапу розроблення специфікації.
10. Розкрийте ризики та основні можливості щодо керування ними.
11. Які ви знаєте можливі ризики програмних проектів?
12. Поясніть методи та підходи до планування ризиків.
13. Проаналізуйте основні стратегії керування ризиками.
14. Поясніть доцільність моніторингу ризиків.
15. Наведіть основні ознаки ризиків.

II. ПРОЕКТУВАННЯ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

5. АРХІТЕКТУРНЕ ПРОЕКТУВАННЯ

5.1. Загальний підхід до проектування архітектури

Архітектурним проектуванням називають перший етап процесу проектування, на якому визначають підсистеми, структуру керування і взаємодію підсистем. Мета архітектурного проектування – опис *архітектури програмного забезпечення*.

Архітектурне проектування є з'єднувальною ланкою між процесом проектування і процесом розроблення вимог до створюваної системи. Технічне завдання, як правило, не містить інформації про структуру системи. Таке твердження справедливе лише для невеликих систем. Архітектурна декомпозиція системи необхідна для структуризації й організації системної специфікації. Модель системної архітектури часто є відправною точкою для створення специфікації різних частин системи. У процесі архітектурного проектування розробляється базова структура системи, тобто визначаються основні компоненти системи і взаємодія між ними.

Існують різні підходи до процесу архітектурного проектування, які залежать від професійного досвіду, а також майстерності та інтуїції розробників. Але можна виділити декілька етапів, загальних для всіх процесів архітектурного проектування.

Структурування. На цьому етапі система подається сукупністю незалежних підсистем та визначається їх взаємодією.

Моделювання керування. Розробляється базова модель керування взаємодіями між частинами системи.

Модульна декомпозиція. Підсистема поділяється на окремі модулі. Тут визначаються типи модулів та їх взаємозв'язки.

Ці етапи перетинаються, накладаються один на одний. Етапи можуть повторюватися для більш детального опрацювання архітектури, поки проект не буде задовольняти системні вимоги.

Підсистема – це незалежний елемент системи, який має власні операції (методи). Підсистеми складаються з модулів і мають певні інтерфейси, за допомогою яких вона взаємодіє з іншими підсистемами.

Модуль – це компонент системи, що надає один або кілька сервісів іншим. Модуль може використовувати сервіси, підтримувані іншими модулями. Цей компонент ніколи не розглядається як незалежна система. Модулі зазвичай складаються з ряду інших, більш простих компонентів.

Результатом процесу архітектурного проектування є документ, що відображає архітектуру системи. Він складається з набору графічних схем подань моделей системи з відповідним описом. В описі повинно бути зазначено, з яких підсистем і модулів складається кожна підсистема. Графічна схема моделі системи дозволяє всебіч-

но розглянути архітектуру. Як правило, розробляються чотири архітектурні моделі.

1. *Статична модель*, у якій підсистеми та розроблювані компоненти розглядаються незалежно.

2. *Динамічна модель*, у якій процеси демонструють часову роботу системи.

3. *Модель інтерфейсу*, що визначає сервіси кожної підсистеми.

4. *Моделі взаємодії підсистем*.

Для опису архітектури доцільно використовувати неформальні моделі та описи системи, які надає *UML*. Архітектура системи будується відповідно до певної моделі. Важливо знати ці моделі, їх недоліки, переваги і можливості застосування. Разом з тим архітектуру великих систем неможливо описати за допомогою будь-якої однієї моделі. Розробляючи окремі частини великих систем, варто використовувати різні архітектурні моделі. При цьому архітектура системи може виявитися надто складною. Завданням розробника є вибір найбільш придатної моделі, яка б відповідала вимогам розроблюваного ПЗ.

Архітектура системи впливає на продуктивність, надійність, зручність супроводження та інші характеристики системи.

Продуктивність. Забезпечується найменшою кількістю підсистем з якнайменшою взаємодією між ними через використання великомодульних компонентів.

Захищеність. Створюється багаторівневою структурою, коли найбільш критичні системні елементи захищені на внутрішніх рівнях, а безпека цих рівнів перевіряється на більш високому рівні.

Безпека. Забезпечується застосуванням якнайменшої кількості підсистем, операції яких впливають на безпеку системи.

Надійність. Забезпечується включенням надлишкових компонентів так, щоб їх можна було замінювати і обновляти, не переривaючи роботу системи.

Зручність супроводу. Для цього архітектуру системи проектують на рівні дрібних структурних компонентів, які можна легко змінювати. Програми, що створюють дані, повинні бути відділені від програм, що використовують ці дані. Варто також уникати структури зі спільним використанням даних.

Очевидно, що деякі архітектури суперечать одна одній. Наприклад, для того щоб підвищити продуктивність, необхідно викорис-

товувати великомодульні компоненти. При цьому супроводження системи набагато спрощується, якщо вона складається із дрібних структурних компонентів. Якщо необхідно врахувати обидві вимоги, варто шукати компромісне рішення. Один зі способів вирішення подібних проблем полягає в застосуванні різних архітектурних моделей для різних частин системи.

5.2. Структурування системи

На першому етапі процесу проектування архітектури система поділяється на кілька взаємодійних підсистем. На абстрактному рівні системну архітектуру зображають графічно у вигляді блок-схеми, у якій окремі підсистеми подаються окремими блоками. Підсистему також можна розбити на кілька частин; на діаграмі ці частини зображуються прямокутниками всередині великих блоків. Потоки даних і/або потоки керування між підсистемами позначають стрілками. Така блок-схема дає загальне уявлення про структуру системи.

Структурну модель архітектури експертної системи показано на рис. 5.1. Вона складається з таких підсистем: підсистеми пояснень та здобування знань, підсистем прийняття рішення, бази знань робочої пам’яті та інтерфейсу.

Із погляду розробника ПЗ подібні блок-схеми є даремним поданням системної архітектури, оскільки з них не можна довідатися ні про природу взаємодії між компонентами системи, ні про їх властивості. Однак такі моделі є ефективними на етапі попереднього проектування системи. Ця модель не перевантажена деталями, за її допомогою зручно виразити взаємодію компонентів системи. Структурна модель визначає всі основні підсистеми, які можна розробляти незалежно від інших підсистем. Отже, завдання керівника проекту – розподілити процес розроблення цих підсистем між різними виконавцями.

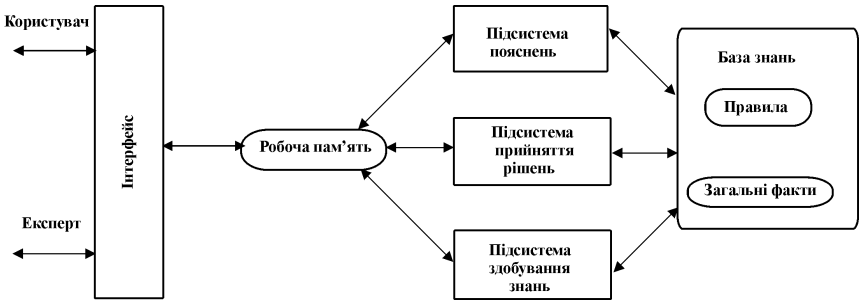


Рис. 5.1. Структурна модель архітектури експертної системи

Розробляються і більш деталізовані моделі структури, у яких показується, як підсистеми поділяють дані і як вони взаємодіють. Розглянемо стандартні моделі: репозиторію, клієнт/сервер і абстрактної машини.

Для того щоб підсистеми, що утворюють систему, працювали ефективніше, між ними має відбуватися обмін інформацією. Обмін можна організувати двома способами:

– всі спільно використовувані дані зберігаються в центральній базі даних, доступній всім підсистемам;

– кожна підсистема має власну базу даних, взаємообмін у них відбувається за допомогою передавання повідомлень.

Модель системи, заснована на спільному використанні бази даних, часто називають *репозиторієм*. Більшість систем, що обробляють великі обсяги даних, організовані навколо спільно використовуваної бази даних, або репозиторію. Така модель придатна для додатків, у яких дані створюються в одній підсистемі, а використовуються іншою. Наприклад, системи керування інформацією, системи автоматичного проектування і *CASE*-засоби.

Приклад архітектури інтегрованого набору *CASE*-інструментів, що ґрунтується на спільно використовуваному репозиторії показано на рис. 5.2. Вважається, що для *CASE*-засобів перший спільно використовуваний репозиторій був розроблений на початку 1970-х років англійською компанією *ICL* у процесі створення своєї операційної системи [5].

Рис. 5.2. Архітектура інтегрованого набору *CASE*-засобів

Переваги моделі:

– ефективність спільного використання великих обсягів даних, оскільки не потребує передавання даних з однієї підсистеми в іншу;

– автономність підсистем, тому не потрібно знати, як використовуються створювані дані в інших підсистемах;

– прозорість спільного використання, якщо нові підсистеми сумісні з узгодженою моделлю даних, їх можна безпосередньо інтегрувати в систему;

– централізованість резервного копіювання, забезпечення безпеки, керування доступом і відновлення даних, оскільки входять у систему керування репозиторієм;

– однакова політика до безпеки, відновлення і резервування даних.

Недоліки моделі:

– необхідність компромісу між вимогами до даних, пропонованими до кожної підсистеми. Вони мають бути узгоджені з моделлю репозиторію. Компромісне рішення може знизити продуктивність підсистем;

– проблемна модернізація таких систем. Оскільки в узгодженій моделі даних генеруються великі обсяги інформації, тому переведення системи на нову модель даних становиться дорогим, складним, іноді навіть неможливим;

– складність розподіленої організації у зв’язку з надмірністю і порушенням цілісності даних.

Розглянута модель репозиторію подає його пасивним елементом, керування покладається на підсистеми, що «витягують» з нього дані.

Архітектура клієнт/сервер – це модель розподіленої системи, у якій показано розподіл даних і процесів між декількома процесорами. Вона включає три основні компоненти: набір автономних серверів, набір клієнтів і мережу.

Набір автономних серверів, які надаються іншим підсистемам. Наприклад, сервер друку, що надає послуги друку, файлові сервери,

які надають сервіси керування файлами, сервер-компілятор, що пропонує сервіси з компіляції вихідних кодів програм, хмарний та ін.

Набір клієнтів – викликають сервіси, що надаються серверами. У контексті системи клієнти є підсистемами. Допускається одночасне виконання декількох екземплярів клієнтської програми.

Мережа – за їх допомогою клієнти отримують доступ до сервісів. Немає жодної заборони на те, щоб клієнти і сервери запускалися на одній машині. Однак на практиці модель клієнт/сервер у такій ситуації не використовується.

Клієнти повинні знати імена доступних серверів і сервісів, які вони викликають. Водночас серверам не потрібно знати імен клієнтів та їх кількість. Клієнти отримують доступ до сервісів за допомогою віддаленого виклику процедури запиту.

Приклад системи, організованої за моделлю клієнт/сервер, показано на рис. 5.3. Це багатокористувацька гіпертекстова система, призначена для підтримання бібліотек фільмів і фотографій. У ній утримується кілька серверів, які розміщають різні типи медіафайлів і керують ними. Відеофайли потребують швидкого і синхронного передавання. Вони зберігаються у стислому стані. Фотографії передаються з високою роздільною здатністю. Каталоги повинні забезпечувати роботу з множиною запитів і підтримувати зв'язки з гіпертекстовою системою. Клієнтська програма є інтегрованим інтерфейсом користувача.

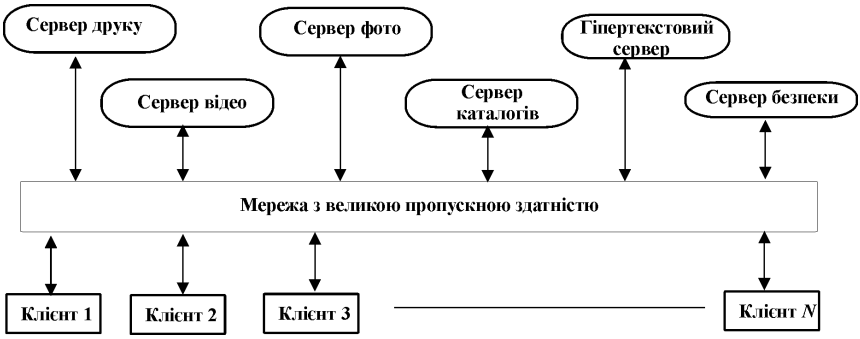


Рис. 5.3. Архітектура бібліотечної системи

Проектування системи за клієнт/серверною архітектурою можна використовувати для реалізації системи, заснованої на репозиторії, що виконує функції сервера системи. Підсистеми, що мають доступ до репозиторію, є клієнтами. Але кожна підсистема керує власними

даними. Під час роботи сервери і клієнти обмінюються даними, однак у разі обміну великими обсягами даних можуть виникати проблеми, пов'язані із пропускною здатністю мережі. Із розвитком високошвидкісних мереж ця проблема нівелюється.

Найбільш важливою перевагою моделі клієнт/сервер полягає в тому, що вона є розподіленою архітектурою. Її ефективно використовувати в мережевих системах з множиною розподілених процесорів. У систему легко додати новий сервер і інтегрувати його з іншою частиною системи або ж оновити сервери, не впливаючи на інші частини системи.

Модель архітектури абстрактної машини (багаторівнева модель) показує взаємодію підсистем. Вона організовує систему у вигляді набору рівнів, кожний з яких надає свої сервіси. Кожний рівень визначає *абстрактну машину*, машинна мова якої (сервіси, надавані рівнем) використовується для реалізації наступного рівня абстрактної машини. Наприклад, найпоширеніший спосіб реалізації мови програмування полягає у визначенні ідеальної «мовної машини» і компіляції програм, написаних цією мовою, у код цієї машини. На наступному кроці трансляції код абстрактної машини конвертується в реальний машинний код.

Добре відомим прикладом такого походу може бути модель *OSI* мережевих протоколів [6]. На рис. 5.4 зображено подібну модель і показано, як за допомогою моделі абстрактної машини можна подати систему адміністрування версій.

Система адміністрування версій ґрунтується на керуванні версіями об'єктів і надає повне керування конфігурацією системи. Для керування конфігурацією використовується система адміністрування об'єктів, що підтримує систему бази даних і сервіси керування об'єктами. У свою чергу, у системі баз даних підтримуються різні сервіси, наприклад керування транзакціями, відкоту назад, відновлення і керування доступом. Для керування базами даних використовуються засоби основної операційної системи і її файлова система.

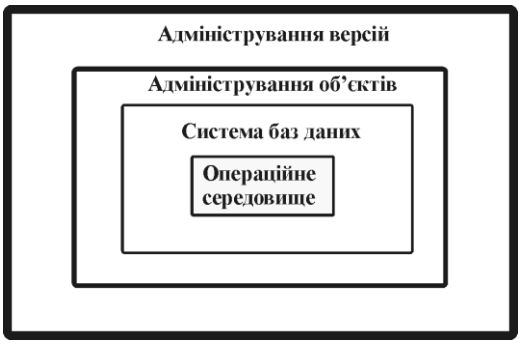


Рис. 5.4. Модель абстрактної машини
для системи адміністрування версій

Багаторівневий підхід забезпечує покроковий розвиток систем – під час розроблення будь-якого рівня надавані нею сервіси стають доступними для користувачів. Така архітектура легко змінювана і придатна до різних платформ. Зміна інтерфейсу будь-якого рівня вплине лише на суміжний рівень, тому що в багаторівневих системах залежність від машинної платформи локалізована на внутрішніх рівнях. Ці системи можна реалізувати на інших платформах, оскільки виникає необхідність змінити лише їх.

Недоліком багаторівневого підходу є досить складна структура системи. Основні засоби, такі як керування файлами, необхідні всім абстрактним машинам, надаються внутрішніми рівнями. Тому сервісам, запитуваним користувачем, можливо, буде потрібний доступ до внутрішніх рівнів абстрактної машини. Така ситуація призводить до руйнування моделі, оскільки зовнішній рівень залежить не лише від попереднього рівня, але і від більш низьких рівнів.

5.3. Моделі керування

У моделі структури системи показані всі підсистеми, з яких вона складається. Для того щоб підсистеми функціонували як єдине ціле, необхідно керувати ними. Структурні моделі не містять ніякої інформації з керування. У моделях керування на рівні архітектури проектується потік керування між підсистемами. Можна виділити такі основні типи керування в програмних системах.

Централізоване керування. Одна з підсистем повністю відповідає за керування, запускає і завершує роботу інших підсистем. Ке-

рування від першої підсистеми може переходити до іншої підсистеми, однак потім обов'язково вертається до першої.

Подійне керування. Тут замість однієї підсистеми, відповідальної за керування під впливом зовнішніх подій – може відповідати будь-яка підсистема. Події, на які реагує система, можуть відбуватися або в інших підсистемах, або в зовнішньому оточенні системи.

Модель керування доповнює структурні моделі. Описані вище структурні моделі можна реалізувати за допомогою централізованого керування або керування, що ґрунтуються на подіях.

У моделі централізованого керування одна з підсистем призначається головною і керує роботою інших. Такі моделі можна поділяти на два класи, залежно від того, послідовно або паралельно реалізоване виконання керованих підсистем.

Модель виклику-повернення. Являє собою відому модель організації виклику програмних процедур «зверху-вниз», у якій керування починається на вершині ієрархії процедур і через виклик передається на більш низькі рівні ієрархії. Ця модель застосовується тільки в послідовних системах.

Модель диспетчера. Застосовується в паралельних системах. Один системний компонент визначається диспетчером і керує пуском, завершенням та координацією інших процесів системи. Процес може відбуватися одночасно з іншими процесами. Модель такого типу застосовна також у послідовних системах, де керувальна програма викликає окремі підсистеми залежно від значень деяких змінних станів. Зазвичай таке керування реалізовується оператором *case.*

Децентралізоване керування. Використовується в розподілених системах.

Модель виклику-повернення показано на рис. 5.5. Із головної програми викликаються підпрограми 1, 2 і 3, з підпрограми 1 – підпрограми 1.1 і 1.2, з підпрограми 3 – підпрограми 3.1 і 3.2 і т.д. Така модель виконання підпрограм не є структурною – підпрограма 1.1 не обов'язково є частиною підпрограми 1.

Керування переходить від програми, розміщеної на найвищому рівні ієрархії, до підпрограми нижнього рівня. Потім керування повертається в точку виклику підпрограми. За керування відповідає та підпрограма, що виконується в поточний момент часу; вона може викликати інші підпрограми або повернути керування підпрограмі, що викликає. Такий стиль програмування, коли керування повертається до певної точки програми, є недосконалим.

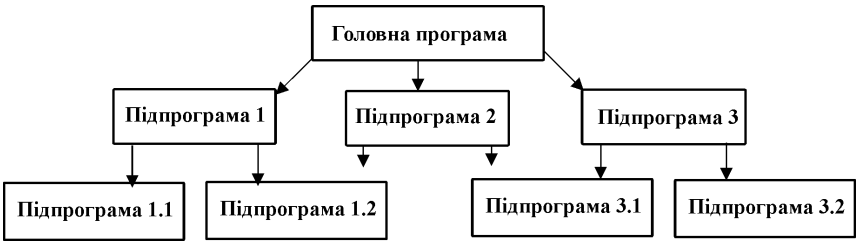


Рис. 5.5. Модель виклику-повернення

Модель виклику-повернення можна використовувати на рівні модулів для керування функціями і об'єктами. Підпрограми в мові програмування, які викликаються з інших підпрограм, є природно функціональними. Однак у багатьох об'єктно-орієнтованих системах операції в об'єктах (методи) реалізовані у вигляді процедур або функцій. Наприклад, об'єкт *Java* запитує сервіс з іншого об'єкта за допомогою виклику відповідного методу.

Жорстка й обмежена природа моделі виклику-повернення є одночасно і перевагою, і недоліком. Переваги моделі проявляються у відносно простому аналізі потоків керування, а також у випадку вибору системи, відповідальної за конкретне введення даних. Недолік моделі полягає в складному обробленні виняткових ситуацій.

Модель централізованого керування для паралельної системи показано на рис. 5.6. Подібна модель часто використовується в системах реального часу, у яких немає надто жорстких часових обмежень. Центральний контролер керує виконанням множини процесів, пов'язаних з датчиками і виконавчими механізмами.

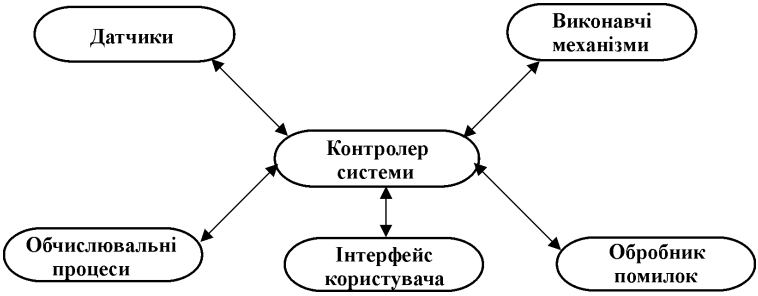


Рис. 5.6. Модель централізованого керування для системи реального часу

Контролер залежно від змінних станів системи визначає моменти запуску або завершення процесів. Він перевіряє, чи генерується в інших процесах інформація для її подальшого оброблення або передавання іншим процесам для оброблення.

Контролер працює постійно, перевіряючи датчики й інші процеси або відслідковуючи зміни стану, тому модель централізованого керування іноді називають моделлю зі зворотним зв'язком.

Системи, керовані подіями. На противагу моделям централізованого керування є системи, керування якими ґрунтується на зовнішніх подіях. У такому контексті під подією розуміють не тільки бінарний сигнал типу «так-ні», але і який може набувати деякого діапазону значень. Розбіжність між подією і звичайними вхідними даними полягає в тому, що планування події виходить за межі керування процесом, що обробляє цю подію. Для оброблення події підсистемі необхідний доступ до інформації стану, однак така інформація не визначається потоком керування.

Є різні типи систем, що керуються подіями. До них належать електронні таблиці, у яких зміна значення в будь-якій комірці змінює вміст інших комірок, системи штучного інтелекту, у яких при виконанні деякої умови ініціюються дії або використовуються активні об'єкти, тоді зі зміною значення властивості об'єкта зініціюється деяка дія.

Моделі передавання повідомлень ефективні для інтеграції підсистем, розподілених на різних комп'ютерах, які об'єднані в мережу. Моделі, керовані перериваннями, використовуються в системах реального часу з жорсткими часовими вимогами.

У моделі передавання повідомлень (рис. 5.7) підсистеми реагують на певні події. Якщо відбулася деяка подія, керування переходить до підсистеми, що обробляє цю подію. Між моделями передавання повідомлень і централізованого керування (див. рис. 5.6) існує відмінність: алгоритм керування не вбудований в обробник повідомлень і подій. Підсистеми визначають події, які їм потрібні, а обробник повідомлень і подій стежить, щоб дані події були відправлені саме їм.

Події можуть генерувати повідомлення всім підсистемам, але при цьому значно збільшується навантаження у процесі оброблення даних. Часто обробник подій і повідомлень керує регістром підсистем і подіями, на які вони реагують. Підсистеми генерують по-

дії, у яких, можливо, є дані для оброблення. Обробник реєструє подію і передає її в ті підсистеми, які на неї реагують. Обробник події завжди підтримує двоточкову взаємодію. Такі підсистеми відправляють повідомлення іншій підсистемі.

Рис. 5.7. Модель керування,
що ґрунтується на передаванні повідомлень

Перевагою моделі передавання повідомлень є відносно проста модернізація систем. Нову підсистему можна інтегрувати в систему, реєструючи її в обробнику подій. Кожна підсистема може активізувати будь-яку іншу підсистему, не знаючи її ім'я або розміщення. Підсистеми також можна реалізувати на різних машинах.

Недоліком цієї моделі є те, що підсистемам невідомо, коли оброблятимуться події. Генеруючи подію, підсистема не знає, яка саме система прореагує на неї. Цілком припустима ситуація, коли різні підсистеми реагують на однакові події. Це може призвести до конфліктів під час отримання доступу до результатів оброблення події.

Системи реального часу, у яких однією з вимог є швидке оброблення зовнішніх подій, мають бути керованими подіями. Наприклад, система реального часу, що керує системою безпеки автомобіля, повинна визначити можливу аварію і встигнути наповнити повітрям подушку безпеки до того, коли голова водія вдариться об кермо. Для того щоб забезпечити швидку реакцію на події, необхідно використовувати керування, що ґрунтується на перериваннях.

Модель керування, заснована на перериваннях, показано на рис. 5.8. Для кожного типу переривань є свій обробник. Кожний тип переривання асоціюється з коміркою пам'яті, у якій зберігається адреса обробника переривання. Отримавши певне переривання, апаратний перемикач негайно передає керування обробнику переривання. У відповідь на подію, викликану перериванням, обробник може запустити або завершити інші процеси.

Модель керування використовується лише у системах реального часу, де потрібне негайне реагування на певні події. Можна ском-

бінувати цю модель із моделлю централізованого керування. Центральний диспетчер оброблює нормальний хід виконання системи, а в критичних ситуаціях використовується керування, що ґрунтується на перериваннях.

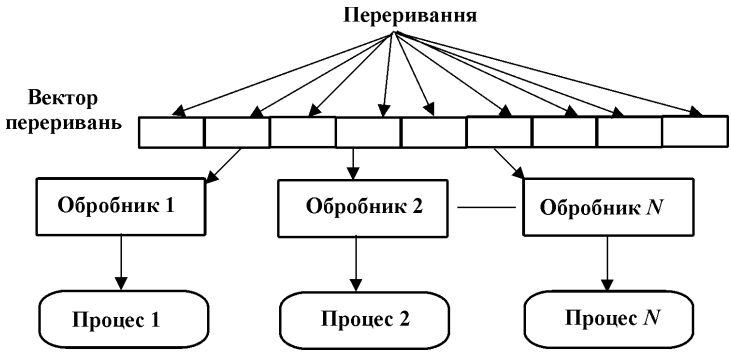


Рис. 5.8. Модель керування, що ґрунтується на перериваннях

Переваги підходу – миттєва реакція системи на виниклі події.

Недолік – складність програмування й атестації системи.

Неможливо імітувати всі переривання в процесі тестування системи. Складно змінювати системи, розроблені на основі такої моделі, оскільки кількість переривань обмежено апаратурою.

Жодні типи подій не обробляються, якщо межу досягнуто. Обмеження іноді можна обійти, якщо на одному перериванні визначити кілька типів подій і надати для їх оброблення окремий обробник. Однак, якщо потрібно дуже швидко реагувати на переривання, такий підхід виявляється непрактичним.

5.4. Модульна декомпозиція

Після розроблення системної структури в процесі проектування відбувається етап декомпозиції підсистем на модулі. Між поділом системи на підсистеми і поділом підсистем на модулі немає істотних відмінностей. Оскільки компонентів модулів зазвичай менше ніж компонентів підсистем, можна використовувати спеціальні моделі декомпозиції: об’єктну модель та модель потоків даних. При цьому система складається з функціональних модулів, які отримують на вході дані і перетворюють їх деяким чином у вихідні дані. Такий підхід часто називають *конвеєрним*.

В *об'єктно-орієнтованій моделі* модулі являють собою об'єкти з власними станами і певними операціями над цими станами. Так, у моделі потоків даних модулі виконують функціональні перетворення. Обидві моделі мають реалізовані модулі або як послідовні компоненти, або як процеси.

По можливості розробникам не варто приймати поспішних рішень про те, чи система буде паралельною, чи послідовною. Проектування послідовної системи має ряд переваг: послідовні програми легше проектувати, реалізовувати, перевіряти і тестувати, ніж паралельні системи, де дуже складно формалізувати, керувати і перевіряти часові залежності між процесами. Краще спочатку розбити систему на модулі, а на етапі реалізації вирішити, як організувати їх виконання – послідовно чи паралельно.

Архітектурна модель з об'єктно-орієнтованою структурою має вигляд сукупності слабко зв'язаних об'єктів із чітко визначеними інтерфейсами. Об'єкти викликають сервіси, надавані іншими об'єктами.

Приклад об'єктно-орієнтованої архітектурної моделі для системи оброблення рахунків показано на рис. 5.9. Ця система виписує рахунки замовникам, одержує платежі, відправляє квитанції за платіжними квитанціями і повідомлення про неоплачені рахунки. У прикладі використовується мова моделювання *UML*, у якій класи об'єктів мають імена і набір атрибутів. Операції, якщо вони є, визначаються в нижній частині прямокутника, що позначає об'єкт. Штрихові стрілки позначають об'єкти, що використовують властивості або сервіси, надавані іншими об'єктами.

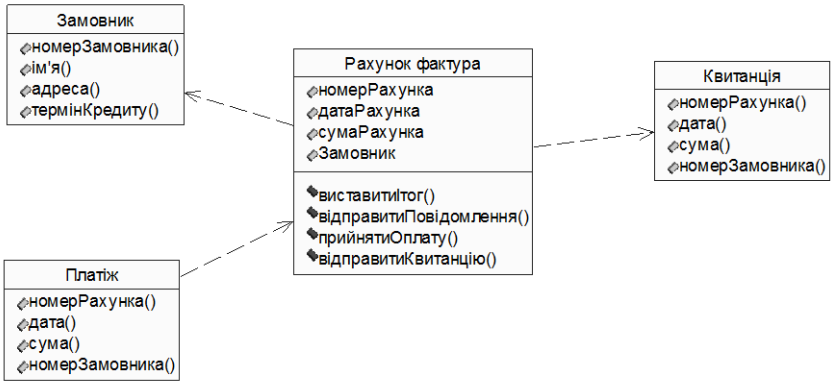


Рис. 5.9. Об'єктна модель системи оброблення рахунків

На етапі об'єктно-орієнтованої декомпозиції визначаються класи об'єктів, їх властивості й операції. За реалізації системи із цих класів створюються об'єкти; для координації операцій об'єктів використовується будь-яка модель керування. На рис. 5.9 клас *Рахунок* має різні зв'язані операції (методи), які реалізовують функціональні засоби системи. Цей клас використовує інші класи – замовників, платежів і квитанцій.

Переваги підходу: легкість модернізації будь-якого об'єкта без впливу на інші об'єкти. Структуру системи легко зрозуміти, оскільки об'єкти часто є об'єктами реального світу. Для безпосередньої реалізації системних компонентів використовують об'єктно-орієнтовані мови програмування.

Недоліки підходу. При використанні сервісів об'єктів необхідно знати імена інших об'єктів і їх інтерфейс. Якщо в разі зміни системи потрібно змінити інтерфейс, необхідно оцінити ефект від такої зміни з урахуванням усіх користувачів змінюваного об'єкта. Багато об'єктів реального світу складно подати у вигляді системних об'єктів.

У *моделі потоків даних* дані через послідовність перетворень і залишають систему. Перетворення даних можуть виконуватися послідовно або паралельно, а їх оброблення – пакетно або поелементно.

Якщо перетворення виконуються окремими процесами, таку модель іноді називають *конвеєром* або моделлю фільтрів відповідно до термінології, прийнятої в системі *Unix*. Остання підтримує конвеєри, які діють як сховища даних, і набір команд, що виконують функціональні перетворення. Тут використовується термін «фільтр», оскільки перетворення «фільтрує» дані під час обробленя потоку даних.

Різні варіанти моделі потоків даних виникли разом з появою перших комп'ютерів, призначених для автоматизованого оброблення даних. Коли перетворення послідовно обробляють пакети даних, така архітектурна модель називається пакетною послідовною моделлю. Вона є основою для багатьох класів систем оброблення даних. Прикладом можуть бути системи, які генерують велику кількість вихідних звітів, отриманих за допомогою нескладних обчислень, але з великою кількістю вхідних записів, наприклад система оброблення рахунків (рис. 5.10).

Організація виписує рахунки замовникам. Один раз на тиждень платіжні квитанції узгоджуються з рахунками. Для оплачених рахунків видається квитанція. Для рахунків, не оплачених протягом установленого терміну, видається відповідне повідомлення.

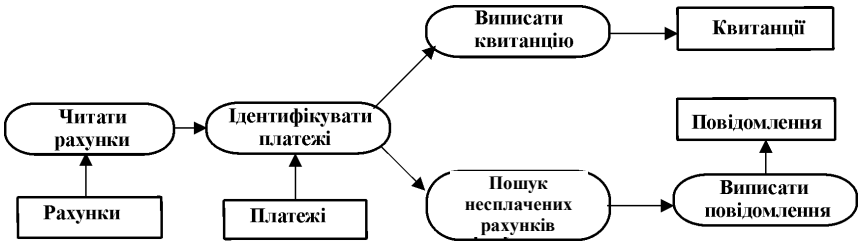


Рис. 5.10. Модель потоків даних для системи оброблення рахунків

Ця модель – лише частина системи оброблення рахунків, у реальних ситуаціях виписування рахунків використовуються інші перетворення. На її відміну об'єктна модель більш абстрактна, оскільки в ній не утримується інформація про послідовність дій.

Переваги архітектури:

– можливе повторне використання перетворень;

– модель зрозуміла, оскільки подана у термінах «вхід – вихід»;

– легка модифікація шляхом безпосереднього додавання нових перетворень;

– проста щодо реалізації як послідовних, так і паралельних систем.

Недоліки архітектури:

– необхідність використання деякого загального формату передавання даних, що повинен розпізнаватися всіма перетвореннями;

– неможливість інтеграції перетворень, що використовують несумісні формати даних;

– важкий опис взаємодійних систем через відсутність прогнозованого потоку оброблюваних даних;

– графічні інтерфейси користувача мають складні формати введення-виведення і керування, що ґрунтуються на різноманітних подіях (наприклад, клацання кнопкою миші або вибір з меню).

5.5. Проблемно-залежні архітектури

Поряд з основними моделями використовуються архітектурні моделі, характерні для конкретної предметної галузі додатка. Ці моделі називаються *проблемно-залежними архітектурами.* Розрізнюють два типи проблемно-залежних архітектурних моделей: моделі класів систем і базові.

Моделі класів систем відображають класи реальних систем з основними їх характеристиками. Архітектурні моделі класів, як

правило, використовуються в системах реального часу, наприклад у системах збирання даних, моніторингу та ін.

Базові моделі є більш абстрактними і надають розробникам інформацію про загальну структуру будь-якого типу систем.

Чітких відмінностей між цими видами моделей немає. У деяких випадках моделі класів є базовими, які використовуються в системах комунікацій і для порівняння можливих системних архітектур. Розрізняються і процеси розроблення цих моделей. Моделі класів розробляються як узагальнення існуючих систем «знизу-вгору», тоді як базові моделі – «зверху-донизу».

Компілятор є найбільш відомим прикладом архітектурної *моделі класу* систем. Натепер розроблено тисячі компіляторів. Компілятор має містити такі модулі (рис. 5.11):

– *лексичний аналізатор* – транслює вхідну мову в деякий внутрішній код;

– *таблицю ідентифікаторів* – видається лексичним аналізатором, де утримується інформація про використовувані в програмі імена і типи;

– *синтаксичний аналізатор* – перевіряє синтаксис компільованого коду;

– *синтаксичне дерево* – внутрішня структура компільованої програми;

– *семантичний аналізатор* – перевіряє семантичну коректність програм на підставі інформації, отриманої із синтаксичного дерева і таблиці ідентифікаторів;

– *генератор коду* – проходить по синтаксичному дереву і генерує машинний код.

Рис. 5.11. Модель потоків даних для компілятора

Етапи лексичного, синтаксичного та семантичного аналізу виконуються послідовно. У компіляторі можуть бути й інші компо-

ненти, які перетворюють синтаксичне дерево, підвищують ефективність і усувають надмірність генерованого машинного коду.

Модель ефективна там, де програми компілюються і виконуються без участі користувача, тобто в пакетному режимі. Однак вони виявляються менш ефективними, якщо компілятор інтегрований з іншими мовними засобами, наприклад системою редагування структур, інтерактивним відладником, програмою підготовки документів до друку та ін. У цьому випадку компоненти системи можна організувати відповідно до моделі репозиторію (рис. 5.12).

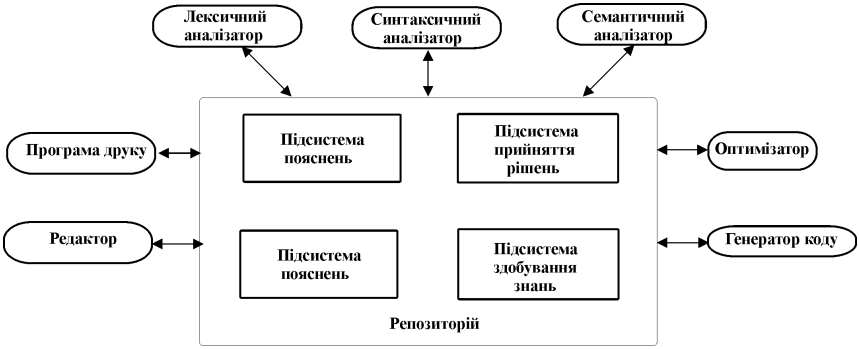


Рис. 5.12. Модель репозиторію для компілятора

Таблиця ідентифікаторів і синтаксичне дерево моделі компілятора використовуються як центральне сховище інформації, або репозиторій, через який взаємодіють інструментальні засоби. Інші дані, такі як граматичні правила і визначення вихідного формату програми, беруться з інструментальних засобів і також містяться в репозиторію.

Натепер розроблено невелику кількість проблемно-залежних моделей класів систем. Організації, що займаються розробленням подібних моделей, розглядають їх як інтелектуальну власність, оскільки вони є основою розроблення програмних систем. Такі моделі часто являють собою архітектуру серії програмних продуктів.

5.6. Аспектно-орієнтоване програмування

Аспектно-орієнтоване програмування (АОП) – парадигма побудови гнучких до змін програмних систем шляхом додавання нових аспектів (функцій), що забезпечують, наприклад, безпеку, взаємо-

дію компонентів з іншим середовищем, а також синхронізацію одночасного доступу частин програмної системи до даних і виклик нових компонентів із загальносистемних середовищ.

Аспектом може бути компонент повторного використання, фрагмент програми, що реалізовує концепцію взаємодії компонентів у середовищі, захист даних тощо. Програмна система, яка створюється з компонентів повторного використання (КПВ), об'єктів, невеликих методів та аспектів, доповнюється необхідними засобами взаємодії, синхронізації та захисту. Отже, вбудовані фрагменти наповнюють компоненти новим змістовним аспектом.

Практична реалізація аспектів, розміщених у різних частинах елементів програмної системи, забезпечується механізмом посилань і точками з'єднання, через які відбувається зв'язок з аспектним фрагментом для отримання визначеної додаткової функції.

В основу АОП покладено метод, подібний до методу поділу завдань предметної галузі на ряд функціональних компонентів, визначення необхідності використання різного роду додаткових аспектів і встановлення точок розміщення аспектів в окремих компонентах, де це потрібно. Ця робота виконується у процесі розроблення, доповнює реалізацію програмної системи засобами забезпечення взаємодії компонентів або їх синхронізації. Підхід АОП застосовується під час налагодження програм, коли додаткові фрагменти коду вбудовуються в певні точки початкової програми для видачі результатів перевірки. Якщо налагодження закінчується позитивно, ці фрагменти вилучаються. Програмні фрагменти аспектів залишаються в основній програмі.

Кінцева програмна система в АОП створюється за технологією, що відповідає розробленню компонентних систем, за винятком того, що використані аспекти визначають особливі умови виконання компонентів у середовищі взаємодії. Аспекти можна асоціювати з особами, що виконують різні ролі. Це наближує аспект до програмного агента, який виконує додаткові функції з визначення архітектури системи та якості компонентів.

В АОП для розроблення використовується механізм фільтрації вхідних повідомлень, за допомогою яких змінюються параметри і імена текстів аспектів у конкретно заданому компоненті системи. Код компонента стає «нечистим», коли його перетинають аспекти, і при композиції з іншими компонентами загальні засоби (виклик процедур, *RPC*, *RMI*, *IDL* і т. ін.) стають недостатніми. Оскільки аспекти потребують декларативного об'єднання описів, особливо коли їх фрагменти беруться з одних об'єктів для інших.

Один з механізмів композиції компонентів і аспектів – фільтр композиції, що оновлює аспекти без зміни їх функціональних можливостей. Фактично фільтрація стосується вхідних і вихідних параметрів повідомлень, які перевизначають відповідні імена об'єктів. Тобто фільтри делегують внутрішнім частинам компонентів параметри, переадресовуючи вже встановлені посилання, перевіряють і розміщують у буфері повідомлення, локалізують обмеження і готують відповідний компонент для виконання.

В об'єктно-орієнтованій системі (ООС) можуть використовуватися методи, призначені для виконання деяких розрахунків у разі звертання до інших зовнішніх методів.

Із погляду моделювання аспекти можна розглядати як каркаси декомпозиції системи, у яких окремі аспекти перетинають КПВ. Це схематично показано на рис. 5.13 для програм *P*1, *P*2 і *P*3, тобто в певних точках різних КПВ може бути звернення до одного аспекту.

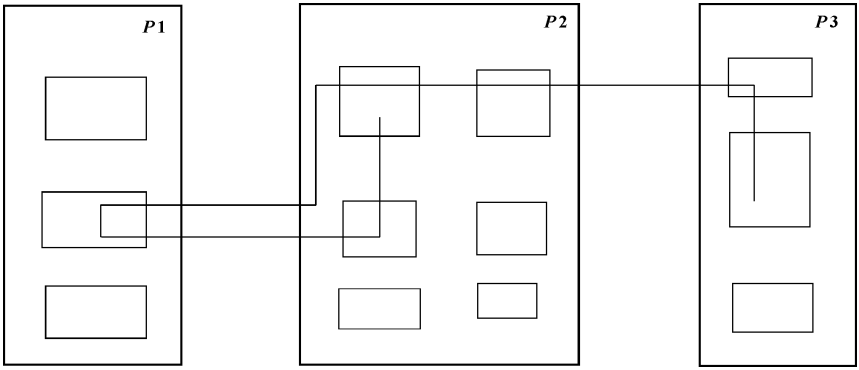


Рис. 5.13. Приклад розташування аспектів у програмах *P*1, *P*2 і *P*3

Програма *P*1 містить аспект, що здійснює звернення до деякої точки програми *P*2, яка після отримання інформації записує її у захищену базу даних, а потім аспект програми *P*2 забезпечує взаємодію з програмою *P*3 для передавання відомостей про запис необхідної інформації в базу даних.

Різним аспектам проектованої системи можуть відповідати і різні парадигми програмування: об'єктно-орієнтовані, структурні та інші, що утворює мультипарадигменну концепцію розроблення проектованої системи.

Через аспектні механізми встановлюються зв'язки з іншими предметними галузями в сім'ї програм або систем ПЗ. Мова АОП дозволяє описувати аспекти для різних систем. Після компіляції описів перетинних аспектів вони генеруються, поєднуються, оптимізуються і орієнтуються на виконання в динаміці. При цьому кожний аспект може стати модулем-посередником, що реалізовує шаблон взаємодії окремих програм або систем.

Інакше кажучи, у процесі розроблення виявляється, що аспекти надто щільно «переплетені» з компонентами і тому потрібна мінімізація зв'язку між аспектами і компонентами через посилання до варіантів використання, зіставлення із шаблоном або блоком коду, у якому встановлені перетини посилань. Природно, що вказані перетини можуть знизити ефективність виконання програм або систем, у яких вони розміщені.

Аналіз програмної системи закінчується побудовою характеристичної моделі та встановленням статичних або динамічних зв'язків з додатковими аспектами моделі. Різним аспектам програмної системи відповідають різні парадигми програмування, які потребують їх удосконалення і узагальнення у разі розроблення нового ПЗ.

Технологія розроблення ПС методами АОП містить у собі ряд загальних процесів (рис. 5.14). Вони такі.

1. Декомпозиція функціональних завдань з умовою багаторазового застосування модулів і виділених аспектів виконання (паралельно, синхронно тощо).

2. Аналіз мов специфікації аспектів для опису виділених аспектів та інших завдань ПЗ.

3. Визначення точок вбудовування аспектів у компоненти і формування посилань та зв'язків з іншими елементами ПЗ.

4. Розроблення фільтрів для їх подання на боці сервера для керування відповідно до заданих аспектів.

5. Визначення механізмів композиції (викликів процедур, методів, зв'язку) функціональних модулів, КПВ та аспектів у точках їх з'єднання як фрагментів керування виконанням або повернення із цих точок до інших модулів.

6. Створення об'єктної або компонентної моделі, доповнення її вхідними і вихідними фільтрами повідомлень, що посилають об'єктам повідомлення із завданням виконання методів або аспектів.

7. Компіляція, спільне налагодження модулів та аспектів і потім композиція їх у готовий програмний продукт.

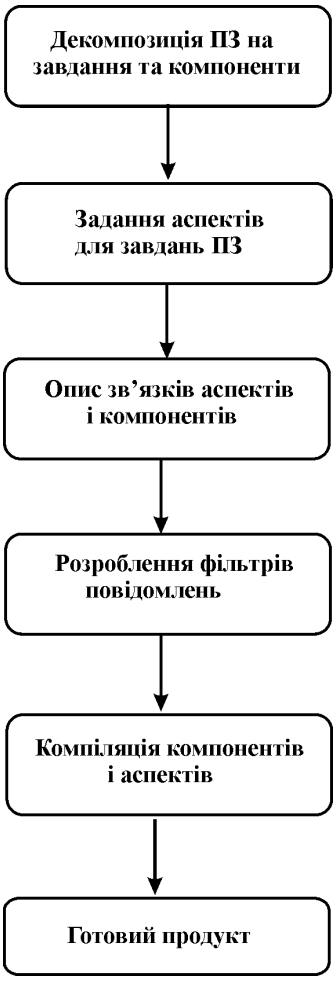


Рис. 5.14. Технологічна схема проектування ПС засобами АОП

Для ефективної реалізації аспектів розроблено системи *Aspect*, IP1-бібліотека розширень, активні бібліотеки, а також розширено мови програмування *Smalltalk* засобами опису аспектів (*Aspect++*, *Aspect*, *AspectC#*, *JAC*).

Систему *Aspect* розробив дослідницький центр *Xerox PARC* для підтримання АОП на базі мови *Java*. Система має компілятор, налагоджувач і генератор документації.

Компілятор видає код, сумісний з віртуальною машиною *Java*.

Розширення мови *Java* стосується способів опису правил інтеграції аспектів і *Java*-об'єктів і базується на таких поняттях:

– точка з'єднання *JoinPoint* у програмі, асоційована з контекстом виконання (виклик методу, конструктора, доступ до поля класу і т. ін.);

– набір точок зрізу *Pointcut* для *JoinPoint*, що задовольняє певні умови;

– набір інструкцій *Advice* у мові *Java*, виконуваних до, після або замість кожної з точок *JoinPoint*, що входять у заданий зріз;

– задання аспекту *Introduction* для зміни структури *Java*-класу шляхом додавання нових полів, методів та ієрархії.

Точка зрізу (точка у програмі, у яку вводяться певні інструкції, що були виконані до або будуть виконуватися замість цієї точки) і набір інструкцій визначають правила інтеграції, які формують відповідний модуль системи.

Загальними принципами розроблення програмної системи з використанням засобів АОП системи *Aspect* є:

– побудова моделі програмної системи за компонентами й аспектами;

– аспектна декомпозиція;

– окрема реалізація кожної вимоги;

– інтеграція аспектів у програмний код.

Інтеграція аспектів відбувається в момент компіляції. Модель побудови готової програмної системи з компонентів, аспектів та фрагментів готового коду показано на рис. 5.15.

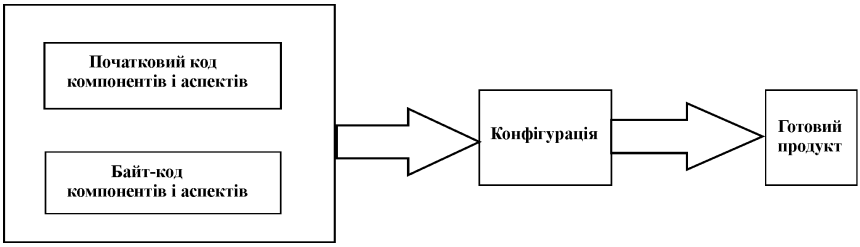


Рис. 5.15. Інтеграція аспектів і компонентів

Після компіляції отримується готова система з функціональністю, інтегрованою за правилами, описаними в аспектних модулях.

Існують інші реалізації АОП: *AspectC++*, *AspectC*, *AspectC#*, розширення мов C++, C, C# аспектами; *JAC* – система, написана мо-

вою *Java*, для створення розподілених програмних систем; *Weave.NET* – проект реалізації механізму підтримання АОП без прив'язки до конкретної мови програмування всередині компонентної моделі.*NET Framework* і т. ін.

У процесі створення програмних систем із застосуванням аспектів можуть використовуватися: *IP*-бібліотека розширень, активні бібліотеки, мова програмування *SmallTalk*, розширені засоби опису аспектів.

В *IP*-бібліотеці розміщені деякі функції компіляторів, методів, засобів оптимізації, редагування, відображення тощо. Наприклад, бібліотека матриць, за допомогою якої обчислюються вирази з масивами, забезпечує швидкість виконання, надання пам'яті й т. ін.

Використання таких бібліотек у розширених середовищах програмування називають родовим програмуванням, а вирішення проблем економії, перебудови компіляторів під кожне нове мовне розширення, використання шаблонів і результатів попереднього оброблення належать до ментального програмування.

Бібліотека *IP* містить: окремі функції компіляторів, засоби оптимізації, редагування, відображення понять, перебудови окремих компонентів компіляторів під нове мовне розширення, а також засоби програмування на основі шаблонів та ін. Бібліотеки з такими можливостями отримали назву бібліотек генерації.

Інший вид бібліотек АОП – активні бібліотеки, які містять не тільки базовий код реалізації понять ПЗ, а і цільовий код забезпечення оптимізації, адаптації, візуалізації і редагування. Активні бібліотеки поповнюються засобами та інструментами інтелектуалізації агентів, за допомогою яких забезпечується розроблення спеціалізованих агентів для реалізації конкретних завдань ПС.

Контрольні запитання і завдання

1. Із яких етапів складається проектування ПЗ?
2. Обгрунтуйте зв'язок продуктивності з надійністю системи.
3. Поясніть сутність етапу структурування системи.
4. Які існують способи обміну інформації в системах? Наведіть приклади.
5. Поясніть призначення моделі репозиторію, проаналізуйте її, наведіть приклади застосування.
6. Наведіть визначення моделі архітектури клієнт/сервер, проаналізуйте її, наведіть приклад застосування.

7. Наведіть визначення моделі архітектури типу абстрактна машина, проаналізуйте її, наведіть приклад застосування.
8. Визначте модель архітектури типу виклик – повернення.
9. Наведіть визначення моделі централізованого керування та наведіть характеристики.
10. Назвіть та поясніть моделі, що використовуються на етапі модульної декомпозиції підсистем.
11. Охарактеризуйте об'єктно-орієнтовану модель за модульної декомпозиції підсистем.
12. Поясніть поняття моделі потоків даних за модульної декомпозиції підсистем
13. Що являють собою моделі потоків даних за модульної декомпозиції підсистем?
14. Наведіть приклад об'єктно-орієнтованої архітектурної моделі виконання розрахунків.
15. Виконайте порівняльний аналіз моделі клієнт-сервер з моделлю репозиторію.

6. АРХІТЕКТУРА РОЗПОДІЛЕНИХ СИСТЕМ

Сучасні програмні системи можна поділити на три великі класи:

– *прикладні програмні системи*, призначені для роботи лише на одному персональному комп'ютері або робочій станції (текстові процесори, електронні таблиці, графічні системи та ін.);

– *вбудовані системи*, призначені для роботи на одному процесорі або на інтегрованій групі процесорів (системи керування побутовими пристроями, різними приладами і т. ін.);

– *розподілені системи*, у яких ПЗ виконується на слабкоінтегрованій групі паралельно працюючих процесорів, зв'язок між якими здійснюється через мережу (системи банкоматів, що належать будь-якому банку, видавничі системи, системи ПЗ колективного користування тощо).

Розподіленою називається система, у якій оброблення інформації зосереджене не на одній обчислювальній машині, а розподілена між декількома комп'ютерами.

Проектування розподілених систем має багато загального із проектуванням будь-якого іншого ПЗ, але існує ряд специфічних особливостей. Деякі з них уже згадувалися під час розгляду архітектури клієнт/сервер (підрозділ 5.1.2).

Межі між класами програмних систем надалі будуть зменшуватися. Наприклад, високошвидкісні бездротові мережі динамічно інтегрують пристрої з убудованими програмними системами, або загальні системи містять електронні органайзери.

Розрізнюють характеристики розподілених систем.

Стійкість до відмов. Наявність декількох комп'ютерів і можливість дублювання інформації означає, що розподілені системи стійкі до певних апаратних і програмних помилок. Розподілені системи у випадку помилки підтримують часткову функціональність. Повне порушення функціонування системи відбувається лише у випадку мережевих помилок.

Прозорість. Ця властивість означає, що користувачам надається прозорий доступ до ресурсів, але інформація про розподіл ресурсів у системі від них прихована. Знання організації системи у багатьох випадках допомагає користувачу оптимально використовувати ресурси.

Складність. Розподілені системи складніші від централізованих. Набагато важче зрозуміти й оцінити властивості розподілених систем у цілому, а також їх тестувати. Наприклад, продуктивність системи в них залежить не від швидкості роботи одного процесора, а від смуги пропускання мережі та швидкості роботи різних процесорів. Застосовуючи ресурси іншої частини системи, можна радикально вплинути на її продуктивність.

Безпека. Доступ до системи можна отримати з декількох різних машин, повідомлення в мережі можуть проглядатися або перехоплюватися. Тому в розподіленій системі набагато складніше підтримувати безпеку.

Керованість. Система може складатися з різнотипних комп'ютерів, на яких можуть бути встановлені різні версії операційних систем. Помилки однієї машини можуть поширитися на інші машини
з непередбаченими наслідками. Тому керування і підтримання системи в робочому стані потребують значно більших зусиль.

Непередбачуваність. Реагування розподілених систем на певні події залежить від повного завантаження системи, її організації і мережевого навантаження. Оскільки всі ці параметри можуть постійно змінюватися, час, витрачений на виконання запиту користувача у той або інший момент може істотно відрізнятися.

Основні проблеми проектування розподілених систем наведено в табл. 6.1. Завдання розробників розподілених систем – проектувати програмне або апаратне забезпечення так, щоб забезпечити потрібні характеристики розподіленої системи. Для цього необхідно знати характеристики різних типів архітектури розподілених систем. Виділяють дві родинні архітектури розподілених систем.

1. *Клієнт/сервер*. Ця модель архітектури являє собою набір сервісів, надаваних серверами клієнтам. При цьому сервери і клієнти мають істотні відмінності.

2. *Розподілені об'єкти* є архітектурою, у якій між серверами і клієнтами немає розбіжностей і систему можна подати набором взаємодійних об'єктів, розміщення яких не має особливого значення. Між постачальником сервісів і їх користувачами немає розбіжностей.

Таблиця 6.1

Проблеми проектування розподілених систем

Проблема проектування	Опис
Ідентифікація ресурсів	Доступу до ресурсів, що забезпечується системою уніфікованого покажчика ресурсів *URL*
Комунікації	Ефективна реалізація протоколів *TCP*/*IP* в Інтернет для більшості розподілених систем є прикладом ефективної організації взаємодії між комп'ютерами. Однак можуть бути використані й альтернативні способи системних комунікацій
Якість системного сервісу	Забезпечується розподілом системних процесів, ресурсів, системними і мережевими апаратними засобами і можливістю адаптації системи
Архітектура ПЗ	Описує розподіл системних функцій за компонентами системи, а також перерозподіл цих компонентів за процесорами. Вибір правильної архітектури є вирішальним фактором для підтримання високої якості системного сервісу

У розподіленій системі різні системні компоненти можуть бути реалізовані різними мовами програмування і виконуватися на різних типах процесорів. Моделі даних, подання інформації і протоколи взаємодії – усе це не обов'язково буде однотипним у розподіленій системі. Отже, для розподілених систем необхідно таке ПЗ,

що могло б керувати цими різнотипними частинами і гарантувати взаємодію та обмін даними між ними. Для цього використовується проміжне ПЗ. Воно стикує різні частини розподілених компонентів системи.

Розподілені системи зазвичай розробляються на основі об'єктно-орієнтованого підходу. Вони створюються зі слабкоінтегрованих частин, кожна з яких може безпосередньо взаємодіяти як з користувачем, так і з іншими частинами системи. Ці частини по можливості повинні реагувати на незалежні події. Програмні об'єкти, побудовані на основі таких принципів, є природними компонентами розподілених систем.

6.1. Багатопроцесорна архітектура

Найпростішою розподіленою системою є багатопроцесорна система. Вона складається з множини різних процесів, які можуть (але не обов'язково) виконуватися на різних процесорах. Цю модель використовують у великих системах реального часу. Ці системи збирають інформацію, приймають на її основі рішення і відправляють сигнали виконавчому механізму, що змінює системне оточення. Усі процеси, пов'язані зі збиранням інформації, прийняттям рішень і керуванням виконавчим механізмом, можуть виконуватися на одному процесорі під керуванням планувальника завдань. Використання декількох процесорів підвищує продуктивність системи і її здатність до відновлення. Розподіл процесів між процесорами може перевизначатися (властиво критичним системам) або ж перебувати під керуванням диспетчера процесів. Спрощену модель системи керування транспортним потоком показано на рис. 6.1.

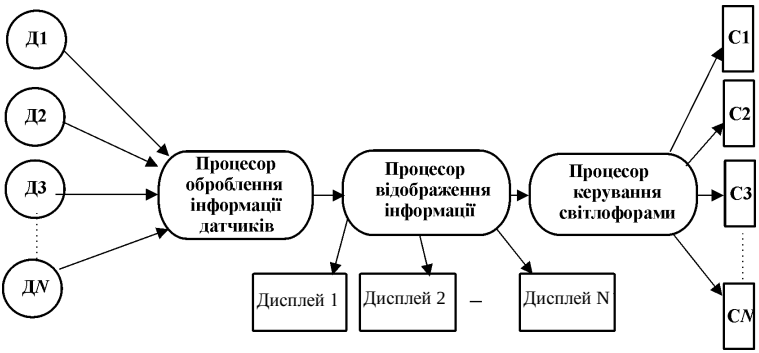


Рис. 6.1. Багатопроцесорна система керування рухом транспорту

Група розподілених датчиків Д1, …, Д*N* збирає інформацію про величину потоку. Зібрані дані перед відправленням диспетчеру обробляються на місці. На підставі отриманої інформації оператори, які стежать за інформацією на дисплеях, приймають рішення і керують світлофорами С1,…, С*N*. У цьому прикладі для керування датчиками, диспетчерським пунктом і світлофорами є окремі логічні процесори. Це може виконуватися як окремим процесором, так і групою процесорів.

Системи ПЗ, одночасно виконуючи множину процесів, не обов'язково є розподіленими. Якщо в системі більше ніж один процесор, реалізувати розподіл процесів нескладно. Тому багатопроцесорні програмні системи не обов'язково створювати на основі розподілених систем.

6.2. Архітектура клієнт/сервер

В архітектурі клієнт/сервер програмний додаток моделюється як набір сервісів, надаваних серверами, і множин клієнтів, що використовують ці сервіси. Схему розподіленої архітектури клієнт/сервер показано на рис. 6.2.

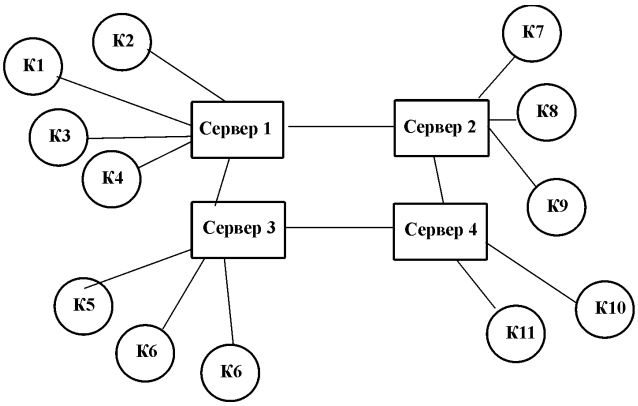


Рис. 6.2. Система клієнт/сервер

У системі між процесами і процесорами необов'язково дотримуватися відношення «один до одного». Фізичну архітектуру системи, що складається із шести клієнтських машин і двох серверів, показано на рис. 6.3. На них запускаються клієнтські і серверні процеси, зображені на рис. 6.2. У загальному випадку клієнти і сервери являють собою швидше логічні процеси, ніж фізичні машини, на яких виконуються ці процеси. Архітектура системи клієнт/сервер повинна відображати логічну структуру розроблюваного програмного додатка. На рис. 6.4 пропонується ще один погляд на програмний додаток, структурований у вигляді трьох рівнів.

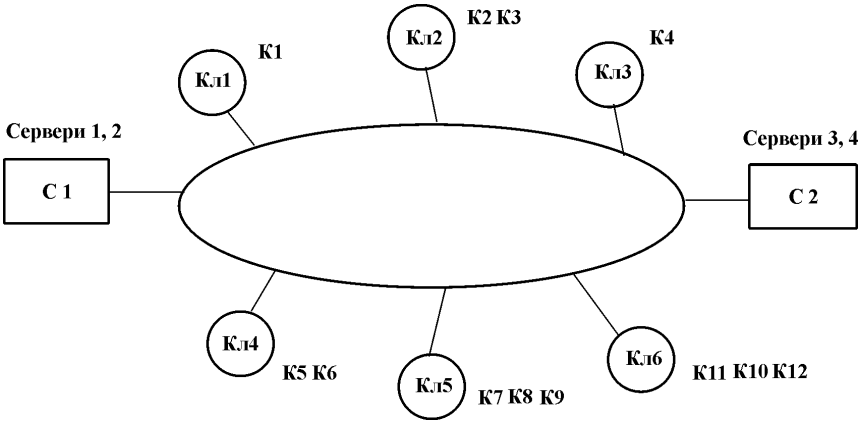


Рис. 6.3. Комп'ютери в мережі клієнт/сервер

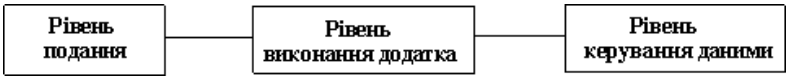


Рис. 6.4. Рівні програмного додатка

Рівень подання забезпечує інформацію для користувачів і взаємодію з ними. Рівень виконання додатка реалізовує логіку роботи додатка. На рівні керування даними виконуються всі операції з базами даних. У централізованих системах між цими рівнями немає чіткого поділу. Однак, проектуючи розподілені системи, необхідно розрізняти ці рівні, щоб потім розмістити кожний рівень на різних комп'ютерах.

Найпростішою архітектурою клієнт/сервер є дворівнева, у якій додаток складається із сервера і групи клієнтів. Розрізнюють два види такої архітектури: модель товстого клієнта і модель тонкого клієнта (рис. 6.5).

Рис. 6.5. Дворівнева архітектура

Модель товстого клієнта. У цій моделі сервер лише керує даними. На клієнтській машині реалізуються робота додатка і взаємодія з користувачем системи.

Модель тонкого клієнта. Це найпростіший спосіб переведення існуючих централізованих систем в архітектуру клієнт/сервер. Користувацький інтерфейс у цих системах розміщується на персональному комп'ютері, а сам програмний додаток виконує функції сервера, тобто виконує всі процеси додатка і керує даними. Модель тонкого клієнта можна також реалізовувати там, де клієнтами є

звичайні мережеві пристрої, а не персональні комп'ютери або робочі станції. Мережеві пристрої запускають інтернет-броузер і користувацький інтерфейс, реалізований усередині системи.

Головний недолік моделі тонкого клієнта – велика завантаженість сервера і мережі. Усі обчислення виконуються на сервері, а це може призвести до щільного мережевого трафіку між клієнтом і сервером. У сучасних комп'ютерах достатньо обчислювальної потужності, але вона майже не використовується в моделі тонкого клієнта банку.

Модель товстого клієнта використовує обчислювальну потужність локальних машин: і рівень виконання додатка, і рівень подання розміщуються на клієнтський комп'ютер. Сервер, по суті, є сервером транзакцій, що керує всіма транзакціями баз даних. Як приклад архітектури такого типу можна навести системи банкоматів, у яких банкомат є клієнтом, а сервер – центральним комп'ютером, що обслуговує базу даних розрахунків із клієнтами.

Мережу системи банкоматів показано на рис. 6.6. Ці банкомати пов'язані з базою даних розрахунків не прямо, а через монітор телеоброблення. Цей монітор є проміжною ланкою, що взаємодіє з віддаленими клієнтами й організовує запити клієнтів у послідовність транзакцій для роботи з базою даних. Використання послідовних транзакцій у разі виникнення збоїв дозволяє системі відновитися без утрати даних.

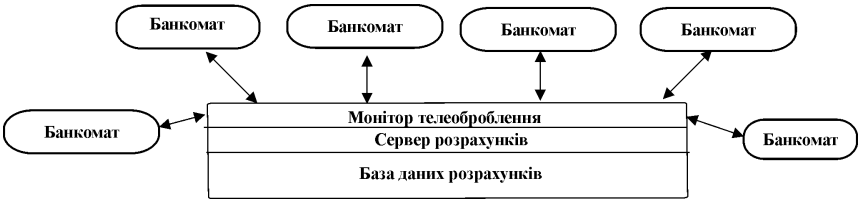


Рис. 6.6. Система клієнт/сервер для мережі банкоматів

Оскільки в моделі товстого клієнта виконання програмного додатка організовано більш ефективно, ніж у моделі тонкого клієнта, керувати такою системою складніше. Тут функції додатка розподілені між множиною різних машин. Необхідність заміни додатка призводить до його повторної інсталяції на всіх клієнтських

комп'ютерах, що потребує більших витрат, якщо в системі сотні клієнтів.

Поява мови *Java* і аплетів, що завантажуються, дозволила розробляти моделі клієнт/сервер, які розміщені посередині між моделями тонкого і товстого клієнта. Частина програм, що становлять додаток, можна завантажувати на клієнтській машині як аплети *Java* і тим самим розвантажити сервер. Інтерфейс користувача будується за допомогою *web*-броузера, що запускає аплети *Java*. Однак *web*-броузери від різних виробників і навіть різні версії *web*-броузерів від одного виробника не завжди виконуються однаково. Такий підхід можна використовувати лише тоді, коли є впевненість, що користувачі системи мають броузери, сумісні з *Java*.

У дворівневій моделі клієнт/сервер істотною проблемою є розміщення на двох комп'ютерних системах трьох логічних рівнів – подання, виконання додатка і керування даними. Такій моделі властиві проблеми з масштабованістю або продуктивністю, якщо обрано модель тонкого клієнта, або проблеми, пов'язані з керуванням системою, якщо використовується модель товстого клієнта. Щоб уникнути цих проблем, необхідно застосувати альтернативний підхід на основі трирівневої архітектури клієнт/сервер (рис. 6.7). У цій архітектурі рівням подання, виконання додатка і керування даними відповідають окремі процеси.

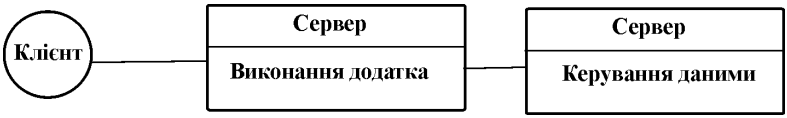


Рис. 6.7. Трирівнева архітектура клієнт/сервер

Архітектура ПЗ, побудована на трирівневій моделі клієнт/сервер, не потребує об'єднання в мережу цих комп'ютерних систем. На одному комп'ютері-сервері можна запустити додаток на виконання, а даними керують окремі логічні сервери. Утім, якщо вимоги до системи підвищаться, можна розділити виконання додатка і керування даними і виконувати їх на різних процесорах.

Банківську систему, що використовує інтернет-сервіси, можна реалізувати за допомогою трирівневої архітектури клієнт/сервер. База даних розрахунків, яка розміщена на головному комп'ютері, надає сервіси керування даними, *web*-сервер підтримує сервіси до-

датка, наприклад засоби переведення грошей, генерацію звітів, оплати рахунків і т.ін. А комп'ютер користувача з інтернет-броузером є клієнтом. Як показано на рис. 6.8, ця система масштабована, оскільки в неї відносно просто додати нові *web*-сервери в разі збільшення кількості клієнтів.

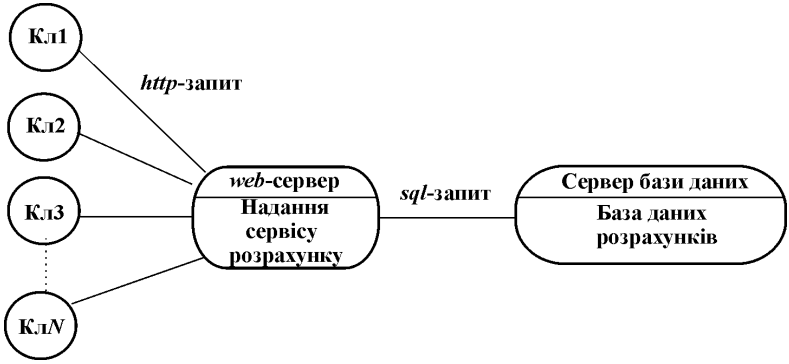


Рис. 6.8. Розподілена архітектура банківської системи з використанням інтернет-сервера

Використання трирівневої архітектури в цьому прикладі дозволяє оптимізувати передавання даних між *web*-сервером і сервером бази даних. Взаємодію між цими системами не обов'язково будувати за стандартами Інтернету, можна використовувати більш швидкі комунікаційні протоколи низького рівня. Інформацію від бази даних обробляє ефективне проміжне ПЗ, що підтримує запити до бази даних мовою структурованих запитів *SQL*.

У деяких випадках трирівневу модель клієнт/сервер можна перевести в багаторівневу, додавши в систему додаткові сервери. Багаторівневі системи можна використовувати і там, де додаткам необхідно мати доступ до інформації, що перебуває в різних базах даних. Тут об'єднувальний сервер розміщується між сервером, на якому виконується додаток, і серверами баз даних. Об'єднувальний сервер збирає розподілені дані і подає їх у додаток таким чином, немовби вони перебувають в одній базі даних.

Розробники архітектур клієнт/сервер, обираючи найбільш придатну, повинні враховувати фактори раціонального застосування різних типів архітектури клієнт/сервер, наведені в табл. 6.2.

Таблиця 6.2

Застосування архітектур клієнт/сервер

Архітектура	Додатки
Дворівнева архітектура тонкого клієнта	Наслідувані системи, у яких недоцільно розділяти виконання додатків і керування даними. Додатки з інтенсивними обчисленнями, наприклад компілятори, але з незначним обсягом керування даними. Додатки, у яких обробляються великі масиви даних (запити), але з невеликим обсягом обчислень у самому додатку
Дворівнева архітектура товстого клієнта	Додатки, у яких користувачу потрібне інтенсивне оброблення даних (наприклад, візуалізація даних або більші обсяги обчислення). Додатки з відносно постійним набором функцій на боці користувача, застосовуваних у середовищі з добре налагодженим системним керуванням
Трирівнева і багаторівнева архітектури клієнт/сервер	Великі додатки із сотнями і тисячами клієнтів. Додатки, у яких часто змінюються і дані, і методи оброблення. Додатки, у яких виконується інтеграція даних з багатьох джерел

6.3. Архітектура розподілених об’єктів

У моделі клієнт/сервер розподіленої системи клієнти і сервери розрізнюються. Клієнт запитує сервіси лише в сервера, але не в інших клієнтів; сервери можуть функціонувати як клієнти і запитувати сервіси в інших серверів, але не в клієнтів; клієнти повинні знати про сервіси, надавані певними серверами, і про те, як вони взаємодіють. Така модель більш придатна для багатьох типів додатків, але водночас обмежує розробників системи, які змушені вирішувати, де надавати сервіси. Вони також повинні підтримувати масштабованість і розробити засоби включення клієнтів у систему на розподілених серверах.

Загальний підхід, застосовуваний у проектуванні розподілених систем, полягає в усуненні розбіжностей між клієнтом і сервером і розробленні архітектури системи за принципом розподілених об’єктів. Основними компонентами цієї архітектури (рис. 6.9) системи є об’єкти, що надають набір сервісів через свої інтерфейси. Інші об’єкти викликають ці сервіси, не розділяючи клієнта і сервера.

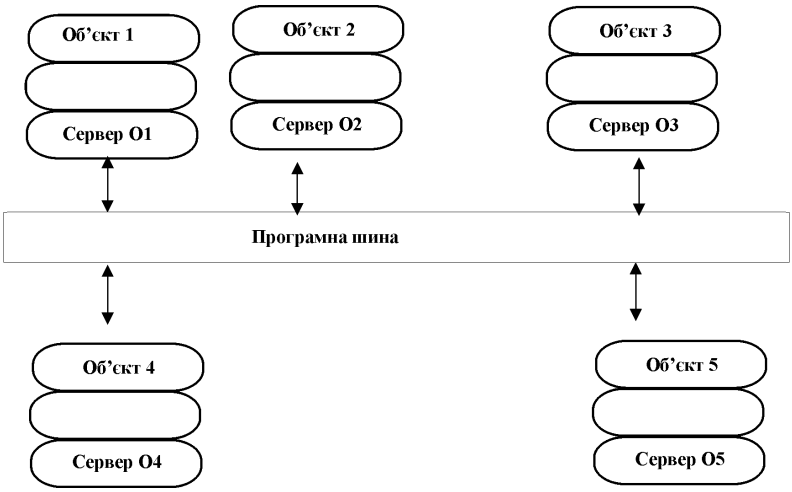


Рис. 6.9. Архітектура розподілених об'єктів

Об'єкти можуть розміщуватися на різних комп'ютерах у мережі і взаємодіяти за допомогою проміжного ПЗ. За аналогією із системною шиною, що дозволяє підключати різні пристрої і підтримувати взаємодію між апаратними засобами, проміжне ПЗ можна розглядати як шину ПЗ. Вона надає набір сервісів, які дозволяють об'єктам взаємодіяти один з одним, додавати або видаляти їх із системи. Проміжне ПЗ називають брокером запитів до об'єктів. Його завдання – забезпечувати інтерфейс між об'єктами.

Переваги моделі архітектури розподілених об'єктів.

1. Неважливо де і як будуть надаватися сервіси. Об'єкти, що надають сервіси, можуть виконуватися в будь-якому місці (вузлі) мережі. Отже, модель товстого клієнта та модель тонкого клієнта розрізняються мало, тому немає потреби заздалегідь планувати розміщення об'єктів для виконання додатка.

2. Системна архітектура відкрита, що дозволяє у разі потреби додавати в систему нові ресурси.

3. Система є гнучкою і масштабованою. Зі збільшенням навантаження в систему додаються нові об'єкти без зупинення роботи інших її об'єктів.

4. Система за допомогою об'єктів, що мігрують у мережі за запитами, динамічно переконфігурується. Об'єкти, що надають сер-

віси, мігрують на той самий процесор, що й об'єкти, що запитують сервіси, для підвищення продуктивності системи.

У процесі проектування систем архітектуру розподілених об'єктів можна використовувати подвійно: як логічну модель і як клієнт/ серверну систему.

Логічна модель дозволяє розробникам структурувати і планувати системи. У цьому випадку функціональність додатка описується в термінах і комбінаціях сервісів. Потім розробляються способи надання сервісів за допомогою декількох розподілених об'єктів. На цьому рівні, як правило, проектують великомодульні об'єкти, які надають сервіси, що відображають специфіку конкретної ділянки додатка.

У клієнт/серверній системі клієнти і сервери реалізовані як розподілені об'єкти, що взаємодіють за допомогою програмної шини. За такого підходу легко замінити систему, наприклад дворівневу на багаторівневу. У цьому випадку ні сервер, ні клієнт не можуть бути реалізовані в одному об'єкті, однак можуть складатися з множини невеликих об'єктів, кожний з яких надає певні сервіси.

Прикладом системи, для якої придатна архітектура розподілених об'єктів, є система оброблення даних, що зберігаються в різних базах даних (рис. 6.10). У цьому прикладі будь-яку базу даних можна подати як об'єкт з інтерфейсом, що надає доступ до даних «тільки читання». Кожний з об'єктів-інтеграторів збирає інформацію з баз даних для відстеження типів залежностей між даними.

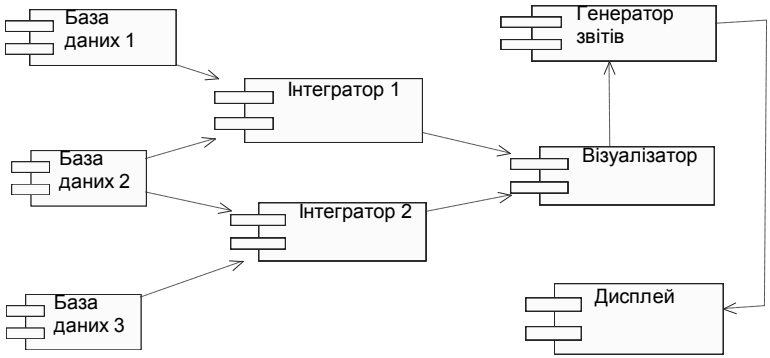


Рис. 6.10. Архітектура розподіленої системи оброблення даних

Об’єкт-візуалізатор взаємодіє з об’єктами-інтеграторами для подання даних у графічному вигляді або для складання звітів за аналізованими даними. Такому типу додатків більше відповідає архітектура розподілених об’єктів ніж архітектура клієнт/сервер з трьох причин.

1. Ці системи на відміну, наприклад, від системи банкоматів не мають одного постачальника сервісу, на якому були б зосереджені всі сервіси керування даними.

2. Збільшувати кількість доступних баз даних можна не перериваючи роботу системи, оскільки кожна база даних являє собою об’єкт, який підтримується спрощеним інтерфейсом, що керує доступом до даних. Доступні бази даних можна розмістити на різних машинах.

3. За допомогою додавання нових об’єктів-інтеграторів можна відслідковувати нові типи залежностей між даними.

Головним недоліком архітектури розподілених об’єктів є те, що їх складніше проектувати, ніж системи клієнт/сервер. Виявляється, що системи клієнт/сервер забезпечують більше природний підхід до створення розподілених систем. У ньому відображаються взаємини між людьми, за яких одні люди користуються послугами інших людей, що спеціалізуються на наданні конкретної послуги. Набагато складніше розробляти архітектуру розподілених об’єктів.

6.4. Архітектура брокерів запитів до загальних об’єктів

Для реалізації архітектури розподілених об’єктів необхідне проміжне ПЗ (брокери запитів до об’єктів), що організовує взаємодію між розподіленими об’єктами. Тут є ймовірність виникнення певних проблем, оскільки об’єкти в системі можуть бути реалізовані різними мовами програмування, можуть запускатися на різних платформах і їх імена не повідомляються всім іншим об’єктам системи. Тому проміжне ПЗ має виконувати роботу для підтримання постійної взаємодії об’єктів.

Натепер підтримують розподілені об’єктні обчислення два основні стандарти проміжного ПЗ.

1. *Common Object Request Broker Architecture* (*CORBA*) – архітектура брокерів запитів до загальних об’єктів. Це набір стандартів для проміжного ПЗ, розроблений групою з керування об’єктами (*Object Management Group*, *OMG*). Стандарти *CORBA* підтримуються операційною системою *UNIX* і операційними системами від

Microsoft. Стандарти *CORBA* визначають загальний машинонезалежний підхід до розподілених об'єктних обчислень. Існує безліч реалізацій цього стандарту.

2. *Distributed Component Object Model* (*DCOM*) – об'єктна модель розподілених компонентів. *DCOM* являє собою стандарт, розроблений і реалізований компанією *Microsoft* і інтегрований у її операційні системи. Ця модель розподілених обчислень менш універсальна, ніж *CORBA* і пропонує більш обмежені можливості мережевих взаємодій. Натепер використання *DCOM* обмежується операційними системами *Microsoft*.

CORBA, *DCOM* та інші технології, наприклад *RMI* (*Remote Method Invocation* – виклик віддаленого методу, технологія побудови розподілених додатків мовою *Java*), будуть поступово зближатися і це зближення ґрунтується на стандартах *CORBA*. Тому немає потреби в ще одному стандарті. Різні стандарти стають перешкодою подальшого розвитку.

OMG є консорціумом фірм-виробників програмного і апаратного забезпечення, серед яких такі компанії, як *Sun*, *Hewlett-Packard* і *IBM* (понад 500 компаній). Роль *OMG* – створення стандартів для об'єктно-орієнтованих розробок, а не забезпечення конкретних реалізацій цих стандартів. Стандарти підтримують об'єктно-орієнтовані розроблення і перебувають у вільному доступі на *web*-вузлі *OMG*. Група визначає не тільки стандарти *CORBA*, але й інші стандарти, включаючи мову моделювання *UML*.

Подання розподілених додатків у межах *CORBA* показано на рис. 6.11.

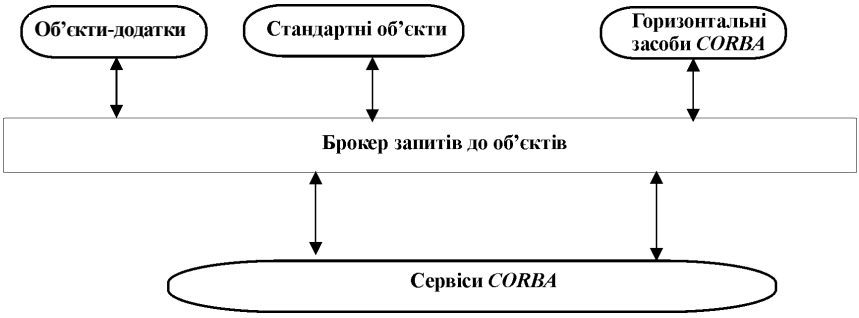


Рис. 6.11. Структура розподіленого додатка, заснованого на стандартах *CORBA*

Стандарти *CORBA* описують чотири основні елементи.

1. Модель об'єктів, у якій об'єкт *CORBA* інкапсулює стани за допомогою чіткого опису мовою *IDL* (*Interface Definition Language*) – мовою опису інтерфейсів.

2. Брокер запитів до об'єктів (*Object Request Broker – ORB*), що керує запитами до сервісів об'єктів. *ORB* розміщує об'єкти, що надають сервіси, готує їх до отримання запитів, передає запит сервісу і повертає результати об'єкту, який зробив запит.

3. Сукупність основних сервісів об'єктів, потрібних багатьом розподіленим додаткам. Прикладами є служби каталогів, сервіси транзакцій і сервіси підтримання тимчасових об'єктів.

4. Сукупність загальних компонентів, побудованих на верхньому рівні основних сервісів. Вони можуть бути як вертикальними, що відображають специфіку конкретної галузі, так і горизонтальними універсальними компонентами, використовуваними в багатьох програмних додатках.

У моделі *CORBA* об'єкт інкапсулює атрибути і сервіси як звичайний об'єкт. В об'єктах *CORBA* має також утримуватися визначення різних інтерфейсів, що описують глобальні атрибути й операції об'єкта. Інтерфейси об'єктів *CORBA* визначаються стандартною універсальною мовою опису інтерфейсів *IDL*. Якщо один об'єкт запитує сервіси, надавані іншими об'єктами, він отримує доступ до цих сервісів через *IDL*-інтерфейс. Об'єкти *CORBA* мають унікальний ідентифікатор, названий *IOR* (*Interoperable Object Reference* – посилання на взаємодійний об'єкт). Коли один об'єкт відправляє запити для сервісу, що надається іншим об'єктам, використовується ідентифікатор *IOR*.

Брокеру запитів до об'єктів відомі об'єкти, що запитують сервіси і їх інтерфейси. Він організовує взаємодію між об'єктами. Взаємодійним об'єктам не потрібно знати розміщення інших об'єктів, а також їх реалізацію. Інтерфейс *IDL* відокремлює об'єкти від брокера, реалізацію об'єктів можна змінювати без урахування інших компонентів системи.

На рис. 6.12 показано, як об'єкти *Ol* і О2 взаємодіють за допомогою брокера запитів до об'єктів. Викличний об'єкт *Ol* зв'язаний із заглушкою *stub IDL*, що визначає інтерфейс об'єкта, який надає сервіс.

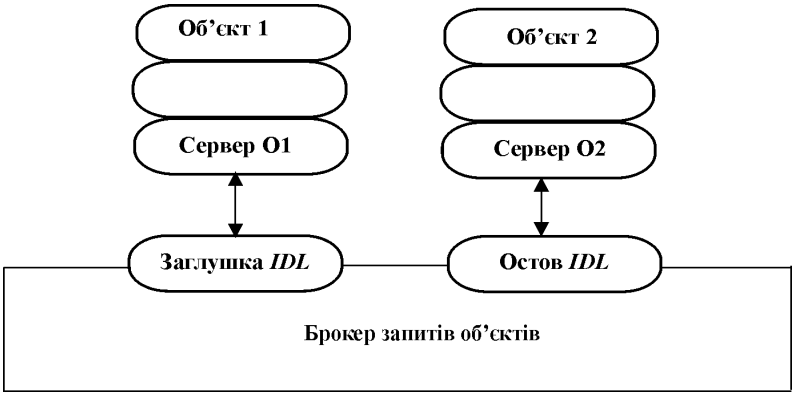


Рис. 6.12. Взаємодія об'єктів за допомогою брокера запитів

Конструктор об'єкта О1 для запиту до сервісу впроваджує виклики в заглушку реалізації об'єкта. Мова *IDL* є розширенням *C*. Опис інтерфейсу об'єкта перекладається *IDL*, можливо й іншими мовами, наприклад, *Ada* або *COBOL*. Але в цих випадках необхідна відповідна інструментальна підтримка.

Об'єкт, що надає сервіс, зв'язаний з остовом (*skeleton*) *IDL*, який зв'язує інтерфейс з реалізацією сервісів. Інакше кажучи, коли сервіс викликається через інтерфейс, кістяк *IDL* транслює виклик до сервісу незалежно від того, яка мова використовувалася в реалізації. Після завершення методу або процедури остов транслює результати в мову *IDL* і вони стають доступними прийнятому об'єкту. Якщо об'єкт одночасно надає сервіси іншим об'єктам або використовує сервіси, які ще надавались, йому потрібні і *IDL*, і заглушка *IDL*. Остання необхідна всім використовуваним об'єктам.

Брокер запитів до об'єктів зазвичай реалізовується не у вигляді окремих процесів, а як каркас, що пов'язаний з реалізацією об'єктів. Тому в розподіленій системі кожний комп'ютер, на якому працюють об'єкти, повинен мати власний брокер запитів до об'єктів, що буде обробляти всі локальні виклики об'єктів. Але якщо запит зроблений до сервісу, що наданий вилученим об'єктом, то потрібна взаємодія між брокерами.

Таку ситуацію ілюструє рис. 6.13, де якщо об'єкт О1 або О2 відправляє запити до сервісів, надаваним об'єктами О3 або О4, то необхідна взаємодія зв'язаних із цими об'єктами брокерів. Ста-

ндарти *CORBA* підтримують взаємодію «брокер – брокер», що забезпечує брокерам доступ до описів інтерфейсів *IDL*, і пропонують розроблений групою *OMG* стандарт узагальненого протоколу взаємодії брокерів *GIOP* (*Generic Inier-ORB Protocol*). Цей протокол визначає стандартні повідомлення, якими можуть обмінюватися брокери під час виконання викликів віддаленого об'єкта і передавання інформації. Разом із протоколом Інтернет низького рівня *TCP/IP* протокол *GIOP* дозволяє брокерам взаємодіяти через Інтернет.

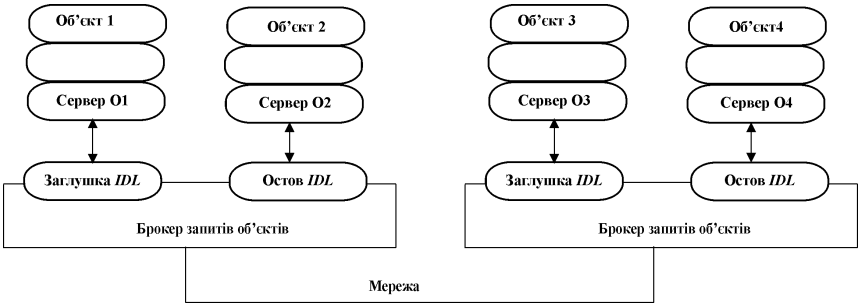


Рис. 6.13. Взаємодія між брокерами запитів до об'єктів

Перші варіанти *CORBA* були розроблені ще в 1980-х роках. Ранні версії *CORBA* пов'язані з підтриманням розподілених об'єктів.

Однак згодом стандарти розвивалися, ставали більше розширеними. Подібно до клієнт-серверних механізмів *CORBA* є засобами, застосовними в багатьох розподілених системах. Ці стандарти визначають приблизно 15 загальних служб (сервісів). Ось деякі з них.

Служба імен – дозволяє об'єктам знаходити інші об'єкти в мережі і посилатися на них. Служба імен є сервісом каталогів, що привласнює імена об'єктам. У разі потреби об'єкти через цю службу можуть знаходити ідентифікатори *IOR* інших об'єктів.

Служба реєстрації – дозволяє об'єктам реєструвати інші об'єкти після здійснення деяких подій. За допомогою цієї служби об'єкти можна реєструвати за їх участі в певній події, а коли певна подія вже відбулася, вона автоматично реєструється сервісом.

Служба транзакцій – підтримує елементарні транзакції і повертається назад у випадку помилок або збоїв. Ця служба є відмовостійким засобом, що забезпечує відновлення у випадку помилок під час операції відновлення. Якщо дії з відновлення об'єкта спричи-

няють помилки або збій системи, об'єкт завжди можна повернути назад до стану, що був перед початком відновлення.

Вважається, що стандарти *CORBA* повинні містити визначення інтерфейсів для широкого діапазону компонентів, які можна використовувати для побудови розподілених додатків. Ці компоненти можуть бути вертикальними або горизонтальними. Вертикальні компоненти розробляються спеціально для конкретних додатків. Їх розробляють фахівці з різних сфер діяльності. Горизонтальні компоненти є більш універсальними, наприклад, компоненти користувацького інтерфейсу.

Контрольні запитання і завдання

1. Які відмінності має розподілена система? Наведіть її структуру.
2. Як можна класифікувати сучасні програмні системи?
3. Сформулюйте переваги розподілених систем.
4. Поясніть проблеми проектування розподілених систем.
5. Наведіть класифікацію архітектури розподілених програмних систем.
6. Яку архітектуру має багатопроцесорна система?
7. Яка архітектура клієнт-серверної системи?
8. Чим відрізняються модель тонкого клієнта від моделі товстого клієнта?
9. Поясніть причини застосування трирівневої архітектури клієнт-сервер.
10. Порівняйте багатопроцесорну і клієнт-серверну архітектури.
11. У яких випадках доцільно використовувати архітектуру розподіленої системи оброблення даних?
12. Які стандарти складають проміжне ПЗ?
13. Які елементи складають основу стандартів *CORBA*?
14. Які сервіси *CORBA* підтримують базові сервіси розподілених обчислень?
15. Поясніть відмінності між горизонтальними і вертикальними засобами *CORBA*.

7. ПРОЕКТУВАННЯ ІНТЕРФЕЙСУ КОРИСТУВАЧА

7.1. Процес проектування інтерфейсу

Розроблення обчислювальних систем охоплює широкий спектр проектних дій – від проектування апаратних засобів до інтерфейсу ко-

ристувача. Організації-розробники часто наймають фахівців для проектування апаратних засобів і дуже рідко для проектування інтерфейсів. Таким чином, фахівці з розроблення ПЗ найчастіше проектують і інтерфейс користувача. Якщо у великих компаніях до цього процесу залучаються фахівці з інженерної психології, то в невеликих компаніях послугами таких фахівців майже не користуються.

Грамотне проектування інтерфейсу користувача надто важливе для успішної роботи системи. Складний у застосуванні інтерфейс призводить до помилок користувача. Іноді користувачі просто відмовляються працювати зі складною ПС, незважаючи на її функціональні можливості. Якщо інформація подається непослідовно, користувачі можуть зрозуміти її неправильно і в результаті їх подальші дії можуть пошкодити дані або навіть порушити роботу системи.

Стандартним пристроєм взаємодії між користувачем і програмою тривалий час був буквено-цифровий (текстовий) термінал, що відображав на чорному полі символи зеленим або синім кольором. Сучасні персональні комп'ютери підтримують графічний інтерфейс користувача (*GUI – Graphical User Interface*), що дозволяє працювати мишею із клавіатурою на кольоровому графічному екрані з високою роздільною здатністю. Незважаючи на те, що текстові інтерфейси ще застосовуються, особливо в наслідуваних системах, найбільш поширеним натепер є *GUI*, основні елементи якого наведено в табл. 7.1.

Таблиця 7.1

Елементи графічних інтерфейсів користувача

Елементи	Опис
Вікна	Відображають на екрані різну інформацію
Піктограми	Подають різні типи даних
Меню	Команди
Покажчики	Миша – використовується як указувальний пристрій для вибору команд із меню та виділення окремих елементів у вікні
Графічні елементи	Візуальне подання тексту

Переваги графічного інтерфейсу:

1) інтуїтивно зрозумілий, легко засвоюється на практиці;

2) можливість організації багатопроцесорного режиму без утрати даних, отриманих у ході виконання програми;

3) масштабування вікон надає прямий доступ до будь-якого місця екрана.

Ітераційний процес проектування користувацького інтерфейсу показано на рис. 7.1. Найбільш ефективним підходом до проектування інтерфейсу користувача є розроблення із застосуванням моделювання дій користувача. На початку процесу прототипування створюються паперові макети інтерфейсу, потім розробляються екранні форми, що моделюють взаємодію з користувачем. Бажано, щоб кінцеві користувачі брали активну участь у процесі проектування інтерфейсу. В одних випадках користувачі допоможуть оцінити інтерфейс, в інших будуть повноправними членами проектної групи.

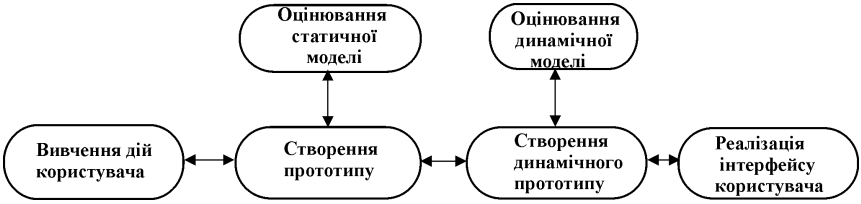


Рис. 7.1. Процес проектування інтерфейсу користувача

Важливим етапом процесу проектування інтерфейсу користувача є аналіз діяльності користувачів, яку має забезпечити обчислювальна система. Для аналізу потрібно (як правило, одночасно) застосовувати різні методики, а саме: аналіз завдань, етнографічний підхід (див. підрозд. 2.7), опитування користувачів і спостереження за їх роботою.

7.2. Принципи проектування інтерфейсів користувача

Розробники інтерфейсів повинні враховувати фізичні і розумові здібності людей, які працюють із програмним забезпеченням. Люди на нетривалий час можуть запам’ятати досить обмежений обсяг інформації, роблять помилки в разі ручного введення великих обсягів даних або, працюючи в напружених умовах, відрізняються фізичними здібностями.

Основою принципів проектування інтерфейсів користувача є людські можливості. Основні принципи проектування будь-якого інтерфейсу користувача наведено в табл. 7.2.

Таблиця 7.2

Принципи проектування інтерфейсу користувача

Принцип	Опис
Орієнтація на знання користувача	Використання термінів і понять, зрозумілих користувачам системи
Узгодженість	Різні частини інтерфейсу повинні бути узгодженими щодо однотипних операцій
Мінімізація несподіванок	Прогнозоване поводження системи
Відновлюваність	Інтерфейс повинен включати засоби, що дозволяють відновлення даних після помилкових дій
Забезпеченість довідковою інформацією	Інтерфейс повинен надавати довідкову інформацію на випадок помилок користувача і підтримувати засоби контекстно-залежної довідки
Орієнтованість на можливості користувачів	Інтерфейс повинен мати засоби для зручної взаємодії з користувачами різного рівня кваліфікації і можливостей

Принцип орієнтованості знань користувача припускає таке: інтерфейс має бути настільки зручним для реалізації, щоб користувачі без особливих зусиль мали звикнути до нього. В інтерфейсі мають використовуватися терміни, зрозумілі користувачу, а об'єкти, керовані системою, безпосередньо пов'язаними з робітничим середовищем користувача. Наприклад, якщо розробляється система, призначена для авіадиспетчерів, то керованими об'єктами в ній повинні бути літаки, траєкторії польотів, сигнальні знаки та ін. Основну реалізацію інтерфейсу в термінах файлових структур і структур даних необхідно ховати від кінцевого користувача.

Принцип узгодженості інтерфейсу користувача припускає, що команди і меню системи повинні бути однакового формату, параметри передаватися в усі команди однаково, а пунктуація команд схожою. Такі інтерфейси скорочують час на навчання користувачів. Знання, здобуті у процесі вивчення будь-якої команди або частини додатка, надалі застосовуються під час роботи з іншими частинами системи.

У цьому випадку йдеться про узгодженість низького рівня. Бажаною є узгодженість більш високого рівня. Наприклад, зручно, коли для всіх типів об'єктів системи підтримуються однакові методи (наприклад, друк, копіювання тощо). Однак повна узгодженість

неможлива і навіть небажана. Наприклад, якщо операцію видалення об'єктів робочого стола доцільно реалізовувати перетягуванням їх у кошик, то в текстовому редакторі такий спосіб видалення фрагментів тексту видається неприродним.

Завжди потрібно дотримувати такий принцип: кількість несподіванок повинна бути мінімальною, оскільки користувачів дратує, коли система раптом починає поводитися непередбачено. Під час роботи із системою в користувача формується певна модель її функціонування. Якщо його дія в одній ситуації викликає певну реакцію системи, природно очікувати, що така сама дія в іншій ситуації зумовить аналогічну реакцію. Якщо ж відбувається зовсім не те, що очікувалося, користувач або дивується, або не знає, що робити. Тому розробники інтерфейсів повинні гарантувати, що схожі дії зумовлять схожий ефект.

Дуже важливим є принцип відновлюваності системи, тому що користувачі завжди допускають помилки. Правильно спроектований інтерфейс може зменшити кількість помилок користувача (наприклад, використання меню дозволяє уникнути помилок, які виникають з уведенням команд із клавіатури), однак всі помилки усунути неможливо. В інтерфейсах повинні бути засоби, що запобігають, по можливості, помилкам користувача, а також дають змогу коректно відновити інформацію після усунення помилок. Ці засоби бувають двох видів.

Підтвердження деструктивних дій. Якщо користувач вибрав потенційно деструктивну операцію, то він повинен ще раз підтвердити свій намір.

Можливість скасування дій. Скасування дій повертає систему в той стан, у якому вона перебувала до їх виконання. Не зайвим буде підтримання багаторівневого скасування дій, оскільки користувачі не завжди відразу розуміють, що зробили помилку.

Засоби підтримання користувачів повинні бути вбудовані в інтерфейс та систему і забезпечувати різні рівні допомоги та довідкової інформації. Має бути кілька рівнів довідкової інформації – від основ для початківців до повного опису можливостей системи. Довідкова система має бути структурованою і не перевантажувати користувача зайвою інформацією у випадках простих запитів до неї.

Орієнтованість на можливості користувачів припускає, що із системою можуть працювати різні їх типи. Частина користувачів працює із системою нерегулярно, час від часу. Але існує й інший тип –

«досвідчені користувачі», які працюють із додатком щодня по кілька годин. Випадкові користувачі мають потребу в інтерфейсі, який «керував» би їх роботою із системою, тоді як досвідченим користувачам потрібен інтерфейс, що дозволив би їм максимально швидко взаємодіяти із системою. Крім того, оскільки деякі користувачі можуть мати різні фізичні недоліки, в інтерфейсі мають бути засоби, які допомогли б їм перелаштовувати інтерфейс «під себе». Це можуть бути засоби, що дозволяють відображати збільшений текст, заміщають звук текстом, створюють кнопки великого розміру та ін.

Визнання різноманітності категорій користувачів може суперечити іншим принципам проектування інтерфейсів, наприклад узгодженості. Аналогічно, що необхідні рівні довідкової інформації для різних типів користувачів можуть радикально розрізнятися. Неможливо створити таку довідкову систему, яка задовольняла б усіх користувачів. Розробник інтерфейсу повинен завжди бути готовим до компромісних рішень залежно від реальних користувачів системи.

7.3. Взаємодія з користувачами

Розробнику інтерфейсу користувача обчислювальних систем необхідно вирішити два головні завдання: яким чином користувач уводитиме дані в систему і як дані будуть подані користувачу. «Правильний» інтерфейс повинен забезпечувати і взаємодію з користувачем, і подання інформації.

Інтерфейс користувача забезпечує уведення команд і даних в обчислювальну систему. На перших обчислювальних машинах був лише один спосіб уведення даних – через інтерфейс командного рядка, причому для взаємодії з машиною використовувалася спеціальна командна мова. Такий спосіб придатний тільки для досвідчених користувачів, тому пізніше були розроблені більш спрощені способи введення даних. Усі ці види взаємодії можуть належати до одного з п’яти основних стилів взаємодії.

1. *Безпосереднє маніпулювання*. Користувач взаємодіє з об’єктами на екрані. Наприклад, для видалення файла користувач перетягує його в кошик.

2. *Вибір команди зі списку пунктів меню*. Обрана команда впливає лише на той об’єкт, що виділений (обраний) на екрані.

3. *Заповнення форм*. Користувач заповнює поля екранної форми. Деякі поля можуть мати своє меню (випадне або списки). Форма може містити командні кнопки, клацанням мишею на які ініцію-

ється деяка дія. Щоб видалити файл за допомогою інтерфейсу, заснованого на формі, треба ввести в поле форми ім'я файла і натиснути на кнопку видалення, яка є у формі.

4. *Командна мова*. Користувач уводить конкретну команду з параметрами, щоб указати на її подальші дії. Щоб видалити файл, користувач уводить команду видалення з іменем файла як параметра цієї команди.

5. *Природна мова*. Щоб видалити файл, користувач може ввести команду «Видалити файл з іменем *XXX*».

Кожний із цих стилів взаємодії має переваги та недоліки і найбільше відповідає різним типам додатків і різним категоріям користувачів. Основні переваги та недоліки основних стилів взаємодії і типи додатків, у яких вони зазвичай використовуються, наведено в табл. 7.3.

Таблиця 7.3

Переваги і недоліки стилів взаємодії користувача із системою

Стиль взаємодії	Переваги	Недоліки	Приклади додатків
Пряме маніпулювання	Швидка й інтуїтивно зрозуміла взаємодія. Легкий для вивчення	Складна реалізація. Придатний тільки там, де є зорове сприйняття завдань і об'єктів	Відеоігри; системи автоматичного проектування

Закінчення табл. 7.3

Стиль взаємодії	Переваги	Недоліки	Приклади додатків
Вибір з меню	Скорочення кількості помилок користувача. Уведення із клавіатури мінімальне	Повільний варіант для досвідчених користувачів. Може бути складним, якщо меню містить велику кількість укладених пунктів	Головним чином системи загального призначення
Заповнення форм	Просте введення даних. Легкий для вивчення	Займає місце на екрані	Системи керування запасами: оброблення фінансової інформації
Командна мова	Потужний і гнучкий	Важкий для вивчення. Складно уникнути	Операційні системи; бібліотечні

		помилок уведення	системи
Природна мова	Придатний для недосвідчених користувачів. Легко налаштовується	Потребує великого ручного набору	Системи розкладу; системи зберігання даних *WWW*

Стилі взаємодії рідко використовуються в чистому вигляді, в одному додатку їх може бути одночасно кілька. Наприклад, в операційній системі *Microsoft Window* підтримується декілька стилів: пряме маніпулювання піктограмами, що позначають файли і папки; вибір команд з меню; ручне введення деяких команд, таких як команди конфігурування системи, використання форм (діалогових вікон).

Користувацькі інтерфейси додатків Інтернет базуються на засобах, надаваних мовою *HTML* разом з іншими мовами, наприклад *Java*, що зв'язує програми з компонентами *web*-сторінок.

Інтерфейси *web*-сторінок проектуються здебільшого для випадкових користувачів і являють собою інтерфейси у вигляді форм. У *web*-додатках можна створювати інтерфейси, у яких застосовувався б стиль прямого маніпулювання, однак до написання специфікації таких інтерфейсів це становило складне для програмування завдання.

Необхідно застосовувати різні стилі взаємодії для керування різними системними об'єктами. У такій моделі розрізняють подання інформації, керування діалоговими засобами і керування додатком. Утім така модель є швидше ідеальною, ніж практичною, однак майже завжди є можливість розділити інтерфейси для різних класів користувачів (наприклад, на основний і повний). Подібну модель із розділеними інтерфейсом командної мови і графічним інтерфейсом, що лежить в основі деяких операційних систем, зокрема *Linux*, зображено на рис. 7.2.

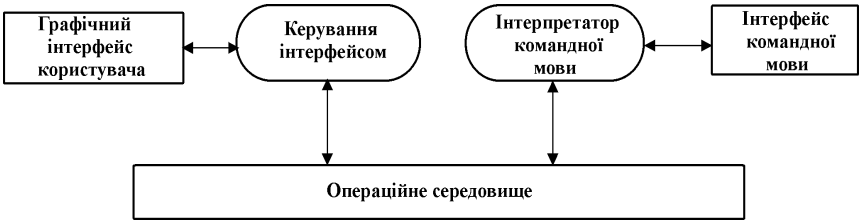


Рис. 7.2. Множинний інтерфейс

Поділ подання, взаємодії й об'єктів, включених в інтерфейс користувача, є основним принципом підходу «модель – подання – контролер».

7.4. Подання інформації

У будь-якій інтерактивній системі повинні бути засоби для подання даних користувачам. Дані в системі можуть відображатися порізному: наприклад, інформація, що вводиться, відображається безпосередньо на дисплеї (як, наприклад, текст у текстовому редакторі) або перетворюється у графічну форму. Під час проектування систем доречно виокремлювати подання даних із самих даних. Певною мірою розроблення такого ПЗ суперечить об'єктно-орієнтованому підходу, за якого методи, виконувані над даними, мають визначати самі дані. Передбачається, що розробник об'єктів завжди знає ефективний спосіб подання даних, хоча це не завжди так. Часто визначити такий спосіб подання даних конкретного типу досить важко, у такому випадку об'єктні структури не повинні бути «жорсткими».

Після того як подання даних у системі виокремлено із самих даних, подання даних на екрані користувача змінюється без зміни самої системи (рис. 7.3).

Підхід «модель подання – контролер», показаний на рис. 7.4, уперше був застосований у мові *Smalltalk* як ефективний спосіб підтримання різних подань даних. Користувач має змогу взаємодіяти з кожним типом подання. Відображувані дані інкапсульовані в об'єкти моделі. Кожний об'єкт моделі може мати кілька окремих об'єктів подань, де кожне подання – це різні відображення моделі.

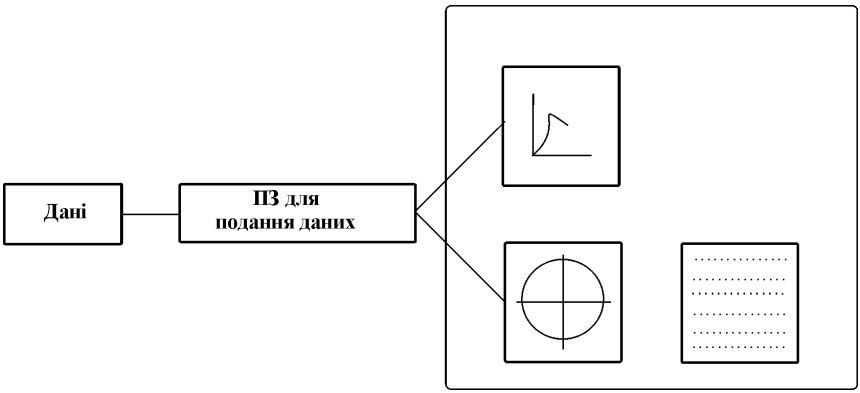


Рис. 7.3. Подання даних

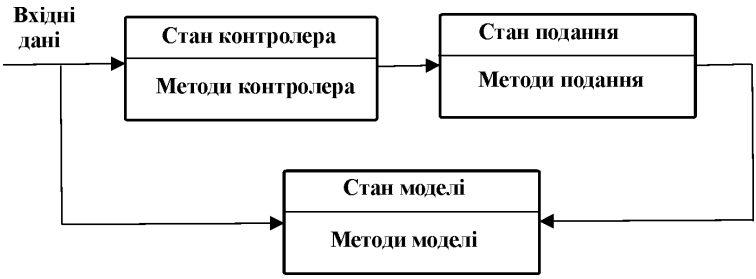


Рис. 7.4. Модель ПК взаємодії з користувачем

Кожне подання має пов'язаний з ним об'єкт контролера, що обробляє введені користувачем дані і забезпечує взаємодію із пристроями. Така модель може відображати числові дані, наприклад, у вигляді діаграм або таблиць.

Модель можна редагувати, змінюючи значення в таблиці або параметри діаграми. Щоб знайти відповідне подання інформації, необхідно знати, з якими даними працюють користувачі і яке зорове сприйняття вони застосовують у системі. Приймаючи рішення щодо подання даних, розробник повинен ураховувати такі фактори:

1) що потрібно користувачу – точні значення даних або співвідношення між значеннями;

2) наскільки швидко відбуватимуться зміни значень даних; чи потрібно негайно показувати користувачу зміну значень;

3) чи належить користувачу вдаватися до якихось дії у відповідь на зміну даних;

4) чи потрібно користувачу взаємодіяти з відображуваною інформацією за допомогою інтерфейсу із прямим маніпулюванням;

5) інформація має відображатися в текстовому (описово) або числовому форматі;

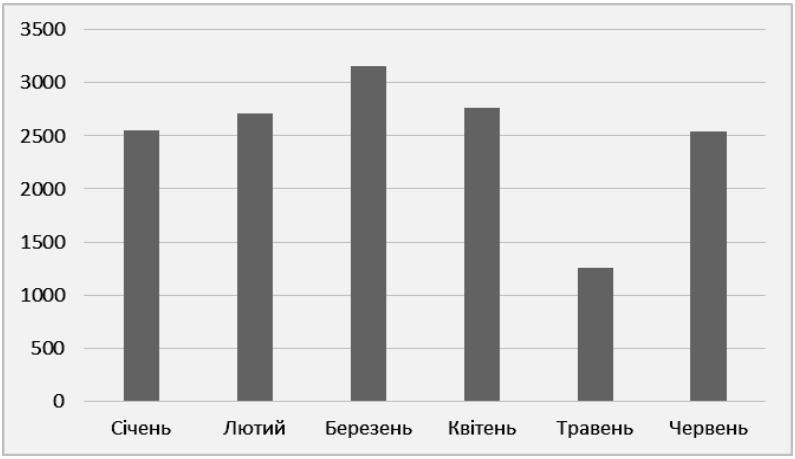
6) чи важливі відносні значення елементів даних.

Якщо дані не змінюються протягом сеансу роботи із системою, їх можна подати або у графічному, або текстовому вигляді залежно від типу додатка. Текстове подання даних займає на екрані мало місця, але тоді їх не можна охопити всебічно. За допомогою різних стилів подання незмінні дані варто відокремити від змінних. Наприклад, статичні дані можна вирізнити особливим шрифтом, особливим кольором або позначити піктограмами.

Якщо потрібна точна цифрова інформація і дані змінюються відносно повільно, їх можна відображати в текстовому вигляді. Там, де дані змінюються швидко, використовується графічне зображення. Як приклад розглянемо систему, яка щомісячно записує і підбиває підсумки даних продажів якоїсь компанії. Із рис. 7.5 видно, що одні й ті самі дані можна подати як у вигляді тексту, так і у графічному вигляді.

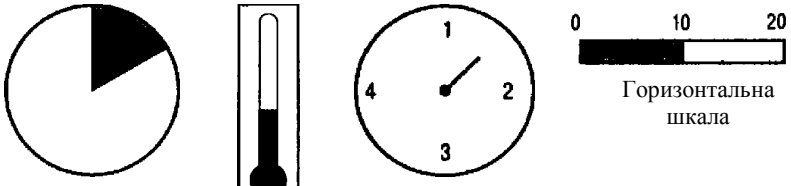
Менеджерам, що вивчають дані з продажів, більш потрібні тенденції зміни або аномальні дані, ніж їх точні значення. Графічне зображення цієї інформації у вигляді гістограми дозволяє виокремити аномальні дані за березень і травень, що значно відрізняються від інших даних. Як видно з рис. 7.5, дані в текстовому поданні займають менше місця, ніж у графічному.

Динамічні зміни числових даних краще відображати графічно, використовуючи аналогові подання. Цифрові екрани, що постійно змінюються, дезорієнтують користувачів, оскільки точні значення даних швидко не сприймаються. Графічне зображення даних у разі потреби доповнюється точними значеннями (рис. 7.6).



Січень	Лютий	Березень	Квітень	Травень	Червень
2555	2715	3164	2764	1255	2543

Рис. 7.5. Альтернативне подання даних



Кругова діаграма

Термометр

Циферблат зі стрілкою

Горизонтальна шкала

Рис. 7.6. Способи подання змінних числових даних

Графічне зображення має включати максимальні та мінімальні значення, оскільки додатковий час, необхідний для їх розрахунків, може призвести до помилок оператора у стресових ситуаціях, коли виникають проблеми і на дисплеї відображаються аномальні дані (рис. 7.7).

Для подання точних буквено-цифрових даних для виділення інформації також використовують графічні елементи. Замість звичайного рядка дані поміщають у прямокутник або відзначають піктограмою (рис. 7.8). Прямокутник з повідомленнями міститься поверх поточного екрана, що привертає до нього увагу користувача.

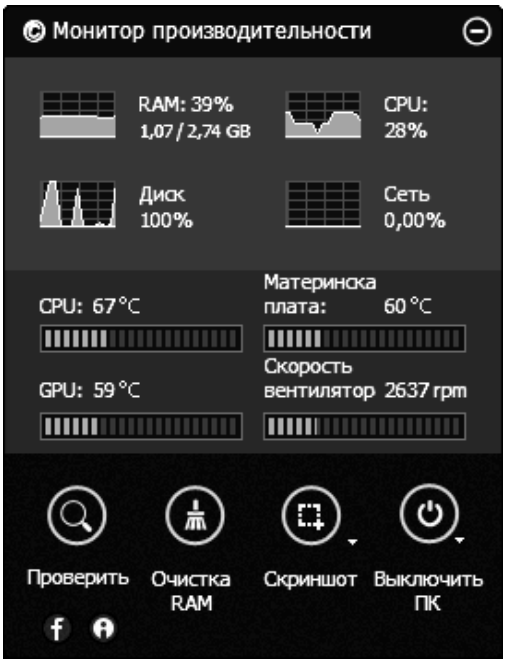


Рис. 7.7. Графічне зображення даних на шкалах

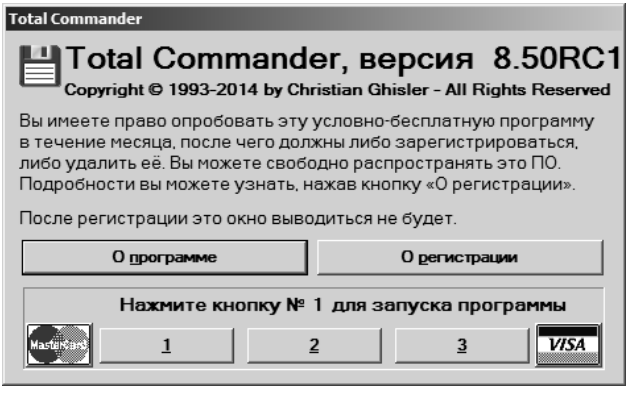


Рис. 7.8. Виділення буквено-цифрових даних

Виділення інформації за допомогою графічних елементів можна також використовувати для привернення уваги до змін, що відбуваються у різних частинах екрана. У випадку дуже швидких змін не

слід використовувати графічні елементи, оскільки відбувається накладення екранів, що спантеличує і дратує користувачів.

Для подання великих обсягів даних можна використовувати різні прийоми візуалізації, які вказують на родинні елементи даних. Розробники інтерфейсів повинні пам'ятати про можливості візуалізації, особливо якщо інтерфейс системи відображає фізичні сутності (об'єкти). Кілька прикладів візуалізації даних:

1) відображення метеорологічних даних, зібраних з різних джерел, у вигляді метеорологічних карт з ізобарами, повітряними фронтами та ін.;

2) графічне зображення стану телефонної мережі у вигляді зв'язаної множини вузлів;

3) візуалізація стану хімічного процесу з показаннями тиску і температури;

4) поведінка динамічної системи у тривимірному просторі за допомогою системи віртуальної реальності;

5) відображення множини *web*-сторінок у вигляді дерева гіпертекстових посилань (карта сайта).

Різні класи візуалізації часто використовуються на практиці, зокрема візуалізація даних з використанням дво- і тривимірних дерев та мереж. Більшість із них пов'язані з відображенням значних обсягів даних, керованих комп'ютером. Найбільше візуалізацію застосовують в інтерфейсах для подання деяких фізичних структур, наприклад молекулярної структури, телекомунікаційних мереж і т.ін. Тривимірні подання, для створення яких використовується спеціальне устаткування віртуальної реальності, є особливо ефективним способом візуалізації.

У всіх інтерактивних системах, незалежно від їх призначення, підтримуються кольорові екрани, тому в користувацьких інтерфейсах часто використовуються різні кольори. У деяких системах кольори застосовують переважно для виділення певних елементів (наприклад, у текстових редакторах для виділення фрагментів тексту); в інших системах (таких, як системи автоматичного проектування) кольорами позначають різні рівні проектів.

Правильне використання кольорів робить інтерфейс користувача більш зручним для розуміння і керування. Разом з тим використання кольорів може бути неправильним, у результаті чого створюються інтерфейси, які візуально неприваблиі і навіть провоку-

ють помилки. Основним принципом розробників інтерфейсів має бути обережне використання кольорів на екранах. Для цього потрібно застосовувати такі правила.

1. *Кількість кольорів обмежувати.* У вікні слід використовувати 4–5 різних кольорів, в інтерфейсі системи не повинно бути більше 7 кольорів.

2. *Зміни у стані системи відображати різними кольорами.* Якщо на екрані змінилися кольори, це означає, що відбулася якась подія. Виділення кольором особливо важливе і для складних екранів, на яких відображаються сотні різних об'єктів.

3. *Використовувати колірне кодування.* Виділяти аномальні елементи кольором; якщо потрібно знайти подібні елементи, їх показувати однаковим кольором.

4. *Колірне кодування має бути продуманим і послідовним.* Якщо в якій-небудь частині системи повідомлення про помилку зображаються, наприклад, червоним кольором, то у всіх інших частинах подібні повідомлення потрібно зображати таким самим кольором. Певні групи користувачів мають своє уявлення про значення окремих кольорів.

5. *Кольори використовувати відповідно до фізіологічних особливостей людського ока*. Послідовність червоних і синіх зображень спричиняє зорову напругу. Деякі комбінації кольорів можуть візуально порушувати або ускладнювати сприйняття інформації.

Різні професійні групи людей мають різне сприйняття кольорів; крім того, у різних професіях існують свої домовленості про значення окремих кольорів. Користувачі на підставі здобутих знань можуть неадекватно інтерпретувати такий самий колір. Наприклад, водієм червоний колір сприймається як небезпека. А для хіміка червоний колір означає гарячий.

Непослідовне використання кольорів також дезорієнтує користувача.

7.5. Засоби підтримання користувачів

Інтерфейс користувача повинен завжди забезпечувати деякий тип оперативної довідкової системи. Довідкові системи – один з основних аспектів проектування інтерфейсу користувача. Довідкову систему додатка становлять:

– повідомлення, що генеруються системою у відповідь на дії користувача;

– діалогова довідкова система;

– документація, що поставляється із системою.

Проектування корисної і змістовної інформації для користувача – досить складне завдання, воно повинно оцінюватися на тому самому рівні, що й архітектура системи або програмний код. Проектування повідомлень потребує значного часу і чималих зусиль. Доречно залучати до цього процесу професійних копірайтерів і дизайнерів. Проектуючи повідомлення про помилки або текстову довідку, необхідно враховувати фактори, наведені в табл. 7.4.

Таблиця 7.4

Фактори проектування текстових повідомлень

Фактор	Опис
Зміст	Довідкова система повинна знати, що робить користувач, і реагувати на його дії повідомленнями відповідного змісту
Досвід користувача	Якщо користувачі добре ознайомлені із системою, їм не потрібні детальні повідомлення. Утім користувачам-початківцям такі повідомлення видаються складними, малозрозумілими і надто короткими. Довідкова система повинна підтримувати обидва типи повідомлень, а також засоби керування складністю повідомлень
Професійний рівень користувача	Повідомлення повинні містити відомості, що відповідають їх професійному рівню. У повідомленнях для користувачів різних рівнів необхідно застосовувати різну термінологію
Стиль повідомлень	Повідомлення мають позитивний відтінок, використовувати активний тон, не бути образливими чи жартівливими
Культура	Розробник повідомлень повинен бути ознайомлений з культурою тієї країни, де планується продавати систему. Повідомлення, цілком доречне в культурі однієї країни, може виявитися неприйнятним в іншій

Приклад запрошення для входу в інтерфейс роутера показано на рис. 7.9. Користувачу пропонується заповнити форму, що складається з двох полів: «Ім’я користувача» і «Пароль».

Перше враження, яке користувач отримує у процесі роботи із ПС, ґрунтується на повідомленнях про помилки. Недосвідчені користувачі, зробивши помилку, повинні розуміти повідомлення, що з’явилося, про помилку. Приклад повідомлення про помилку в табличному редакторі *Excel* подано на рис. 7.10.

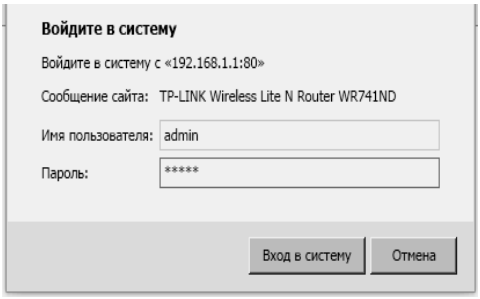


Рис. 7.9. Форма для входу в інтерфейс роутера

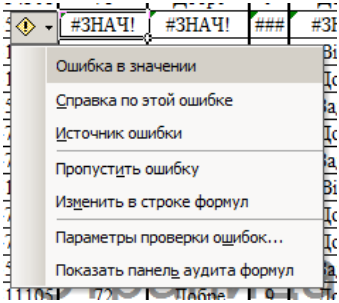


Рис. 7.10. Приклад повідомлення про помилку

Отримавши повідомлення про помилку, користувач часто не знає, що робити, і звертається до довідкової системи за інформацією. Довідкова система повинна надавати різні типи інформації: як ту, що допомагає користувачу в скрутних ситуаціях, так і конкретну, яку шукає користувач. Для цього довідкова система має містити різні засоби і різні структури повідомлень.

Довідкова система повинна забезпечувати користувача кількома точками входження (рис. 7.11). Користувач може ввійти в неї на верхньому рівні її ієрархічної структури і переглянути всі розділи довідкової інформації. Інші точки входження в довідкову систему – вікна повідомлень про помилки або опис конкретної команди додатка.

Усі довідкові системи мають складну мережеву структуру, у якій кожний розділ довідкової інформації може посилатися на кілька інших інформаційних розділів. Структура такої мережі, як правило, ієрархічна, з перехресними посиланнями, як показано на рис. 7.11. На верхньому рівні структурної ієрархії утримується загальна інформація, на нижньому – більш детальна.

Під час використання довідкової системи виникають проблеми, зумовлені тим, що користувачі входять у мережу після допущення помилки і далі переміщаються по ній. Через деякий час вони плутаються і не можуть визначити, у якому місці довідкової системи перебувають. У такій ситуації користувачі мають завершити сеанс роботи з довідковою системою і знову почати роботу з деякої відомої точки довідкової мережі.

Відображення довідкової інформації в декількох вікнах спрощує подібну ситуацію. Екран, на якому розміщені три рівні довідки, показано на рис. 7.12. Простір екрана завжди обмежений і розробнику варто

пам’ятати, що додаткові вікна можуть сховати іншу потрібну інформацію.

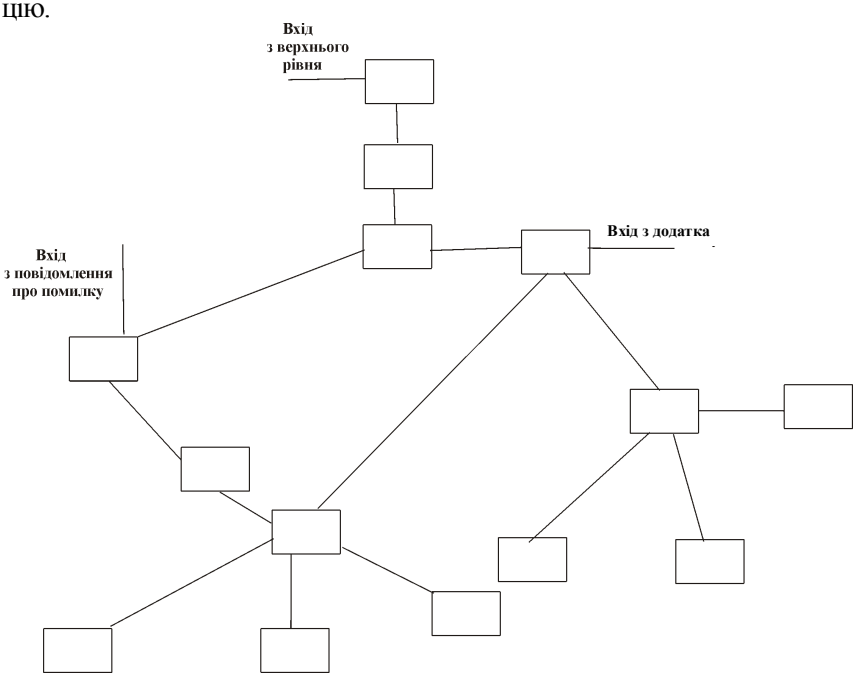


Рис. 7.11. Точки входу в довідкову систему

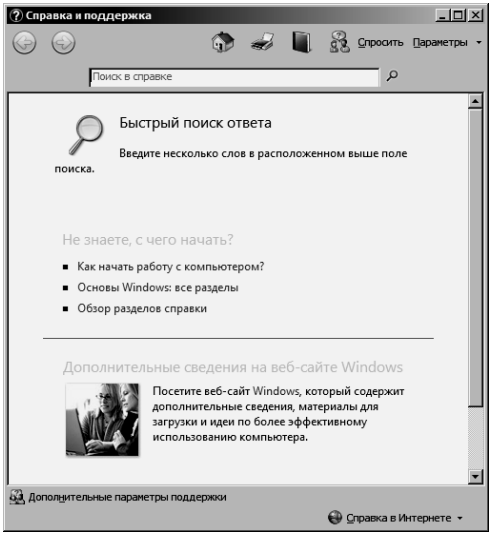


Рис. 7.12. Вікно довідкової системи операційної системи *Windows* 7

Тексти довідкової системи необхідно готувати разом з розробниками додатка. Довідкові розділи не повинні бути простим відтворенням посібника користувача, оскільки інформація на папері і на екрані сприймається по-різному. Сам текст (його розміщення і стиль) потрібно ретельно продумувати, щоб його можна було читати у вікні малого розміру. Розділ довідки «Швидкий пошук відповіді» на рис. 7.12 порівняно невеликий – у будь-якому довідковому розділі має бути лише необхідна інформація. У вікні, що відображає довідковий розділ, розміщено контекстне меню.

Користувачі мають кілька можливостей переміщення між розділами довідкової системи: перейти до розділу безпосередньо з відображуваного розділу, вибрати потрібний розділ з вікна «Журнал» для повторного перегляду і вибрати відповідний вузол на карті довідкової мережі і перейти до нього.

Довідкову систему можна реалізувати у вигляді групи зв’язаних *web*-сторінок або за допомогою узагальненої гіпертекстової системи, інтегрованої з додатком. Ієрархічна структура легко реалізується у вигляді гіпертекстових посилань. Перевага *web*-системи: проста в реалізації і не потребує спеціального ПЗ.

Щоб поставлена документація разом із програмною системою була корисна всім системним користувачам, вона повинна містити п’ять документів, які наведено в табл. 7.5.

Таблиця 7.5

Типи користувацької документації

Тип документа	Зміст	Споживач
Опис сервісів	Функціональний опис	Експерти
Рекомендації щодо установлення системи	Документація з інсталяції системи	Системні адміністратори
Початок роботи	Курс для початківців	Початкові користувачі
Детальний опис	Довідкове керівництво	Досвідчені користувачі
Обслуговування	Керівництво з експлуатації	Системні адміністратори

1. *Функціональний опис* – містить функціональні можливості системи.
2. *Документ з інсталяції системи* – містить інформацію щодо установлення системи.
3. *Курс для початківців* – оперує інформацією про роботу із системою та використовує загальні можливості.
4. *Довідкове керівництво* – описує можливості системи і їх використання, список повідомлень про помилки та можливі причини їх появи; розглянуто способи відновлення системи після виявлення помилок.
5. *Керівництво з експлуатації* – потрібне для деяких типів програмних систем. Подаються опис повідомлень, що генеруються системою при взаємодії з іншими системами, і способи реагування на ці повідомлення. Якщо в систему включено апаратну частину, то керівництво має містити інформацію про те, як виявити й усунути несправності, пов’язані з апаратурою, як підключити нові периферійні пристрої і т. ін. Зручні предметні покажчики допомагають швидко орієнтуватися в довідковій системі.

7.6. Оцінювання та попереднє тестування інтерфейсу

Процес, у якому оцінюється зручність використання інтерфейсу і ступінь його відповідності вимогам користувача, називається оцінюванням інтерфейсу. Таким чином, оцінювання інтерфейсу є частиною загального процесу тестування та атестації систем ПЗ.

У системах на основі ЕОМ чільне місце займають специфічні питання узгодження роботи людини-оператора і технологічної частини системи. Уведення–виведення (темп, форми подання) необхідно узгоджувати з людським фактором і відображенням інформації; клавіатурою та іншими органами керування; засобами комунікації; конструктивним виконанням пристроїв. Інтерфейс користувача – це видима частина програми. Ергономіка включається в процеси розроблення і тестування програмного продукту як частина системи якості. Користувацький інтерфейс розробляється одночасно з дизайном програмного продукту і передує його імплементації.

Процес розроблення користувацького інтерфейсу містить такі етапи:

1) аналіз трудової діяльності користувача, об'єднання бізнес-функцій у ролі;

2) побудова користувацької моделі даних, узгодження об'єктів і ролей та формування робочих місць;

3) формулювання вимог до роботи користувача і вибір показників оцінювання призначеного для користувача інтерфейсу;

4) розроблення узагальненого сценарію взаємодії користувача з програмним модулем (функціональною моделлю) і його попереднє оцінювання користувачами і замовником;

5) коригування і деталізація сценарію взаємодії, вибір і доповнення стандарту (керівництва) для побудови прототипу;

6) розроблення макетів і прототипів користувацького інтерфейсу та їх оцінювання в діловій грі, вибір остаточного варіанта;

7) розроблення засобів підтримання користувача і їх вбудовування в програмний код;

8) тестування тестової версії користувацького інтерфейсу за набором визначених показників;

9) підготовка документації для користувачів і розроблення програми навчання.

Із погляду ергономіки, найважливіше в програмі – створити призначений для користувача такий інтерфейс, який зробить роботу ефективною і продуктивною, а також задовольнить користувача.

Ефективний інтерфейс надає точність, функціональну повноту і завершеність виконання завдань на робочому місці користувача.

Користувацький інтерфейс має такі показники ефективності:

– точність роботи, що визначається відповідністю вимогам користувача до продукту;

– функціональна повнота – відображає ступінь використання первинних і оброблених даних, необхідних процедур оброблення або звітів. Вимірюється кількістю пропущених технологічних операцій або етапів під час виконання користувачем завдання;

– завершеність роботи – ступінь виконання виробничого завдання, усереднена за групою користувачів за певний термін або період, частка (або довжина черги) незадоволених (необроблених) заявок, відсоток продукції, що перебуває на проміжній стадії готовності, а також кількість користувачів, які виконали завдання у фіксований термін. Треба уникати такого інтерфейсу, який не відповідає алгоритму вирішення користувацьких завдань;

– простота вивчення предметної галузі та системи функціонування продукту;

– природність, звичність дій користувача при взаємодії з прикладною програмою;

– своєчасне і детальне інформування користувача в режимі запиту довідки і його інформаційне супроводження в онлайн режимі про об'єкти, дії і режими роботи;

– збалансоване, гармонійне використання колірних можливостей апаратного та ПЗ для відображення інформації;

– емоційна виразність текстів повідомлень;

– своєчасна і достатня інформація про суб'єкти автоматизованої діяльності, їх дії і правила роботи з ними;

– установлення, налаштування окремих характеристик інтерфейсу під користувача.

Для вирішення завдань користувача розробник повинен з'ясувати:

– яка інформація потрібна для вирішення завдання, а яку можна не враховувати;

– спільно з користувачем поділити інформацію на сигнальну, відображальну, що редагується, пошукову і результуючу;

– які рішення необхідно приймати в процесі роботи з програмою;

– чи може користувач виконувати кілька різних дій (вирішувати кілька завдань) одночасно;

– які типові операції використовує користувач під час вирішення завдань;

– які можуть бути наслідки в разі невиконання користувачем пропонованого алгоритму.

Дизайн користувацького інтерфейсу має мінімізувати зусилля користувача під час виконання роботи, що зумовить:

– скорочення тривалості операцій читання, редагування і пошуку інформації;

– зменшення часу навігації і вибору команди;

– підвищення загальної продуктивності користувача, що полягає в обсязі оброблених даних за певний період часу;

– збільшення тривалості стійкої роботи користувача тощо.

Вимоги до зручності і комфортності інтерфейсу зростають зі збільшенням складності робіт і відповідальності користувача за кінцевий результат. Задоволеність від роботи досягається в разі:

– прозорої для користувача навігації і цільової орієнтації в програмі операції, яку програма після цього кроку зробить;

– розуміння користувачем текстів і значення іконок. У програмі повинні бути ті текстові значення і графічні образи, які користувач знає або знає за характером його роботи чи займаної посади;

– інтенсивності навчання у ході роботи з програмою (мають переважати стандартні елементи взаємодії);

– наявності допоміжних засобів підтримання користувача (пошукових, довідкових, нормативних), у тому числі і для прийняття рішення в невизначеній ситуації (уведення за замовчуванням, обхід «зависання» процесів тощо).

Для оцінювання рівня зручності користувацького інтерфейсу використовують опитувальники, формуляри, чек-листи. До цієї роботи краще залучати фахівців з ергономіки. Зручний інтерфейс допомагає впоратися із втомою і напругою під час роботи в умовах високої відповідальності за результат.

Важливу роль відіграють питання технічної естетики, доцільного формування предметно-просторового середовища, тому на етапі розроблення прототипу необхідно:

1. Враховувати особливості пристроїв уведення–виведення інформації, використовуваних користувачем:

– розмір екрана монітора, розрізнення екрана, колірну палітру, мишу (з роликом або без), тип клавіатури та ін.

2. Вибирати технології і методи ведення діалогу програми з користувачем: ступінь активності користувача при взаємодії; ступінь врахування ситуації; відповідність очікуванням користувача; стійкість, терпимість до помилок користувача шляхом виправлення ти-

пових помилок; дублювання вручну окремих функцій системи і додаткові контрольні процедури роботи окремих режимів; налаштовування користувацького інтерфейсу на різний рівень підготовки користувача, специфіку завдання, ступінь адаптивності користувацького інтерфейсу до переваг користувача.

3. Розміщувати інформацію і керувальні елементи в поле екрана, у вікні. При композиції екрана необхідно враховувати обмежені розміри простору екрана для оптимального розміщення максимально можливого обсягу інформації.

4. Формувати зворотний зв'язок між користувачем і додатком: показ актуального стану системи, режиму роботи системи і режиму взаємодії; виведення окремих, важливих для робочої операції даних і показників; відображення дій користувача; зрозумілість та інформативність повідомлень системи.

5. Проектувати панелі меню та інструментів (панелі інструментів) і вибір пунктів у них: логічне і смислове угруповання пунктів; фіксована позиція панелей на екрані; обмеження на ширину списку і глибину меню; використання звичних назв, поширених іконпіктограм, традиційних іконок-символів і акуратне введення скорочень.

6. Розробляти засоби орієнтації і навігації: легкість визначення місцезнаходження, зручний перехід, швидкий пошук у списку і т. ін.

7. Створювати форми для введення даних: визначення способів уведення даних, виділення редагованих обов'язкових і необов'язкових.

Принципи реалізації призначеного для користувача інтерфейсу:

– стильова гнучкість – можливість використовувати різні інтерфейси з одним і тим самим додатком;

– спільне нарощування функціональності – можливість розвивати додаток без руйнування існуючого інтерфейсу;

– масштабованість – можливість легко налаштовувати і розширювати як інтерфейс, так і сам додаток при збільшенні кількості користувачів, робочих місць, обсягу і характеристик даних;

– адаптивність до дій користувача – додаток має припускати можливість уведення даних і команд різними способами і багатоваріативність доступу до прикладних функцій. У програмі має враховуватись можливість переходу і повернення від вікна до вікна, від режиму до режиму, і правильно обробляти такі ситуації;

– незалежність ресурсів – для створення призначеного для користувача інтерфейсу повинні надаватися окремі ресурси, спрямовані

на зберігання і оброблення даних, необхідних для підтримання користувача;

– кросплатформеність – для переходу на іншу апаратну (програмну) платформу. Здійснюється перенесення користувацького інтерфейсу і кінцевого додатка, про що потрібно інформувати користувача. Один зі способів інформування користувача про хід виконання роботи – використовувати індикатор процесу.

В ідеалі оцінювання належить проводити відповідно до показників зручності використання інтерфейсу (табл. 7.6). Кожний показник потрібно подавати у числовій формі. Наприклад, навченість оцінюється за кількістю помилок, що здійснює досвідчений оператор після тригодинного навчання. Позитивним вважається оцінювання, коли використовується 80 % функціональних можливостей системи. Однак частіше зручність використання інтерфейсу оцінюється якісно, а не через числові значення показників.

Таблиця 7.6

Показники зручності використання інтерфейсу

Показник	Опис
Навченість	Тривалість навчання для початку продуктивної роботи із системою

Закінчення табл. 7.6

Показник	Опис
Швидкість роботи	Швидкість реакції системи на дії користувача
Стійкість	Стійкість системи до помилок користувача
Відновлюваність	Здатність системи відновлюватися після помилок користувачів
Адаптованість	Здатність системи «підбудовуватися» до різних стилів роботи користувачів

Повне оцінювання користувацького інтерфейсу може виявитися досить дорогим. У цей процес будуть залучені фахівці з когнітивної психології і дизайнери. Процес оцінювання може охоплювати розроблення і виконання ряду статистичних експериментів з користувачами в спеціально створених лабораторіях і з необхідним для спостереження устаткуванням. Таке оцінювання інтерфейсу економічно нерентабельне для систем, розроблюваних у невеликих організаціях з обмеженими ресурсами.

Існують більш прості і менш дорогі методики оцінювання інтерфейсів користувача, що дозволяють виявляти окремі дефекти в інтерфейсах. Це зокрема:

1) анкети, за допомогою яких користувач оцінює інтерфейс;

2) спостереження за роботою користувачів з наступним обговоренням їх способів використання системи для вирішення конкретних завдань;

3) відеоспостереження типового використання системи;

4) додавання в систему програмного коду, що збирає інформацію про найчастіше використовувані системні сервіси та найпоширеніші помилки.

Анкетування користувачів – відносно дешевий спосіб оцінювання інтерфейсу. Запитання повинні бути точними, а не загальними. Не слід використовувати питання типу «прокоментуйте практичність системи», оскільки відповіді, імовірно, будуть істотно розрізнятися. Запитання слід конкретизувати, наприклад: «Оцініть зрозумілість повідомлень про помилки за шкалою від 1 до 5. Оцінювання 1 означає повністю зрозуміле повідомлення, 5 – малозрозуміле». На такі питання легше відповісти і більш імовірно отримати в результаті корисну для поліпшення інтерфейсу інформацію.

Під час заповнення анкети користувачі мають обов'язково оцінити власний досвід і знання. Такі відомості дозволяють розробникам зафіксувати рівень знань користувачами інтерфейсу. Якщо проект інтерфейсу вже створений і оцінений у паперовому вигляді, анкети можна використовувати навіть до повної реалізації системи.

Під час спостереження за роботою користувачів оцінюється, як вони взаємодіють із системою, які використовують сервіси, які роблять помилки тощо. Разом зі спостереженнями можуть проводитися семінари, на яких користувачі розповідають про свої спроби вирішити ті або інші проблеми, а також як вони розуміють систему і як використовують її для досягнення цілей.

Відеоустаткування відносно недороге, тому до безпосереднього спостереження можна додати відеозапис користувацьких семінарів для наступного аналізу. Повний аналіз відеоматеріалів дорогий і потребує спеціально оснащеного комплекту з декількома камерами, спрямованими на користувача та екран. Однак відеозапис окремих дій користувача може виявитися корисним лише для виявлення дій

користувача, що зумовлюють проблеми. Тоді варто вдатися до інших методів оцінювання.

Аналіз відеозаписів дозволяє розробнику встановити кількість маніпуляцій руками користувача (частоту переходу із клавіатури на мишу) і виявити неприродні рухи очима. Якщо під час роботи з інтерфейсом потрібно часто зміщувати зоровий фокус, користувач може зробити більше помилок і пропустити деякі елементи.

Уставлення в програму коду, що збирає статистичні дані в ході використання системи, поліпшує інтерфейс, оскільки виявляються найбільш часто використовувані операції. Інтерфейс змінюється так, щоб ці операції вибиралися швидше порівняно з іншими. Наприклад, у вертикальному меню або меню, що випадають, часто використовувані команди мають розміщуватися вгорі списку. Такий код також дозволить виявити і змінити команди, що спричиняють появу помилок.

Кожна програма має містити нескладні засоби, за допомогою яких користувач зможе передавати розробникам повідомлення зі «скаргами». Такі засоби переконують користувачів у тому, що їх думку поважають. А розробники інтерфейсу й інші фахівці можуть отримати швидкий зворотний зв'язок щодо окремих проблем інтерфейсу.

Жоден із цих методів оцінювання користувацького інтерфейсу не є надійним і не гарантує вирішення всіх проблем інтерфейсу. Перед випуском системи ці методи застосовують групи добровольців. Це сприяє виявленню і виправленню більшості проблем інтерфейсу користувача.

Контрольні запитання і завдання

1. Запропонуйте придатні об'єкти маніпулювання для таких типів користувачів систем з урахуванням понять предметної галузі додатка ПЗ:
   – автоматизований каталог товарів складу асистента;
   – система спостереження за безпекою літака цивільної авіації;
   – фінансова база даних менеджера;
   – система керування патрульними поліцейськими машинами.
2. Опишіть ситуації, у яких недоцільно або неможливо підтримувати інтерфейс користувача.
3. Які фактори варто враховувати під час проектування інтерфейсів, що використовують меню, для таких систем, як банкомати? Опишіть основні ознаки відомого інтерфейсу банкомата.

4. Запропонуйте способи адаптації користувацького інтерфейсу в системах електронної комерції (наприклад, віртуальної книгарні або магазину музичних дисків) для користувачів, що мають фізичні вади, наприклад поганий зір або проблеми опорно-рухового апарата.
5. Сформулюйте переваги графічного способу відображення інформації і наведіть чотири приклади додатків, у яких більш доречно використовувати графічне подання числових даних, а не табличне.
6. Якими основними принципами варто керуватися у разі використання кольорів в інтерфейсах користувача? Запропонуйте більш ефективний спосіб використання кольорів в інтерфейсі будь-якого відомого додатка.
7. Розгляньте повідомлення про помилки, що генеруються операційними системами MS Windows, Unix, MacOS або будь-якою іншою. Запропонуйте шляхи їх поліпшення.
8. Складіть анкету для збирання даних про інтерфейс якої-небудь відомої вам програми (наприклад, текстового редактора). Якщо є можливість поширити цю анкету серед інших користувачів, спробуйте оцінити результати анкетування. Що ви довідалися про інтерфейс програми з анкет?
9. Чи етично розробляти програмні системи, не узгодивши з кінцевими користувачами ті елементи системи, які вони будуть контролювати?
10. З якими етичними проблемами зіштовхуються розробники інтерфейсів, коли намагаються узгодити запити кінцевих користувачів системи з вимогами організації, що оплачує розроблення системи?
11. Сформулюйте принципи проектування інтерфейсу користувача.
12. Наведіть п'ять різних стилів взаємодії користувача із програмними системами.
13. Назвіть стилі подання інформації. У яких випадках доцільне графічне зображення даних?
14. Які правила проектування засобів підтримання користувача застосовуються у відомому програмному забезпеченні?
15. Назвіть показники зручності використання систем.

ІІІ. РЕАЛІЗАЦІЯ ОБ'ЄКТНО-ОРІЄНТОВАНИХ ПРОГРАМНИХ СИСТЕМ

8. ОСНОВИ ОБ'ЄКТНО-ОРІЄНТОВАНОГО ПОДАННЯ ПРОГРАМНИХ СИСТЕМ

8.1. Принципи об'єктно-орієнтованого подання програмних систем

Розгляд будь-якої складної системи потребує застосування техніки декомпозиції – поділу на складові елементи. Відомі дві схеми декомпозиції: *алгоритмічна та об'єктно-орієнтовна*. В основу алгоритмічної декомпозиції покладено поділ за діями. Цю схему подання застосовують у звичайних ПС. Об'єктно-орієнтована декомпозиція забезпечує поділ за об'єктами реального (або віртуального) світу. Ці об'єкти – великі елементи, кожний з яких містить описи

дій і даних. Вони створюються завдяки механізмам абстракції, інкапсуляції, модульності та ієрархії.

8.1.1. Абстрагування

Апарат абстракції – зручний інструмент спрощення реальних систем. Створюючи певні поняття для будь-якого завдання, необхідно відволікатися (абстрагуватися) від неістотних характеристик конкретних об'єктів, визначаючи лише істотні. Наприклад, в абстракції «блок живлення» виокремлюють характеристику «забезпечення електроенергією», не беручи до уваги такі характеристики, як тип електроенергії, показники вхідної та вихідної напруг, габаритні розміри, типи рознімів, виробник. Тобто, абстрагування зводять до формування абстракцій. Кожна абстракція фіксує основні характеристики об'єкта, які відрізняють його від інших видів об'єктів і забезпечують понятійні межі предметної галузі.

Абастракція зосереджує увагу на основних функціональних характеристиках поводження об'єкта на відміну від його реалізації. Її зручно будувати шляхом виділення функцій об'єкта. Вона є основою об'єктно-орієнтованого програмування і дозволяє працювати з об'єктами, не вдаючись в особливості їх реалізації. Така можливість надається завдяки спеціальному інтерфейсу, де зосереджується опис усіх можливих застосувань програми. Таким інтерфейсом може бути деяка множина функцій, за допомогою яких створюється «абстракція». Інтерфейс запобігає спотворенню даних і вивільняє користувача від необхідності знати деталі реалізації.

Програмісту абстракція дає змогу визначати нові типи даних – класи. Членам класу доступні лише функції з явно оголошеного набору.

Приклад 8.1. Мовою *C* створюється абстрактний клас, який містить дані, доступні деяким функціям у тому вигляді, у якому вони визначені в класі. Загальний вигляд класу:

```
class data_type
    { // опис даних
        /* список функцій ("дружні" функції) */
    };
```

Приклад 8.2. Абстрактний клас певного типу даних, у якому є множина функцій, що має доступ до змінних класу, і самі є складовою частиною самого типу; записується так:

```
class object_type
```

```
{
    /* опис, що використовується для реалізації object_type */
 public:
    /* опис, що специфікує інтерфейс із object_type */
 };
```

8.1.2. Інкапсуляція

Інкапсуляція й абстракція – взаємодоповняльні поняття: абстракція виділяє зовнішнє поводження об'єкта, а інкапсуляція містить і приховує реалізацію, що забезпечує це поводження. Інкапсуляція досягається за допомогою інформаційної закритості. Зазвичай приховуються структура об'єктів і реалізація їх методів.

Інкапсуляція є процесом поділу елементів абстракції на секції з різною видимістю. Вона відділяє інтерфейс абстракції від її реалізації. З точки зору мови програмування інкапсуляція є механізмом, що обмежує доступ до складових об'єкта компонентів (методів та властивостей і робить їх приватними). Із погляду іншого програміста об'єкт подається «чорним ящиком», який має лише входи та виходи. Механізм застосування «чорного ящика» робить непомітними для користувача зміни роботи об'єкта, а не програми.

Приклад 8.3. Мовою С:

```
Class A
{ public:
        Int a, b; /* дані відкритого інтерфейсу */
        Int ReturnSomething(); /* метод відкритого інтерфейсу */
  private:
        int Aa, Ab; /* приховані дані */
        Void Do_Something(); /* прихований метод */
}.
```

Клас *A* інкапсулює властивості *Aa*, A*b*, метод *Do_Something* подає зовнішній інтерфейс *ReturnSomething* властивостей *a*, *b*.

Приклад 8.4. Фізичний об'єкт – регулятор швидкості може вмикатися, вимикатися, збільшувати швидкість, зменшувати швидкість, відображати свій стан. Подамо його роботу псевдокодом:

```
Class_Датчикшвидкості.Клас_Порт;
Class_Регуляторшвидкості
    { private
        Тип Режим (Збільшення, Зменшення);
```

```
        Тип даних: Розміщення;
        тип Регуляторшвидкості;
    function Новрегуляторшвидкості
    {номер: Розміщення;
                                напр.:  Напрямок
    };
    function Увімкнути(Регуляторшвидкості);
    function Увімкнути(Регуляторшвидкості);
    function Збільшитишвидкість(Регуляторшвидкості);
    function Зменшитишвидкість(Регуляторшвидкості);
    function ЗапитСтану(Регулятор);
private
    зробити запис Регуляторшвидкості
        Номер;
        Розміщення;
        Стан: Режим;
        Керування: укз_напорт;
    завершити зробити запис;
завершити Class_Регуляторшвидкості;
}.
```

Допоміжний тип «Режим» використовується для задання основного типу класу, клас «Датчикшвидкості» забезпечує клас регулятора описом допоміжного типу «Напрямок», клас «Порт» фіксує абстракцію порту, через який посилаються повідомлення для регулятора. Три властивості «Номер», «Стан», «Керування» – формулюють інкапсульоване подання основного типу класу «Регулятор швидкості». За спроби клієнтів отримати доступ до цих властивостей фіксується семантична помилка.

Повне інкапсульоване подання класу «Регулятор швидкості» включає опис реалізацій його методів – він утримується в тілі класу. Інкапсуляція закриває деталі внутрішнього подання абстракцій.

8.1.3. Модульність

У мовах *C++*, *C#*, *Object Pascal*, *Java*, *Ada* 95 абстракції класів і об'єктів формують логічну структуру системи. Для побудови фізичної структури ці абстракції розміщують у модулях. Великі системи, які складаються із сотень класів, мають модулі, які допомагають керу-

вати складністю. Модулі є фізичними контейнерами, у яких оголошуються класи й об'єкти логічного розроблення.

Модульність визначає здатність системи піддаватися декомпозиції на ряд сильнозв'язаних і слабкоз'єднаних модулів.

Загальна мета декомпозиції на модулі – зменшення термінів розроблення і вартості проектованої системи за рахунок виділення модулів, які проектуються і змінюються незалежно. Кожна модульна структура має бути досить простою, щоб бути повністю зрозумілою. Реалізація модулів має виконуватися без знання реалізації інших модулів і без впливу на їх поводження.

Класи й об'єкти визначаються в ході логічного розроблення, а модулі – у ході фізичного розроблення системи. Ці дії є ітераційними і взаємозалежними.

У C++ модулями є файли, традиційним – розміщення інтерфейсної частини модулів в окремі файли з розширенням *.h* (файл-заголовок), текст модуля зберігається у файлах з розширеннями *.cpp*, *.cxx*, *.cc*. Зв'язок між модулями оголошується за допомогою директиви *#include*.

Приклад 8.5. Файл *Book.h* директивою *#include* вставляється в поточний файл перед початком його компіляції під час першого проходження компілятора по програмі препроцесора. Файл *Book.h* називають файлом заголовків.

```
#include "Book.h"
. . .
Book b;
```

Різні мови програмування мають власні засоби щодо створення модулів. Так, мова *Pascal* для запису модулів використовує слово *unit*, в *Ada* потужним засобом забезпечення модульності є пакет. Модульність показує шлях угруповання зв'язаних за певною логікою абстракцій.

8.1.4. Ієрархічна організація

Доповненням до абстракції, інкапсуляції та модульності є ієрархічна організація програм та систем. Визначенням ієрархії в проекті спрощується розуміння проблем замовників і їх реалізація – складна система стає доступною для уявлення її людиною. Ієрархічна організація задає розміщення абстракцій на різних рівнях опису системи.

Двома важливими інструментами ієрархічної організації в об'єктно-орієнтованих системах є:

– структура з класів;

– структура з об'єктів.

Найчастіше ієрархічна структура будується за допомогою спадкування, яке визначає відношення між класами, що розділяє структуру або поводження, один до одного (одиничне спадкування) або один до багатьох (множинне спадкування).

Приклад 8.6. Для програми керувань польотом другого ступеня, яка подібна до програми керування польотом ракети-носія першого ступеня визначимо клас керування польотом другого ступеня, що інкапсулює її спеціальне поводження за допомогою псевдокоду:

```
Class_Керпольотом1;
{Class _Керпольотом2
    { тип Траєкторія (Гнучка, Вільна);
     тип керпольотом2.Керпольотом1
     private:
     function Встановити: Керпольотом2:
        тип: Траєкторія; орієнтація: Кути;
        параметри: Координати_Швидкість; команди: Графіки);
     function Умови_відділення_ступеня (Керпольотом2;
        критерій: Критерій_відділення);
     function Прогноз_відділення_ступеня (Керпольотом2);
     function Виконання_команд (Керпольотом2);
private:
        тип Керпольотом2 {Керпольотом1
        тип траєкторії: траєкторія; довідділення: Бортовий_час;
              виконання_команд: бульовий;
    }
}.
```

Із прикладу 8.6 бачимо, що клас Керпольотом2 – це різновид класу Керпольотом1, що є батьківським класом. До класу Керпольотом2 додані:

– допоміжний тип Траєкторія;

– три нові властивості (тип_траєкторії, умови_відділення, виконання_команд);

– три нові методи (Умови_відділення_ступеня, Прогноз_відділення_ступеня, Виконання_команд).

Крім того, у класі «Керпольотом2» перевизначено метод батьківського класу «Встановити». Тобто до спадщини класу «Керпольотом2» додано набір методів і властивостей класу «Керпольотом1». Зокрема, тип «Керпольотом2» включає поля типу «Керпольотом1», що забезпечують приймання даних про координати та швидкість ракети-носія, її кутову орієнтацію і послідовність команд, а також про критерії відділення наступного ступеня.

Класи «КерПольотом1» і «КерПольотом2» утворюють спадкоємну ієрархічну організацію. У ній загальна частина структури і поводження зосереджені у верхньому, найбільш загальному класі (батьківському класі). Батьківський клас відповідає загальній абстракції, а підклас – спеціалізованій абстракції, у якій елементи суперкласу доповнюються, змінюються і навіть приховуються. Тому спадкування часто називають відношенням *узагальнення – спеціалізація*.

Ієрархію спадкування можна продовжити. Наприклад, використовуючи клас Керпольотом2, можна оголосити ще більш спеціалізований підклас – «Керпольотом космічним апаратом».

Інший різновид ієрархічної організації – структура з об'єктів, яка базується на відношенні агрегації. Агрегація не є поняттям, унікальним для об'єктно-орієнтованих систем. Наприклад, будь-яка мова програмування, що дозволяє використовувати структури типу «запис», підтримує агрегацію. Але агрегація особливо корисна у сполученні зі спадкуванням:

1) агрегація забезпечує фізичне угруповання зв'язаної за певною логікою структури;
2) спадкування дозволяє легко і багаторазово використовувати загальні групи в інших абстракціях.

Приклад 8.7. Застосування класу «ВимірювачСК» (вимірювач системи керування літального апарата):

*Class*_Налаштування_датчиків. Клас_Датчик;
Modul Клас_Вимірювач СК
{
    тип Вимірювач СК;
    private;
    опис методів;
{*private*:
    тип вкз_на_датчик;
    тип Вимірювач СК
        датчики: масив(1..30) з вкз_на_датчик;

функція настроювання: Налаштування_датчиків;
}
}.

Очевидно, що об'єкти типу «Вимірювач СК» є агрегатами, що складаються з масиву датчиків і процедури налаштування. Введена абстракція «Вимірювач СК» дозволяє використовувати в системі керування різні датчики. Зміна датчика не змінює індивідуальності вимірювача в цілому. Адже датчики вводяться в агрегат за допомогою покажчиків, а не величин. Таким чином, об'єкт типу «Вимірювач СК» і об'єкти типу «Датчик» мають відносну незалежність. Наприклад, час життя вимірювача і датчиків незалежні один від одного. Навпаки, об'єкт типу «Налаштування датчиків» фізично включається в об'єкт типу «Вимірювач СК» і незалежно існувати не може. Звідси висновок – руйнуючи об'єкт типу «Вимірювач СК», своєю чергою, руйнується екземпляр «Налаштування датчиків».

Якщо порівняти елементи ієрархій спадкування і агрегації щодо рівнів складності, то можна помітити, що при спадкуванні нижній елемент ієрархії (підклас) має більші рівні складності (більші можливості), при агрегації – навпаки (агрегат «Вимірювач СК» має більші можливості, ніж його елементи – датчики і процедура налаштування).

8.2. Об'єкти, класи, відношення між класами й об'єктами

Об'єкти – конкретні сутності, які існують у часі і просторі, вид подання абстракції. Об'єкт має індивідуальність, стан і поводження. Структуру і поводження подібних об'єктів визначено в їх загальному класі. Терміни «екземпляр класу» і «об'єкт» взаємозамінні.

Індивідуальність – це характеристика об'єкта, що відрізняє його від інших об'єктів.

Стан об'єкта характеризується переліками властивостей об'єкта і поточних значень кожної із цих властивостей. Об'єкти не існують ізольовано один від одного. Вони зазнають впливу або самі впливають на інші об'єкти.

Приклад об'єкта за іменем «*Стілець*», що має певний набір властивостей і операцій, подано на рис. 8.1.

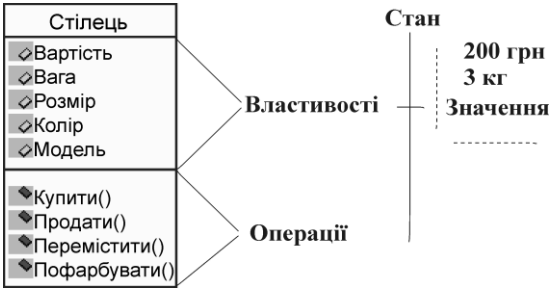


Рис. 8.1. Подання об’єкта з іменем «Стілець»

Поводження характеризується у термінах зміни стану та передавання повідомлень, тобто показує, як об’єкт впливає на інші об’єкти (або зазнає впливу). Поводження об’єкта є функцією як його стану, так і функцією виконуваних ним операцій (купити, продати, зважити, перемістити, пофарбувати).

Операція позначає обслуговування – об’єкт пропонує своїм клієнтам.

8.2.1. Основні операції, виконувані клієнтом з об’єктом

Можливі п’ять видів операцій клієнта над об’єктом:

1) модифікатор змінює стани об’єкта;
2) селектор надає доступ до стану, але не змінює його;
3) ітератор надає доступ до змісту об’єкта його окремими частинами, у строго визначеному порядку;
4) конструктор (створює об’єкт та ініціалізує його стан);
5) деструктор (руйнує об’єкт і звільняє займану ним пам’ять).

Приклади операцій наведено в табл. 8.1.

Таблиця 8.1

Різновиди операцій

Види операцій	Приклади
Модифікатор	Змінити (*V*)
Селектор	Визначити напругу ( ) : *integer*
Ітератор	Показати асортимент товарів ( ) : *string*
Конструктор	Увімкнути в роботу (параметри)
Деструктор	Вимкнути з роботи ( )

В об’єктно-орієнтованих мовах програмування операції можуть оголошуватися лише як методи – елементи класів, екземплярами яких є об’єкти. Гібридні мови (*C++*, *Ada* 95) дозволяють писати операції як вільні підпрограми (поза класами). Відповідні приклади показано на рис. 8.2.

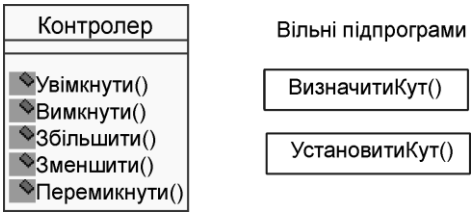


Рис. 8.2. Методи та вільні підпрограми

Усі методи та вільні підпрограми, асоційовані з конкретним об’єктом, утворюють його протокол. Таким чином, протокол визначає оболонку допустимих поводжень об’єкта і тому містить цільне (статичне і динамічне) подання об’єкта.

Великий протокол корисно поділити на логічні угруповання поводжень. Ці угруповання, що поділяють простір поводжень об’єкта, позначають ролі, які може виконувати об’єкт. Принцип виділення ролей ілюструє рис. 8.3.

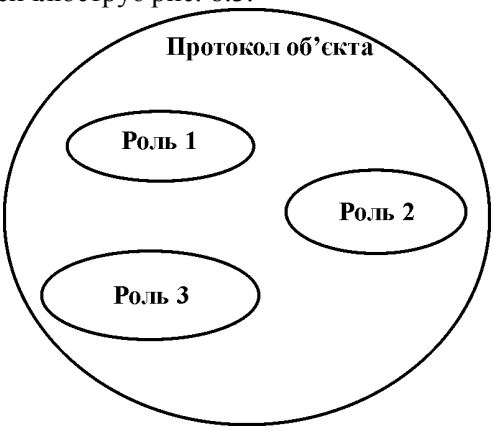


Рис. 8.3. Простір поводжень об’єкта

Із погляду зовнішнього середовища значущим є поняття «обов'язки об'єкта». Обов'язки означають зобов'язання об'єкта забезпечити відповідне поводження. Його обов'язками є всі види обслуговування, які він пропонує клієнтам. У світі об'єкт відіграє певні ролі, виконуючи свої обов'язки.

Наявність внутрішніх станів об'єкта означає, що важливим є порядок виконання операцій. Тобто об'єкт може бути незалежним автоматом. За аналогією з автоматами можна виділяти активні і пасивні об'єкти. Активний об'єкт має власний канал (потік) керувань, пасивний – не має такого каналу (рис. 8.4).

Рис. 8.4. Активний та пасивний об'єкти

Активний об'єкт автономний, він може проявляти поводження без впливу на нього інших об'єктів. Пасивний об'єкт, навпаки, може змінювати свої стани лише під впливом інших об'єктів.

8.2.2. Типи відношень між об'єктами

Розробники ПЗ працюють не з поодинокими об'єктами, а із взаємодійними об'єктами, оскільки саме взаємодія об'єктів реалізовує поводження системи. Відношення між парою об'єктів ґрунтуються на взаємній інформації про дозволені операції і очікуване поводження. Особливо привабливі два види відношень між об'єктами: зв'язки й агрегація.

Зв'язок – це фізичне або понятійне поєднання об'єктів. Об'єкт взаємодіє з іншими об'єктами через їх зв'язки. Зв'язок позначає з'єднання, за допомогою якого:

– об'єкт-клієнт викликає операції об'єкта-постачальника;

– один об'єкт переміщує дані до іншого об'єкта.

Зв’язки між об’єктами показано на рис. 8.5 за допомогою сполучних ліній. Зв’язки являють собою можливі шляхи для передавання повідомлень. Самі повідомлення показано стрілками, що позначають їх напрямок та імена викличних операцій.

Рис. 8.5. Зв’язки між об’єктами

У зв’язку об’єкт виконує одну із трьох ролей:

– актора – активного об’єкта, що впливає на інші об’єкти і не знає впливу інших об’єктів;

– сервера – пасивного об’єкта, що не впливає на інші об’єкти, використовується іншими об’єктами;

– агента – об’єкта, що може як впливати на інші об’єкти, так і використовуватися ними.

Агент створюється для виконання роботи від імені актора або іншого агента. Керувальна програма виконує функції актора, двигун та датчик швидкості – сервера, регулятор – агента на рис. 8.5.

Видимість об’єктів. Розглянемо об’єкти *A* і *B*, між якими є зв’язок. Для того щоб об’єкт *A* міг надіслати повідомлення об’єкту *B*, потрібно, щоб *B* був видимий для *A*.

Розрізняють чотири форми видимості між об’єктами.

1) об’єкт-постачальник (сервер) є глобальним для клієнта;

2) об’єкт-постачальник (сервер) є параметром операції клієнта;

3) об’єкт-постачальник (сервер) є частиною об’єкта-клієнта;

4) об’єкт-постачальник (сервер) є локально оголошеним об’єктом в операції клієнта.

На етапі аналізу фактор видимості зазвичай не беруть до уваги. На етапах проектування і реалізації фактор видимості зв'язків обов'язково має розглядатися.

Агрегація. Зв'язки позначають рівноправні (клієнт-серверні) відношення між об'єктами. Агрегація позначає відношення об'єктів в ієрархії «ціле/частина». Агрегація забезпечує можливість переміщення від цілого (агрегату) до його частин (властивостей).

Агрегація може позначати, а може і не позначати фізичне включення частини в ціле. Приклад фізичного включення (композиції) частин (вхідного фільтра, високочастотного перетворювача, вихідного фільтра) в агрегат «блок живлення» показано на рис. 8.6. У цьому випадку частини включені в агрегат за величиною.

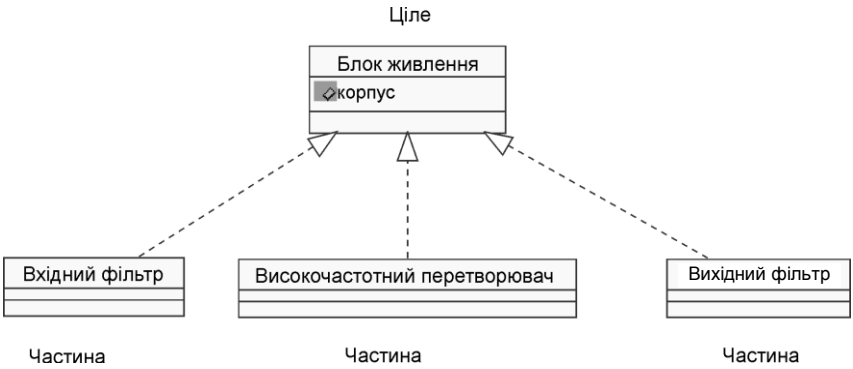


Рис. 8.6. Фізичне включення частин в агрегат (за величиною)

Приклад нефізичного включення частин «користувача», «комп'ютера» в агрегат «комп'ютерна система» показано на рис. 8.7. Очевидно, що користувач і комп'ютер є елементами комп'ютерної системи, але вони не входять у нього фізично. У цьому випадку частини включені в агрегат за посиланням.

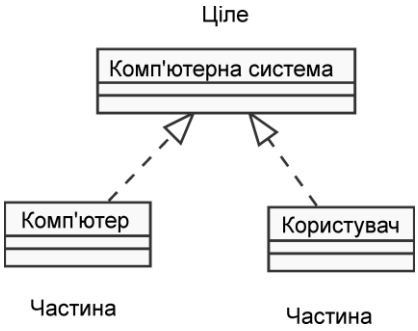


Рис. 8.7. Нефізичне включення частин в агрегат
(за посиланням)

Отже, між об’єктами існують два види відношень – зв’язки і агрегація. Вибираючи вид відношення, потрібно враховувати такі фактори:

– зв’язки забезпечують слабке з’єднання між об’єктами;

– агрегація інкапсулює частини як складові цілого.

8.2.3. Класи

Поняття об’єкта і класу тісно пов’язані. Проте існує важлива розбіжність між цими поняттями. Клас – це абстракція істотних характеристик об’єкта.

Клас – опис множини об’єктів, які мають однакові властивості, операції, відношення і семантику (зміст). *Об’єкт* – екземпляр класу.

Розрізняють внутрішнє подання класу (реалізацію) і зовнішнє подання класу (інтерфейс), рис. 8.8.

Інтерфейс повідомляє можливості (послуги) класу, але приховує його структуру і поводження. Інакше кажучи, інтерфейс демонструє зовнішньому середовищу абстракцію класу, його зовнішній вигляд. Він складається з оголошень усіх операцій, застосовних до екземплярів класу, а також може включати оголошення типів даних, змінних, констант і виключень, необхідних для повноти даної абстракції. Інтерфейс може поділятися на три частини:

Клас	
Інтерфейсні	
Частина	Публічна

1) публічну (*public*), яка доступна всім клієнтам;

2) захищену (*protected*), яка доступна лише самому класу,

Рис. 8.8. Структура подання класу

його підкласам і друзям;

3) приватну (*private*), яка доступна лише самому класу і його друзям.

Дружнім класом називають клас, що має доступ до всіх частин цього класу (публічної, захищеної і приватної). Реалізація класу описує його поводження. Вона включає реалізації всіх операцій, що визначені в інтерфейсі класу.

8.2.4. Типи відношень між класами

Класи, подібно до об'єктів, не є ізольованими. Навпаки, з окремою проблемною галуззю їх пов'язують ключові абстракції, відношення між якими формують структуру із класів системи.

Між класами є чотири основні види відношень:

– асоціація (фіксує структурні відношення – зв'язки між екземплярами класів);

– залежність (відображає вплив одного класу на інший клас);

– узагальнення–спеціалізація;

– ціле–частина.

Для покриття основних відношень більшість об'єктно-орієнтованих мов програмування підтримує такі відношення:

1) асоціацію;

2) спадкування;

3) агрегацію;

4) залежність;

5) конкретизацію;

6) метаклас;

7) реалізацію.

Асоціація забезпечує взаємодію об'єктів, що належать до різних класів. Завдяки асоціаціям отримується працююча система. Без асоціацій система перетворюється в набір ізольованих класів-одинаків. Асоціація класів позначає семантичний їх зв'язок.

Спадкування – найбільш популярний різновид відношень узагальнення – спеціалізація. Альтернативою спадкуванню вважається делегування. Об'єкти делегують своє поводження родинним об'єктам. При цьому класи стають не потрібними.

Агрегація забезпечує відношення ціле – частина, що повідомляється для екземплярів класів.

Залежність часто подається у вигляді окремої форми – використання, що фіксує відношення між клієнтом, що запитує послугу, і сервером, що надає цю послугу.

Конкретизація є різновидом відношень узагальнення – спеціалізація. Застосовується в мовах *Ada* 95, *C++*, Ейфель.

Метаклас – це клас класів, поняття, що дозволяє поводитися з класами як з об'єктами. Відношення метакласів підтримуються в мовах *SmallTalk* і *CLOS*.

Реалізація визначає відношення, за якого клас-приймач забезпечує власну реалізацію інтерфейсу іншого класу-джерела. Ідеться про спадкування інтерфейсу. Семантично реалізація – це «схрещування» відношень залежності й узагальнення-спеціалізації.

Приклад 8.9. У системі обслуговування читачів є дві ключові абстракції – «книга» і «бібліотека». Клас «книга» відіграє роль елемента, збереженого в бібліотеці, клас «бібліотека» – сховища для книг. Відношення асоціації між класами зображено на рис. 8.9.

Рис. 8.9. Асоціація

Очевидно, що асоціація припускає двобічні відношення:

– для екземпляра «книга» виділяється екземпляр «бібліотека», що забезпечує її зберігання;

– для екземпляра «бібліотека» виділяються всі збережені «книги».

Показано асоціацію один-до-багатьох. Кожний екземпляр «книга» має покажчик на екземпляр «бібліотека». Кожний екземпляр «бібліотека» має набір покажчиків на кілька екземплярів «книга».

Асоціація позначає лише семантичний зв'язок. Вона не вказує напрямок і точну реалізацію відношень. Асоціація придатна для аналізу проблеми, коли потрібно лише ідентифікувати зв'язки. Семантичні зв'язки, їх ролі, потужність (кількість елементів) учасників визначаються за допомогою створення асоціацій.

Асоціація один-до-багатьох, наведена в прикладі, означає, що кожному екземпляру класу «бібліотека» є 0 або більше екземплярів класу «книга», а для кожного екземпляра класу «книга» є один

екземпляр «бібліотека». Множинність означає потужність асоціації. Потужність асоціації буває одною із трьох типів:

– один-до-одного;

– один-до-багатьох;

– багато-до-багатьох.

Приклади асоціацій з різними типами потужності подано у вигляді рис. 8.10:

– персональний комп'ютер має одного власника, а у власника є лише один комп'ютер;

– користувач може працювати з однією мережею, а в мережі може бути задіяно скільки завгодно комп'ютерів;

– користувачів мережею може бути багато, а в мережі може перебувати скільки завгодно серверів.

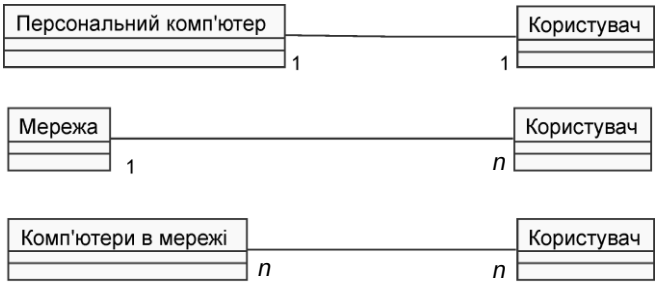


Рис. 8.10. Асоціації з різними типами потужності

Спадкування – це відношення, за якого наслідуваний клас розділяє структуру і поводження, визначені в одному (просте спадкування) або в декількох батьківських (множинне спадкування) класах.

Між *n* класами спадкування визначає ієрархію «є», за якої підклас щось успадковує від одного або декількох більш загальних суперкласів. Підклас є спеціалізацією його батьківського класу (за рахунок доповнення або перевизначення існуючої структури чи поводження).

Приклад 8.10. Організуємо систему запису параметрів польоту в «чорний ящик» літака у вигляді ієрархії класів, що побудована на основі спадкування. Абстракція батьківського класу має вигляд:

модуль *Class*_Параметрипольоту();

{тип Параметрипольоту

private;

function Ініціювати Параметрипольоту;
function Записувати (Параметрипольоту);
function Поточний-час (Параметрипольоту, Бортовийчас);

private
тип Параметрипольоту;
Позначкачасу: Бортовийчас;
};

Запис параметрів кабіни літака може забезпечуватися наступним класом, що є його нащадком:

*Class*_Параметрипольоту; ...
Modul Клас_Кабіна ();
{тип Кабіна Параметрипольоту
private;
function Ініціювати (*P*:Тиск; К:Кисень;Т:Температура);
function Записувати (Кабіна);
function Перепадтиску (Кабіна);
private
тип Кабіна (Параметрипольоту:
параметр1: Тиск;
параметр2: Кисень;
параметр3: Температура)
}.

Цей клас успадковує структуру і поводження класу Параметрипольоту, але нарощує його структуру (вводить три нові елементи даних), перевизначає його поводження (процедура «Записувати») і доповнює його поводження (функція «Перепадтиску»).

Ієрархічну структуру класів системи для запису спадкованих параметрів польоту показано на рис. 8.11.

Тут «Параметрипольоту» – базовий (кореневий) суперклас, підкласами якого є «Екіпаж», «Параметрируху», «Прилади», «Кабіна».

Своєю чергою, клас «Параметрируху» є суперкласом для його підкласів «Координати», «Швидкість», «Прискорення».

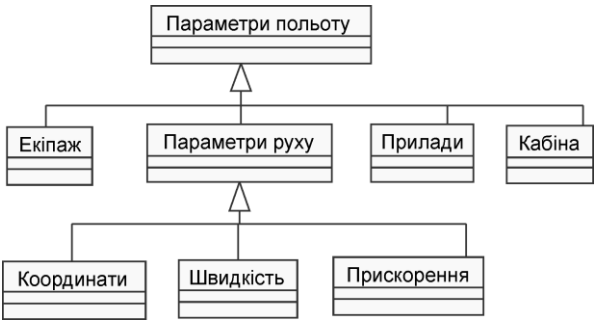


Рис. 8.11. Ієрархія простого спадкування

Поліморфізм – можливість за допомогою одного імені позначати операції з різних класів (але стосовно до загального суперкласу). Виклик обслуговування за поліморфним іменем приводить до виконання одного з деякого набору операцій.

Розглянемо різні реалізації функції «Записувати». Для класу «Параметрипольоту» її реалізація має вигляд

function Записувати (Параметрипольоту)
{
---і записувати ім'я параметра
---і записувати оцінку часу
};
У класі Кабіна передбачена інша її реалізація:
функція Записувати (Кабіна)
{
Записувати (Параметрипольоту ());
---і виклик методу;
---і суперкласу;
---і записувати значення тиску
---і записувати процентний вміст кисню;
---і записувати значення температури;
};
Припустимо, що є по екземпляру кожного із цих двох класів:
Впольоті: Параметрипольоту:= Ініціювати;
Вкабіні: Кабіна:= Ініціювати (768. 21.20);
Припустимо також, що є вільна функція:
функція Зберігатиновдані (*d*: Параметрипольоту; *t*: Бортовийчас)
{

якщо поточнчас(*d*) >= *t*, то

Записувати (*d*): ---і диспетчеризація;

}.

У чому відмінність виконання таких операторів? Кожний з операторів викликає операцію «Записувати» потрібного класу. У першому випадку диспетчеризація зумовить операцію «Записувати» із класу «Параметрипольоту». У другому випадку буде виконуватися операція із класу «Кабіна». Як бачимо, у вільній функції змінна *d* може позначати об'єкти різних класів, отже, тут записано виклик поліморфної операції.

Агрегація. Відношення агрегації між класами аналогічні агрегації між об'єктами. Повторимо приклад з описом класу Контролер кута:

*Class*_Графікрозвороту, Клас_Регуляторкута;

Modul Клас_Контролеркута

тип вкз_нарафік для Графікрозвороту;

тип Контролеркута

private:

function Обробляти (Контролеркута; вкз_нарафік);

function Заплановано (Контролеркута; укз_нарафік, вих: Секунда)

{

private

Контролеркута;

регулятор: Регуляторкута;

…

};

Бачимо, що клас «Контролеркута» є агрегатом, а екземпляр класу «Регуляторкута» – це одна з його частин. Агрегацію тут визначено як включення за величиною. Це – приклад фізичного включення, що означає, що об'єкт-регулятор не існує незалежно від його екземпляра, що включає, «Контролеркута». Час життя цих двох об'єктів нерозривно зв'язаний.

Графічну ілюстрацію відношень агрегації за величиною (композицією) показано на рис. 8.12.

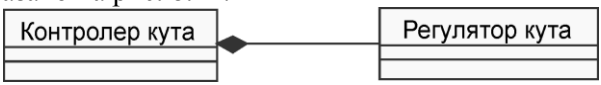


Рис. 8.12. Відношення агрегації за величиною (композиція)

Можливий непрямий тип агрегації – включенням за посиланням. Якщо записати в окремій частині класу Контролеркута:

```
…
{private
        тип вкз_нарегуляторкута для Регуляторкута;
        тип Контролеркута;
                регулятор: вкз_нарегуляторкута;
                …
},
```

то регулятор як частина Контролера буде доступний додатково. Тепер зв'язок об'єктів зменшено. Екземпляри кожного класу створюються і знищуються незалежно.

Ще два приклади агрегації за посиланням та величиною (композиція) ілюструє рис. 8.13. Тут показано клас-агрегат «Мережа» і клас-агрегат «Вікно перегляду» із зазначенням ролей та множинності частин агрегату (відповідні позначки мають лінії відношень). Укладені частини демонструють свою множинність (потужність, кратність) у правому верхньому куті свого символа. Якщо мітка множинності опущена, за замовчуванням вважають, що її значення «багато». Укладений елемент може мати власну роль в агрегаті. Використовується синтаксис – роль : ім'я класу.

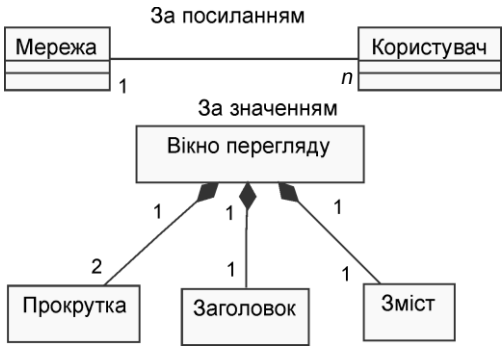


Рис. 8.13. Агрегація класів

Ця роль відповідає тій ролі, яку відіграє частина в неявному (у цій нотації) відношенні композиції між частиною і цілим (агрегатом).

Відзначимо, що властивості (атрибути) класу перебувають відносно композиції між усім класом і його елементами-властивостями. Проте в загальному випадку властивості повинні мати примітивні значення (числа, рядки, дати), а не посилатися на інші класи, оскільки в «атрибутній» нотації не помітні інші відношення класів-частин. Крім того, властивості класів не можуть бути в спільному користуванні декількома класами.

Залежність – це відношення, що показує, яка зміна в одному класі (незалежному) впливає на інший клас (залежний), що використовує його. Графічно залежність зображується пунктирною стрілкою, напрямленою на той клас, від якого він залежить. За допомогою залежності уточнюють, яка абстракція є клієнтом, а яка – постачальником певної послуги. Пунктирна стрілка залежності напрямлена від клієнта до постачальника. Найчастіше залежності показують, що один клас використовує інший клас як аргумент у сигнатурі своєї операції. У попередньому прикладі клас «Графік развороту» з'являється як аргумент у методах «Обробляти» і «Заплановано» класу «Контролер кута». Тому, як показано на рис. 8.14, «Контролер кута» залежить від класу «Графік развороту».

Конкретизація (за М. Бучем) визначається як процес наповнення шаблона батьківського або параметризованого класу [22]. Метою є отримання класу, від якого залежить можливе створення екземплярів.

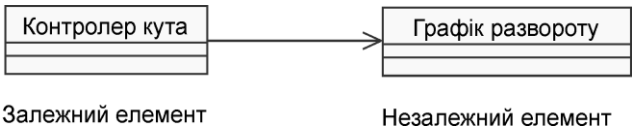


Рис. 8.14. Відношення залежності

Батьківський клас є шаблоном, параметри якого можуть наповнюватися (налаштуватися) іншими класами, типами, об'єктами, операціями. Він може бути родоначальником великої кількості звичайних (конкретних) класів. Можливості налаштування родового класу виражаються списком формальних родових параметрів. Ці параметри в процесі налаштування повинні замінюватися фактичними родовими параметрами. Процес налаштування родового класу називають конкретизацією.

У різних мовах програмування родові класи оформляються порізному. Для мови *Ada* 95, де вперше була реалізована ідея налаштування-параметризації, формальні родові параметри записуються між словом *generic* і заголовком пакета, що розміщує клас.

Приклад 8.11 Подамо родовий (параметризований) клас Черга:

```
generic
    type Елемент is private;
package Клас_Черга is
    type Черга is limited tagged private;
    …
    procedure Додати (В_0черга: in out Черга;
                      елт: Елемент );
…
private
…
end Клас_черга;
```

Один формальний родовий параметр цього класу – тип «елемент». Замість цього параметра можна підставити будь-який тип даних.

Виконаємо налаштування, тобто оголосимо два конкретизовані класи – Чергацілихелементів і Чергапакетів:

```
package Клас_Чергацілихелементів is new Клас_черга
            (Елемент => Integer);
package Клас_Чергапакетів is new Клас_черга
            (Елемент => Пакет);
```

У першому випадку побудовано клас для конкретного типу *Integer* (фактичний родовий параметр), у другому – для конкретного типу «Пакет». Класи «Чергацілихелементів» і «Чергапакетів» можна використовувати як звичайні класи. Вони містять всі засоби родового класу, але лише ці засоби налаштовані на використання конкретного типу, заданого при конкретизації.

Графічну ілюстрацію відношень конкретизації показано на рис. 8.15. Відношення конкретизації відображається за допомогою підписаної стрілки відношення залежності. Це логічно, оскільки конкретизований клас залежить від родового класу (класу-шаблона).

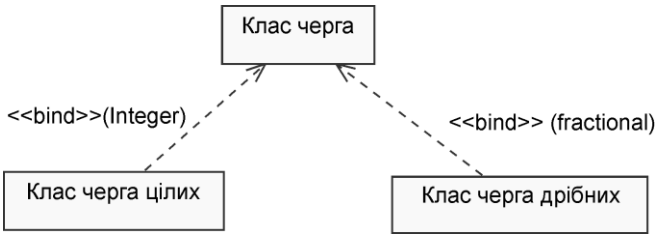


Рис. 8.15. Відношення конкретизації батьківського класу

Контрольні запитання і завдання

1. Чим відрізняється абстрагування від інкапсуляції в об'єктно-орієнтованому програмуванні?
2. Наведіть загальну характеристику об'єкта (літака, ракети, космічного апарата).
3. Що таке поводження об'єкта?
4. Що таке протокол об'єкта?
5. Чим відрізняються активні об'єкти від пасивних?
6. Чим відрізняється об'єкт від класу?
7. Які існують зв'язки між об'єктами?
8. Охарактеризуйте відношення агрегації між об'єктами. Які різновиди агрегації ви знаєте?
9. Які секції існують в інтерфейсній частині класу?
10. Назвіть види відношень між класами.
11. Наведіть приклад асоціації між класами.
12. Наведіть приклад спадкування класів.
13. Наведіть приклад поліморфізму.
14. Наведіть приклад відношень агрегації між класами.
15. Наведіть приклад відношень залежності між класами.

9. МОВА ВІЗУАЛЬНОГО МОДЕЛЮВАННЯ

9.1. Базис мови візуального моделювання

Моделі аналізу і проектування об'єктно-орієнтованих програмних систем створюються мовами візуального моделювання. Натепер розрізняють три покоління мов візуального моделювання. Перше покоління утворювалася 10 мовами, чисельність другого покоління вже перевищила 50 мов. Серед найбільш популярних мов другого покоління визнають мови: Буча (*G. Booch*), Рамбо (*J.*

Rumbaugh), Джекобсона (*I. Jacobson*), Коада-Йордона (*Coad-Yourdon*), Шлеєра-Меллора (*Shlaer-Mellor*) та ін. [17]. Кожна мова має свої засоби вираження, орієнтовані на власний синтаксис і семантику, тобто претендує на роль єдиної і неповторної мови. У результаті розробники і користувачі перестали розуміти один одного. Виникла гостра необхідність уніфікації мов.

Ідея уніфікації привела до появи мов третього покоління. Як стандартну мову третього покоління взято *Unified Modeling Language* (*UML*), що створювався в 1994–1997 рр. Г. Бучем, Дж. Рамбо, І. Джекобсоном.

UML – стандартна мова для написання моделей аналізу, проектування і реалізації об'єктно-орієнтованих програмних систем, яку можна використовувати для візуалізації, специфікації, конструювання і документування результатів програмних проектів. Моделі, що створюються за допомогою цієї мови, транслюються безпосередньо в текст мовами програмування *Java*, *C++*, *Visual Basic*, *Ada* 95, *Object Pascal* і навіть у таблиці для реляційної бази даних, підтримуються технології *COM*, *DDL* (*Data Definition Language*), можуть генеруватися схеми *ORACLE*, *SQL* та здійснювати зворотне проектування. Мова *UML* підтримується багатьма пакетами програмних продуктів, але кращим засобом для проектування, що неодноразово відзначалося різними експертами, є програмний продукт *IBM Rational Rose*.

Словник *UML* утворюють три різновиди структурних блоків: предмети, відношення, діаграми. Предмети – це абстракції, які є основними елементами в моделі, відношення зв'язують ці предмети, діаграми групують колекції предметів.

9.2. Предмети в *UML*

Базовими об'єктно-орієнтованими структурними блоками *UML* є предмети, які використовуються для написання моделей. Є чотири різновиди предметів: структурні, поводження, групові та пояснювальні.

Із погляду синтаксису мови структурні предмети є іменниками в *UML*-моделях. Вони являють собою статичну частину моделі – понятійні або фізичні елементи. Розрізнюють вісім різновидів структурних предметів.

1. Клас – опис множини об'єктів, які мають однакові властивості, операції, відношення і семантику (зміст). Клас реалізовує один або декілька інтерфейсів. Як показано на рис. 9.1, графічно клас зображується у вигляді прямокутника, що має секції з іменем, властивостями (атрибутами) і операціями.

Рис. 9.1. Клас «Оперативна пам'ять»

2. Інтерфейс – набір операцій, які виконує клас або компонент. Інтерфейс описує поводження елемента, видимі ззовні. Інтерфейс може надавати повні послуги класу або компонент або частину таких послуг. Інтерфейс визначає набір лише специфікацій операцій, а не набір реалізацій операцій. Графічно інтерфейс зображується у вигляді кружка з іменем, як показано на рис. 9.2. Ім'я інтерфейсу зазвичай починається з букви «*I*». Інтерфейс рідко показують самостійно. Його приєднують до класу або компонента, що реалізовує інтерфейс.

Рис. 9.2. Інтерфейс

3. Кооперація (співпраця) визначає взаємодію і є сукупністю ролей та інших елементів, які працюють разом для забезпечення колективного поводження більш складного, ніж проста сума всіх елементів. Таким чином, кооперації мають як структурний, так і поведінковий вимір. Конкретний клас може брати участь у декількох кооперація. Ці кооперації виражають реалізацію паттернів (зразків), які формують систему. Графічно кооперація зображується пунктирним еліпсом, поряд записується її ім'я (рис. 9.3).

Рис. 9.3. Кооперація

4. Актор – набір узгоджених ролей, які можуть виконувати користувачі, взаємодіючи із системою (елементами *Use Case*). Кожна роль очікує від системи певного поводження. Актор традиційно зображується дротовим чоловічком з іменем (рис. 9.4).

Рис. 9.4. Актор

5. Елемент Use Case (прецедент) – опис послідовності дій (або декількох послідовностей), виконуваних системою

для окремого актора і для отримання результату. Елемент *Use Case* застосовується для структурування предметів поводження та реалізується кооперацією. На моделях елемент *Use Case* зображується еліпсом, під яким записується назва прецеденту (рис. 9.5).

6. Активним називається клас, об'єкти якого мають один або кілька процесів (або потоків) і тому можуть ініціювати керувальну діяльність. Активний клас подібний до звичайного класу за винятком того, що його об'єкти діють одночасно з об'єктами інших класів. Активний клас зображується прямокутником зі стовщеними стінками.

7. Компонент – фізична і замінна частина системи, що відповідає набору інтерфейсів і забезпечує реалізацію цього набору інтерфейсів. У систему включаються компоненти, що є результатами процесу розроблення (файли вихідного коду), різні різновиди використовуваних компонентів (*COM+*, *Java Beans*). Зазвичай компонент – це фізичне упакування різних логічних елементів (класів, інтерфейсів і кооперацій). Компонент зображується прямокутником із вкладками, усередині міститься його ім'я (рис. 9.6).

Рис. 9.5. Елемент *Use Case*

Рис. 9.6. Компонент

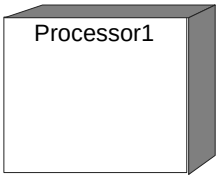


Рис. 9.7. Вузол «бізнес-логіка»

8. Вузол – фізичний елемент, що існує в період роботи системи і надає ресурс, який має пам'ять і можливості оброблення. У вузлі розміщується набір компонентів, які можуть переміщуватися від вузла до вузла. Вузол зображується кубом з іменем (рис. 9.7).

9. Динамічну частину UML-моделей подають у вигляді предметів поводження. Вони є дієсловами моделей, поданням поводження в часі і просторі. Є два основні різновиди предметів поводження: взаємодія і кінцевий автомат.

Взаємодія – поводження, що містить набір повідомлень, якими обмінюються об'єкти у конкретному контексті для досягнення певної мети.

Взаємодія може визначати динаміку як сукупності об'єктів, так і окремої операції. Елементами взаємодії є повідомлення, послідовність дій і зв'язки (з'єднання об'єктів). Повідомлення зображуються у вигляді напрямленої лінії з іменем її операції (рис. 9.8).

Кінцевий автомат – поводження, що визначає послідовність станів об'єкта або взаємодії, виконувані в ході його існування у відповідь на події (з урахуванням обов'язків щодо цих подій). За допомогою кінцевого автомата може визначатися поводження індивідуального класу або кооперації класів. Елементами кінцевого автомата є стани, переходи (від стану до стану), події (предмети, що викликають переходи) і дії (реагування на перехід). Стан зображується прямокутником із закругленими кутами (рис. 9.9), позначається іменем і, якщо є, підстаном.

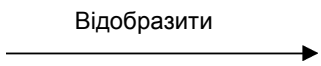


Рис. 9.8. Повідомлення «*display*»

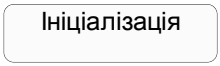


Рис. 9.9. Стан «регулювання»

Взаємодія і кінцевий автомат є базисними предметами поводження, які доповнюють *UML*-моделі для кращого розуміння поведінки проектованої системи. Семантично ці елементи асоціюються з різними структурними елементами (насамперед із класами, операціями і об'єктами).

Груповані предмети – організаційні частини *UML*-моделей. Це ящики, по яких може бути розкладена модель. Передбачено лише один різновид предмета, що групує, – пакет. *Пакет* – загальний механізм для розподілу елементів по групах. У пакет можуть міститися структурні предмети, предмети поводження і навіть інші угруповання предметів. На відміну від компонента (який існує в період виконання), пакет – суто концептуальне поняття. Це означає, що пакет існує лише в період розроблення. Пакет зображується папкою із закладкою, на якій позначено його ім'я та його зміст (рис. 9.10).

Пояснювальні предмети – частини *UML*-моделей, що призначені для роз'яснення особливостей моделі. Вони є зауваженнями, які можна застосовувати для опису, пояснення, і коментарів до будь-якого елемента моделі. Передбачено лише один різновид пояснювального предмета – примітка.

Примітка – символ для відображення обмежень і зауважень, що приєднуються до елемента або сукупності елементів. Зображується у вигляді прямокутника із загнутим кутом, у який вписується текстовий або графічний коментар (рис. 9.11).

Рис. 9.10. Пакет «обчислювальні засоби»

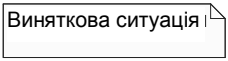


Рис. 9.11. Примітка

9.3. Відношення *UML*

Базовими структурними блоками, що використовуються для створення *UML*-моделей, є відношення. Розрізнюють чотири їх типи.

1. *Залежність* – семантичне відношення між двома предметами, у якому зміна в одному предметі (незалежному) може впливати на семантику іншого предмета (залежного). Залежність зображується у вигляді пунктирної лінії, напрямленої на незалежний предмет, і іноді має мітку (рис. 9.12).

2. *Асоціація* – структурне відношення, яке описує набір зв'язків, що з'єднує об'єкти. Агрегація – це спеціальний різновид асоціації, що являє собою структурне відношення між цілим і його частинами. Асоціацію зображують у вигляді суцільної лінії, яка іноді має мітку і часто включає інші пояснювальні доповнення, такі як потужність та імена ролей (рис. 9.13).

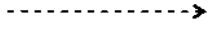


Рис. 9.12. Залежність

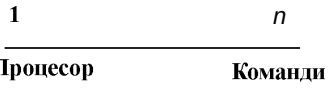


Рис. 9.13. Відношення асоціації між замовником та замовленнями, які він виконує

3. *Узагальнення* – відношення спеціалізації–узагальнення, у якому об'єкти спеціалізованого елемента (нащадка, дочки) можуть замінювати об'єкти узагальненого елемента (предка, батька). Інакше кажучи, нащадок розділяє структуру і поводження батька. Узагальнення зображують у вигляді суцільної стрілки з порожнім нако-нечником, що вказує на батька (рис. 9.14).

4. *Реалізація* – семантичне відношення між класифікаторами, де один класифікатор визначає завдання, що інший класифікатор зобов'язується виконувати (до класифікаторів належать класи, інтерфейси, компоненти, елементи *Use Case*, кооперації). Відношення реалізації застосовують у двох випадках: між інтерфейсами і класами (або компонентами), що реалізовують їх; між елементами *Use Case* і кооперациями, які реалізовують їх. Реалізація зображується як середнє між узагальненням і залежністю (рис. 9.15).

Рис. 9.14. Відношення узагальнення

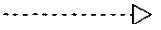


Рис. 9.15. Зовнішній вигляд відношення реалізації

9.4. Діаграми в *UML*

Діаграма – графічне зображення множини елементів, які мають вигляд зв'язного графу з вершин (предметів) і дуг (відношень). Їх набір візуалізує систему з різних точок зору, потім вони відображаються в програмну систему. Зазвичай діаграма дає неповне подання елементів, які складають систему. Хоча один і той самий елемент може з'являтися у всіх діаграмах, на практиці він з'являється лише в деяких діаграмах. Теоретично діаграма може містити будь-яку комбінацію предметів і відношень, на практиці обмежуються малою кількістю комбінацій, які відповідають п'ятьом поданням архітектури програмної системи. Із цієї причини *UML* включає десять видів діаграм:

1. *Діаграма класів* – набір класів, інтерфейсів, кооперацій і їх відношень. Під час моделювання об'єктно-орієнтованих систем діаграми класів використовуються найчастіше. Діаграми класів забезпечують статичне проектне подання системи. Діаграми класів, що включають активні класи, забезпечують статичне подання процесів системи.

2. *Діаграма об'єктів* – набір об'єктів і їх відношення. Діаграма об'єктів подає статичний «моментальний знімок» з екземплярів предметів, які перебувають у діаграмах класів. Як і діаграми класів, ці діаграми забезпечують статичне проектне подання або статичне подання процесів системи (але з погляду реальних або фототипових випадків).

3. *Діаграма Use Case* (прецедентів) – набір елементів *Use Case*, акторів і їх відношень. За допомогою діаграм *Use Case* для системи створюється статичне подання *Use Case*. Ці діаграми важливі для організації і моделювання поводження системи, задання вимог замовника до системи.

4. *Діаграма взаємодії* – взаємодія, що включає набір об'єктів і їх відношень, і повідомлення, що пересилаються між об'єктами. Ці діаграми забезпечують динамічне подання системи.

5. *Діаграма послідовності* – це діаграма взаємодії, яка впорядковує повідомлення за часом.

6. *Діаграма кооперації* – діаграма взаємодії, що виділяє структурну організацію об'єктів, що посилає і приймає повідомлення. Діаграми послідовності та діаграми кооперації ізоморфні. Це означає, що одну діаграму можна трансформувати в іншу діаграму. Це різновиди діаграм взаємодії.

7. *Діаграма схем станів* – кінцевий автомат; містить стани, переходи, події і дії. Ці діаграми забезпечують динамічне подання системи. Вони важливі для моделювання поводження інтерфейсу, класу або кооперації. Діаграми схем станів виділяють таке поводження об'єкта, яке керується подіями, що особливо корисно для моделювання реактивних систем.

8. *Діаграма діяльності* – спеціальний різновид діаграми схем станів, що показує потік від дії до дії всередині системи. Ці діаграми забезпечують динамічне подання системи. Вони важливі для моделювання функціональності системи і виділяють потік керування між об'єктами.

9. *Компонентна діаграма* – організація набору компонентів і залежності між компонентами. Ці діаграми забезпечують статичне подання реалізації системи. Вони пов'язані з діаграмами класів, а саме: у компонент відображається один або кілька класів, інтерфейсів або кооперацій.

10. *Діаграма розміщення* (розгортання) – конфігурація обробних вузлів періоду виконання, а також компоненти, що в них розміщені. Ці діаграми забезпечують статичне подання розміщення системи. Вони пов'язані з компонентними діаграмами, а саме: вузол включає один або кілька компонентів.

9.5. Механізми розширення в *UML*

Оскільки *UML* – розвинута мова, яка має великі можливості, тому для відображення деяких особливостей, які можуть виникнути під час створення різних моделей; вона припускає контрольовані розширення. Механізмами розширення в *UML* є обмеження, тегові величини, стереотипи.

Обмеження розширюють семантику структурного *UML*-блока, дозволяючи додати нові правила або існуючі модифікувати. Обмеження показуються у вигляді текстового рядка, взятого у фігурні дужки {}. Наприклад, на рис. 9.16 вводиться просте обмеження на атрибут «результат класу» класу «цикл розрахунку». Тоді обмеження значення результату буде мати два знаки після коми. Крім того, тут показано обмеження на два елементи (дві асоціації); воно розміщується біля пунктирної лінії, що сполучає елементи, і має такий зміст: власником конкретного рахунка не може бути ні організація, ні персона.

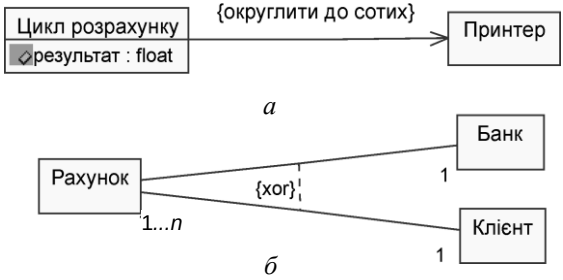


Рис. 9.16. Приклади обмежень на моделях:
а – просте обмеження; *б* – обмеження на декілька елементів

Тегова величина розширює характеристики будівельного *UML*-блока, дозволяючи створити нову інформацію у специфікації конкретного елемента. Її показують рядком у фігурних дужках {}. Рядок має вигляд: ім'я тегової величини = значення. Іноді (у випадку визначених тегів) вказується лише ім'я тегової величини.

Приклад 9.1. Клас «редактор формул» може бути розширений відповідно до вказівки на його версію й автора шляхом використання тегів. Це дозволяє відслідковувати версію й автора певних блоків (рис. 9.17).

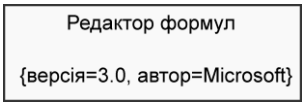


Рис. 9.17. Застосування розширення класу

Вони можуть бути додані до будь-якого структурного блока (наприклад, до класу) уведенням нових тегових значень.

Стереотип (*stereotype*) розширює словник мови, дозволяє створювати нові види структурних блоків, похідні від існуючих, і враховувати специфіку нової проблеми. Елемент зі стереотипом є варіацією існуючого елемента, що має таку саму форму, але відрізняється по суті. У нього можуть бути додаткові обмеження і тегові величини, а також інше візуальне подання. Він інакше обробляється при генерації програмного коду. Відображають стереотип іменем, яке вказується в подвійних кутових дужках (або в кутових лапках).

Приклади елементів зі стереотипами показано на рис. 9.18. Клас «Втрата значущості» зі стереотипом «*exception*» тепер розглядається як спеціальний клас, якому дозволяється лише генерація й оброблення сигналів виключень. Особливі можливості метакласу отримав клас «опис». Застосування стереотипу «*call*» до відношення залежності вносить у нього новий зміст.

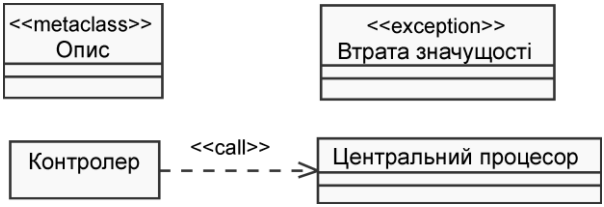


Рис. 9.18. Застосування стереотипів

Таким чином, механізми розширення дозволяють адаптувати *UML* під потреби конкретних проектів і під нові програмні технології. Можливе додавання нових структурних блоків, модифікація специфікацій існуючих блоків і навіть зміна їх семантики. Дуже важливо забезпечити контрольоване введення розширень.

9.6. Статичні моделі об’єктно-орієнтованих програмних систем

Статичні моделі забезпечують подання структури систем у термінах базових структурних блоків і їх відношень. Статичність цих

моделей полягає в тому, що в них не показується динаміка змін системи в часі. Разом з тим варто розуміти, що ці моделі містять не лише структурні описи, але й опис операцій, що реалізовують задане поводження системи. Основним засобом для подання статичних моделей є діаграми класів. Вершини діаграм класів навантажені класами, а дуги (ребра) – їх відношенням. Діаграми використовуються:

– під час аналізу для указання ролей і обов'язків сутностей, які забезпечують поводження системи;

– у ході проектування для фіксації структури класів, які формують системну архітектуру.

Вершини в діаграмах класів – клас. Ім'я класу вказується завжди, властивості й операції – вибірково. Передбачено задання області дії властивості (операції). Якщо властивість підкреслюється, її областю дії є клас, у противному випадку областю дії є екземпляр (рис. 9.19). У цьому випадку всі його екземпляри (об'єкти) використовують загальне значення цієї властивості, у противному випадку кожний екземпляр має своє значення властивості.

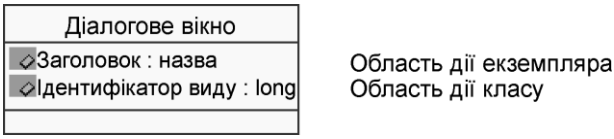


Рис. 9.19. Властивості рівнів класу та екземпляра

Властивості. Загальний синтаксис подання властивості має вигляд

Видимість Ім'я [Множинність]: Тип = Початкове значення {Характеристики}

Мова *UML* має три рівні видимості:

public – властивість (операція) класу, яку може використовувати будь-який клієнт;

protected – властивість (операція), яку може використовувати будь-який нащадок класу;

private – властивість (операція), яку використовує сам клас.

Якщо видимість не зазначено, вважають, що властивість оголошено з видимістю *public*.

Властивості мають три характеристики:

changeable – немає обмежень на модифікацію значення властивості;

addOnly – для властивостей із множинністю, більшою від одиниці; додаткові значення можуть бути додані, але після створення значення не може віддалятися або змінюватися;

frozen – після ініціалізації об'єкта значення властивості не змінюється.

Якщо характеристика не зазначена, вважають, що властивість оголошена з характеристикою *changeable*.

Приклад 9.2. Оголошення властивостей:

1) друк – лише ім'я;

2) + друк – видимість і ім'я;

3) початок: координати – ім'я і тип;

4) ім'я прізвище [0..1]: *String* – оголошені ім'я, множинність, тип;

5) лівий кут: координати=(0, 10) – оголошені ім'я, тип, початкове значення;

6) сума: *Integer* {*frozen*} – ім'я і характеристика.

Операції. Загальний синтаксис подання операції має вигляд

Видимість Ім'я (Список Параметрів): тип, що повертається {Характеристики}

Приклад 9.3. Оголошення операцій:

записати – лише ім'я;

+ записати – оглошені видимість та ім'я;

зареєструвати (і: Ім'я, п: Прізвище) – ім'я і параметри;

баланс рахунка ( ) : *Integer* – ім'я і тип, що повертається;

нагрівати ( ) (*guarded*) – ім'я і характеристика.

У сигнатурі операції можна вказати нуль або більше параметрів, форма подання параметра має такий синтаксис:

Напрямок Ім'я : Тип = Значення за замовчуванням

Елемент Напрямок може набувати одне з таких значень:

in – вхідний параметр, не може модифікуватися;

out – вихідний параметр, може модифікуватися для передавання інформації об'єкта;

inout – вхідний параметр, може модифікуватися.

Припустимо застосування таких характеристик операцій:

leaf – кінцева операція, операція не може бути поліморфною і не може перевизначатися (у ланцюжку спадкування);

isQuery – виконання операції не змінює стани об'єкта;

sequential – у кожний момент часу в об'єкт надходить лише один виклик операцій. Як наслідок у кожний момент часу виконується лише одна операція об'єкта. Інакше кажучи, допустимий тільки один потік викликів (потік керування);

guarded – припускається одночасне надходження в об'єкт декількох викликів, але в кожний момент часу обробляється лише один виклик охоронюваної операції. Тобто паралельні потоки керування виконуються послідовно (за рахунок розміщення викликів у чергу);

concurrent – в об'єкт надходить кілька потоків викликів операцій (з паралельних потоків керування). Дозволяється одночасне (і множинне) виконання операції. Мається на увазі, що такі операції є атомарними.

Організація властивостей і операцій. Відомо, що піктограма класу включає три секції (ім'я, властивості й операції). Порожня секція не означає, що клас не має властивостей або операцій, просто в цей момент вони не вказуються. Можна явно визначити наявність у класі великої кількості властивостей або атрибутів. Для цього в кінці показаного списку ставляться три крапки. Як показано на рис. 9.20, у довгих списках властивостей і операцій дозволяється угруповання – кожна група починається зі свого стереотипу.

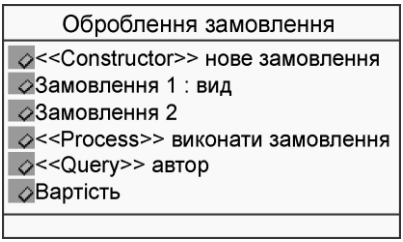


Рис. 9.20. Стереотипи для характеристик класу

Асоціації відображають структурні відношення між екземплярами класів, тобто з'єднання об'єктів. Кожна асоціація може мати мітку – *ім'я*, що описує природу відношень. Як показано на рис. 9.21, імені можна додати напрямок.

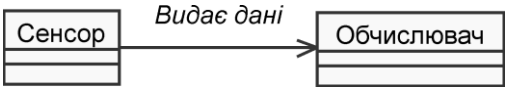


Рис. 9.21. Асоціація з іменем

Коли клас бере участь в асоціації, він відіграє щодо неї певну роль. Роль визначає, як подається клас на одному кінці асоціації відносно класу, що міститься на протилежному кінці асоціації (рис. 9.22).

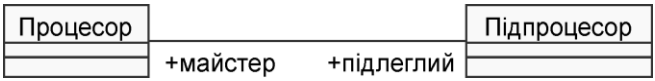


Рис. 9.22. Ролі класів

Один і той самий клас у різних асоціаціях може відігравати різні ролі. Часто важливо знати, як багато об'єктів може з'єднуватися через екземпляр асоціації. Ця кількість називається хибністю ролі в асоціації, записується у вигляді виразу, що задає діапазон величин або одну величину (рис. 9.23).

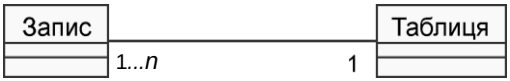


Рис. 9.23. Потужність

Запис потужності на одному кінці асоціації визначає кількість об'єктів, що з'єднуються з кожним об'єктом на протилежному кінці асоціації. Наприклад, можна задати такі варіанти потужності:

5 – точно п'ять;

необмежена кількість;

0..* – нуль або більше;

1..* – один або більше;

3..7 – певний діапазон;

1..3, 7 – певний діапазон або число.

Досить часто виникає така проблема – як для об'єкта на одному кінці асоціації виділити набір об'єктів на протилежному кінці. Наприклад, розглянемо взаємодію між банком і клієнтом — вкладником. Як показано на рис. 9.24, установлюється асоціація між кла-

сом «банк» і класом «клієнт». «Банк» ідентифікує конкретного «клієнта» за «номером рахунка». Тоді «номер рахунка» є атрибутом асоціації. Він не є характеристикою «клієнта», оскільки «клієнту» не обов'язково знати службові параметри його рахунка. Тепер для екземпляра «банку» і значення «номера рахунка» можна виявити нуль або один екземпляр «клієнта». В *UML* для вирішення цієї проблеми вводиться кваліфікатор – атрибут асоціації, значення якого виділяють набір об'єктів, пов'язаних з об'єктом через асоціацію. Кваліфікатор зображується малим прямокутником, з'єднаним з кінцем асоціації. У прямокутник уписується властивість – атрибут асоціації.

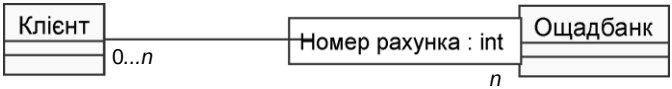


Рис. 9.24. Застосування кваліфікатора

Крім того, ролі в асоціаціях можуть мати позначки *видимості.* На кінці асоціації можна задати три рівні видимості, додаючи символ видимості до ім'я ролі:

– за замовчуванням для ролі задається публічна видимість (*public*);

– видимість (*private*) указує, що об'єкти недоступні для будь-яких об'єктів поза асоціацією;

– захищена видимість (*protected*) указує, що об'єкти недоступні для будь-яких об'єктів поза асоціацією, за винятком нащадків того класу, що зазначений на протилежному кінці асоціації.

Асоціації у мові *UML* можуть мати властивості. Як показано на рис. 9.25, такі можливості відображаються за допомогою класів-асоціацій. Ці класи сполучаються з лінією асоціації пунктирною лінією і розглядаються як класи із властивостями асоціацій або як асоціації із властивостями класів. Властивості класу-асоціації характеризують не один, а пари об'єктів; у цьому випадку – пара екземплярів, «лікар» і «клініка».

Відношення агрегації і композиції в *UML* уважаються різновидами асоціації, застосовуваними для відображення структурних відношень між «цілим» (агрегатом) і його «частинами». Агрегація показує відношення за посиланням (в агрегат включені лише покажчики на частині), композиція – відношення фізичного включення (в агрегат включені самі частини).

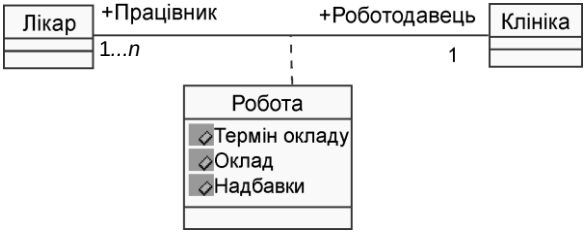


Рис. 9.25. Клас-асоціація

Залежність є відношенням між клієнтом (залежним елементом) і постачальником (незалежним елементом). Операції клієнта:

– викликають операції постачальника;

– мають сигнатури, що належать класу постачальника, повертаються у значення або аргументи.

Наприклад, на рис. 9.26 показано залежність класу «Замовлення» від класу «Книга», оскільки «Книга» використовується в операціях «Опрацювати замовлення», «Створити» і «Видалити» класу «Замовлення», а також залежність, яка показує, що клас «перегляд замовлення» використовує клас «Замовлення», який не має відомостей про «Перегляд замовлення»; залежність позначено стереотипом «*friend permission*», що розширює просту залежність певною мовою. Відзначимо, що відношення залежності дуже різноманітне. Натепер мова передбачає 17 різновидів залежності, що розрізняються стереотипами.

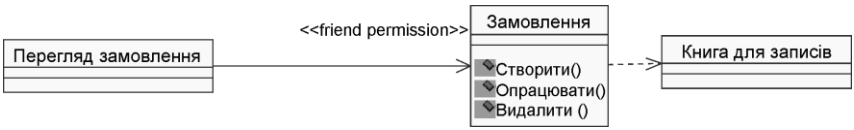


Рис. 9.26. Відношення залежності

Узагальнення – відношення між загальним предметом (батьківським класом) і спеціалізованим різновидом цього предмета (підкласом). Підклас може мати одного батька (один батьківський клас) або кілька батьків (батьківських класів) – множинне спадкування (рис. 9.27).

Множинне спадкування складне і небезпечне, має багато «підводних каменів». Наприклад, підклас «Літак-винищувач» не слід робити від батьківських класів «Літак» і «Ракета». Це типове неправильне використання множинного спадкування: нащадок успа-

дковує всі властивості від його батька, попри те, що не всі властивості застосовні до нащадка. Очевидно, що «Літак-винищувач» є «Літаком», але не є «Ракетою», тому що ракета має обмежений ресурс.

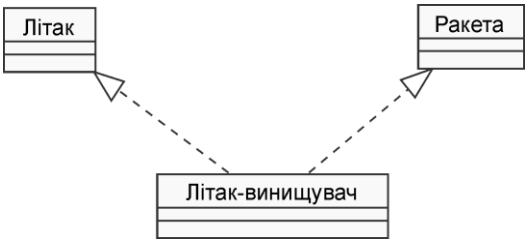


Рис. 9.27. Множинне спадкування

Ще більш складне спадкування від двох класів, що мають загального батька. У результаті утворяться ромбоподібні ґратки спадкування (рис. 9.28).

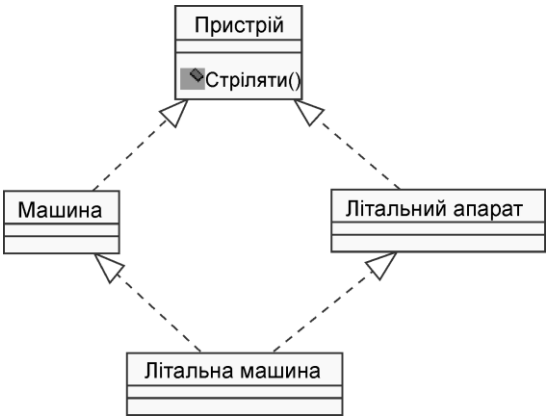


Рис. 9.28. Ромбоподібні ґратки спадкування

У підкласах «Машина» і «Літальний апарат» операція «стріляти» суперкласу «Пристрій» перевизначена відповідно до обов'язку підкласу («Машини» їздять, а «Літальний апарат» – літає). Чи буде наслідувати операцію «стріляти» «літальна машина»? А що робити із властивостями, що дісталися в спадщину від батьків і загального прабатька? Чи матимемо декілька копій властивості, чи лише одну?

Усі ці проблеми ускладнюють реалізацію, призводять до введення множинних правил для оброблення особливих випадків.

Реалізація – семантичне відношення між класами, у якому клас-приймач виконує реалізацію операцій інтерфейсу класу-джерела. Як показано на рис. 9.29, клас «Аудіопрогравач» повинен реалізовувати інтерфейс «Обробник функцій», тобто «Обробник функцій» розглядається як джерело, а «Аудіопрогравач» – як приймач.

Рис. 9.29. Реалізація інтерфейсу

Інтерфейс «Обробник функцій» дозволяє клієнтам взаємодіяти з об'єктами класу «Аудіопрогравач» без знання тієї дисципліни доступу, яку тут реалізовану (*LIFO* – останній увійшов, перший вийшов; *FIFO* – перший увійшов, перший вийшов і т.д.).

Дерева спадкування. Для використання відношень узагальнення будується ієрархія класів. Деякі класи в цій ієрархії можуть бути абстрактними. *Абстрактним* називають клас, що не має екземплярів. Імена абстрактних класів зображуються курсивом. Наприклад, на рис. 9.30 показано абстрактні класи «Літальні засоби», «Цивільні», «Ракети».

Крім того, тут є конкретні класи «Винищувач», «Міг-29», кожний з яких може мати екземпляри. Зазвичай клас успадковує деякі характеристики класу-батька і передає свої характеристики класу-нащадка. Іноді потрібно визначити кінцевий клас, що не може мати дітей. Наприклад, на рис. 9.30 показано кінцевий клас «Міг-29».

Аналогічні властивості мають і операції. Операція є поліморфною, це означає, що в різних точках ієрархії можна визначати операції зі схожою сигнатурою. Такі операції з дочірніх класів перевизначають поводження відповідних операцій з батьківських класів.

Під час оброблення повідомлення (у період виконання) здійснюється поліморфний вибір однієї з операцій ієрархії відповідно до типу об'єкта. Наприклад, «Встановити тип ()» і «Перевезти вантаж ()» – поліморфні операції. До того ж операція «Літальні засоби:

призначення ()» є абстрактною, тобто неповною і потрібною для реалізації нащадка.

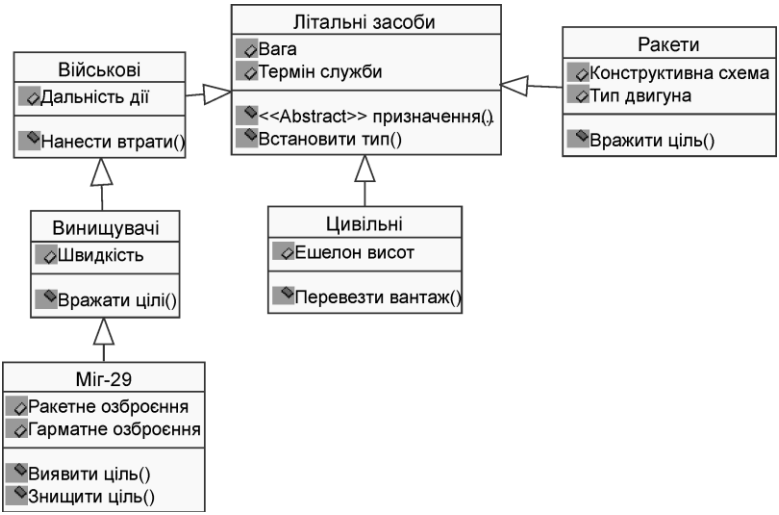


Рис. 9.30. Абстрактність і поліморфізм

9.7. Діаграма класів в *UML*

Розглянемо діаграму класів системи керування польотом літального апарата (рис. 9.31).

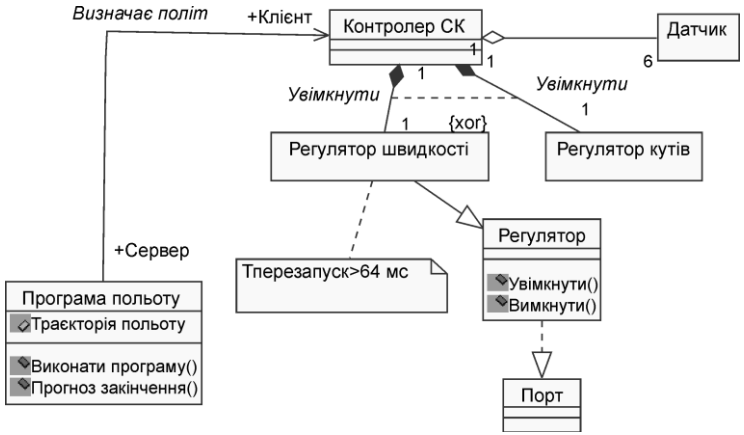


Рис. 9.31. Діаграма класів системи керування польотом

Клас «Програма польоту» має властивість «Траєкторія польоту», операцію-модифікатор «Виконувати програму ()» і операцію-селектор «Прогнозувати закінчення ()». Асоціація між цим класом і класом «Контролер СК» – екземпляри програми задають параметри руху, які повинні забезпечувати екземпляри контролера.

Клас «Контролер СК» – агрегат, що містить по одному екземпляру класів «Регулятор швидкості» і «Регулятор кутів», а також шість екземплярів класу «Датчик». Екземпляри «Регулятор швидкості» і «Регулятор кутів» включені в агрегат фізично (за допомогою відношення композиції), а екземпляри «Датчик» – за посиланням, тобто екземпляр «Контролер СК» включає лише покажчики на об'єкти-датчики. Абстрактний суперклас «Регулятор» передає підкласам «регулятор швидкості» і «Регулятор кутів» у спадщину абстрактні операції «Увімкнути ( )» і «Вимкнути ()». Клас «Регулятор» використовує конкретний клас «Порт».

Асоціацію на діаграмі позначено іменем («Визначає політ»), ролі учасників асоціації зазначено («Сервер», «Клієнт»). Відношення композиції також супроводжуються іменем («Увімкнути»), причому на ці відношення накладено обмеження – контролер не може вмикати «Регулятор швидкості» і «Регулятор кутів» одночасно.

На клас «Контролер СК» встановлене обмеження на множинність – припускається не більше трьох екземплярів цього класу. Клас «Регулятор швидкості» має обмеження іншого типу – повторне включення його екземпляра дозволяється не раніше, ніж через 64 мс.

Контрольні запитання і завдання

1. Яке покоління мов візуального моделювання має найбільшу чисельність?
2. Назвіть правильне призначення *UML*: програмування, моделювання, розрахунки.
3. Чим візуально відрізняються клас, стан та діяльність?
4. Які предмети *UML* призначені для відображення динаміки моделі, а які показують стан моделі в поточний момент часу?
5. Які предмети поводження існують в *UML*?
6. Предмети, що пояснюють, в *UML*, призначені для опису роботи, специфікування чи для коментарів?
7. Чим візуально на діаграмах «відношення» «асоціація» відрізняється від відношень «агрегація», «композиція» та «реалізація»?

8. На діаграмі *Use Case* відображаються класи, діяльності, компоненти чи прецеденти?
9. Як можна позначити версію та автора моделі на діаграмі класів?
10. Як позначають обмеження на використання 50-гривневих банкнот у моделі видачі коштів банкомата?
11. Яку діаграму доцільно використати для моделювання часового графіка старту польоту ракети?
12. Що означає відсутність пояснень на окремих секціях графічного позначення класу?
13. Як указати сферу дії властивості (операції)?
14. Як обмежити кількість екземплярів класу?
15. Чим відрізняється агрегація від композиції? Різновидами якого відношення (в *UML*) вони є?

10. ДИНАМІЧНІ МОДЕЛІ ОБ'ЄКТНО-ОРІЄНТОВАНИХ ПРОГРАМНИХ СИСТЕМ

10.1. Моделювання поводження програмних систем

Динамічні моделі забезпечують подання поводження систем, яке полягає в зміні станів у часі у процесі роботи системи. Мова *UML* для цього має різноманітні засоби, що орієнтовані не лише на програмні системи, але і на відображення вимог замовника до поводження таких систем.

Для моделювання поводження системи використовують: автомати і взаємодії. Автомат описує поводження в термінах послідовності станів, через які проходить об'єкт протягом життя. Взаємодія описує поводження в термінах обміну повідомленнями між об'єктами.

Таким чином, автомат задає поводження системи як цільної, єдиної сутності; моделює життєвий цикл єдиного об'єкта. Тому автоматний підхід зручно застосовувати для формалізації динаміки окремого, важкого для розуміння блока системи.

Взаємодії визначають поводження системи у вигляді комунікацій між його частинами (об'єктами), подаючи систему спільно працюючими об'єктами. Саме тому взаємодії вважають основним апаратом для фіксації повної динаміки системи.

Автомати відображають за допомогою діаграм схем станів та діаграм діяльності, а взаємодії – за допомогою діаграм співпраці (кооперації) і діаграм послідовності.

10.2. Діаграми схем станів

Діаграма схем станів відображає кінцевий автомат, виділяючи потік керування від стану до стану. Кінцевий автомат – поводження, що визначає послідовність станів у ході існування об'єкта. Їх послідовність розглядається як відповідь на події і включає реакції на ці події.

Діаграма схем станів показує:

1) набір станів системи;

2) події, які викликають перехід з одного стану в інший;

3) дії, які відбуваються в результаті зміни стану.

У мові *UML* станом називають період життя об'єкта, протягом якого він задовольняє певну умову, провадить певну діяльність або очікує деяку подію. Події зумовлюють переходи, а дії є реакціями на переходи.

Приклади події: баланс < 0 – зміна стану; перешкоди – сигнал (об'єкт з іменем); тиск – виклик дії; *after* (5 *seconds*) – плинність часу; *when* (*time* = 16:30) – настання абсолютного моменту часу.

Приклади дії: зупинити об'єкт ( ) – виклик однієї операції; *flt*:= *new*(фільтр); оцінити якість ( ) – виклик двох операцій; *send* Регулятор.стоп – надсилання сигналу «стоп» в об'єкт Регулятор.

Для відображення надсилання сигналу використовують спеціальне службове слово «*send*», а для відображення переходу в початковий і кінцевий стани використовують позначення, показане на рис. 10.1, 10.2.

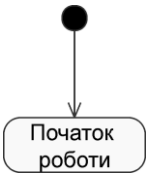


Рис. 10.1. Перехід
у початковий стан

Рис. 10.2. Перехід
у кінцевий стан

Діаграму схем станів для системи охоронної сигналізації показано на рис. 10.3. Початковий стан для системи «Ініціалізація», потім переходить у стан «Очікування». У цьому стані через кожні

10 с (подія «*after* 10 *с*») виконується операція «Самоперевірка». Із настанням події «Тривога» реалізуються дії, пов'язані із блокуванням периметра охоронного об'єкта, – виконується операція «Блокувати периметр» і здійснюється перехід у стан «Активний». В активному стані кожні 5 с за подією «*after* 5 *с*» запускається операція «Очікування команди». Якщо команду отримано (настає подія «Скинути тривогу»), система повертається в стан «Очікування». У процесі повернення периметр охоронного об'єкта розблокується (операція «Розблокувати периметр»).

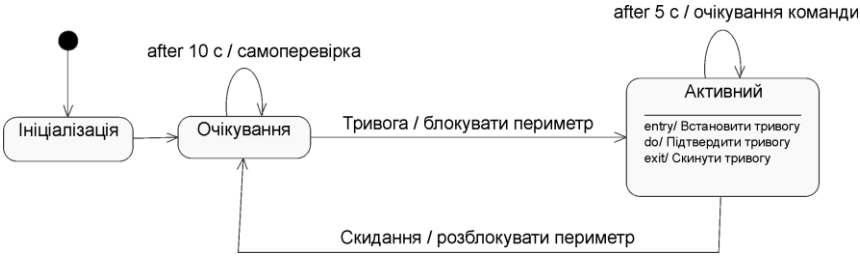


Рис. 10.3. Діаграма схем станів системи охоронної сигналізації

Дії станів позначаються спеціальними мітками. Так, для входження в стан і виходу зі стану використовуються мітки «*entry*» і «*exit*» відповідно. Під час входження в стан «Активна» виконується операція «Установити тривогу» із класу «Контролер», а для виходу зі стану – операція «Скинути тривогу».

Мітка «*do*» вказує дію, яка повинна виконуватися, коли система перебуває в поточному стані. Така дія починається із входження у стан і закінчується із виходом з нього. Наприклад, у стані «Активний» – це дія «Підтвердити тривогу».

Між станами можливі різні типи переходів. Зазвичай перехід ініційований подією. Допускаються переходи і без подій. Дозволено умовні або охоронні переходи. Правила позначення стрілок умовних переходів ілюструє рис. 10.4.

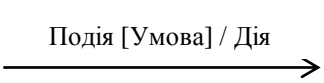


Рис. 10.4. Позначення умовного переходу

Послідовність умовного переходу:

1) відбувається подія;

2) обчислюється «Умова» переходу;

3) якщо дійсна «Умова переходу = *true*» запускається перехід і активізується дія, у противному випадку перехід не виконується.

Приклад умовного переходу показано на рис. 10.5 (перехід радара в режим пошуку за подією «Живлення подано»). Перехід здійснюється між станами «Умикання» і «Вимірювання».

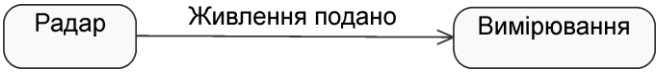


Рис. 10.5. Умовний перехід між станами

Однією з найбільш важливих характеристик кінцевих автоматів в *UML* є підстан, який значно спрощує моделювання складного поводження. Підстан – це стан, укладений в інший стан. Складний стан, що містить у собі два підстани, ілюструє рис. 10.6.

Семантика вкладеності така: якщо система перебуває в стані «Активний», то він має бути точно в одному з підстанів: «Перевірка», «Сигнал», «Очікування». У підстан можуть вкладатися інші підстани. Ступінь вкладеності підстанів не обмежується. Така семантика відповідає випадку послідовних підстанів.

Можлива наявність паралельних підстанів, які виконуються одночасно всередині складеного стану. Графічно зображення паралельних підстанів відділяються один від одного пунктирними лініями.

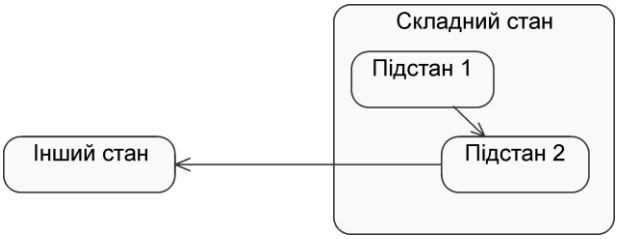


Рис. 10.6. Позначення підстанів

Внутрішню структуру складного стану «Активна» показано на рис. 10.7.

Іноді під час повернення в складний стан виникає необхідність потрапити в той його підстан, що минулого разу був останнім. Такий підстан називають історичним. Інформація про історичний

стан запам’ятовується. Застосування історичного підстану показано на рис. 10.8, який відображається літерою Н усередині кружка.

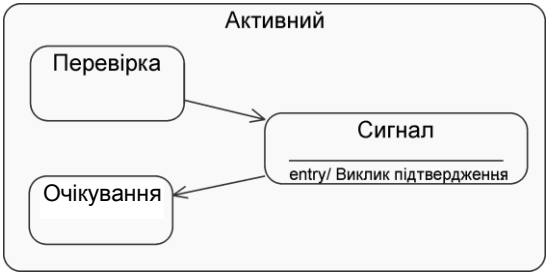


Рис. 10.7. Переходи в стані «Активний»

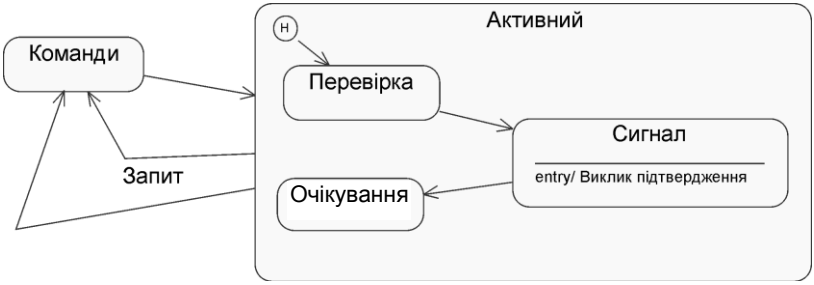


Рис. 10.8. Історичний стан

Під час першого відвідування стану «Активний» автомат не має історії, тому відбувається простий перехід у підстан «Перевірка». Припустимо, що в підстані «Сигнал» відбулася подія «Запит». Засоби керування змушують автомат покинути підстан «Сигнал» (стан «Активний») і повернутися у стан «Команди». Коли робота в стані «команди» завершується, виконується повернення в історичний підстан стану «Активний». Оскільки тепер автомат запам’ятав історію, він переходить безпосередньо в підстан «Сигнал» (минаючи підстан «Перевірка»).

10.3. Діаграми діяльності

Діаграма діяльності є особливою формою кінцевого автомата, у якому показується процес обчислень і потоки робіт. У ній виділяються не звичайні стани об’єкта, а стани виконуваних обчислень –

стани дій. При цьому процес обчислень не переривається зовнішніми подіями. Діаграми діяльності схожі на блок-схеми алгоритмів.

Стан дії вважається атомарним (дію не можна перервати) і виконується за один квант часу, його не можна піддати декомпозиції. Якщо потрібно виразити складну дію, яку можна надалі декомпозитувати (розбити на ряд більш простих дій), то використовують стан піддіяльності. Зображення стану піддіяльності містить піктограму в правому нижньому куті. У вершину вписується ім'я іншої діаграми, що має внутрішню структуру. Переходи між станами дій виконуються по закінченні дій, вони зображуються стрілками. Крім станів, у діаграмах діяльності використовуються допоміжні вершини:

– рішення (ромбик з однією вхідною і декількома вихідними стрілками);

– об'єднання (ромбик з декількома вхідними і однією вихідною стрілками);

– лінійка синхронізації – поділ (жирна горизонтальна лінія з однією вхідною і декількома вихідними стрілками);

– лінійка синхронізації – злиття (жирна горизонтальна лінія з декількома вхідними і однією вихідною стрілками);

– початковий стан (чорний кружок);

– кінцевий стан (незафарбований кружок, у якому розміщений чорний кружок меншого розміру).

Вершина «Рішення» дозволяє відобразити розгалуження обчислювального процесу, стрілками, що позначаються сторожовими умовами розгалуження. Вершина «об'єднання» позначає точку злиття альтернативних потоків дій. Лінійки синхронізації дозволяють показати паралельні потоки дій, відзначаючи точки їх синхронізації при запуску (момент поділу) і по завершенні (момент злиття).

Приклад 10.1. Дії покупця в інтернет-магазині описуються діаграмою, у якій зображені дві точки розгалуження – для вибору способу пошуку товару та для прийняття рішення про купівлю. Є три лінійки синхронізації: верхня показує поділ на два паралельні процеси, середня – поділ і злиття процесів, а нижня – лише злиття процесів (рис. 10.9).

Додатково на цій діаграмі показано дві плавальні доріжки – покупця і магазину, які розділені вертикальною лінією. Кожна доріжка має ім'я і фіксує ділянку діяльності конкретної особи, позначаючи зону її відповідальності.

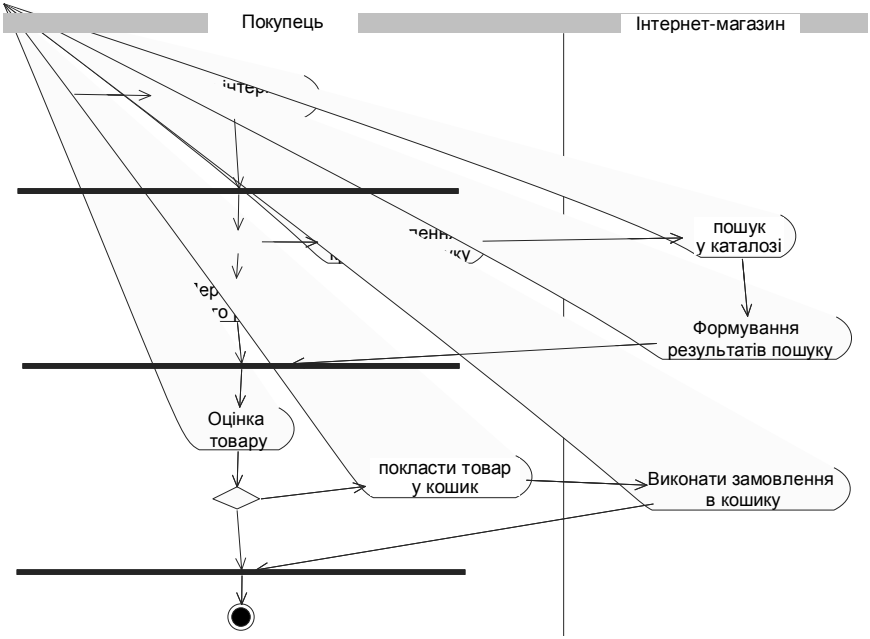


Рис. 10.9. Діаграма дій покупця в інтернет-магазині

10.4. Діаграми взаємодії

Діаграми взаємодії призначені для моделювання динаміки системи. Вони показують взаємодію, що включає набір об'єктів і їх відношень, а також повідомлення, що пересилаються між об'єктами. Існують два різновиди діаграми взаємодії – послідовності і співпраці. Діаграма послідовності виділяє впорядкування повідомлень за часом. Діаграма кооперації – виділяє структурну організацію об'єктів, що надсилають і приймають повідомлення. Елементами діаграм взаємодії є об'єкти, зв'язки, повідомлення.

10.5. Діаграми кооперації

Діаграми кооперації відображають взаємодію об'єктів у процесі функціонування системи. Такі діаграми моделюють сценарії поводження системи. Ім'я об'єкта підкреслюється і вказується завжди, властивості вказуються вибірково. Синтаксис подання імені має вигляд: «Ім'я об'єкта : Ім'я класу».

Вид запису імені*:*

«Тачскрін : Планшет» – ім’я об’єкта і класу;

«: Користувач» – лише ім’я класу (анонімний об’єкт);

«мій комп’ютер» – лише ім’я об’єкта (мається на увазі, що ім’я класу відомо);

«агент:» – об’єкт-сирота (мається на увазі, що ім’я класу невідомо).

Синтаксис подання властивості має вигляд: «Ім’я : Тип = Значення».

Запис властивості*:*

«номер: Телефон = " 7350-420"» – ім’я, тип, значення;

«активний = *True*» – ім’я і значення.

Об’єкти взаємодіють один з одним за допомогою зв’язків – каналів для передавання повідомлень. Зв’язок між парою об’єктів розглядається як екземпляр асоціації між їх класами. Інакше кажучи, зв’язок між двома об’єктами існує лише тоді, коли є асоціація між їх класами. Неявно всі класи є асоціаційованими, отже, об’єкт може надіслати повідомлення самому собі.

Отже, зв’язок – це шлях для пересилання повідомлення. Шлях може супроводжуватися характеристикою видимості, яка показується стандартним стереотипом на далекому кінці зв’язку. Для зв’язку передбачено такі стандартні стереотипи видимості:

«*global*» – об’єкт-постачальник перебуває в глобальній ділянці визначення;

«*local*» – об’єкт-постачальник перебуває в локальній ділянці визначення об’єкта-клієнта;

«*parameter*» – об’єкт-постачальник є параметром операції об’єкта-клієнта;

«*self*» – один і той самий об’єкт є і клієнтом, і постачальником.

Повідомлення – це специфікація передавання інформації між об’єктами в очікуванні того, що буде забезпечено необхідну діяльність. Приймання повідомлення розглядається як подія.

Результатом оброблення повідомлення є дія. Мовою *UML* моделюються такі різновиди дій: виклик – в об’єкті запускається операція; повернення – значення повертається об’єкту; відправлення (*send*) – об’єкту надсилається сигнал; створення – створюється об’єкт за стандартним повідомленням «*create*»; знищення – об’єкт знищується за стандартним повідомленням «*destroy*».

Для запису повідомлень у мові *UML* використовується такий синтаксис: «Повернзнач := Ім'я повідомлення (Аргументи)», де «Повернзнач» задає величину, що повертається як результат оброблення повідомлення.

Запис повідомлень: «коорд := теперзнаходж(літак1)» – виклик операції, повернення значення; «оповіщення( )» – відправлення сигналу; «встановити маршрут(*x*)» – виклик операції з дійсним параметром; «*create*» – стандартне повідомлення для створення об'єкта.

Коли об'єкт надсилає повідомлення іншому об'єкту (делегуючи деяку дію отримувачу), об'єкт-отримувач, у свою чергу, може надіслати повідомлення третьому об'єкту і т.д. Так формується потік повідомлень – послідовність керування. Очевидно, що повідомлення в послідовності мають бути пронумеровані. Номери записуються перед іменами повідомлень, напрямок повідомлень вказується стрілками (розміщуються над лініями зв'язків).

Найбільш загальну форму керування задає процедурний або вкладений потік (потік синхронних повідомлень). Процедурний потік зображується стрілками із заповненими наконечниками (рис. 10.10).

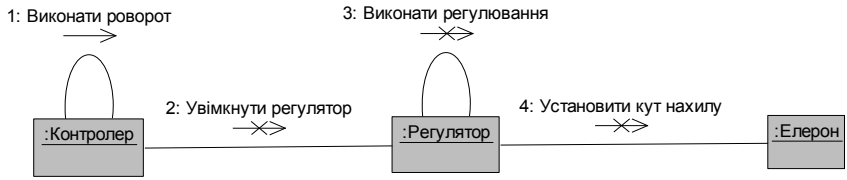


Рис. 10.10. Потік синхронних повідомлень

Тут повідомлення «1: виконати розворот» є синхронним, що послідовно викликає повідомлення «2: увімкнути регулятор», «3: виконати регулювання», «4: установити кут нахилу». Робота із синхронним повідомленням підлягає такому правилу: передавач чекає доти, доки отримувач не прийме і не обробить повідомлення. Це означає, що третє повідомлення буде надіслано лише після оброблення повідомлень 2 і 4.

Менш загальну форму керування задає асинхронний потік керування. Як показано на рис. 10.11, асинхронний потік зображується звичайними стрілками. Тут всі повідомлення вважаються асинх-

ронними, відповідно до яких передавач не чекає реакції від отримувача повідомлення. Такий вид комунікації має семантику поштової скриньки – одержувач приймає повідомлення в міру готовності. Тобто передавач і одержувач не синхронізують свою роботу, один об'єкт передає повідомлення другому об'єкту. У прикладі повідомлення «зробити запит» визначається друге повідомлення в послідовності.

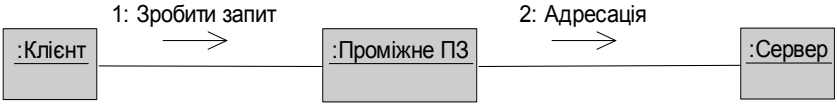


Рис. 10.11. Потік асинхронних повідомлень

Таким чином, для формування діаграми кооперації потрібні такі дії:

1) відобразити взаємодійні об'єкти;

2) установити зв'язки, що з'єднують взаємодійні об'єкти;

3) позначити зв'язки повідомленнями, які посилають і стримують виділені об'єкти.

Останні версії середовища *IBM Rational Rose* 2006 [17, 18] у розділі «Властивості повідомлень» не містять збережуваної умови. Тому для переходу за повідомленням 3 вводиться змінна *x*. Дії пілота залежать від оцінки *x*, яку дає регулювальник (рис. 10.12).

Рис. 10.12. Ітерація і розгалуження

У підсумку формується візуальне подання потоку керування (у контексті структурної організації об'єктів, що співпрацюють). Діаграму кооперації системи керування польотом літального апарата показано на рис. 10.13.

На діаграмі показано п'ять об'єктів з характеристиками видимості всіх зв'язків системи. Потік керування в системі містить вісім повідомлень: чотири асинхронні і чотири синхронні. Екземпляр «Контролер СК» чекає приймання і оброблення повідомлень: «Вимкрегшвидкість», «Вимкрегкутів», «Поточна швидкість», «Поточний кут». Порядок проходження повідомлень задано їх номерами.

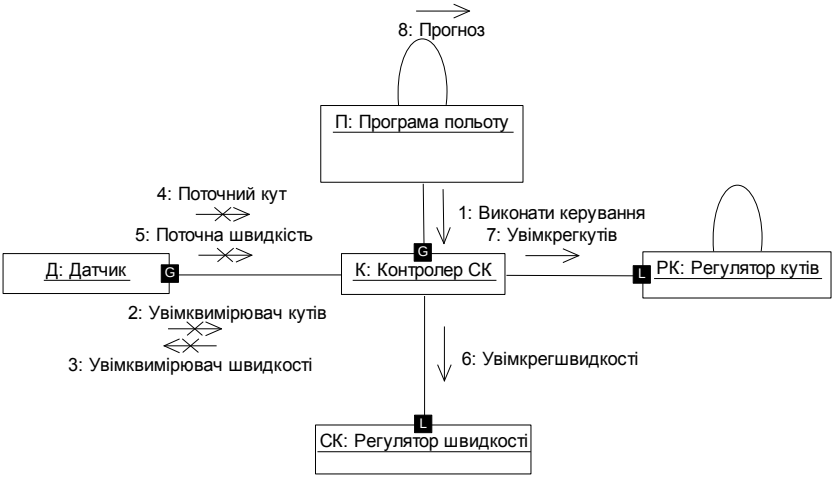


Рис. 10.13. Діаграма кооперації системи керування польотом

10.6. Діаграми послідовності

Це різновид діаграм взаємодії. Відображаючи сценарій поводження в системі, ця діаграма забезпечує наочну послідовність передавання повідомлень, але вона не дозволяє показати такі деталі, які видно на діаграмі кооперації (структурні характеристики об'єктів і зв'язків). Графічно діаграма послідовності являє собою таблицю, що показує об'єкти, розміщені горизонтально, і повідомлення, упорядковані за часом, – вертикально (рис. 10.14). Об'єкти, що беруть участь у взаємодії, містяться на вершині діаграми. Ліворуч розміщується об'єкт, що ініціює взаємодію, а праворуч – за підпорядкованістю.

Повідомлення, що посилаються і приймаються об'єктами, містяться уздовж осі *Y* в міру зростання часу від вершини до основи діаграми. Використовуються ті самі синтаксис і позначення синхронізації, що й у діаграмах кооперації. Таким чином, забезпечується просте візуальне подання потоку керування в часі.

Від діаграм кооперації діаграми послідовності відрізняють дві важливі характеристики. Перша характеристика – лінія життя об'єкта (вертикальна пунктирна лінія, що позначає період існування об'єкта). Більшість об'єктів існують протягом усього часу взаємодії, їх лінії життя простираються від вершини донизу діаграми.

При цьому об’єкти можуть створюватися в ході взаємодії. Їх лінії життя починаються з моменту приймання повідомлення «*create*». Крім того, об’єкти можуть знищуватися в ході взаємодії. Момент приймання повідомлення «*destroy*» означає закінчення лінії життя. Знищення лінії життя позначається × у кінці лінії (рис. 10.15).

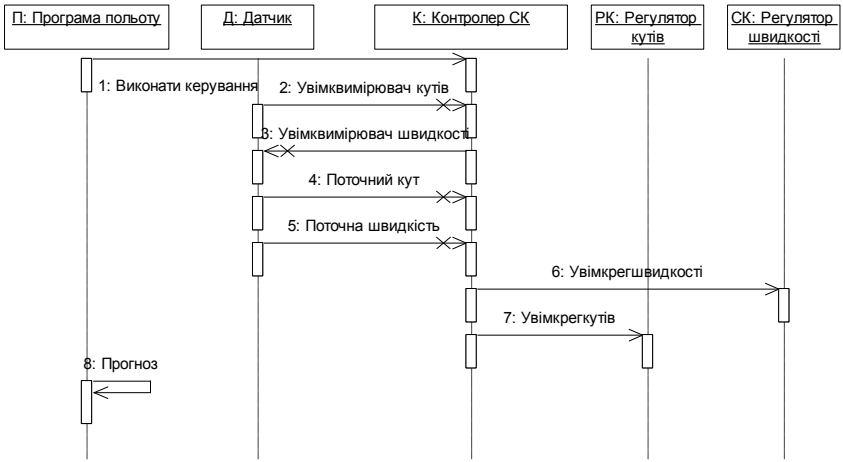


Рис. 10.14. Діаграма послідовності системи керування польотом

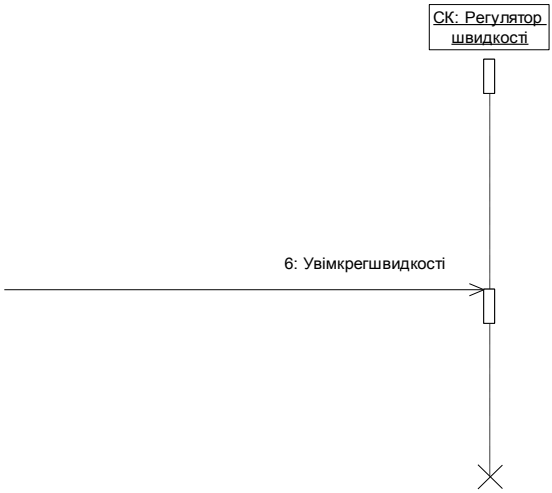


Рис. 10.15. Створення і знищення об’єкта

Інша характеристика – фокус керування, що має вигляд високого тонкого прямокутника, що відображає період часу, протягом якого об'єкт виконує дію (свою або залежну процедуру). Вершина прямокутника позначає початок дії, а основа – його завершення. Момент завершення може маркуватися повідомленням повернення, що зображується пунктирною стрілкою.

10.7. Діаграми *Use Case*

Діаграма *Use Case* (прецедентів, варіантів використання) визначає поводження системи з погляду користувача і розглядається як головний засіб для первинного моделювання динаміки системи. Її основне застосування визначається з'ясуванням вимог до розроблюваної системи, фіксацією цих вимог у формі, що дозволить її розробляти надалі.

До складу діаграм *Use Case* входять прецеденти, актори, а також відношення залежності, узагальнення і асоціації. Ці діаграми можуть містити пакети, використовувані для угруповання елементів моделі у великі фрагменти. Як і інші діаграми *Use Case* можуть включати примітки і обмеження.

10.7.1. Елементи діаграми

Вершинами в діаграмі *Use Case* є актори і прецеденти. Їх позначення показано на рис. 10.16. Актори користуються роботою системи, прецеденти – зображають дії, виконувані системою в інтересах акторів.

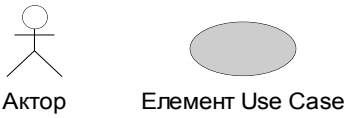


Рис. 10.16. Позначення актора і прецедента

Актор – роль об'єкта поза системою, що прямо взаємодіє з її частиною – конкретним прецедентом. Розрізняють акторів і користувачів. Користувач – це фізичний об'єкт, що використовує систему. Він може виконувати декілька ролей і тому може моделюватися декількома акторами. Справедливим є і те, що актором можуть бути різні користувачі.

Наприклад, для комерційного літального апарата можна виділити двох акторів: пілота і касира. Сидоров – користувач, що іноді діє як пілот, а іноді – як касир. Як зображено на рис. 10.17, залежно від ролі Сидоров взаємодіє з різними прецедентами.

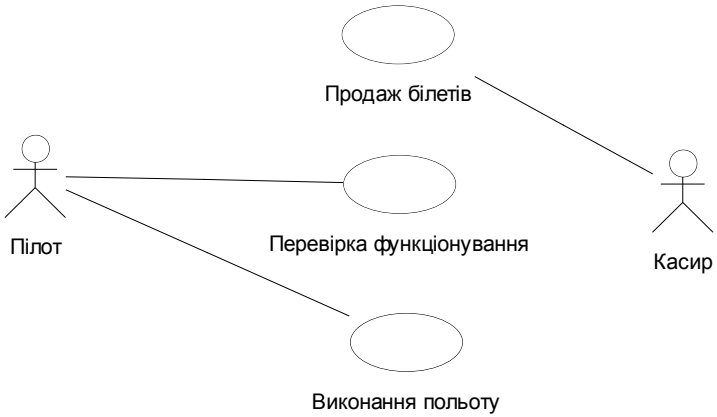


Рис. 10.17. Модель діаграми *Use Case*

Прецедент – це опис послідовності дій (або декількох послідовностей), які виконуються системою і досягають для окремого актора видимий результат. Один актор може використовувати кілька прецедентів, і навпаки, один прецедент може мати декілька акторів, що використовують його. Кожний прецедент задає певний маршрут використання системи. Набір усіх елементів *Use Case* визначає повні функціональні можливості системи.

10.7.2. Типи відношень у діаграмах

Між актором і елементом *Use Case* можливий лише один вид відношень – асоціація, що відображає їх взаємодію (рис. 10.18). Будь-яка асоціація може мати ім'я, роль, потужність.

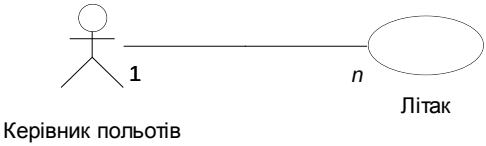


Рис. 10.18. Відношення асоціації

Між акторами припустиме відношення узагальнення (рис. 10.19). Це означає, що екземпляр нащадка може взаємодіяти з такими ж різновидами екземплярів елементів *Use Case*, що й екземпляр батька.

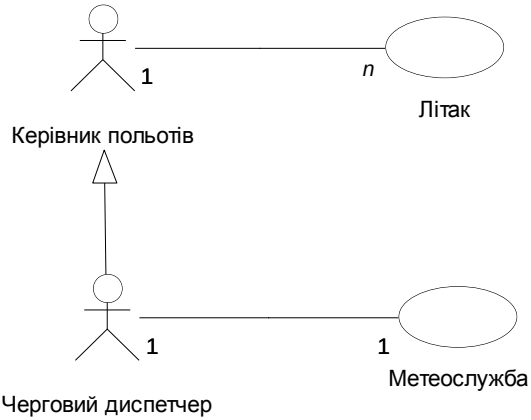


Рис. 10.19. Відношення узагальнення між акторами

Між прецедентами визначено відношення узагальнення і два різновиди відношень залежності – включення і розширення. Відношення узагальнення (рис. 10.20) фіксує, що нащадок успадковує поводження батька.

Рис. 10.20. Відношення узагальнення між елементами *Use Case*

Крім того, нащадок може доповнити або перевизначити поводження батька. Прецедент, який є нащадком, може заміщати елемент *Use Case*, що є батьком, у будь-якому місці діаграми.

Відношення включення (рис. 10.21) між прецедентами означає, що базовий прецедент явно включає поводження іншого прецедента у точці, що визначена в базі. Елемент, що включається, ніколи не використовується самостійно – його конкретизація може бути лише частиною іншого, більшого прецедента. Відношення включення –

приклад відношень делегації. Окремий прецедент містить певний набір обов'язків системи. Далі інші частини системи можуть агрегувати ці обов'язки (у разі потреби).

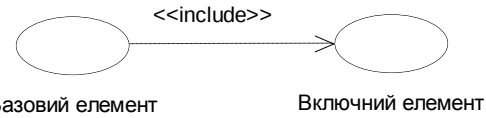


Рис. 10.21. Відношення включення між елементами *Use Case*

Відношення розширення (рис. 10.22) між прецедентами означає, що базовий прецедент неявно включає поводження іншого прецедента у точці, що визначається побічно розширюваним прецедентом. Базовий елемент *Use Case* може бути автономним, але за певних умов його поводження розширюватиметься поводженням іншого прецеденту. Базовий прецедент розширюватиметься лише в певних точках – точках розширення. Відношення розширення застосовується для моделювання обраного поводження системи. Таким способом можна відокремити обов'язкове поводження від необов'язкового поводження. Наприклад, можна використовувати відношення розширення для окремого підпотоку, що виконується лише за певних умов, що перебувають поза полем зору базового прецеденту. Нарешті, можна моделювати окремі потоки, уставлення яких у певну точку керується актором.

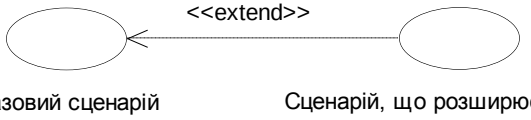


Рис. 10.22. Відношення розширення між елементами *Use Case*

Приклад найпростішої діаграми *Use Case*, у якій використано відношення включення і розширення, показано на рис. 10.23.

10.7.3. Робота з елементами діаграми

Прецеденти описують функціонування системи, але не вказують на послідовність дій. Моделювання дозволяє відокремлювати зовнішнє подання системи від її внутрішнього подання. Поводження

прецеденту описується потоком подій. Початковий опис виконується в текстовій формі, прозорій для користувача системи. У потоці подій виділяють:

- основний потік і альтернативні потоки поводження;
- початок і закінчення прецеденту;
- взаємодію прецеденту з акторами;
- дані, якими обмінюються актор і система.

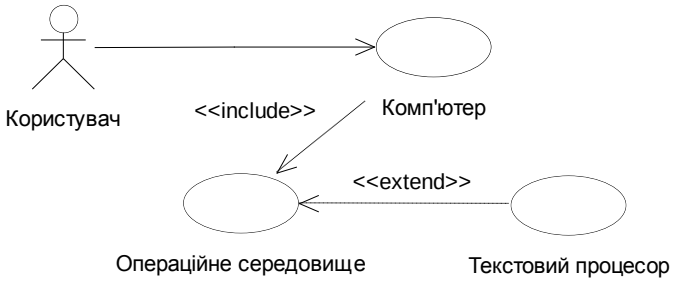


Рис. 10.23. Найпростіша діаграма *Use Case* для банку

Для уточнення і формалізації потоків подій використовують діаграми послідовності. Зазвичай одна діаграма послідовності визначає головний потік у прецеденті, а інші діаграми – потоки виключень.

У загальному випадку один прецедент описує набір послідовностей, у якому кожна послідовність являє собою можливий потік подій. Кожна послідовність називається сценарієм. Сценарій – конкретна послідовність дій, що ілюструє поводження, для прецеденту він є майже тим же, чим є екземпляри для класу. Сценарій – це екземпляр прецеденту.

Основним джерелом інформації для аналізу і проектування системи є специфікація прецеденту. Істотною вимогою до змісту специфікації є її повнота і конструктивність. У загальному випадку специфікація включає головний потік, підпотоки і альтернативні потоки поводження.

Якщо самостійний прецедент містить підпотік, то його варто з'єднати з базовим прецедентом відношенням *include*. Своєю чергою, самостійний прецедент з альтернативним потоком підключається до базового прецеденту відношенням *extend*.

Приклад 10.2. Діаграму *Use Case* для дій користувача банкомата під час виконання простих операцій – перевірки рахунка, зняття грошей – зображено на рис. 10.24.

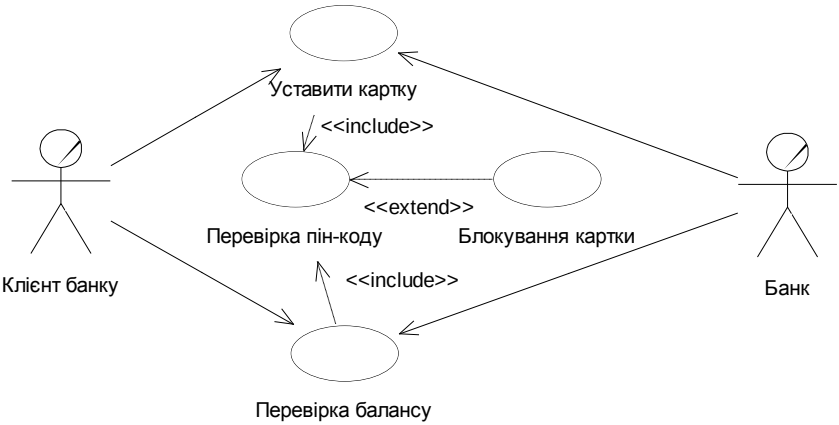


Рис. 10.24. Діаграма *Use Case* для сеансу взаємодії з банкоматом

У цій діаграмі актор «клієнт» взаємодіє з двома базовими прецедентами «Вставити картку» та «Перевірка балансу». Базові прецеденти включають прецедент «Перевірка пін-коду», який розширюється на прецедент «Блокування картки».

10.7.4. Розширення функціональних можливостей

Для додавання прецеденту нових дій зручно застосовувати відношення розширення, за допомогою якого базовий прецедент може бути розширений новим прецедентом.

Приклад 10.3. Базовий прецедент «Сеанс роботи» може бути розширений на прецеденти «Компіляція», «Робота з поштою», «Текстове оброблення». Базовий прецедент при цьому залишається майже без змін.

Відношення розширення визначає переривання базового прецеденту для вставлення іншого прецедента. Базовий прецедент не знає чи буде виконуватися переривання, чи не буде. Обчислення умов переривання перебуває поза компетенцією базового прецеденту.

У розширюваному прецеденті вказується посилання на те місце базового прецедента, куди він буде вставлятися (у разі переривання). Після виконання розширюваного прецедента триває виконання базового прецеденту.

Розширення використовують для моделювання:

– варіантних частин прецеденту;

– складних і рідко виконуваних альтернативних послідовностей;

– залежних послідовностей, які виконуються лише в певних випадках;

– систем з вибором за допомогою меню.

Рішення про вибір, підключення варіанта на основі розширення приймається поза базовим прецедентом. Якщо вводиться в базовий прецедент умовна конструкція, конструкція вибору, то необхідно застосовувати відношення включення. Це випадок, коли керування надається базовому прецеденту.

10.8. Уточнення моделі вимог

Уточнення моделі зводиться до пошуку однакових частин у прецеденті та їх вилучення. Будь-які зміни в такій частині, виділеній в окремий прецедент, будуть автоматично впливати на всі прецеденти, які використовують її спільно.

Витягнуті прецеденти називають абстрактними. Вони не можуть бути конкретизовані самі по собі, застосовуються для опису однакових частин в інших, конкретних прецедентах. Таким чином, опис абстрактних прецедентів використовують в описах конкретних прецедентів. Конкретний прецедент перебуває у відношенні «Увімкнення» з абстрактним прецедентом.

Приклад 10.4. Два конкретні прецеденти «Повернення елемента» та «створення денного звіту» мають загальну частину – дії друкування квитанції. Тому, як показано на рис. 10.25, можна виділити абстрактний прецедент «Друк». Цей прецедент буде спеціалізуватися на друкуванні квитанцій.

Рис. 10.25. Застосування відношення включення

Своєю чергою, абстрактні прецеденти можуть використовувати інші абстрактні прецеденти. Так утворюється ієрархія. Під час побудови ієрархії абстрактних прецедентів керуються правилом: виділення прецедентів припиняється із досягненням рівня окремих операцій над об’єктами.

Виділення абстрактних прецедентів можна спростити за допомогою абстрактних акторів.

Абстрактний актор – це загальний фрагмент ролі декількох конкретних акторів. Абстрактний актор виражає подібність прецедентів. Конкретні актори перебувають у відношенні спадкування з абстрактним актором. Так, конкретні актори мають одне загальне поводження: вони можуть отримувати квитанцію. Тому можна визначити одного абстрактного актора – «Отримувача квитанції», спадкоємцями цього актора є «Клієнт» і «Касир» (рис. 10.26).

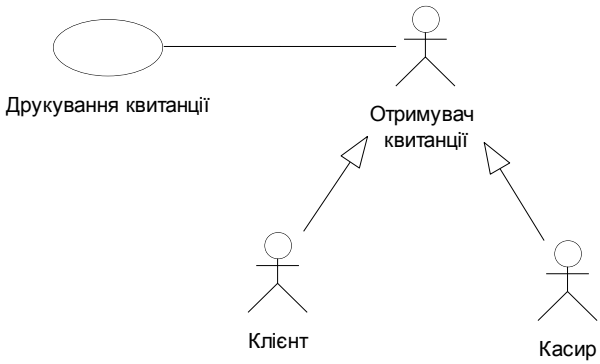


Рис. 10.26. Виділення абстрактного актора

Таким чином, абстрактні прецеденти знаходять добуванням загальних послідовностей з різних прецедентів. Відношення «Увімкнення» застосовується, якщо кілька прецедентів мають загальне поводження для усунення повторів, ліквідації надмірності. Крім того, це відношення часто використовують для обмеження складності великого прецеденту. Відношення «Розширення» застосовується, коли описується варіація, що доповнює нормальне поводження.

Контрольні запитання і завдання

1. Поясніть, який підхід доцільно використовувати для моделювання поводження системи в часі.
2. Яка відмінність між дією і подією, як вони відображаються на діаграмі станів ?
3. Як задаються вкладені стани в діаграмах схем станів?
4. Коли не слід застосовувати діаграму діяльності?
5. Які засоби діаграми діяльності дозволяють відобразити одночасні дії?

6. Навіщо в діаграму діяльності введені плавальні доріжки?
7. Запишіть повідомлення мовою *UML*. Поясніть зміст повідомлення.
8. У якому відношенні перебувають повідомлення і дії? Назвіть різновиди дій.
9. У чому полягає відмінність між процедурним і асинхронним потоками повідомлень?
10. Що загального в діаграмі послідовності і діаграмі кооперації? Яка між ними відмінність?
11. Як відображається передавання повідомлень у діаграмі послідовності?
12. Коли зручніше застосовувати діаграму послідовності?
13. У чому полягає відмінність між відношеннями включення і розширення з погляду керування?
14. Яке призначення специфікації прецеденту і як вона оформлюється?
15. Як документуються відношення включення, розширення?

11. МОДЕЛІ РЕАЛІЗАЦІЇ ОБ'ЄКТНО-ОРІЄНТОВАНИХ ПРОГРАМНИХ СИСТЕМ

11.1. Компонентні діаграми

Статичні і динамічні моделі описують логічну організацію системи, відображають логіку програмного додатка. Моделі реалізації забезпечують фізичне подання системи у вигляді логічних елементів, упакованих у компоненти, та їх розміщення в апаратних вузлах.

Компонентною називається діаграма, яка моделює фізичні аспекти об'єктно-орієнтованих систем. Вона показує організацію набору компонентів та залежності між ними.

Елементами компонентних діаграм є компоненти й інтерфейси, а також відношення залежності та реалізації. Як і інші діаграми, компонентні діаграми можуть включати примітки й обмеження. Крім того, ці діаграми можуть містити пакети або підсистеми, використовувані для угруповання елементів моделі у великі фрагменти.

11.1.1. Компоненти

За суттю компоненти є фізичним фрагментом реалізації системи, що містить виконуваний програмний код, сценарні описи або набори команд операційної системи (командні файли).

Компонент – фізична і замінна частини системи, що відповідає набору інтерфейсів і забезпечує реалізацію цього набору інтерфей-

сів. Графічно компонент зображується як прямокутник із вкладками, що має ім'я (рис. 11.1).

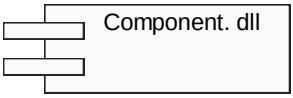


Рис. 11.1. Позначення компонента

Оскільки компонент є базисним будівельним блоком фізичного подання ПЗ, тому доцільно порівняти його з базисним будівельним блоком логічного подання ПЗ-класом.

Подібність між компонентом і класом:

– наявність ім'я;

– реалізація набору інтерфейсів;

– участь у відношеннях залежності;

– можливість вкладення;

– наявність екземплярів (екземпляри компонентів можна використовувати лише в діаграмах розміщення).

Проте між компонентами і класами є істотна відмінність, її характеризує табл. 11.1.

Таблиця 11.1

Відмінність між компонентами і класами

Клас	Компонент
Логічна абстракція	Фізичний предмет, одиниця вимірювання – біт; містить різні логічні елементи
Властивості, операції	Має лише операції, доступні через його інтерфейс

Клас потребує оболонки. Такою оболонкою є компонент. В оболонці може перебувати кілька класів і кооперацій. Отже, оболонка містить набір логіки. Клас – відкритий, він може показати свої властивості. Компонент завжди закритий, але доступ до нього здійснюється через інтерфейс можливих операцій.

За останні півстоліття розробники апаратури пройшли шлях від комп'ютерів розміром з кімнату до міні-ноутбуків, що забезпечують великі функціональні можливості. За ті ж півстоліття розробники ПЗ пройшли шлях від великих систем в Асемблері і Фортрані до ще більших систем у *C++* і *Java*. Проте програмний інструментарій розвивається повільніше, ніж апаратний інструментарій.

Розробник апаратури створює систему з готових апаратних компонентів (мікросхем), що виконують певні функції і надають набір послуг через прозорі інтерфейси. Завдання конструкторів спрощується за рахунок повторного використання результатів, отриманих раніше.

Повторне використання – магістральний шлях розвитку програмного інструментарію. Створення нового ПЗ з існуючих, працездатних програмних компонентів дає змогу отримувати більш надійний і дешевий код. При цьому рядки розроблення істотно скорочуються.

Основна мета програмних компонентів – складати систему із двійкових замінних частин. Вони повинні забезпечити початкове створення системи з компонентів , а потім і її розвиток – додавання нових компонентів і заміну деяких старих компонентів без перебудови системи в цілому. Ключ до втілення такої можливості – інтерфейси. Після того як інтерфейс визначений, до виконуваної системи можна підключити будь-який компонент, що задовольняє його або забезпечує цей інтерфейс. Для розширення системи виробляються компоненти, які надають додаткові послуги через нові інтерфейси. Такий підхід ґрунтується на таких особливостях компонента.

1. Компонент – фізичний елемент, який не містить логічних понять і не залежить від мови програмування.

2. Компонент – замінний елемент. Властивість замінності дозволяє замінити один компонент іншим компонентом, що задовольняє той самий інтерфейс. Механізм заміни підтримується сучасними компонентними моделями (*COM, COM+, CORBA, Java Beans*), що потребують незначних перетворень або надають утиліти автоматизації.

3. Компонент – частина системи. Частіше компонент взаємодіє з іншими компонентами та існує в архітектурному або технологічному середовищі, призначеному для його використання. Компонент зв’язаний і фізично, і логічно, він позначає фрагмент великої системи.

4. Компонент відповідає набору інтерфейсів і забезпечує реалізацію цього набору інтерфейсів.

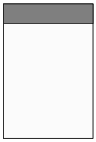
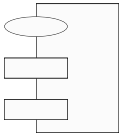
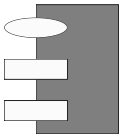
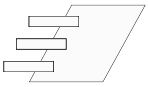
Таким чином, компоненти – базисні будівельні блоки, з яких проектується і складається система. Компонент може з’являтися на різних рівнях ієрархії подання складної системи. Система на одному рівні абстракції може стати простим компонентом на більш високому рівні абстракції.

Світ сучасних компонентів досить широкий і різноманітний. У мові *UML* для позначення нових видів компонентів використовують механізм стереотипів. На діаграмах компонентів відображається у виглядів стереотипів; їх характеристику наведено

в табл. 11.2. Кожному з компонентів відповідає окремий файл вихідного складання програмного додатка.

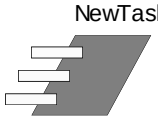
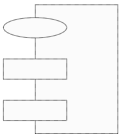
Таблиця 11.2

Графічне зображення стереотипів компонентів та їх характеристика

Графічне зображення	Назва	Зміст
	*Subprogram Specification*	Специфікація підпрограми. Містить опис змінних, процедур, функцій
	*Subprogram Body*	Підпрограма. Містить реалізацію процедур та функцій
	*Main Program*	Головна програма. Реалізовує базову логіку роботи програмного додатка і містить посилання на інші компоненти моделі
	*Package Specification*	Специфікація пакета. Містить визначення класу, його атрибутів і операцій. У мові програмування C++ специфікації пакета відповідає окремий файл з розширенням *h*
	*Package Body*	Пакет. Містить код реалізації операцій класу. У мові програмування C++ специфікації пакета відповідає окремий файл з розширенням *cpp*
	*Task Specification*	Специфікація задачі. Складається з визначення класу, його атрибутів і операцій, які передбачається використовувати в незалежному потоці керу-

		вання

Закінчення табл. 11.2

Графічне зображення	Назва	Зміст
	*Task Body*	Тіло задачі. Містить реалізацію операцій класу, які мають незалежний потік керування
	*Generic Subprogram*	Типова підпрограма. Містить опис змінних, процедур і функцій, які можуть бути використані в деяких програмних додатках. Типова підпрограма не містить визначень класів
	*Generic Package*	Типовий пакет. Містить визначення класу, його атрибутів і операцій, яке може бути використано в деяких програмних додатках
	*Database*	База даних. Містить визначення одного або декількох класів, їх атрибутів та, можливо, операцій. Відповідні класи можуть бути реалізовані у вигляді однієї або декількох таблиць бази даних

11.1.2. Інтерфейси

Інтерфейс – список операцій, які визначають функціональні можливості класу чи компонента до виконання завдань. Інакше інтерфейс є рознімом, через який доступні операції компонента. За допомогою інтерфейсу компоненти стикаються один з одним, утворюючи систему.

Інтерфейс подібний до абстрактного класу, який не має властивостей і операцій, а є лише абстрактні операції (які не мають тіл). Усі операції інтерфейсу відкриті і видимі клієнту, вони вказують лише на пропоновані послуги.

За способом зв’язку компонента з інтерфейсом розрізняють:

– експортований інтерфейс, у якому реалізовується і пропонується компонент як послуга клієнтам;

– імпортований інтерфейс, у якому компонент використовується як послуга іншого компонента.

Один компонент може мати декілька експортованих і декілька імпортованих інтерфейсів. Той факт, що між двома компонентами завжди міститься інтерфейс, усуває їх безпосередню залежність. Компонент, що використовує інтерфейс, буде функціонувати правильно незалежно від того, який компонент реалізовує цей інтерфейс. Це забезпечує гнучку заміну компонентів, що сприяє розвитку системи.

11.1.3. Діаграми розгортання

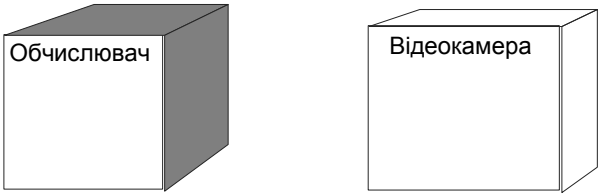
Діаграма розміщення (розгортання) – друга із двох різновидів діаграм реалізації *UML*, що моделюють фізичні аспекти об'єктно-орієнтованих систем, і розробляється, як правило, для територіально розподілених систем.

Цей тип діаграм застосовується для подання загальної конфігурації і топології розподіленої програмної системи і містить зображення розміщення компонентів в окремих вузлах системи. Крім того, діаграма розгортання вказує наявність фізичних сполук – маршрутів передавання інформації між апаратними пристроями, задіяними в реалізації системи.

Призначення діаграми розгортання полягає у візуалізації елементів і компонентів програми, які наявні лише на етапі розміщення системи. При цьому подаються лише ті компоненти програми, які здійснюються файлами або динамічними бібліотеками. Компоненти, які не використовуються на етапі виконання, на діаграмі розгортання не показуються. Компоненти з вихідними текстами програм можуть бути лише на діаграмі компонентів. На діаграмі розгортання вони не показуються.

Діаграма розгортання містить графічні зображення процесорів, пристроїв, процесів і зв'язків між ними. На відміну від діаграм логічного подання вона є єдиною для системи в цілому, оскільки відображає особливості її реалізації. Цією діаграмою завершується процес проектування конкретної програмної системи та її розроблення останній етап специфікації моделі. Вона розробляється спільно системними аналітиками, мережевими інженерами і системотехніками.

Вузол являє собою фізично існуючий елемент системи, який може містити обчислювальний ресурс, що має пам'ять, а можливо, і можливість оброблення. Він зображується у формі куба з боковими гранями сірого кольору (рис. 11.2, *а*). Інший стереотип – пристрій (рис. 11.2, *б*). Графічно вузол зображується кубом з іменем.



а *б*

Рис. 11.2. Зображення вузла

Елементами діаграм розміщення є вузли та асоціації, які зображуються відрізками ліній без стрілок. Наявність такої лінії вказує на необхідність організації фізичного каналу для обміну інформацією між відповідними вузлами. Характер з'єднання може бути додатково специфікований приміткою, стереотипом, супроводжується значенням або обмеженням. Як і інші діаграми, діаграми розміщення можуть включати примітки й обмеження. Так, на рис. 11.3 показано фрагмент діаграми розгортання, на якій визначено рекомендації з технології фізичної реалізації з'єднання у вигляді примітки.

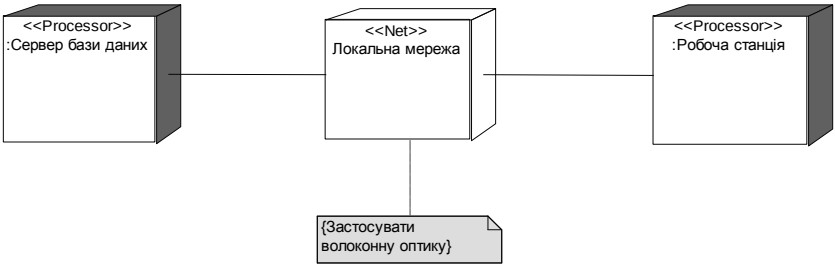


Рис. 11.3. Фрагмент діаграми розгортання

Розроблення діаграми розгортання починається з ідентифікації всіх апаратних, механічних та інших типів пристроїв, які необхідні

для виконання системою всіх функцій. Спочатку специфікуються обчислювальні вузли системи, що містять процесор і пам'ять. При цьому використовуються наявні в мові *UML* стереотипи, а якщо їх немає, розробники можуть визначити нові стереотипи. Окремі вимоги до складу апаратних засобів можуть задаватись у формі обмежень і помічених значень.

Подальша деталізація діаграми розгортання пов'язана з розміщенням усіх виконуваних компонентів діаграми у вузлах системи.

Із моделі має бути вилучена ситуація, коли окремі виконувані компоненти виявилися не розміщеними у вузлах. Із цією метою можна ввести в модель додаткові вузли, що містять процесор і пам'ять. Для розроблення простих програм, які виконуються локально на одному комп'ютері, потреби в діаграмі розгортання немає. Побудова діаграми розгортання доцільна для моделювання:

– клієнт-серверних систем за умови чіткого поділу повноважень між клієнтськими робочими станціями та сервером бази даних з метою уточнення не лише топології системи, але й її компонентного складу;

– систем з неоднорідною розподіленою архітектурою, окремі вузли якої можуть бути віддалені один від одного на сотні кілометрів, для візуалізації загальної топології системи і контролю міграції окремих компонентів між вузлами;

– систем реального часу із вбудованими мікропроцесорами для візуалізації складу всіх пристроїв і їх взаємозв'язків у системі.

Діаграму розгортання можна розробляти для аналізу існуючої системи для її подальшої деталізації та модифікації. Розроблення діаграми на початкових етапах аналізу характеризує її загальний напрямок від фізичного подання до логічного.

11.1.4. Генерація програмного коду

Розроблення складних систем потребує створення програмного коду, який розкиданий по багатьох файлах. У разі використання *Java* вихідний код зберігається в *java*-файлах, у разі використання *C++* – у заголовних файлах (.*h*-файлах) і тілах (.cpp-файлах), а в разі використання *Ada* 95 – у специфікаціях (.*ads*-файлах) і реалізаціях (.*adb*-файлах).

Можливість генерації тексту програми тією чи іншою мовою програмування залежить від установленого середовища для ство-

рення *UML*-проекту. Програма *IBM Rational Rose* має можливість генерації програмного коду декількома мовами програмування і може бути використана розробником після побудови моделі без інсталяції додаткового ПЗ. Вибір мови залежить від версії програми.

Генеровані програмою *IBM Rational Rose* файли з текстом програмного коду містять мінімум інформації. Для введення додаткових елементів у програмний код слід змінювати властивості генерації програмного коду, установлені за замовчуванням.

Ефект від використання таких програмних засобів проявляється під час розроблення масштабних проектів у складі команди або проектної групи, коли необхідно виконати проект із декількома десятками варіантів використання і сотнею класів. Саме для подібних проектів виявляється перевага використання засобу *IBM Rational Rose* і нотації мови *UML* для документування та реалізації відповідних моделей.

11.2. Основи компонентної об'єктної моделі

Компонентна об'єктна модель (*COM*) – фундамент компонентно-орієнтованих засобів сім'ї операційних систем *Windows*.

COM визначає стандартний механізм, за допомогою якого одна частина ПЗ надає свої послуги іншій частині. Загальна архітектура надання послуг у бібліотеках, додатках, системному і мережевому ПЗ дозволяє *COM* змінити підхід до створення програм.

COM установлює поняття і правила, необхідні для визначення об'єктів і інтерфейсів; крім того, у її склад входять програми, що реалізовують ключові функції.

У *COM* будь-яка частина ПЗ реалізовує свої послуги за допомогою об'єктів *COM*. Кожний об'єкт «*COM*» підтримує декілька інтерфейсів. Клієнти можуть отримати доступ до послуг об'єкта *COM* тільки через виклики операцій його інтерфейсів – вони не мають безпосереднього доступу до даних об'єкта.

Розглянемо об'єкт *COM* з інтерфейсом «Робота з файлами». Нехай у цей інтерфейс входять операції «Відкрити файл», «Записати файл» і «Закрити файл». Якщо розробник має намір увести в об'єкт *COM* функцію перетворення форматів, то об'єкт потребує ще одного інтерфейсу перетворення форматів, можливо, з єдиною операцією «Перетворити формат». Операції кожного з інтерфейсів спільно

надають спільні послуги: або роботу з файлами, або перетворення їх форматів.

Як показано на рис. 11.4, об'єкт *COM* завжди реалізовується усередині деякого сервера, який може бути бібліотекою, що підключається динамічно (*DLL*), завантажується під час роботи додатка, або окремим самостійним процесом (*EXE*). Зазначимо, що тут не застосовуються графічні засоби мови *UML*, а використовуються упроваджені в *COM* позначення.

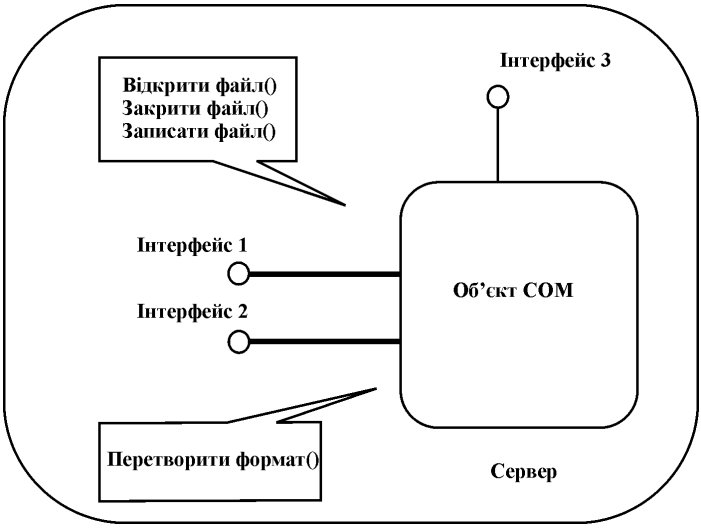


Рис. 11.4. Організація об'єкта *COM*

Для виклику операції інтерфейсу клієнт об'єкта *COM* повинен отримати покажчик на його інтерфейс. Клієнту потрібен окремий покажчик для кожного інтерфейсу, операції якого він має намір викликати. Наприклад, як показано на рис. 11.5, клієнту об'єкта *COM* потрібен один покажчик інтерфейсу для виклику операцій інтерфейсу «Робота з файлами», а інші – для виклику операцій інтерфейсу «Перетворення форматів».

Отримавши покажчик на потрібний інтерфейс, клієнт може використовувати послуги об'єкта, викликаючи операції цього інтерфейсу. Із погляду програміста виклик операції аналогічний виклику локальної процедури або функції. Але код, що виконується насправді при цьому, може бути частиною або бібліотеки, або окремого про-

цесу, або операційної системи (він навіть може розміщуватися на іншому комп'ютері).

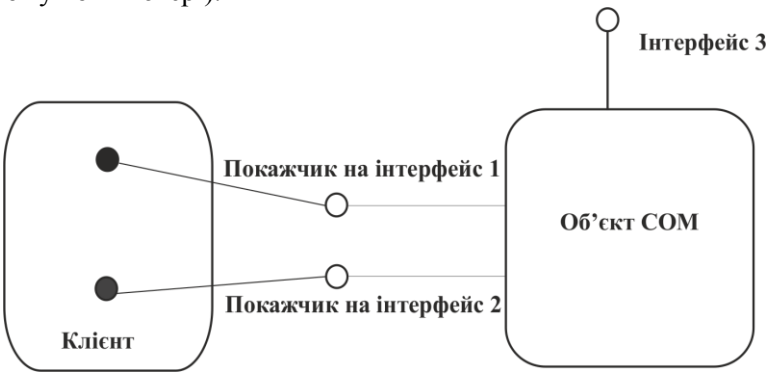


Рис. 11.5. Доступ клієнта до інтерфейсів об'єкта *COM*

Завдяки *COM*-клієнтам немає потреби враховувати ці відмінності – доступ до його функцій здійснюється одноманітно. Інакше кажучи, у *COM* для доступу до послуг, надаваним будь-якими типами ПЗ, використовується одна загальна модель.

11.3. Організація інтерфейсу *COM*

Кожен інтерфейс об'єкта *COM* – контракт між об'єктом і його клієнтами. Вони зобов'язуються: об'єкт – підтримувати операції інтерфейсу в точній відповідності з його визначеннями, а клієнт – коректно викликати операції. Для забезпечення контракту треба задати:

– ідентифікацію кожного інтерфейсу;
– опис операцій інтерфейсу;
– реалізацію інтерфейсу.

Кожному інтерфейсу об'єкта *COM* надається два імені. Просте символьне ім'я призначено для людей, воно не є унікальним. Інше, складне ім'я призначене для використання програмами. Програмне ім'я унікальне, це дозволяє точно ідентифікувати інтерфейс.

За замовченням символьні імена *COM*-інтерфейсів починаються з літери *I* (від *Interface*). Наприклад, інтерфейс для роботи з файлами має називатися «Іроботазфайлами», а інтерфейс перетворення їх форматів – «Іперетворенняформатів».

Програмне ім’я будь-якого інтерфейсу утворюється за допомогою глобально унікального ідентифікатора *GUID* (*Globally Unique Identifier*), який являє собою 16-байтову величину, що генерується автоматично.

Унікальність у часі досягається введенням в кожен *GUID* мітки часу – покажчика моменту створення. Унікальність у просторі забезпечується числовими параметрами комп’ютера, що використовувався для генерації *GUID*.

11.3.1. Опис інтерфейсу COM

Для визначення інтерфейсів застосовують спеціальну мову опису інтерфейсів *IDL* (*Interface Definition Language*).

Приклад 11.1. Опис мовою *IDL* інтерфейсу «Іроботазфайлами» має вигляд:

```
[ object
        uuid(E7 CDODOO-1827-11 CF-9946-444553540000) ]
        interface Іработасфайлами: IUnknown
{
    import "unknown.idl"
    HRESULT Відкритифайл ([in] OLECHAR ім’я [31]);
    HRESULT Записатифайл ([in] OLECHAR ім’я [31]);
    HRESULT Закритифайл ([in] OLECHAR ім’я [31]);
}
```

Опис інтерфейсу починається зі слова *object*, яке означає, що будуть використовуватися розширення, додані *COM* до оригінального запису мовою *IDL*. Далі записується програмне ім’я (*IID* інтерфейсу), воно починається із ключового слова *uuid* (*Universal Unique Identifier* – універсальний унікальний ідентифікатор), який є синонімом терміна *GUID*.

У третьому рядку записується ім’я інтерфейсу «Роботазфайлами», далі двокрапка й ім’я іншого інтерфейсу – *IUnknown*. Такий запис означає, що інтерфейс «Роботазфайлами» успадковує всі операції, визначені для інтерфейсу *IUnknown*, тобто клієнт, у якого є покажчик на інтерфейс «Іроботазфайлами», може викликати й операції інтерфейсу *IUnknown*. Цей інтерфейс є головним для *COM*, усі інші – його нащадки.

У четвертому рядку вказується оператор «*import*». Оскільки інтерфейс успадковується від «*IUnknown*», тоді потрібен *IDL*-опис

для *IUnknown*. Аргумент оператора «*import*» указує, у якому файлі перебуває потрібний опис.

Нижче в описі інтерфейсу наводяться імена і параметри трьох операцій – «Відкрити файл», «Записати файл» і «Закрити файл». Усі вони повертають величину типу «*HRESULT*», що вказує на коректність оброблення виклику. Для кожного параметра в *IDL* наводяться напрямок передавання (у цьому прикладі «*in*»), тип і назва. Вважається, що такого опису достатньо для укладання контракту між об'єктом *COM* і його клієнтом.

За правилами *COM* забороняється проводити будь-яку зміну інтерфейсу (після його публікації). Для реалізації змін потрібно вводити новий інтерфейс. Такий інтерфейс може бути спадкоємцем старого інтерфейсу, але відмінним від нього і мати інше унікальне ім'я.

Значення правила заборони на зміну інтерфейсу важко переоцінити. Воно є основою стабільної роботи в середовищі, де множина клієнтів взаємодіє з множиною об'єктів *COM* і де незалежна модифікація об'єктів *COM* – звичайне правило. Саме це правило дозволяє клієнтам старих версій не перейматися у разі введення нових версій об'єктів *COM*. Нова версія зобов'язана підтримувати і старий *COM*-інтерфейс.

COM задає стандартний двійковий формат, що повинен реалізувати кожний *COM*-об'єкт і для кожного інтерфейсу. Стандарт гарантує, що будь-який клієнт може викликати операції будь-якого об'єкта, причому незалежно від мов програмування, якими написані клієнт і об'єкт.

Структуру інтерфейсу «Іроботазфайлами», що відповідає двійковому формату, пояснює рис. 11.6.

Зовнішній покажчик на інтерфейс (покажчик клієнта) посилається на внутрішній покажчик об'єкта *COM*. Внутрішній покажчик – це адреса віртуальної таблиці, яка містить покажчики на всі операції інтерфейсу.

Перші три елементи віртуальної таблиці є покажчиками на операції, успадковані від інтерфейсу «*Iunknown*». Видно, що на власні операції інтерфейсу «Іроботазфайлами» вказують 4, 5 і 6-й елементи віртуальної таблиці. Така ситуація типова для будь-якого *COM*-інтерфейсу.

Клієнтський виклик обробляється в такому порядку:

– витягається покажчик на необхідну операцію інтерфейсу за допомогою покажчика на віртуальну таблицю;

– надається доступ до її реалізації покажчиком на операцію;
– виконується код операції, який забезпечує необхідну послугу.

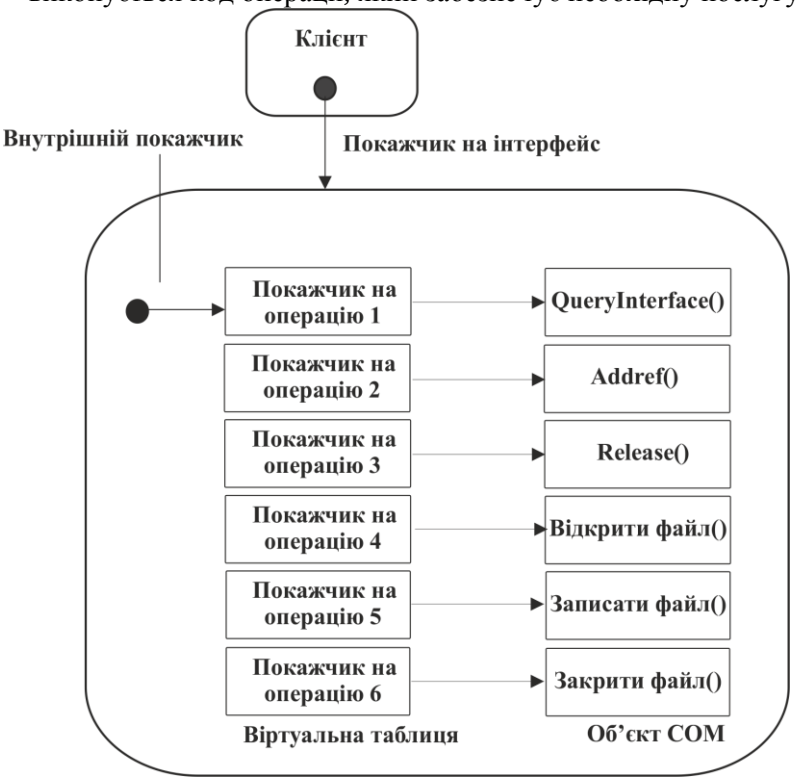


Рис. 11.6. Внутрішня структура інтерфейсу «ІроботазФайлами»

11.3.2. Організація базового інтерфейсу COM

Інтерфейс *IUnknown* містить три операції і надає будь-якому об'єкту *COM* дві функціональні можливості:

– операцію *QueryInterface*(), що дозволяє клієнту отримати покажчик на будь-який інтерфейс об'єкта (з іншого покажчика інтерфейсу);

– операції *AddRef*() і *Release*(), що забезпечують механізм керування часом життя об'єкта.

Свій перший покажчик на інтерфейс об'єкта клієнт отримує під час створення об'єкта *COM*. Порядок отримання інших покажчиків на інтерфейси (для виклику їх операцій) пояснює рис. 11.7, де розписані три кроки роботи. Ідентифікатор необхідного інтерфейсу

(*IID*) задається параметром операції *QueryInterface*. Якщо потрібного інтерфейсу немає, операція повертає значення *NULL*.

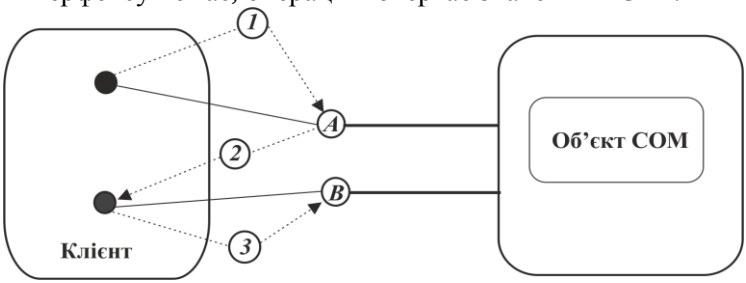


Рис. 11.7. Отримання покажчика на інтерфейс за допомогою *QueryInterface*: *1* – за допомогою покажчика на інтерфейс *A* клієнт запитує покажчик на інтерфейс *B*, викликаючи *QueryInterface* (*IID_B*); *2* – об'єкт повертає покажчик на інтерфейс *B*; *3* – клієнт може викликати операції з інтерфейсу *B*

У сполученні з вимогою незмінності інтерфейсів *COM* операція *QueryInterface* дозволяє:

– розвивати компоненти;

– забезпечувати стабільність клієнтів, що використовують компоненти.

За правилами новий об'єкт *COM* повинен містити також старий інтерфейс *COM*, а операція *QueryInterface* завжди забезпечить доступ до нього.

Фінал існування *COM*-об'єкта полягає в тому, що він повинен сам себе знищити. Це відбувається тоді, коли він перестає бути потрібним усім своїм клієнтам і закінчиться час його існування. Для того є лічильник посилань (ЛП) *COM*-об'єкта.

Правила фіналізації об'єкта *COM* дуже прості:

– у разі видачі клієнту покажчика на інтерфейс виконується ЛП+1;

– якщо викликається операція *AddRef*, виконується ЛП+1;

– якщо викликається операція *Release*, виконується ЛП-1;

– якщо ЛП = 0, об'єкт знищує себе.

Клієнт повинен допомагати цьому:

– у разі отримання від іншого клієнта покажчика на інтерфейс *COM*-об'єкта повинна викликатися в цьому об'єкті операція *AddRef*;

– наприкінці роботи з об'єктом він зобов'язаний викликати опе-

рацію *Release*.

11.3.3. Сервери об'єктів COM

Кожний *COM*-об'єкт існує всередині конкретного сервера. Цей сервер містить програмний код реалізації операцій, а також дані активного *COM*-об'єкта. Один сервер може забезпечувати кілька об'єктів і навіть декілька *COM*-класів. На рис. 11.8 показано використання трьох типів серверів:

1) у процесі (*in-process*), коли об'єкти перебувають у бібліотеці, що підключається динамічно, і, отже, виконуються в тому самому процесі, що і клієнт;

2) локальний (*out-process*), коли об'єкти перебувають в окремому процесі, що виконується на тому самому комп'ютері, що і клієнт;

3) віддалений сервер, коли об'єкти перебувають у *DLL* або в окремому процесі, які розміщені на віддаленому від клієнта комп'ютері.

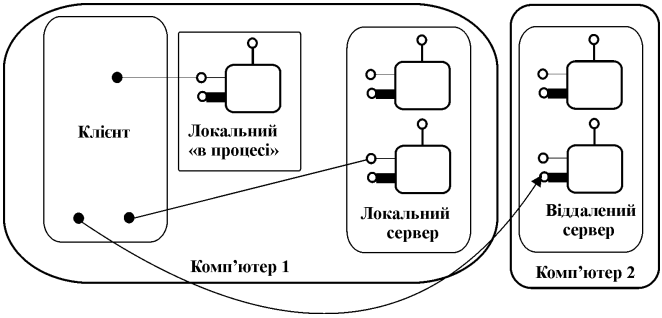


Рис. 11.8. Сервери об'єктів *COM*

Із погляду логіки, клієнту байдуже, у сервері якого типу перебуває *COM*-об'єкт – створення об'єкта, отримання покажчика на його інтерфейси, виклик його операцій і фіналізація виконуються завжди однаково.

Основні переваги *COM*.

1. *COM* забезпечує зручний спосіб фіксації послуг, надаваних різними фрагментами ПЗ.

2. Загальний підхід до створення всіх типів програмних послуг

у *COM* спрощує процес розроблення.

3. *COM* не є важливою мовою програмування, на якій пишуться *COM*-об'єкти і клієнти.

4. *COM* забезпечує ефективне керування зміною програм – заміну поточної версії компонента на нову версію з додатковими можливостями.

11.3.4. Робота з об'єктами COM

Під час роботи з об'єктами *COM* доводиться їх створювати, повторно використовувати, розміщувати в інших процесах, описувати в бібліотеці операційної системи.

Створення об'єкта *COM* ґрунтується на використанні функцій бібліотеки *COM*. Бібліотека *COM*:

– містить функції, що пропонують базові послуги об'єктам і їх клієнтам;

– надає клієнтам можливість запуску серверів *COM*-об'єктів.

Доступ до послуг бібліотеки *COM* виконується за допомогою викликів звичайних функцій. Найчастіше імена функцій бібліотеки *COM* починаються із префікса «Co». Наприклад, у бібліотеці є функція *CoCreateInstance*.

Для створення *COM*-об'єкта клієнт викликає функцію бібліотеки *COM CoCreateInstance*. Параметрами цієї функції є ідентифікатор класу об'єкта *CLSID* і *IID* інтерфейсу, підтримуваного об'єктом. За допомогою *CLSID* бібліотека шукає сервер класу за допомогою диспетчера керування сервісами *SCM* (*Service Control Manager*). Пошук здійснюється в системному реєстрі (*Registry*), що відображає *CLSID* на адресу коду сервера. У системному реєстрі повинні бути зареєстровані класи всіх *COM*-об'єктів. Після закінчення пошуку бібліотека *COM* запускає сервер класу. У результаті створюється неініціалізований *COM*-об'єкт (об'єкт з невизначеними даними). Описаний процес ілюструє рис. 11.9.

Після отримання покажчика на створений об'єкт *COM* клієнт пропонує об'єкту самоініціалізуватися, тобто завантажити себе конкретними значеннями даних. Цю процедуру забезпечують стандартні *COM*-інтерфейси *IPersistFile*, *IPersistStorage* і *IPersistStream*.

Параметри функції «*CoCreateInstance*», використовуваної клієнтом, дозволяють задати тип сервера, який потрібно запустити (наприклад, «у процесі» або локальний). У більш загальному випадку клієнт

може створити декілька об’єктів *COM* одного і того самого класу. Для цього клієнт використовує фабрику класу (*class factory*) – *COM*-об’єкт, здатний генерувати об’єкти одного конкретного класу.

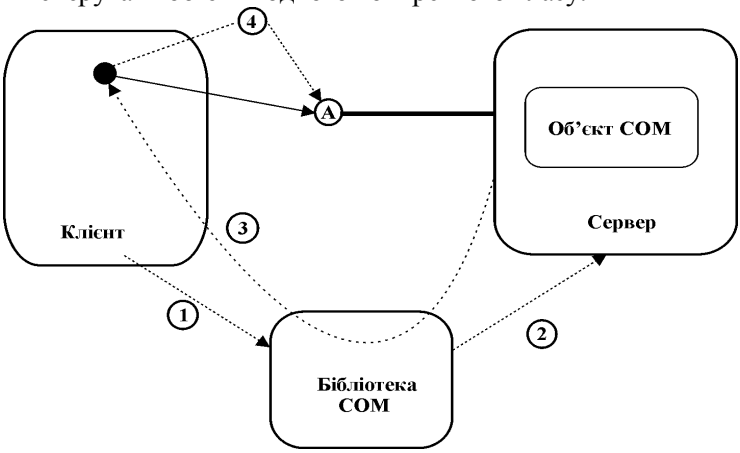


Рис. 11. 9. Створення одиночного *COM*-об’єкта: *1* – клієнт викликає *CoCreateInstance*; *2* – бібліотека *COM* знаходить сервер і запускає його; *3* – бібліотека *COM* повертає покажчик на інтерфейс A; *4* – клієнт може викликати операції *COM*-об’єкта

Фабрика класу підтримує інтерфейс «*Iclassfactory*», що включає дві операції. Операція «*CreateInstance*» створює *COM*-об’єкт – екземпляр конкретного класу, має параметр – ідентифікатор інтерфейсу, покажчик на який треба повернути клієнту. Операція «*LockServer*» дозволяє зберігати сервер фабрики завантаженим до чергового звернення. Клієнт викликає фабрику за допомогою функції бібліотеки *COM CoGetClassObject*:

CoGetClassObject (<*CLSID* створюваного об’єкта>,
< *IID* інтерфейсу *IClassFactory*>).

Третім параметром функції задається тип сервера, що запускається. Бібліотека *COM* запускає фабрику класу і повертає покажчик на інтерфейс «*IclassFactory*» цієї фабрики. Подальший порядок роботи за допомогою фабрики ілюструє рис. 11.10.

Клієнт викликає операцію *IClassFactory*::*CreateInstance* фабрики, параметр якої задає ідентифікатор необхідного інтерфейсу об’єкта

(*IID*). У відповідь фабрика класу створює *COM*-об’єкт і повертає покажчик на заданий інтерфейс. Тепер клієнт застосовує повернений покажчик для виклику операцій *COM*-об’єкта.

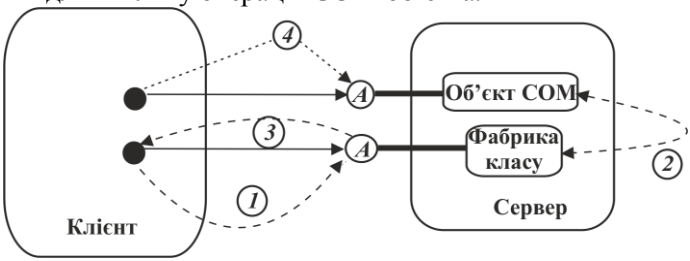


Рис. 11.10. Створення *COM*-об’єкта за допомогою фабрики класу: *1* – клієнт викликає *IClassFactory* :: *CreateInstance* (*IID A*); *2* – фабрика класу створює *COM*-об’єкт і отримує покажчик на його інтерфейс; *3* – фабрика класу повертає покажчик на інтерфейс *A COM*-об’єкта; *4* – клієнт може викликати операції *COM*-об’єкта

Дуже часто виникає така ситуація – існуючий *COM*-клас замінили іншим, який підтримує як старі, так і додаткові інтерфейси, що мають інший *CLSID*. З’являється завдання – забезпечити використання нового *COM*-класу старими клієнтами. Звичайним рішенням є запис до системного реєстру відповідності між старим і новим *CLSID*. Запис виконується за допомогою бібліотечної функції *CoTreatAsClass*: *CoTreatAsClass* (<старий *CLSID*>, <новий *CLSID*>).

Основним засобом повторного використання існуючого коду є спадкування реалізації (новий клас успадковує реалізацію операцій існуючого класу). *COM* не підтримує цей засіб. Причина – у типовому *COM*-середовищі базові об’єкти й об’єкти-нащадки створюються, випускаються і обновляються незалежно. За цими умовами зміни базових об’єктів можуть спричиняти непередбачені наслідки в об’єктах-спадкоємцях їх реалізації. *COM* пропонує інші засоби повторного використання – додавання й агрегування. Застосовуються такі терміни:

– внутрішній об’єкт;

– зовнішній об’єкт.

У разі додавання зовнішній об’єкт є звичайним клієнтом внутрішнього об’єкта. Коли клієнт викликає операцію зовнішнього об’єкта, ця операція, своєю чергою, викликає операцію внутрішнього об’єкта (рис. 11.11). При цьому внутрішній об’єкт нічого не помічає.

Перевагою додавання є простота, недолік – низька ефективність у разі довгого ланцюжка доданих об'єктів.

Недолік включення усувається агрегуванням. Воно дозволяє зовнішньому об'єкту обманювати клієнтів – подавати власні інтерфейси, реалізовані внутрішнім об'єктом. Коли клієнт запитує в зовнішнього об'єкта покажчик на такий інтерфейс, йому вертається покажчик на внутрішній, агрегований інтерфейс (рис. 11.12). Клієнт про агрегування нічого не знає, зате внутрішній об'єкт обов'язково повинен знати про те, що він агрегований.

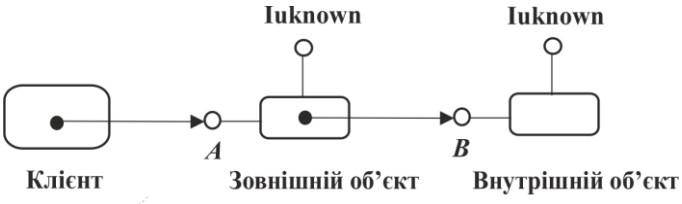


Рис. 11.11. Повторне використання об'єкта *COM* за додаванням

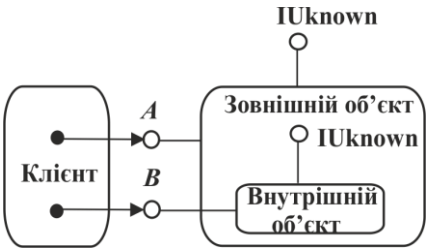


Рис. 11.12. Використання об'єкта *COM* за допомогою агрегування

Це потрібно для того, щоб забезпечити правильний розрахунок посилань і коректну роботу операції *QueryInterface*. Подамо два практичні завдання:

– запит клієнтом у внутрішнього об'єкта (за допомогою операції *QueryInterface*) покажчика на інтерфейс зовнішнього об'єкта;

– зміна клієнтом лічильника посилань внутрішнього об'єкта (за допомогою операції *AddRef*) та інформування про це зовнішнього об'єкта.

За допомогою автономного і незалежного внутрішнього об'єкта їх вирішити не можна. У противному випадку вирішення є простим: внутрішній об'єкт повинен відмовитися від власного інтерфейсу *IUnknown* і застосовувати лише операції *IUnknown* зовнішнього

об’єкта. Інакше кажучи, адреса власного *IUnknown* має бути заміщена адресою *IUnknown* агрегованого об’єкта. Це вказується при створенні внутрішнього об’єкта (додаванням адреси як параметра операції *CoCreate*I*nstance* або операції *IClassFactory*::*CreateInstance*).

Клієнт може містити пряме посилання на *COM*-об’єкт лише в одному випадку, якщо об’єкт *COM* розміщений у сервері «у процесі». У випадку локального або віддаленого сервера, як показано на рис. 11.13, він посилається на посередника.

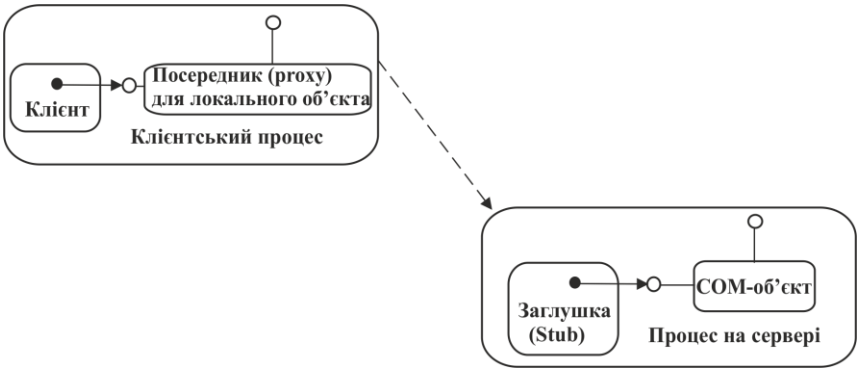


Рис. 11.13. Організація маршалінга і демаршалінга

Посередник – об’єкт *COM*, розміщений у клієнтському процесі і надає йому ті самі інтерфейси, що і запитуваний об’єкт. Запит клієнтом операції через таке посилання приводить до виконання коду посередника.

Посередник приймає параметри, передані клієнтом, і упаковує їх для подальшого пересилання. Ця процедура називається *маршалінгом*. Потім посередник (за допомогою засобу комунікації) надсилає запит у процес, що насправді реалізовує *COM*-об’єкт.

Після прибуття в процес локального сервера запит передається заглушці, яка розпаковує параметри запиту і викликає операцію *COM*-об’єкта. Ця процедура називається демаршалінгом. Після завершення *COM*-операції результати вертаються у зворотному напрямку.

Код посередника і заглушки автоматично генеруються компілятором *MIDL* (*Microsoft IDL*) за *IDL*-описом інтерфейсу.

Крім інформації про інтерфейси, *IDL*-опис може містити інформацію про бібліотеку типу, яка визначає важливі для клієнта характерис-

тики *COM*-об'єкта: ім'я його класу, підтримувані інтерфейси, імена й адреси елементів інтерфейсу.

Приклад 11.2. Об'єкт для роботи з файлами, який складається з трьох частин, описується мовою *IDL*. Перші дві частини описують інтерфейси «Іроботазфайлами» і «Іперетворенняформатів», третя частина – бібліотеку типу «Файлибібл». За першими двома частинами компілятор *MIDL* генерує код посередників і заглушок, третя частина – код бібліотеки типу:

```
--- --- --- --- --- - 1-а частина
[ object,
      uuid(E7 CDODOO-1827-11 CF-9946-444553540000) ]
interface Іроботазфайлами; IUnknown
      {import "unknown.idl"
      HRESULT Відкритифайл ([in] OLECHAR ім'я[31]);
      HRESULT Записатифайл ([in] OLECHAR ім'я[31]);
      HRESULT Закритифайл ([in] OLECHAR ім'я[31]);
      }
--- --- --- --- --- - 2-а частина
[ object.
      uuid(5FBDD 020-1863-11 CF-9946-444553540000) ]
interface ІперетворенняФорматів: IUnknown
      {HRESULT ПеретворитиФормат ([in] OLECHAR ім'я[31],
      [in] OLECHAR формат[31]);
      }
--- --- --- --- --- --- 3-я частина
[ uuid(B253E 460-1826-11 CF-9946-444553540000),
         version (1.0)]
library Файлибібл
{importlib ("stdole32.tlb");
[uuid(B2 ECFAAO-1827-11 CF-9946-444553540000) ]
coclass Софайли
      {  interface ІРоботазФайлами;
         interface ІперетворенняФорматів;
      }
}
```

Опис бібліотеки типу починається з її унікального імені (записується після службового слова *uuid*), потім вказується номер версії бібліотеки. Після службового слова *library* записується символьне

ім’я бібліотеки (Файлибібл). Далі в операторі *importlib* вказується файл зі стандартними визначеннями *IDL – stdole*32.*tlb*. Тіло опису бібліотеки включає лише один елемент – *COM*-клас (*coclass*), на основі якого створюється *COM*-об’єкт. На початку опису *COM*-класу наводиться його унікальне ім’я (це і є ідентифікатор класу – *CLSID*), потім символьне ім’я – «Софайли». У тілі класу імена підтримуваних інтерфейсів – «Роботазфайлами» і «Іперетворенняформаті» (рис. 11.14).

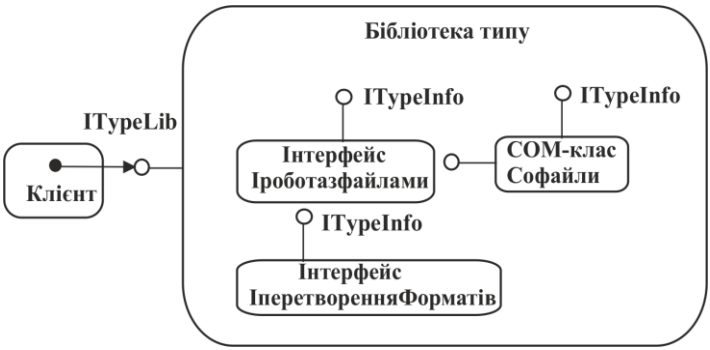


Рис. 11.14. Доступ до бібліотеки типу

Доступ до бібліотеки типу виконується за стандартним інтерфейсом «*ItypeLib*», а доступ до окремих елементів бібліотеки – за інтерфейсом «*Itype*I*nfo*».

Контрольні запитання і завдання

1. У чому полягає основне призначення моделей реалізації?
2. Чим компонент відрізняється від класу?
3. Які форми подання інтерфейсу ви знаєте?
4. Чим корисний інтерфейс?
5. Чим відрізняється вузол від компонента?
6. Де можна використовувати і де не можна використовувати екземпляри компонентів?
7. Для чого використовують компонентні діаграми?
8. Яке призначення *COM*? Які переваги використання *COM*?
9. Чим *COM*-об’єкт відрізняється від зазвичайго об’єкта?
10. Що повинен мати клієнт для використання операції *COM*-об’єкта?
11. Чого не можна робити із *COM*-інтерфейсом? Обгрунтуйте відповідь.
12. Як забезпечити використання нового *COM*-класу старими клієнтами?

13. Наведіть особливості повторного використання *COM*-об’єктів.
14. Назвіть вимоги агрегації до внутрішнього *COM*-об’єкта.
15. Навіщо потрібна бібліотека типу і як вона описується?

IV. ФОРМАЛЬНІ СПЕЦИФІКАЦІЇ І ВЕРИФІКАЦІЯ ПРОГРАМ. МЕТОДИ ПЕРЕВІРКИ І ТЕСТУВАННЯ ПРОГРАМ ТА СИСТЕМ

12. ТЕСТУВАННЯ ПРОГРАМ ТА СИСТЕМ

Тестування – невід'ємна складова процесу програмної інженерії, один з методів подальшого поліпшення якості розробленого ПЗ системи за допомогою виявлення дефектів, не виявлених раніше всіма іншими видами перевірок.

Термін «тестування» широко використовують у літературі з програмування, але визначають по-різному.

Стандарт *ISO*/*IEC*/*IEEE* 24765:2010 [20] визначає термін *testing* у найширшому його змісті як будь-які дії для аналізу ПЗ (включаючи методи динамічної і статичної перевірок).

Г. Майєрс визначає цей термін у найвужчому змісті: «тестування – процес *виконання* програми (або її частини) з метою виявлення помилок... Налагодження – діагностування точної природи відомих помилок і їх виправлення» [21].

Тестування трактується і як випробування (ДСТУ 2844–94 [22], ДСТУ 2853–94 [23]). Однак поняття «випробування» звичніше пов'язують із завершальними стадіями життєвого циклу, на яких виконується не пошук помилок, а підтвердження придатності розробленого програмного продукту. Далі розглядатимемо тестування як процес виконання програмної системи або її елементів для перевірки відповідності встановленим вимогам і виявлення дефектів.

У міру розвитку програмної інженерії розуміння цілей і завдань тестування змінювалося. Розуміння цілей тестування і його місць у життєвому циклі програмної системи з часом змінювалось і має такі періоди розвитку:

1) до 1956 р. – орієнтація на налагодження;

2) 1957–1978 рр. – орієнтація на встановлення відповідності програмної системи вихідним вимогам;

3) 1979–1982 рр. – орієнтація на виявлення дефектів, що залишилися після реалізації;

4) 1983–1987 рр. – орієнтація на аналіз, перевірку і тестування для оцінювання якості програмної системи на всіх стадіях розроблення;

5) 1988–1995 рр. – інтеграція дій з перевірки і тестування в життєвий цикл програмної системи з метою запобігання появі дефектів на всіх стадіях розроблення з охопленням усіх дій з верифікації, валідації і тестування.

У 1995 р. відповідно до стандарту *ISO*/*IEC* 12207 [24], у якому всі дії зі створення програмної системи подано у вигляді окремих процесів життєвого циклу, завдань верифікації, валідації і тестування поділено на окремі процеси. Тестування віднесено до основних процесів, але подано не одним, а групою процесів. Крім того, ряд завдань тестування вирішуються в межах інших процесів розроблення, зокрема, завдання планування тестування поділено на процеси:

– проектування архітектури системи;

– аналіз вимог до ПЗ;

– проектування ПЗ.

Автономне й інтеграційне тестування ПЗ виконуються в межах нерозривно пов’язаних процесів побудови ПЗ і інтеграція ПЗ.

Таким чином, відбулася інтеграція тестування із процесами розроблення. Натепер тестування розглядається як безперервна діяльність, виконувана протягом усього процесу розроблення. Планування тестування має починатися на стадії аналізу вимог, а плани і процедури тестування потрібно систематично і постійно уточнювати в міру розроблення системи.

У *SWEBOK* 2004 р. [25] галузь знань тестування ПЗ подано п’ятьма основними розділами (рис. 12.1).

Тестування полягає в динамічній перевірці поводження програми на скінченній множині тестових даних, обраних з нескінченного вхідного простору, на відповідність установленому очікуваному поводженню.

Динамічність тестування завжди припускає виконання програми, скінченність: навіть для невеликої програми теоретично можна створити таку кількість тестів, виконання яких потребуватиме багато часу. Неповнота – одна з основних проблем тестування,

оскільки на практиці повна множина тестів розглядається як нескінченна. Кількість же тестів, які можуть бути виконані в обмежений термін, зазвичай скінченна. Таким чином, тестування завжди припускає деякий «компроміс» між обмеженими термінами і потенційно необмеженою кількістю тестів. Це призводить до відомих проблем тестування, таких як прийняття рішень про адекватність тестування, і проблем керування – оцінювання витрат (вартості, часу, персоналу) на тестування.

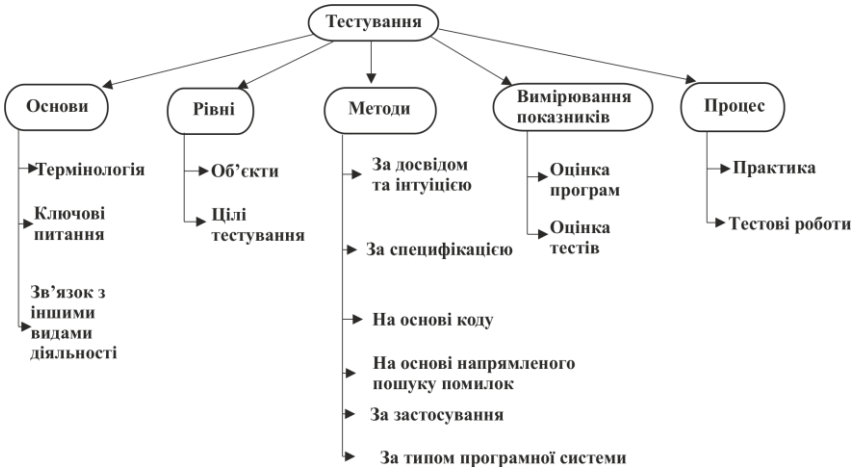


Рис. 12.1. Галузь знань тестування ПЗ відповідно до *SWEBOK*-2004

Проблема адекватності тестування пов'язана з проблемою вибору обмеженої множини тестів. Методи тестування в основному відрізняються підходами до вибору множини тестових даних із вхідного простору. Очікуване поводження дозволяє визначати, чи правильні отримані результати виконання програми, чи відповідає спостережуване виконання очікуванням користувача або специфікаціям.

Інші цілі тестування – зручність застосування, продуктивність і т. ін. Звідси випливають два основні підходи до виконання тестування – деструктивний (негативне, руйнівне тестування) і конструктивний (позитивне або демонстраційне тестування).

Згідно з деструктивним підходом тести вибираються для виявлення дефектів; ефективним вважається тест із високою ймовірністю їх виявлення.

У разі конструктивного підходу тести вибираються для демонстрації відповідності характеристик програмної системи установленим вимогам користувача або цілям якості.

Тестування програмної системи протягом процесу розроблення виконується на декількох рівнях. Для кожного рівня тестування визначається категорія об'єктів тестування (уся система, програмні компоненти, модулі) і набір цілей, що перевіряються (тестованих характеристик).

Цілі та об'єкти спільно визначають критерій вибору множини тестів щодо як його повноти (який обсяг тестування достатній для досягнення встановленої мети), так і його складу (які тести потрібно обрати для досягнення поставленої мети – критерій вибору типів тестів).

На кожному рівні тестування виконується багаторазово, утворюючи цикли тестування – виправлення – повторне тестування.

Розрізнюють поняття «валідація» та «верифікація». Валідація означає забезпечення відповідності продукту вимогам замовника. Верифікація – це забезпечення відповідності кодів проекту.

Тестування охоплює такі основні види робіт:

– розроблення програмного продукту на етапах життєвого циклу та верифікацію результатів на кожному етапі;

– упорядкування плану тестування і підготовки тестів для перевірки окремих елементів розробленої програми та програми в цілому;

– керування виконанням тестів та аналіз результатів тестування;

– повторне тестування.

Кожний тест (тестовий варіант) визначає:

– свій набір вихідних даних і умов для запуску програми;

– набір очікуваних результатів роботи програми.

Повну перевірку програми гарантує вичерпне тестування: перевірка всіх наборів вихідних даних, усіх варіантів їх оброблення, велика кількість тестових варіантів. Вичерпне тестування в багатьох випадках залишається недосяжним через ресурсні обмеження (насамперед, обмеження у часі).

Успішним вважають тестовий варіант з високою ймовірністю виявлення ще не розкритої помилки.

Метою проектування тестових варіантів є систематичне виявлення різних класів помилок за мінімальних витрат часу і вартості.

Тестування забезпечує:

– виявлення помилок;

– демонстрацію відповідності функцій програми її призначенню;

– демонстрацію реалізації вимог до характеристик програми;

– відображення надійності як індикатора якості програми.

Тестування не може показати відсутності дефектів, воно може показувати лише наявність дефектів.

У сучасній програмній інженерії всі види дій, пов'язані з тестуванням, починаючи від планування і до оцінювання результатів, повинні інтегруватися в чітко визначений, документований і контрольований процес тестування. Документування процесу тестування полегшує взаємодію між розробниками, групою тестування і керівництвом проекту, а також дозволяє зробити процес видимим, повторюваним і вимірюваним. У *SWEBOK* ключовими питаннями тестування є такі.

1. Критерії вибору тестів/Критерії адекватності тестів (або правила припинення тестування). Ці критерії можна застосовувати як для створення набору тестів, так і для перевірки адекватності обраних тестів розв'язуваним завданням (тестування), а також для визначення часу припинення тестування.

2. Ефективність тестування/Цілі тестування. Тестування – це спостереження за поводженням програми, виконуваної з метою тестування із заданими параметрами, за заданим сценарієм або з іншими заданими початковими умовами або цілями тестування. Ефективність тесту може бути визначена лише в контексті заданих умов.

3. Тестування для виявлення дефектів. У тестуванні для виявлення дефектів застосовується деструктивний підхід, а успішним вважається тест, що виявляє дефект. Цей підхід принципово відрізняється від іншого підходу, коли тести запускаються для демонстрації того, що програма задовольняє пропоновані до неї вимоги і, відповідно, тест є успішним, якщо не знайдено дефектів.

4. Проблема оракула. Оракул у тестуванні – це будь-який агент (людина або програма), що оцінює поводження програми і формує висновок про результат тесту. Цей висновок істотно залежить від трактування понять «відмова» і «дефект» у конкретному контексті.

5. Теоретичні і практичні обмеження тестування. Обмеження зумовлено неможливістю вичерпного тестування і його економічною недоцільністю для конкретних ПС. Тестування потрібне проводити з урахуванням ризику відмови ПС і ґрунтуватися на таких стратегіях

тестування, що набули поширення в сучасних методологіях розроблення, як тестування, що ґрунтується на ризику (*risk-based testing*), і кероване ризиком тестування (*risk- driven testing*).

6. Проблема нездійсненних маршрутів. Нездійсненні маршрути – це потік керування програми, який не може бути виконаний за жодних вхідних параметрів. Це найскладніша проблема тестування й особливо його автоматизації.

7. Тестопридатність. Цей термін має два різні значення. Перше – це ступінь легкості опису критеріїв тестового покриття для програмної системи, другий – імовірність того, що виконання програми при тестуванні призведе до її відмови за наявності дефекту.

Тестування – основний процес у життєвому циклі, виконуваний завжди, для всіх об'єктів ПЗ системи незалежно від її критичності. Процеси валідації та верифікації у сучасному трактуванні стандартів *IEEE Std*. 1012 [29] і *ISO*/*IEC* 12207 [26] – підтримувальні процеси, які можуть застосовуватися до обраних об'єктів тестування для перевірки планів їх тестування і підтвердження того, що виконане тестування адекватне рівню критичності програмної системи. Стосовно процесу тестування валідація та верифікація виконують контрольну функцію і підтверджують відповідність об'єктів установленим вимогам.

Тестування програмної системи тісно пов'язане з налагодженням і програмуванням, але охоплює набагато більше проблем і учасників – програмістів, тестувальників, аналітиків та інженерів з якості.

12.1. Рівні тестування

Структурування процесу тестування програмної системи за рівнями зазвичай залежить або від типів тестованих об'єктів (модуля, системи або її частин), або з перевірними цілями щодо якості програмної системи (функціональності, безпеки, надійності та ін.) на різних стадіях її створення й експлуатації.

Традиційно виділяють три рівні тестування ПЗ: автономне або модульне, інтеграційне і системне. Стандарт *ISO*/*IEC* 12207 передбачає чотири рівні тестування:

- модульне (у процесі побудови ПЗ);
- інтеграційне (у процесі інтеграції ПЗ);
- тестування ПЗ (як процес);
- системне (у процесі системної інтеграції).

Модульне тестування передбачає перевірку функціонування об'єктів ізольованих один від одного. Воно виконується розробниками з доступом до коду і чергується з налагодженням. Об'єктами тестування є окремі процедури (методи і класи за об'єктного підходу), програмні модулі та програмні компоненти, що складаються із сильно зв'язаних модулів.

Мета модульного тестування – виявлення дефектів у реалізації функцій об'єктів і підтвердження відповідності об'єкта специфікаціям проекту (технічному проекту).

Інтеграційне тестування призначено для перевірки правильності взаємодії між програмними об'єктами, протестованими автономно. Сучасні систематичні стратегії інтеграції визначаються архітектурою і моделлю розроблення (ітераційною зі збільшеннями). Інтеграційне тестування виконується під час кожного складання нової версії для виявлення дефектів в інтерфейсах між інтегрувальними компонентами та підтвердження їх відповідності проекту архітектури програмної системи.

Тестування ПЗ полягає в перевірці функціонування інтегрованої версії ПЗ системи в модельованому середовищі. Мета тестування на цьому рівні – виявлення дефектів у реалізації зовнішніх функцій ПЗ і підтвердження його відповідності специфікації функціональних вимог.

Виділення рівня системного тестування зумовлено необхідністю забезпечення і оцінювання якості сполучення ПЗ з іншими, зокрема, непрограмними компонентами сучасних систем. На цьому рівні тестування перевіряється відповідність показникам якості, установленим у вимогах до системи (з акцентом на нефункціональні вимоги), таким як надійність, стійкість, продуктивність, переносність, а також зовнішнім інтерфейсам з іншими системами, середовищем, апаратним забезпеченням. Більша частина функціональних відмов і дефектів має бути ідентифікована й усунута на попередніх рівнях тестування. Мета тестування на цьому рівні – виявлення дефектів, пов'язаних з незадовільними технічними характеристиками функціонування системи, і підтвердження її відповідності проекту архітектури системи.

Взаємозв'язок рівнів і цілей тестування можна подати у вигляді модифікованої каскадної моделі життєвого циклу (*V*-подібної моделі) [30], що відображає процеси декомпозиції й інтеграції

програмної системи і відповідні рівні тестування (рис. 12.2). Ця модель розглядається як безперервний процес, інтегрований у процес розроблення програмної системи і включає два взаємозалежні підпроцеси – планування тестування в межах процесів розроблення системи (ліве розгалуження на рис. 12.2) і проведення тестування відповідних об'єктів (праве розгалуження).

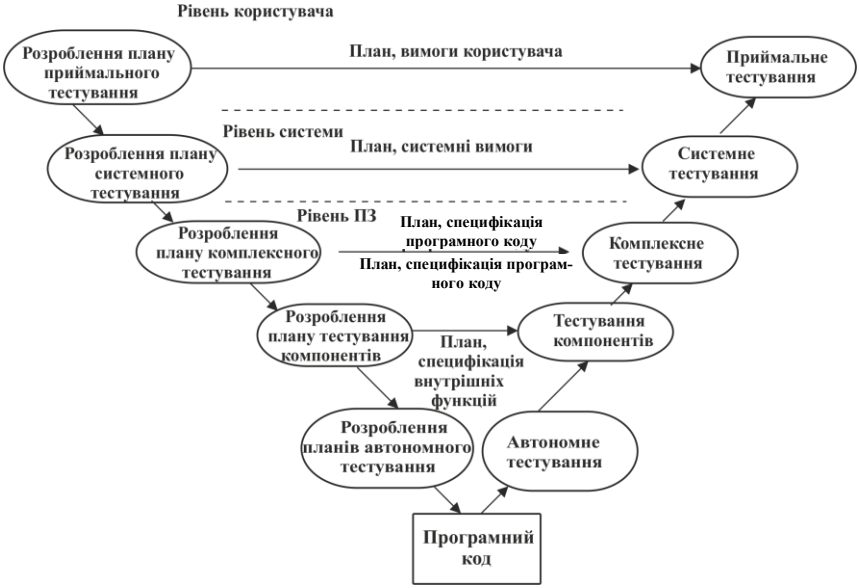


Рис. 12.2. *V*-подібна модель тестування

Для наочності рівні тестування узгоджені з традиційною каскадною моделлю розроблення, але на практиці залежно від архітектури системи, а також моделі розроблення, ці рівні можна поєднувати, а різні завдання тестування – виконувати одночасно.

На практиці завдання тестування ПЗ і системного тестування часто зводяться в єдиний процес, але для складних програмних систем тестування технічних характеристик потрібно виконувати окремо.

Крім розглянутих вище рівнів тестування об'єктів, існують види тестування, виконувані на заключних стадіях її розроблення та в ході експлуатації. Це, насамперед, випробування. У ДСТУ 2853-94 визначено такі види випробувань: попередні, приймальні, настановні та експлуатаційні.

Попередні випробування програмної системи можна вважати найвищим рівнем її «формального» тестування, виконуваного в середовищі розроблення для виявлення можливих дефектів і оцінювання досягнутого рівня якості. Тут «формальне» означає, що тестування виконується відповідно до встановлених документованих процедур і за участю представників замовника. Випробування здійснюються за два прийоми – тестування ПЗ і тестування системи. Отже, мета попередніх випробувань – виявлення невідповідності програмної системи зовнішнім специфікаціям функціональних і технічних характеристик.

Приймальні й усі наступні випробування безпосередньо не пов'язуються з поняттям «тестування», оскільки їх мета – підтвердити відповідність (або невідповідність) системи вихідним вимогам користувача. Відповідно до стандарту *ISO/IEC* 12207 ці випробування виконуються в межах різних процесів життєвого циклу.

Мета приймальних випробувань (приймального, кваліфікаційного тестування) – визначення ступеня відповідності розробленої програмної системи вихідним вимогам замовника (користувача). Ці випробування проводяться винятково в контексті вимог замовника (користувача) і з його особистою участю і, як правило, у середовищі експлуатації. Приймальне тестування відповідно до стандарту *ISO/IEC* 12207 виконується в межах процесів «постачання» і «приймання замовником» (споживачем) і пов'язується із процесом «валідація».

Тестування інсталяції (налагоджувальні випробування) виконуються в середовищі експлуатації (у межах процесу «інсталяція програмної системи») і належать до рівня системного тестування. Включає тестування процедур інсталяції програмної системи із зовнішніх носіїв.

Перед завершенням розроблення системи проводиться *альфа і бета тестування*. Для цього система передається невеликій групі користувачів – внутрішніх (альфа) або зовнішніх (бета), які експлуатують систему й інформують розробника про виявлені проблеми. Виявлені відмови і дефекти свідчать про якість виконаного раніше тестування.

Альфа і бета тестування подібні до експлуатаційних випробувань, виконуваних на етапі дослідної експлуатації системи. Але, на відміну від них, проводяться для систем, розроблюваних для широкого кола користувачів, а не для конкретного замовника.

Далі наводяться види тестування, які не регламентуються планом тестування і безпосередньо не входять у процес тестування.

Регресійне (*regression test*) *і повторне* (*re-test*) *тестування.* Ці поняття в літературі часто ототожнюють, оскільки обидва види тестування стосуються повторного виконання тестів. Однак вони мають деякі відмінності. Повторне тестування виконується на будь-якому рівні тестування для перевірки внесених змін і не регламентується планом тестування. Регресійне тестування пов'язане з розвитком програмної системи і особливо широко застосовується в ітераційних моделях розроблення (для тестування нових версій програмної системи). Воно полягає в повторенні підмножини раніше виконаних тестів, а також у розробленні нових тестів для перевірки правильності внесених змін. На відміну від повторного тестування регресійне тестування планується. Під час його планування вирішується проблема відбору мінімального набору регресійних тестів [23].

Види тестування характеристик програмної системи перевіряються відповідно до характеристик і цілей якості будь-якого об'єкта системи. Можна виділити такі види тестування.

Функціональне тестування (тестування на відповідність або тестування коректності). Спрямоване на перевірку відповідності спостережуваного поводження системи (або окремих елементів) специфікаціям (зовнішнім і внутрішнім). Воно виконується, як правило, у середовищі розроблення за планами, розробленими на основі специфікацій функціональних вимог.

Мета функціонального тестування полягає у виявленні невідповідностей між реальним поводженням реалізованих функцій і вихідних вимог, поданих у специфікації функцій, і визначенні повноти функціонального покриття специфікації функцій. Виконується на рівнях модульного тестування і тестування ПЗ у цілому. Функціональні тести можуть бути отримані за зовнішніми специфікаціями функцій, проектною інформацією або прототипом ПЗ.

Тестування безпеки (*security testing*). Полягає в перевірці можливості несанкціонованого доступу до програмної системи. Виконується моделюванням можливих атак зовнішніх користувачів (у тому числі інших програмних систем) з метою «зламати» систему захисту.

Тестування зручності застосування (ергономічності). Призначається для оцінювання простоти застосування і вивчення системи

кінцевими користувачами, ефективності її використання для вирішення завдань користувача та здатності до відновлення у разі помилок користувача. Часто включає тестування документації (*on-line*, довідкової служби, підсистеми навчання).

Цей вид тестування потребує знань сучасних стандартів з ергономіки, наприклад, *ISO* 9241-210:2010 [32], а також погоджених із замовником вимог до користувацького інтерфейсу. Застосовується в межах процесу аналізу вимог (тестування прототипу), тестування програмних компонентів, що реалізовують інтерфейс, тестування ПЗ і приймальних випробувань системи.

Тестування технічних характеристик. Це процес пошуку невідповідності програмної системи показникам якості (цільовим вимогам), зафіксованим у зовнішніх специфікаціях поряд з описом її функцій. Під час його виконання визначаються кількість показників і принципова досяжність установлених цільових вимог до надійності, продуктивності, сумісності, конфігурації тощо.

Тестування на надійність. Цей вид тестування розглядають як основний засіб поліпшення надійності, оскільки зі збільшенням кількості виявлених і усунутих дефектів надійність програмної системи зростає. Тестування на надійність виконується на підставі наборів даних, обраних випадковим чином із вхідного простору відповідно до операційного профілю [19, 23]. Усі відмови під час виконання тестів реєструються за інтервалами часу між відмовами (або за часом настання відмов від початку тестування). Процес виникнення відмов розглядається як випадковий (стохастичний) процес (марковський або пуассонівський), а для кількісної оцінки досягнутої надійності застосовуються моделі підвищення надійності.

Тестування продуктивності. Виконується для перевірки досягнення встановлених вимог до продуктивності або для оптимізації продуктивності. Іноді поділяється на такі види:

– *навантажувальне тестування* – перевірка працездатності програмної системи за очікуваних нормальних завантажень;

– *тестування на стійкість* – перевірка працездатності в нестандартних умовах (за надмірних і відсутніх завантажень);

– *тестування обсягу* – перевірка внутрішньопрограмних або системних обмежень, пов’язаних, наприклад, з великими масивами даних, великими обсягами баз даних та іншими характеристиками обсягу.

Тестування конфігурації. Виконується для перевірки працездатності програмної системи у різних конфігураціях (операційних системах).

Порівняльне тестування. Вид тестування, за якого один і той самий набір тестів виконується з двома реалізованими версіями, а результати порівнюються. Актуальне в ітераційних моделях розроблення.

Тестування відновлення. Виконується для перевірки процедур відновлення системи після відмов.

Кероване тестами розроблення. Підхід до розроблення, відповідно до якого тести модулів створюються до початку їх кодування. Тести виконують роль прототипу будь-якого модуля або замінника специфікації вимог. Підхід застосовується в екстремальному програмуванні.

12.2. Методи тестування

Традиційно методи тестування поділяють на дві категорії – «чорний ящик» (без доступу до вихідного коду) і «білий ящик» (з доступом до вихідного коду) [14–19; 23].

Більш детальну класифікацію методів тестування, що ґрунтуються на підходах до проектування тестів, показано на рис. 12.3.

12.2.1. Методи тестування, що ґрунтуються на досвіді та інтуїції

Це найпоширеніший підхід, відповідно до якого тести розробляються виходячи з інтуїції тестувальника і його досвіду в тестуванні подібних систем. Ефективність підходу повністю визначається майстерністю тестувальника.

Підхід не потребує детальної специфікації функцій ПЗ, але і не забезпечує оцінювання повноти тестового покриття. Розвитком підходу можна вважати «дослідницьке тестування» (*exploratory testing*), основні принципи якого – поєднання вивчення програмного продукту із проектуванням тестів і їх виконанням. Таке тестування зазвичай виконують незалежні тестувальники для завершених програмних продуктів.

12.2.2. Методи тестування, що ґрунтуються на специфікації

Ці методи широко відомі і детально описані в літературі [14–19; 23]. У традиційній класифікації їх відносять до методів «чорного ящика» і застосовують для тестування зовнішніх та внутрішніх

функцій програми, припускаючи наявність специфікації (формальної або неформальної), використовуваної як еталон. Розрізняються підходами до вибору тестових даних із множини входів (вхідного простору) функцій.

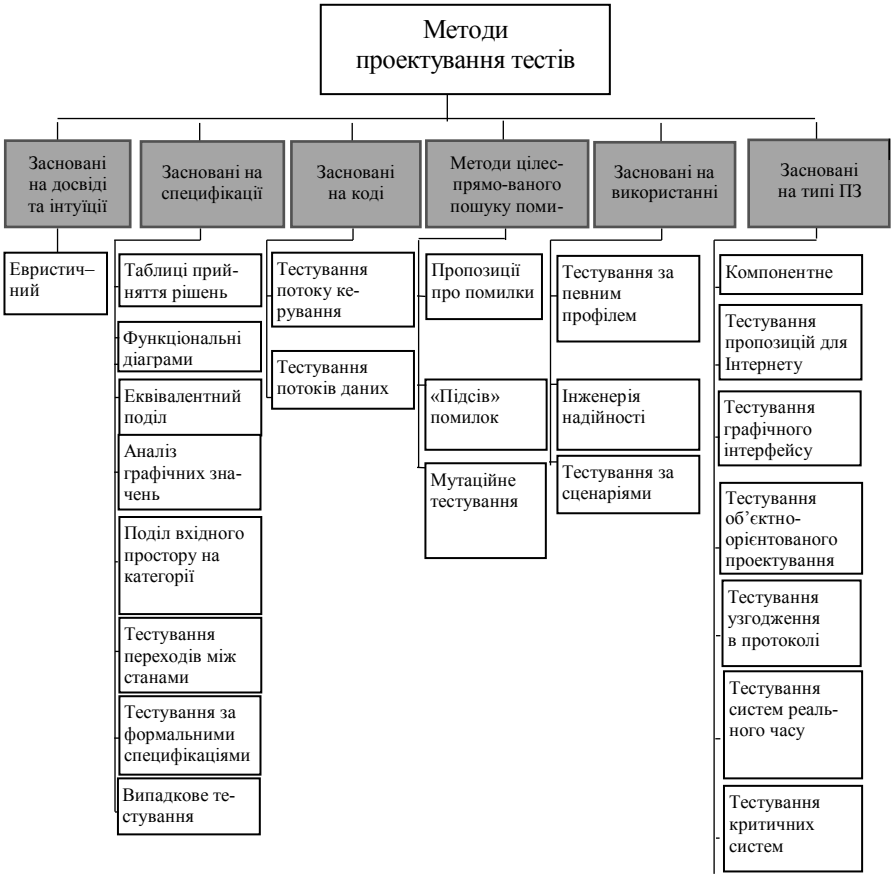


Рис. 12.3. Класифікація методів тестування

До основних методів функціонального тестування належать:
– таблиці прийняття рішень;
– функціональні діаграми;
– еквівалентний поділ;
– аналіз граничних значень;
– поділ вхідного простору на категорії;

– тестування переходів між станами;

– тестування за формальними специфікаціями.

Метод, що використовує таблиці прийняття рішень для проектування тестів, є простим візуальним засобом для верифікації розроблення ПЗ. Стовпчики таблиці містять комбінацію умов, які можуть істотно вплинути на виконання програми. Ці умови ідентифікуються на основі аналізу специфікацій. Потім визначається множина їх можливих значень і встановлюються обмеження щодо їх сумісності. Таким чином, скорочується кількість тестових предикатів (наприклад, обмеження може означати, що якщо умова 1 виконується, то ні умова 2, ні умова 3 не можуть виконуватися).

Метод, запропонований Б. Ельмендорфом і описаний Г. Майєрсом, полягає в перетворенні специфікації у функціональні діаграми [23]. За цим методом спочатку ідентифікується кожна окрема функція, потім визначаються всі причини, що істотно впливають на її поводження, і всі відповідні реагування (наслідки). Наступний крок полягає в побудові графу (діаграми), що пов'язує комбінації причин з очікуваними реагуванням на них. Далі для кожного наслідку, зазначеного на графі, визначаються тестові набори перебиранням усіх комбінацій причин, що породжують цей наслідок.

Хоча жорстке дотримання методу може забезпечити побудова ефективних тестів, він складний у практичному застосуванні для функціонально складних програм, оскільки зі збільшенням причин ускладнюється граф причинно-наслідкових зв'язків, а встановлення обмежень потребує додавання нових проміжних вузлів.

У методі еквівалентного поділу множина значень вхідних даних функції, що утворюють її вхідний простір, поділяється на набір підмножин таким чином, що в кожну підмножину потрапляють значення, еквівалентні один одному з погляду їх використання в тестах для виявлення помилок. Тести, які можуть бути побудовані на підставі еквівалентних значень, належать до одного класу еквівалентності, і для тестування досить вибрати лише найефективніші з них.

Метод аналізу граничних значень ґрунтується на методі еквівалентного поділу. У ньому дані вибираються на межах вхідної області, оскільки багато відмов спричиняються дефектами, що виникають у процесі оброблення граничних значень входів.

Просте розширення цього методу – тестування стійкості, коли тестові дані вибираються також і поза областю для тестування відмовостійкості програми до неприпустимих входів.

Методи еквівалентного поділу й аналізу граничних значень вважаються базовими методами функціонального тестування і застосовуються разом під час проектування набору тестів для кожного рівня тестування.

Розвитком цих методів є метод, що ґрунтується на специфікаціях функцій і використовує для побудови функціональних тестів поділ вхідного простору функцій на певні категорії. Суть методу полягає в ряді послідовних декомпозицій функції, починаючи з вихідної функціональної специфікації, і закінчуючи окремими деталями кожної тестованої процедури, і створенні специфікацій тестів на основі виділення категорій інформації про параметри функції та умови її виконання.

У методі тестування переходів між станами програмний код подається у вигляді моделі, що відображає всі можливі стани її виконання, переходи між цими станами, події, що викликають переходи, і подальші дії з оброблення даних, зумовлені цими переходами.

Опис специфікацій формальною мовою має визначений синтаксис і семантику та дозволяє автоматично будувати на їх основі функціональні тести і водночас забезпечує еталон для перевірки результатів. Існує ряд методів генерації тестів з формальних специфікацій. Згадані методи функціонального тестування належать до систематичних методів на відміну від випадкових (стохастичних) і статистичних методів.

У разі випадкового тестування вхідні дані для тестів вибираються випадково. Як інструмент для генерації вхідних даних можуть застосовуватися датчики випадкових чисел. Для інтерактивних програм тестувальник використовує випадкову комбінацію дій із програмою, намагаючись виявити області її нестійкого функціонування.

12.2.3. Методи тестування, що ґрунтуються на аналізі коду

У традиційній класифікації структурні методи належать до методів «білого ящика». Згідно із цими методами структура програми набуває вигляду напрямленого графу, у якому ідентифікуються потоки керування або потоки даних. Відповідно методи діляться на дві основні категорії: тестування потоку керування (шляхів) і тестування потоків даних. Методи цієї категорії достатньо вивчені і описані в літературі [14–19; 23].

У методах тестування потоку керування дані із вхідного простору вибираються таким чином, щоб забезпечити максимальне покриття коду. Розглянемо основні методи:

Тестування рядків. Вибираються дані, що забезпечують виконання всіх рядків (операторів) програми. Цей метод найслабкіший критерій покриття, названий «покриттям рядків», і прийнятний для програм, що не містять логічних умов і циклів.

Тестування розгалужень. Вибираються дані, що забезпечують виконання шляхів, виділюваних у програмі за допомогою логічних умов, що набувають значення «Так» і «Ні». Для перевірки розгалуження досить двох тестів (якщо $A<B$ і якщо $A > B$):

if ($A<B$) *then*
else
endif.

Для задоволення критерію покриття рядків досить виконати один із цих двох тестів.

Тестування логічних умов. Якщо розгалуження в програмі утворюються в результаті виконання складних логічних умов, дані для їх тестування потрібно вибирати таким чином, щоб перевірити всі значення логічних умов. Наприклад, для повної перевірки логічної умови

if ($A<B$ *and* $C = 1$) *then*
else
endif

необхідно вже чотири тести (один для випадку, коли умова виконується, і три для інших випадків):

$A < B, C = 1$;
$A < B, C \neq 1$;
$A \geq B, C=1$;
$A \geq B, C\neq1$.

Цей метод дає найбільш повний критерій покриття коду програми – критерій *«логічні умови».* Проте для задоволення критерію покриття рядків досить одного тесту, а для покриття розгалужень – двох.

Тестування циклів. Тести розробляються для перевірки кожного циклу на граничних значеннях змінних циклу і в їх середині (для тестування стійкості цикли перевіряються поза граничними значеннями). Цей метод дає критерій покриття, названий «всі цикли».

У методі тестування потоку даних тестові дані вибираються таким чином, щоб відстежити шляхи кожної змінної в програмі від призначення значень до останнього використання (для всіх змінних). Цей метод потребує великої кількості тестів, тому на практиці трасуються лише найбільш критичні значення змінних. Особливий інтерес, наприклад, становлять значення змінних, що беруть участь в операціях ділення (необхідно запобігати можливості перетворення дільника в 0!), умови і цикли, виконання яких залежить від значень змінних. Метод тестування потоку даних може також застосовуватися для пошуку помилок, що виявляються важко, часу виконання.

Таким чином, структурні методи зазвичай застосовуються на рівні модульного тестування програми її розробником і чергуються з налагодженням, хоча можуть використовуватися і у процесах верифікації та валідації для перевірки критичних програм.

Методи функціонального і структурного тестування потрібно розглядати не як альтернативні, а як взаємодоповняльні, оскільки вони використовують різні джерела інформації для побудови тестів і призначені для виявлення різних типів помилок.

12.2.4. Методи напрямленого пошуку помилок

Ці методи використовуються для проектування тестів, спеціально призначених для виявлення помилок деяких типів.

До основних методів цієї категорії:

– припущення про помилки (*error guessing*);

– підсів помилок (*error seeding*);

– мутаційне тестування (*mutation testing*).

Відповідно до методу висування припущень про помилки на підставі «історичної» інформації про помилки, виявлені у подібних програмах, досвіду тестувальника, а також каталогів відомих помилок, складається список можливих помилок і помилкових ситуацій. Приклади можливих типів помилок подано в різних публікаціях. Одним з «класичних» прикладів застосування методу є перевірка ділення на 0. У традиційній класифікації метод належить до категорії методів «чорного ящика».

Методи підсіву помилок і мутаційного тестування призначені для перевірки якості вже виконаного тестування. У традиційній класифікації вони належать до категорії методів «білого ящика».

У методі підсіву помилок у код, протестований на певному наборі тестів, спеціально вноситься невелика кількість помилок, а по-

тім програма повторно тестується. Якщо під час тестування виявляються не всі внесені помилки, набір тестів вважається недостатнім. Відношення кількості виявлених унесених помилок до загальної кількості внесених помилок передбачається приблизно рівним відношенню кількості знайдених реальних помилок до загальної кількості помилок, що утримуються в програмі. Якщо виявлені всі внесені помилки, то вважається, що набір тестів достатній або що ці помилки було надто легко знайти.

У разі мутаційного тестування створюється багато копій основної програми, у кожну з яких вноситься невелика зміна, названа «мутацією», для імітації помилки в операторі або операнді, що не позначається на синтаксичній коректності програми. Кожна копія тестується на одному і тому самому наборі тестів. Якщо в процесі тестування всі внесені зміни виявлені, програма вважається протестованою адекватно і коректно (можуть бути також виявлені раніше не виявлені помилки). Для того щоб метод був ефективним у виявленні помилок, потрібна велика кількість мутантів, що можливо лише в разі автоматичної їх генерації.

12.2.5. Методи тестування, що ґрунтуються на аналізі очікуваного використання

Ця категорія містить два основні методи:

– статистичне тестування;

– тестування за операційним профілем.

За статистичного тестування вхідні дані для проектування тестів вибираються із вхідного простору відповідно до частоти їх появи в майбутніх сценаріях використання програми. Під час виконання тестів фіксуються моменти відмов і обчислюються інтервали між відмовами *MTBF* (*Mean Time Between Failure*) зазвичай в одиницях процесорного часу (*CPU*) або часу виконання. Отримана інформація використовується для оцінювання надійності і прогнозування моменту завершення тестування. Метод застосовується на рівні системного тестування. Для визначення частоти використання функцій програми, а також моментів відмов у код вставляються команди-лічильники.

Тестування за операційним профілем застосовується в межах методології інженерії надійності (*SRE – Software Reliability Engineering*) і називається методом *SRET* (*Software Reliability Engineered Testing*). Методологія *SRE* охоплює повний цикл розроблення програмних сис-

тем з підвищеними вимогами до надійності, а тестування *SRET* (що є по суті статистичним) ґрунтується на пріоритетах входів не лише за частотою, але і з урахуванням критичності функцій, режимів і наслідків відмов систем у процесі експлуатації.

До категорії методів, що ґрунтуються на аналізі очікуваного використання, можна віднести також метод тестування за сценаріями можливого використання, суть якого полягає в ідентифікації можливих сценаріїв роботи користувача і розроблення за цими сценаріями тестування (*test-cases*). Цей метод застосовується у всіх сучасних інструментах автоматизації функціонального тестування програм, що мають інтерфейс користувача (інструменти *Capture-Replay*). Він також використовується в популярній методології *RUP* (*Rational Unify Process*) [33], відповідно до якої сценарії тестування формуються на етапі аналізу і проектування з набором «бізнес-сценаріїв» (*use-cases*).

Тестування об'єктно-орієнтованих програм (ООП). Специфіка тестування ООП пов'язана, у першу чергу, з:

– ітераційним інкрементним підходом до розроблення, що призводить до ускладнення інтеграційного і регресійного тестування;

– подійно-керованою природою ОПП, що потребує тестування не лише структури програми, але і її поводження.

На модульному рівні виконується автономне тестування класів: їх структури і методи (функцій-членів) класу традиційними методами, що ґрунтуються на коді.

На рівні інтеграції класів виконуються тестування ієрархії спадкування і тестування взаємодій між об'єктами. Потрібно перевірити всі можливі зовнішні виклики методів класу – як безпосередні розрахунки, так і виклики, ініційовані отриманням повідомлень. Формальними моделями є модель класів ПЗ і моделі кожного класу.

На рівні системи застосовують тестування за сценаріями використання (*use-cases*) – починаючи з аналізу вимог і проектування ПС. Основні методи (підходи) тестування ООП наведено в табл. 12.1.

Таблиця 12.1

Основні методи тестування ООП

Завдання тестування	Методи/підходи
Розроблення тестових наборів (*test-case*)	Тестування, що ґрунтується на помилках (шаблонах помилок) Тестування, що ґрунтується на сценаріях використання

Закінчення табл. 12.1

Завдання тестування	Методи/підходи
Тестування внутрішньої структури класу	Стохастичне тестування класів. Еквівалентний поділ на рівні класу. Таблиці прийнятих рішень
Тестування взаємодії класів	Стохастичне тестування класів. Еквівалентний поділ на рівні класів. Тестування переходів між станами

Парадигма компонентного програмування і компонентна програмна інженерія – відносно нові напрями, що ґрунтуються на концепціях повторного використання і розподіленого розроблення. Основні особливості побудови системи з існуючих компонентів такі:

– процес розроблення програмної системи з компонентів (фактично складання) відділений від процесу розроблення самих компонентів;

– розроблення програмної системи включає пошук (придбання) потрібних компонентів; їх адаптацію і тестування;

– основні зусилля в життєвому циклі зосереджені на інтеграції («вбудовуванні») компонентів у проектовану систему.

Компонентне тестування складається з декількох рівнів :

– тестування компонента в ізоляції (автономне);

– тестування у процесі складання;

– системне тестування.

Тестування компонента в ізоляції виконується під час створення нового компонента. Крім традиційних методів тестування коректності реалізації, тестуються також можливості повторного використання компонента за різних конфігурацій середовищ.

Тестування web-додатків ґрунтується передусім на трьох основних факторах: моделі життєвого циклу, специфіці об’єктів (архітектурі додатка) і важливості тестування технічних характеристик якості (безпеки, конфігурації, продуктивності і т. ін.).

Тестування графічного інтерфейсу користувача завжди включає тестування зручності застосування, перевірку на відповідність стандартам подання (наприклад, *Windows*), а також тестування конфігурації (на різних типах моніторів). Процес планування і проектування тестів систематизується застосуванням контрольних запитань, які відображають вимоги до інтерфейсу, що підлягають перевірці.

Тестування протоколів. Бурхливий розвиток мережевих технологій, ускладнення їх архітектури, взаємодія декількох різнорідних мереж багаторазово збільшують кількість інтерфейсів, що підлягають тестуванню.

Тестування протоколів полягає у тестуванні:

– відповідності (атестаційне);
– продуктивності;
– спільного функціонування;
– взаємодії;
– функціональності;
– моніторингу.

Тестування на відповідність заданої специфікації – це стандартизований і поширений метод перевірки коректності реалізації протоколу. Стандартизація підходів до тестування відповідності протоколів здійснюється міжнародними організаціями *ETSI*, *ITU-T* і *ISO*. Основний документ – стандарт *ISO* 9646:1994 [33].

Проте одне лише тестування відповідності не може гарантувати повної коректності реалізації, тому що не припускає проведення тестування «під навантаженням» і перевірку поводження системи у всьому діапазоні можливих значень параметрів специфікації. Тому поряд з ним необхідно проводити й інші види тестування.

Тестування спільного функціонування виконується із застосуванням еталонної системи або з використанням системи тестування, що імітує еталонну. Для імітації еталонної системи, з якою має стикатися тестоване устаткування, емулятори протоколів і генератори викликів. Під час використання реальної системи як еталона застосовують аналізатори протоколів, які здійснюють моніторинг інтерфейсу, що поєднує тестовану систему з еталонною.

Під час тестування систем реального часу система завжди подається у вигляді формальної моделі. Тому на модульному рівні застосовуються формальні методи, що ґрунтуються на формальній специфікації. Одним з методів функціонального тестування, спеціально призначеним для систем цього класу, є тестування переходів між станами (*state transition testing*). Обов'язковим також є застосування методів статистичного тестування.

Тестування критичних систем включає не лише динамічні, але і статичні методи верифікації, такі як аналіз критичності, аналіз дерев подій і дерев відмов [34–36], а також інспекції документів і ви-

хідного коду. Для мінімізації вартості тестування і ризику відмови в ході аналізу проектованої системи виділяються критичні модулі, що потребують поглибленого тестування. Для них застосовуються найбільш жорсткі критерії покриття коду.

12.2.6. Дослідницьке тестування

Методи дослідницького тестування призначені для вивчення програми і виявлення в ній помилок одночасно. Ці методи застосовують у стратегіях тестування, що ґрунтуються на ризику.

Один з методів дослідницького тестування і процедура його застосування для тестування функціональності та стійкості програмних продуктів у середовищі *Windows* включає такі кроки.

Крок 1. Дослідження. Вивчення призначення і функцій програмного продукту, типів оброблюваних даних і областей його можливої нестійкості (факторів ризику відмови).

Успіх виконання кроку залежить від наявності інформації про програмний продукт і його потенційних користувачів, а також часу для його вивчення. Для отримання інформації про продукт може використовуватися документація (у тому числі *Help*), відомості про аналогічні продукти, вивчення продукту шляхом його короткочасного використання. Завдання кроку:

– формування списку функцій (ієрархії функцій);

– поділ функцій на основні та другорядні. Приклад критерію поділу за важливістю наведено в табл. 12.2;

– виявлення областей можливої нестійкості продукту (зумовлених відмовами функцій і даних).

Таблиця 12.2

Приклад поділу функцій на основні і другорядні

Визначення	Критерії віднесення
Основна функція – будь-яка зовнішня функція (видима користувачу), відмови в якій означають невідповідність продукту своєму призначенню	Функції, що є визначальними для використання продукту за призначенням. Функції, часто використовувані звичайними користувачами в сеансі роботи. Функції, які можуть використовуватися рідко, але їх відмови призводять до значних негативних наслідків

Закінчення табл. 12.2

Визначення	Критерії віднесення
Другорядна функція – функція, без якої продукт може використовуватися	Рідко використовувані функції, відмови під час виконання яких не можуть мати істотних наслідків

Приклади можливої нестійкості функцій:

– функції оброблення зовнішніх подій, що інтенсивно використовують оперативну пам'ять, надскладні функції, функції, що використовують засоби *Windows* і/або її параметри (налаштування параметрів);

– функції, що маніпулюють конфігурацією *Windows*;

– функції аналізу вхідних даних і оброблення помилок, функції, що замінюють базові функції *Windows* (наприклад, відновлення вилучених файлів);

– функції або групи функцій, що використовують багато паралельних процесів;

– функції, що працюють із багатьма файлами одночасно;

– функції, що працюють із файлами, розміщеними в мережі.

Приклади можливої нестійкості під час оброблення даних: документи (багато одночасно відкритих), багато порожніх і складних записів, багатоколонкові списки, поля з великою кількістю символів, чисельні, великі та складні об'єкти і т. ін.

Крок 2. Проектування тестів. Визначення стратегій виконання тестів і критеріїв оцінювання результатів (наприклад, табл. 12.3).

Таблиця 12.3

Проходження тестів

Мета	Проходження тесту	Відмова
Функціональність – здатність продукту виконувати необхідні функції	Функція виконується відповідно до її призначення	Хоча б одна функція не виконується відповідно до її призначення
	Будь-яке неправильне виконання функції не призводить до наслідків у разі коректного використання	Неправильне виконання призводить до істотних наслідків навіть у разі коректного використання

Закінчення табл. 12.3

Мета	Проходження тесту	Відмова
Стійкість – здатність продукту функціонувати без відмов за підвищених навантажень (за часом, пам'яттю)	Робота продукту за підвищених навантажень не спричиняє руйнування *Windows*. Робота продукту не призводить до втрати даних, «зависання», відмови. Під час тестування не виявлено відмов або будь-яких дефектів у разі виконання основних функцій	Робота продукту руйнує *Windows*. Робота продукту може призвести до втрати даних, «зависання», відмови. Під час тестування принаймні однієї з основних функцій спостерігалися відмови

Крок 3. Виконання тестів. Виконання тестів, спостереження та фіксація результатів і використання цієї інформації для формування думки про роботу продукту. Завдання кроку:

– тестування всіх основних функцій;
– тестування ідентифікованих областей потенційної нестійкості;
– вибіркове тестування другорядних функцій;
– реєстрація відмов;
– фіксація виявлених проблем (для подальшого вивчення).

Крок 4. Побудова евристик. Евристики виражають неформальні правила (що ґрунтуються на досвіді тестувальника і здоровому глузді) прийняття рішення про те, чи вважати роботу продукту прийнятною чи ні, і як надалі продовжувати тестування.

Крок 5. Визначення критерію покриття. У цій процедурі використовуються такі критерії покриття:

– протестовані всі основні функцій;
– протестовані обрані другорядні функції;
– протестовані обрані області можливої нестійкості (варто вибрати від п'яти до десяти функцій і протестувати їх на тестових даних, які можуть призвести до нестійкої роботи продукту).

Розподіл рекомендованого часу тестування: 80 % часу – на основні функції, 10 % – на другорядні і 10 % – на області потенційної нестійкості. Процедура тестування за цим методом виконується циклічно.

Еквівалентний поділ. Суть методу еквівалентного поділу полягає в тому, що вся множина тестів, які теоретично можна побуду-

вати для тестування програми, що реалізує функцію, поділяється на підмножини тестів, які утворять класи еквівалентності, із яких для виконання вибираються лише найбільш представницькі (з найбільшою ймовірністю виявлення помилки).

Належність групи тестів до одного класу еквівалентності може ґрунтуватися на таких практичних критеріях [37]:

– тести включають значення одних і тих самих вхідних даних;

– для запуску тестів виконуються одні й ті самі операції;

– від виконання тестів очікуються однакові результати;

– або жоден з тестів не зумовлює виконання блока оброблення помилок, або всі тести зумовлюють виконання цього блока (у припущенні, що програма взагалі містить блоки оброблення помилок).

Класи еквівалентності поділяються на припустимі класи еквівалентності, що являють собою правильні вхідні дані і вхідні події (події-входи), і неприпустимі класи еквівалентності, що визначають усі інші дані або події. Для подання класів еквівалентності можна скористатися табличною формою (її приклад наведено в табл. 12.4). Для виділення класів еквівалентності можна керуватися такими загальними правилами:

– якщо подія-вхід асоціюється з областю значень (наприклад, як у табл. 12.4), то визначається один припустимий клас еквівалентності і ряд неприпустимих (у таблиці їх п’ять);

– якщо подія-вхід асоціюється зі значенням, що відповідає інтервалу (наприклад, від 1 до 20), то визначається один припустимий клас еквівалентності ($1 \le X \le 20$) і ряд неприпустимих ($X < 1$ і $X > 20$);

– якщо для параметра програми допускається лише визначений перелік значень (наприклад, перелік конкретних найменувань чогонебудь, що зберігаються в базі даних), то визначається один припустимий клас еквівалентності для значень із цього переліку й один неприпустимий (наприклад, для значень, яких немає у переліку). У цьому випадку граничні значення не визначаються;

– будь-який елемент переліку параметрів програми або об’єктів (списки, меню, кнопки панелі інструментів) належать до окремого класу еквівалентності.

Тести розробляються спочатку для припустимих класів еквівалентності, а потім – для неприпустимих:

– для припустимих – по одному тесту всередині класу і по одному на межах класу (наприклад, якщо параметр програми містить

список, потрібно зі списку вибрати перший, останній і будь-який проміжний елемент, а для числових значень у діапазоні 1–100 вибрати 1, 100 і будь-яке число всередині діапазону);

– для неприпустимих – по одному тесту для кожного класу (наприклад, для числових значень у наведеному вище прикладі вибрати 0 і 101, пробіл і нечисловий символ).

Таблиця 12.4

Опис класів еквівалентності

Вхідні події	Припустимі класи еквівалентності	Неприпустимі класи еквівалентності
Уведення числа	Числа від 0 до 9000	Числа менші за 0 і числа більші за 9000. Пробіл. Нечислові символи. Вираз, що обчислюється, результат обчислення якого перевищує припустимий інтервал
Уведення першої букви прізвища	Перший символ повинен бути великою буквою	Перший символ – мала буква і не буква

Аналіз граничних значень. За цим методом тести розробляються для спрямованого тестування класів еквівалентності. Якщо область вхідних даних класу є безперервним діапазоном значень, то вибираються припустимі значення, розміщені на нижній і верхній межах діапазону. Якщо діапазон дискретний (наприклад, елементи списку значень), вибираються перші й останній припустимі елементи.

Аналізуються також усі вихідні класи еквівалентності. Якщо очікувані результати тестів являють собою значення з неперервного діапазону значень, то вхідні дані для їх отримання вибираються так, щоб значення розміщувалися на верхній і нижній межах припустимого діапазону. У випадку, коли результати являють собою набори значень результатів, вхідні дані вибираються для отримання першого й останнього елементів (наприклад, перевіряється виведення на друк першого й останнього рядків звіту).

Відмінність методу аналізу граничних значень від методу еквівалентного поділу полягає в такому:

– для аналізу граничних значень вибираються лише ті елементи в класі еквівалентності, які дозволяють перевірити тестом кожну межу цього класу;

– для розроблення тестів розглядаються не лише вхідні значення, але й очікувані результати.

Поділ вхідного простору на категорії. Процедура застосування цього методу включає таку послідовність кроків.

Крок 1. Декомпозиція функції на функціональні елементи, які можуть тестуватися незалежно. Для більших і складних функцій декомпозиція виконується ієрархічно. Із кожним функціональним елементом зв'язується один або кілька модулів, що його реалізують. Побічним позитивним ефектом виконання цього кроку є аналіз якості декомпозиції проекту за функціональною ознакою.

Крок 2. Ідентифікація параметрів і умов середовища, що впливають на поводження функції. Параметри є входами до функціонального елемента, а умови середовища визначають спосіб реалізації функціонального елемента і характеристики стану системи під час його виконання. Наприклад, для функцій інтерфейсу користувача параметрами можуть бути поля введення, поля виведення, список обраних елементів тощо, а умовами середовища – файли, поля таблиці бази даних та ін.

Реальні тести для функціонального елемента являють собою композицію певних значень параметрів і умов середовища, обраних для максимізації шансів пошуку дефектів у реалізації елемента.

Крок 3. Виділення категорій інформації для кожного параметра й умови середовища. Категорія є основною властивістю (або характеристикою) параметра або умови середовища. У наведеному вище прикладі параметрів і умов середовища для функцій інтерфейсу користувача можна виокремити такі категорії:

– для параметра «поле введення» – тип, розмірність, стан, зміст, обов'язковість та ін.;

– для умови середовища «вікно» – координати, колір вікна, колір символів, рамка та ін.

Категорії вибираються шляхом пошуку в специфікації функції ключових фраз, які описують поводження функціонального елемента за певних зв'язаних параметрів і умов середовища.

Крок 4. Поділ кожної категорії на чіткі альтернативи, які включають різні види значень, можливих для категорії. Альтернативи є класами поділу, з яких будуть вибиратися представницькі елементи для побудови тестів. Приклади категорій і можливих альтернатив параметрам функцій інтерфейсу користувача наведено в табл. 12.5.

Таблиця 12.5

Приклад поділу на категорії та альтернативи

Параметр	Категорія	Альтернативи
Поле вибору	Зміст	Текст Піктограма Значок з написом
	Стан	Поле не зазначене/обране Поле не зазначене/не обране Поле зазначене/обране Поле зазначене/не обране
	Тип поля	Однозначний вибір Багатозначний вибір Розширений вибір
Меню	Спосіб вибору	Вибір курсором Виділена буква/(комбінація букв) Функціональна клавіша Вибір за допомогою клавіатури

Крок 5. Формування формальної специфікації тесту для кожного функціонального елемента. Специфікація тесту складається зі списку категорій і списку альтернатив у кожній категорії. Інформація зі специфікації тесту використовується далі для отримання множини фреймів тесту, що є основою для створення реальних тестових наборів. Фрейм тесту складається з множини виділених альтернатив, причому кожна категорія представлена не більш ніж однією альтернативою (або нулем). Побудована специфікація тесту майже не обмежена, оскільки в ній не враховані можливі відношення між альтернативами. Повна кількість фреймів, згенерованих за такою специфікацією, дорівнює добутку кількості альтернатив у кожній категорії. З метою скорочення кількості можливих тестових фреймів установлюються обмеження для взаємозалежних альтернатив.

Типове обмеження означає, що альтернатива з однієї категорії не може з'явитися у тестовому фреймі з певними альтернативами з інших категорій.

Крок 6. Генерація тестових фреймів. Оскільки процес формування тестових фреймів полягає у перебиранні великої кількості комбінацій альтернатив з урахуванням обмежень, він повинен бути, по можливості, автоматизований. Генерований тестовий фрейм буде містити по одній альтернативі для кожної категорії.

Крок 7. Перетворення тестових фреймів у тестові набори і розроблення тестових сценаріїв. Реальні тестові набори для функціонального елемента є композицією певних значень параметрів і умов середовища, поданих альтернативами відповідних категорій і підібраних з метою максимізації шансів виявлення дефектів у реалізації функціонального елемента.

Тестовий сценарій формується з послідовності зв'язаних тестів для одного або більше функціональних елементів, що потребують однакових умов середовища. Наприклад, один тестовий сценарій для тестування інтерфейсу користувача повинен забезпечувати проходження одного сценарію роботи користувача в програмі у фіксованих умовах середовища на різних типах даних (припустимих, граничних, неприпустимих). Цей метод має ряд переваг:

– дозволяє охопити відразу два основні аспекти тестування – перевірку повноти реалізації функцій і виявлення різних класів дефектів у найбільш уразливих частинах програми;

– функціональний аналіз є невід'ємною частиною методу і виконується одночасно зі специфікацією функціональних вимог (або відразу після її формулювання), забезпечуючи своєчасне усунення дефектів специфікацій;

– хоча специфікація тесту має охоплювати всі можливі категорії інформації і варіанти поєднання альтернатив, застосовуючи механізм обмежень і керуючись практичними міркуваннями, можна керувати обсягом тестування;

– процес перебирання альтернатив і «відсіювання» неможливих або небажаних їх поєднань може бути автоматизований, що позбавить тестувальника від рутинної роботи.

Тестування переходів між станами. За цим методом тести проектуються для перевірки переходів між станами програми (наприклад, зміни зображення на екрані, зміни складу й активності елементів меню тощо). Оскільки кількість можливих станів і переходів між ними може бути набагато більшою, ніж можна протестувати, проектуючи тести, варто керуватися практичними міркуваннями:

– набір тестів повинен охоплювати найбільш імовірні послідовності дій користувачів;

– якщо є залежні стани і синхронізовані події, для кожного з них потрібно розробляти окремі тести;

– окремі тести варто проектувати також для перевірки неприпустимих переходів між станами (для виявлення можливих небажаних побічних ефектів);

– після виконання мінімального набору тестів варто провести набір випадкових тестів (вибираючи довільні режими і сценарії виконання).

Як інструменти для проектування тестів використовують:

– діаграми переходів у стани;

– таблиці переходів у стани.

Цей метод завжди призначався для тестування програм реального часу, але згодом став застосовуватися під час проектування тестів для інтерактивних програм: тестування станів меню (активно, неактивно, обрано), навігації між вікнами, синхронізації елементів інтерфейсу в діалогових вікнах.

Діаграма переходу в стани відображає множину станів у розглянутому контексті, події, які зумовлюють переходи між станами, і можливі дії. Загальний вигляд діаграми показано на рис. 12.4.

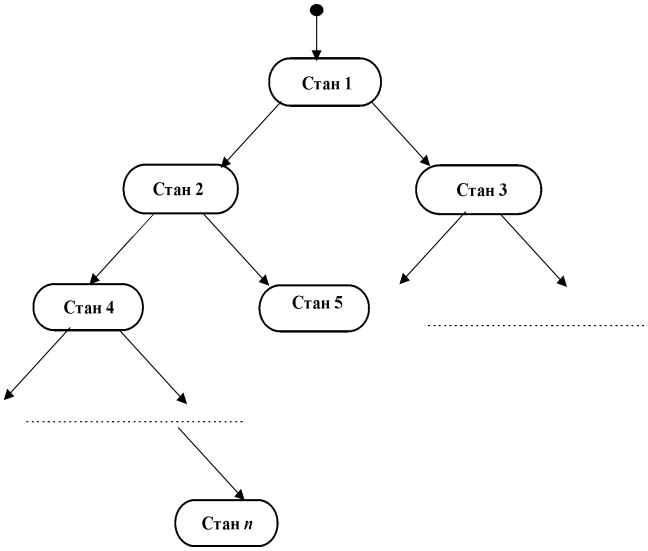


Рис. 12.4. Загальний вигляд діаграми переходів станів

12.2.7. Тестування, що ґрунтується на моделях програмної системи

Як самостійний напрям тестування на базі моделей (*Model-based Testing* (*MBT*)) оформився до середини 90-х років минулого

століття у зв'язку з автоматизацією розроблення тестів [38]. Методи тестування застосовують моделі; для цього напряму характерними є: модель, подана у формальному вигляді, і модель, що застосовується для генерації тестів або еталона.

В енциклопедії [39] тестування на базі моделей визначається як тестування ПЗ, за якого тести розробляються повністю або частково за моделлю, що описує деякі аспекти (як правило, функціональні) тестованої системи.

Тести в цілому утворюють набір, названий абстрактним набором тестів (*abstract test suite*), що безпосередньо не може застосовуватися для тестування, а є основою для створення виконуваного набору тестів, генерованого відповідно до моделі. Сама модель розробляється одночасно з розробленням або безпосередньо для існуючої програмної системи.

Тестування на базі моделей – це узагальнення і розширення відомих «класичних» методів, застосовуваних для автоматичної побудови тестів, наприклад: теорія кінцевих автоматів; переходи між станами; доведення теорем і т. ін.

12.2.8. Тестування web-додатків

Web-додатки можна розглядати як клієнт-серверні додатки, у яких функціональність реалізується як на серверній, так і на клієнтській частині, а користувацький інтерфейс має стандартизовану архітектуру, у якій:

– для взаємодії з користувачем використовується *web*-браузер;

– взаємодія з користувачем чітко поділяється на етапи, протягом яких браузер працює з одним описом інтерфейсу, а ці етапи, своєю чергою, поділяються однозначно локалізовані викликами від браузера до додатка;

– для опису інтерфейсу застосовується стандартне подання;

– взаємодія між браузером і додатком здійснюється за стандартним протоколом (*HTTP*) і чітко формалізована;

– функціональність *web*-додатка розподілена між вилученим сервером і клієнтськими комп'ютерами користувачів.

Таким чином, тестування *web*-додатків виконується на трьох рівнях:

– інтерфейсу користувача;

– сервера (серверів);

– протоколів їх взаємодії.

У цілому тестування ґрунтується на використанні сценаріїв, що описують послідовність дій віртуальних користувачів.

Функціональне тестування виконується як на рівні інтерфейсу додатка, так і на рівні його взаємодії із сервером.

Тестування інтерфейсу зазвичай зводиться до перевірки коректності введення даних, правильності навігації по сайту та інших показниках інтерфейсу і характеризується зручністю застосування *web*-додатка.

Приклад загального контрольного запитальника для перевірки зручності застосування інтерфейсу *web*-сайта, що може бути корисним для планування тестування (опублікований компанією *IT-Online*) наведено в додатку.

Тестування серверної частини *web*-додатків розглянуто в джерелах [40, 41], наприклад, у праці [42] запропоновано оригінальну технологію тестування *UniTesk*.

Поряд з функціональним тестуванням у *web*-додатках підвищується значущість тестування технічних характеристик, насамперед, надійності і відновлюваності, безпеки, продуктивності, конфігурації (у різних браузерах і платформах).

Тестування безпеки охоплює, крім перевірки коректності введення, аналіз таких поширених аспектів «уразливості» *web*-додатків, як механізми аутентифікації, логічні помилки, ненавмисне розкриття інформації, а також традиційні помилки у звичайних додатках. Методологія систематичного тестування охоплює такі аспекти перевірки.

Ідентифікація оточення web-додатка. Містить аналіз інформації про використовувані мови скриптів, *web*-сервера, операційної системи. Для пошуку такої інформації рекомендується:

– проаналізувати відгук на *HTTP*-запити *HEAD* і *OPTIONS* (із заголовка або будь-якої сторінки, що містить рядок *SERVER*);

– досліджувати формат і текст інформації про 404-у помилку сервера (та інших). Деякі системи мають легко пізнавані повідомлення про помилки, а також часто дозволяють довідатися версії використовуваних мов. Можна навмисно запросити сторінки, що призводять до подібних помилок, а також використовувати альтернативні методи запиту (*POST*, *PUT* і т. ін.) для добування інформації із сервера;

– перевірити розпізнавані типи файлів/розширення/каталоги. Багато *web*-серверів по-різному реагують на запити файлів з під-

тримуваними і невідомими розширеннями. Варто спробувати запросити файли зі стандартними розширеннями, такими як *.ASP, .HTM, .PHP, .EXE*, і стежити за появою якого-небудь незвичайного результату або кодів помилок;

– перевірити вихідні тексти доступних сторінок. Вихідний текст сторінки, згенерованої *web*-додатком, може вказувати на використовуване програмне оточення;

– спробувати спотворити вхідні дані для виявлення помилок у скриптах. Це також дозволить отримати інформацію про оточення *web*-додатка;

– використовувати сканування *TCP*/*ICMP* і сервісів. Для цього рекомендується використовувати аналізатори додатків (зокрема, *Amap* і *WebServerFP*) або засоби сканування *Nmap* і *Queso*.

Тестування прихованих елементів форм і розкриття вихідних текстів. Включає перевірку всіх вихідних текстів сторінок на наявність будь-якої корисної інформації, ненавмисно залишеної розробником – це можуть бути фрагменти скриптів, розміщених усередині *HTML*-коду, посилання на підключення чи зв'язані скрипти, неправильно роздані права доступу до критичних файлів з вихідним текстом. Потрібно перевірити наявність кожної програми і скрипта, посилання на які були знайдені. У випадку виявлення їх теж потрібно протестувати.

Визначення механізмів аутентифікації. Необхідно перевірити, як обраний механізм застосовується до кожного ресурсу, використовуваного *web*-додатка. Для цього варто спробувати отримати доступ до всіх ресурсів через кожну точку входу. Багато *web*-систем пропонують засоби підтримання сесій, що ґрунтуються на збереженні в *cookie* або *session-ID* псевдоунікального рядка, що характеризує їх статус. Якщо цей рядок є простим хешем або рядком, складеним з відомих елементів, система може виявитися уразливою до таких видів атаки, як, наприклад, пряме перебирання, повторне відправлення, або спроба відновлення.

Деякі особливості тестування *web*-додатків пов'язані із процесом розроблення. Як правило, для створення *web*-додатків використовуються методи «прискореного» розроблення, у яких:

– тестування вбудовано в життєвий цикл і виконується одночасно з розробленням;

– є невеликі проектні групи (або один *web*-дизайнер) з активною участю замовника;

– дуже стислі терміни розроблення версій обмежують час на ретельне планування і виконання тестування;

– фази інтеграції *web*-додатка зазвичай немає, тому основна увага приділяється функціональному тестуванню додатків у модельованому середовищі і системному – у реальному середовищі;

– стадія супроводження і розвитку – невід'ємна частина життєвого циклу, отже, потрібне постійне регресійне тестування.

Стислі терміни тестування зумовлюють потребу в його автоматизації. Особливо це стосується тестування продуктивності та надійності сервера.

12.3. Методи доведення правильності програм

Роботи з доведення правильності програм появилися ще у 80-і роки [11]. Найвідоміші методи доведення програм:

– метод рекурсивної індукції, індуктивних тверджень Флойда–Наура;

– метод структурної індукції Хоара;

– метод Дейкстрі тощо.

Метод Флойда–Наура. Суть методу полягає у визначенні умови для вхідних і вихідних даних, вибиранні контрольних точок програми так, щоб будь-який циклічний шлях проходив хоча б через одну вибрану точку; для точок формулюються твердження про стан змінних у цих точках (для циклів ці твердження мають бути істинними під час кожного проходження циклу – інваріанти циклу).

Кожна точка розглядається як індуктивне твердження (формула, яка залишається істинною для кожного звернення в цю точку програми і залежить не лише від вхідних та вихідних, а і від проміжних змінних). Далі за індуктивними твердженнями й умовами на аргументи та результати програми будуються умови правильності цієї програми.

Потім для кожного шляху програми між двома точками ставляться у відповідність умови правильності і доводиться істинність усіх цих умов, а також завершення програми за даними, які задовольняють вхідні умови. Доведення правильності програми випливає з доведення істинності умов правильності.

Метод Хоара. Це удосконалений метод Флойда–Наура, що базується на аксіоматичному описі семантики мови програмування, у якому кожна аксіома виражає зміну значень змінних за допомогою

операторів цієї мови. У цьому методі основну увагу приділено формалізації операторів переходу та виклику процедур за допомогою правил виведення, що мають індуктивне висловлення для кожної мітки та функції вихідної програми.

Метод Дейкстрі. Включає два підходи до доведення правильності програм. Перший ґрунтується на моделі обчислень, які оперують історіями обчислень, що виникають у роботі програм. Цей підхід потребує оброблення великого обсягу інформації під час аналізу способів виконання в моделі обчислень.

Другий підхід ґрунтується на формальному дослідженні тексту програми за допомогою предикатів першого порядку. Він орієнтований на клас асинхронних програм, у яких виникають проблеми зі збереженням станів програм під час виконання їх.

Метод Р. Андерсена. Ґрунтується на припущенні, що правильність програм можна доводити як теореми в математиці. В основу методу доведення покладено апарат математичної індукції, суть якого полягає у неформальному доведенні правильності, унаслідок чого можуть допускатися помилки доведення, до того ж не всі помилки в програмі виявляються. Цей метод слід розглядати як системний для перевірки правильності програми по коду.

Метод доведення правильності програм на основі математичної індукції дозволяє довести істинність деякого припущення $P(n)$, що залежить від параметра – цілого числа n, для всіх $n \geq 0$, починаючи з доведення істинності $P(n_0)$. Потім, припускаючи істинність $P(n–1)$ для значення $n–1$, доводиться істинність $P(n)$. Цього достатньо, щоб довести істинність $P(n)$ для всіх $n \geq 0$.

Схема застосування методу полягає в такому. Нехай потрібно довести справедливість деякого твердження A, коли виконання програми досягає визначеної точки. Проходячи через цю точку n разів, можемо визначити справедливість $A(n)$, якщо доведемо:

– справедливе $A(1)$ (тобто справедливе висловлення A під час першого проходження через задану точку);

– справедливе $A(n–1)$ під час $(n–1)$-го проходження через задану точку, то справедливе і $A(n)$ за умови потрапляння в задану точку n разів.

Для доведення правильності програми необхідно описати її функціонування у формі висловлень про правильність або просто тверджень. Якщо припущення, що працююча програма зрештою завершиться, то після її закінчення справедливість твердження A означатиме правильність програми.

Для доведення цими методами застосовується теоретико-множинний підхід, методи верифікації створюваних програм на етапах життєвого циклу, а також тестування отриманих програм на множині тестових даних для встановлення їх правильності.

Переваги і недоліки доведення правильності. Під час конструювання ручного або автоматичного доведення виявляються помилки в коді алгоритму. Техніка доведення додатково забезпечує формальне розуміння програми, оскільки перевіряється основна логічна структура. Регулярне використання цього підходу приводить до більш точного і строгого специфікування даних, структур даних і алгоритмічних правил.

Проте точність досягається нелегко. Багато хто відмовляється від доведення, оскільки, наприклад, алгоритм пузиркового сортування набагато простіший, ніж його логічний опис і доведення. Крім того, великі і складні компоненти можуть включати логічні діаграми, перетворювачі та верифікацію великої кількості частин.

Наприклад, програми для оброблення нечислових даних можуть бути складнішими для розуміння логіки, ніж для числових. Паралельне оброблення також важко перевіряти, структури даних дають результат лише після складного перетворення операторів для такого виконання.

Техніка доведення ґрунтується лише на перевірці того, як вхідні твердження трансформуються у вихідні відповідно до логічних правил. Доведення коректності програми в логічному сенсі ще не означає, що в програмі немає помилок. Справді, ця техніка не розпізнає помилок у проекті, в інтерфейсах з іншими компонентами, в інтерпретації специфікацій, у синтаксисі та семантиці мов програмування або в документації.

Техніка логічного доведення ігнорує структуру і синтаксис мови програмування, у яких тестові програми виконуються. Імовірний випадок, коли ця техніка доводить, що спроектовані компоненти є правильними, але вони не завжди виконуються. Інші техніки беруть до уваги характеристики мов.

12.4. Відмови та помилки

За міжнародним стандартом *IEEE Std* 729:1983 усі дефекти розроблення програм поділяються на помилки, дефекти та відмови.

Помилка (*error*) – це стан програми, ігри, у якому видаються неправильні результати, причиною яких є недоліки в операторах

програми або в технологічному процесі її розроблення, що призводить до неправильної інтерпретації вхідної інформації, а отже, і до неправильного розв'язання.

Дефект (*fault*) у програмі є наслідком помилок розробника на кожному з етапів розроблення і може утримуватися у вхідних або проектних специфікаціях, текстах кодів програм, в експлуатаційній документації тощо.

Відмова (*failure*) – це відхилення програми від функціонування або неможливість виконувати функції, визначені вимогами і обмеженнями, і розглядається як подія, що спричиняє перехід програми в непрацездатний стан у разі помилок або відмов у програмі або в середовищі функціонування.

Відмова може бути наслідком таких причин:

– помилкова специфікація або пропущена вимога, тобто специфікація точно не відображає припущень, які робив користувач;

– специфікація може містити вимогу, яку неможливо виконати;

– проект програми може мати помилки (наприклад, базу даних спроектовано без захисту від несанкціонованого доступу користувача, потрібен захист);

– проект програми може мати помилки (наприклад, опис компонентів містить алгоритм контролю несанкціонованого доступу, що не керується коректно);

– програма може бути неправильною, тобто вона виконує невластивий алгоритм або його виконано неповністю.

Таким чином, відмови, як правило, є результатами однієї або більшої кількості помилок у програмі, а також наявності всіляких дефектів.

Причини появи помилок на етапах життєвого циклу:

– ненавмисне відхилення розробників від робочих стандартів або планів реалізації;

– специфікації функціональних та інтерфейсних вимог виконано без дотримання стандартів розроблення, що призводить до порушення функціонування програм;

– недосконала організація процесу розроблення.

Усі помилки, що виникають у програмах, поділяють на класи:

– логічні та функціональні помилки;

– помилки обчислень і часу виконання;

– помилки введення–виведення і маніпулювання даними;

– помилки інтерфейсів;

– помилки обсягу та ін.

Логічні помилки є причиною порушення логіки алгоритму, внутрішньої неузгодженості змінних в операторах, а також правил програмування. Функціональні помилки є наслідком неправильно визначених функцій, порушення порядку застосування або неповноти реалізації їх тощо.

Помилки обчислень виникають через неточності вихідних даних і реалізованих формул, помилок методів, неправильного застосування операцій обчислень або операндів. Помилки часу виконання виникають через незабезпечення необхідної швидкості оброблення запитів або часу відновлення програми.

Помилки введення – виведення і маніпулювання даними є наслідком неякісної підготовки даних для виконання програми, збоїв під час занесення їх до бази даних або під час зчитування з неї.

Помилки інтерфейсу належать до помилок взаємозв'язку окремих елементів, що виявляється під час передавання даних між ними, а також у разі взаємодії із середовищем функціонування.

Помилки обсягу належать до даних і є наслідком реалізованих методів доступу і розмірів бази даних, які не задовольняють обсяги інформації та інтенсивності їх оброблення в базі даних.

Наведені основні класи помилок властиві багатьом типам компонентів ПЗ і виявляються вони в програмах по-різному. Так, у роботі з базою даних виникають помилки подання і маніпулювання даними, логічні помилки задання прикладних процедур оброблення даних та ін. У програмах обчислювального характеру переважають помилки обчислень, а в програмах керування й оброблення – логічні та функціональні помилки. У ПЗ з множини різнопланових програм реалізації кількох функцій можуть бути помилки кількох типів. Помилки інтерфейсів і порушення обсягу характерні для ПЗ будь-якого типу.

Аналіз можливих типів помилок у програмах є необхідною умовою створення планів і методів тестування для забезпечення правильності ПЗ.

На сучасному етапі розвитку засобів підтримання розроблення ПЗ (*CASE*-технології, об'єктно-орієнтовані методи та засоби проектування моделей і програм) передбачають таке проектування, за якого ПЗ захищається від найбільш типових помилок і тим самим запобігається поява програмних дефектів.

Джерела помилок. Помилки можуть виникати під час розроблення проекту, компонентів, коду, документації, тобто виявлятися в кількох місцях розроблення або супроводження.

Деякі помилки в програмі можуть бути наслідком недоробок під час визначення вимог, проекту, генерації коду або документації. Помилки породжуються також у процесі розроблення програми або інтерфейсів її елементів (наприклад, у разі порушення порядку задання параметрів зв'язку – менше або більше, ніж потрібно тощо).

Причиною появи помилок часто є нерозуміння вимог замовника і неточність їх специфікації в документах проекту. Це призводить до того, що реалізуються деякі функції системи, які будуть працювати не так, як вимагає замовник. У зв'язку із цим замовник домовляється з розробником для усунення непорозуміння й уточнення вимог до системи.

Команда розробників системи може також допустити помилку в синтаксисі та семантиці опису системи. Компілятор відшукує подібні помилки до початку роботи програми. Проте деякі помилки можуть не виявлятися. Наприклад, якщо синтаксис операторів правильний, а деякі індекси або значення змінних цих операторів – неправильні.

Правильно відтрансльована програма може також видавати помилки, якщо для її виконання неправильно задаються деякі граничні умови. Аналогічна ситуація виникає, якщо використовується повторний компонент, значення параметрів звернення до якого задається неправильно.

Виходячи з того, що кожна організація розроблення ПЗ (особливо загальносистемного призначення) стикається з проблемами тестування для пошуку помилок, вона змушена класифікувати типи помилок і визначати своє ставлення до цих помилок. На основі багаторічної діяльності в галузі створення ПЗ багато фірм пропонували класифікацію помилок, характеризуючи їх появу під кількома кутами зору: процес розроблення, функції проектованої системи, типові ділянки і дії. Відомо чимало підходів до класифікації помилок.

Найбільш удалий підхід до класифікації помилок розробила фірма *IBM* [11], який передбачає поділ помилок за категоріями зі ступенем відповідальності за них розробників (рис. 12.5).

Рис. 12.5. Види помилок на етапах життєвого циклу системи

Схема класифікації є продукто- й організаційно незалежна і може застосовуватися на всіх стадіях розроблення ПЗ різного призначення. Перелік помилок відповідно до класифікації (рис. 12.5) наведено в табл. 12.6.

Таблиця 12.6

Ортогональна класифікація дефектів, виконана *IBM*

Контекст помилки	Класифікація дефектів
Функція	Помилки інтерфейсів кінцевих користувачів ПЗ, що зумовлюються апаратурою або глобальними структурами даних
Інтерфейс	Помилки у взаємодії з іншими компонентами, у викликах, макросах, керувальних блоках або у списку параметрів
Логіка	Помилки в програмній логіці, не охопленій валідацією, та помилки, що виникають під час використання значень змінних
Присвоювання	Помилки у структурі даних або в ініціалізації змінних окремих частин програми

Закінчення табл. 12.6

Контекст помилки	Класифікація дефектів
Зациклювання	Помилки, спричинені ресурсом часу, реальним часом або поділом часу
Середовище	Помилки в репозиторії, у керуванні змінами або в контрольованих версіях проекту
Алгоритм	Помилки, пов'язані із забезпеченням ефективності, коректності алгоритмів або структур даних системи
Документація	Помилки в записах документів супроводження або в публікаціях

Ортогональність схеми класифікації полягає в тому, що будь-який її термін належить лише до однієї категорії, тобто простежувана помилка в системі має бути в одному з класів, що дає змогу двом розробникам класифікувати помилки однаковим способом.

Фірма *Hewlett-Packard* установила відсоткове співвідношення помилок, що виявляються у ПЗ на стадіях розроблення (рис. 12.6) [11].

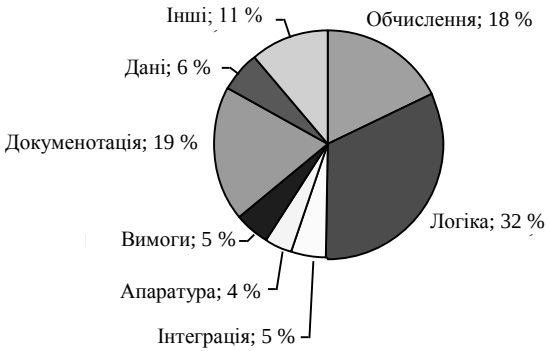


Рис. 12.6. Відсоткове співвідношення помилок, що виникають у процесі розроблення ПЗ

Для перевірки правильності програм спеціально розробляються тести і тестові дані. Під тестом розуміють певну програму, призначену для перевірки працездатності іншої програми і виявлення в ній помилкових ситуацій.

Тестові дані призначені для перевірки роботи системи і готуються по-різному: генератором тестових даних, проектною групою на підставі документів або файлів, користувачем із специфікації вимог тощо. Дуже часто розробляються спеціальні форми вхідних документів, у яких відображається процес виконання програми за допомогою тестових даних.

Тестами перевіряють:

– повноту функцій;
– узгодженість інтерфейсів;
– структуру програми;
– обчислення і коректність виконання функцій;
– правильність функціонування в заданих умовах;
– надійність виконання програм;
– ефективність захисту від збоїв апаратури і невиявлених помилок;
– зручність застосування, супроводження тощо.

Тестові дані готуються як для перевірки окремих програмних елементів, так і для груп програм або комплексів на стадіях процесу розроблення. Класифікацію тестів перевірки за об'єктами тестування на основних стадіях розроблення подано на рис. 12.7.

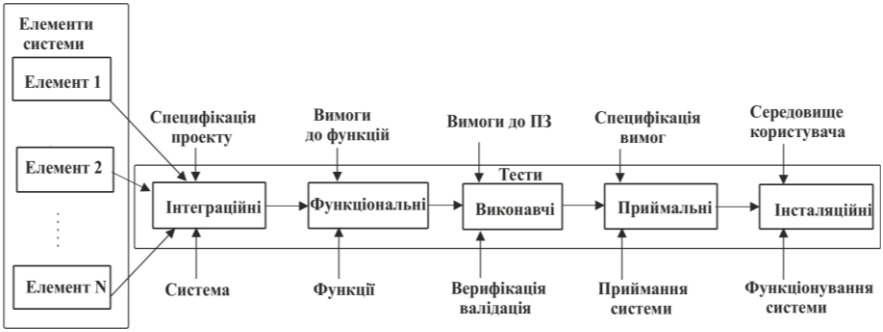


Рис. 12.7. Тести і стадії інтеграційного тестування

Тести інтегрованої системи. Тести для перевірки окремих елементів системи і тести інтегрованої системи мають загальні і відмінні ознаки.

Так, на рис. 12.7 як приклад зображено кроки стадії інтеграції системи з готових після тестування елементів, задано зв'язки між кроками тестування, які інтегруються програмною системою.

Для кожного кроку команда тестувальників готує тести і набори даних, що використовуються для перевірки станів інтегрованої системи і відповідності обмеженням на кожному кроці та вимогам до елемента тестування.

Початковий крок цього процесу – інтеграція елементів, що ґрунтується на інтеграційних тестах, утворюваних за специфікаціями проекту. Розглянемо цей крок детальніше на прикладі схеми інтеграції окремих елементів (рис. 12.8).

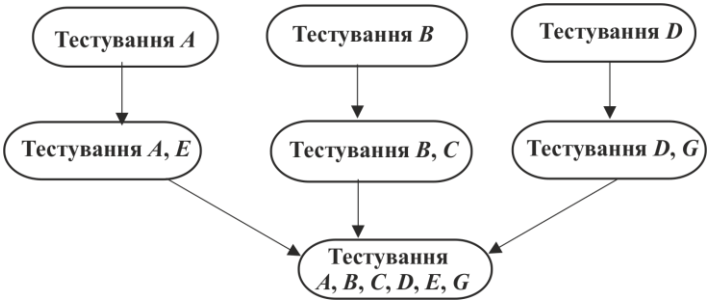


Рис. 12.8. Інтеграційне тестування компонентів

Кожний компонент цієї схеми тестується окремо від інших компонентів за допомогою тестів, які включають набори даних і сценарії, складені відповідно до їх типів і функцій, специфікованих у проекті системи. Тестування проводиться в контрольному середовищі на визначеній множині тестів їх даних та операціях, виконуваних з ними.

Тести забезпечують перевірку внутрішньої структури, логіки і межових умов виконання кожного компонента.

Відповідно до схеми (див. рис. 12.7) спочатку тестуються компоненти *A*, *B*, *D* незалежно один від одного і кожен з окремим тестом. Після їх перевірки виконується такий етап, як перевірка інтерфейсів для їх наступної інтеграції, суть якої полягає в аналізі виконання операторів виклику $A \to E$, $B \to C$, $D \to G$ на нижчих рівнях графу: компоненти *E*, *C*, *G*. Передбачається, що зазначені компоненти, які викликаються, так само мають бути налагоджені окремо. Аналогічно перевіряються звертання до компонента *F*, що є ланкою, яка пов’язує елементи.

При цьому можуть виникати помилки в разі неправильного задання параметрів в операторах виклику або коли обчислюються

процедури чи функції. Знайдені помилки у зв'язках усуваються, і повторно перевіряється зв'язок з компонентом F у вигляді «трійок»: компонент – інтерфейс – компонент.

Наступним кроком тестування системи (рис. 12.8) є перевірка функціонування системи за допомогою функціональних тестів, які включають сценарії перевірки функцій і вимог до них. В основу цього тестування покладено проектні специфікації та функціональні вимоги до системи.

Після перевірки системи за функціональними тестами перевіряється система за допомогою виконавчих і випробувальних тестів, що ґрунтуються на вимогах до ПЗ, до апаратури і виконуваних функцій. Випробувальному тесту передує верифікація і валідація ПЗ.

Тест для перевірки роботи всієї системи є складнішим, до нього включаються набори даних для перевірки функцій на відповідність їх описів та вимог до них. Тест випробувань системи відповідно до вимог замовника перевіряється в реальному середовищі, у якому система буде використовуватися.

Тест інсталяції системи використовується для перевірки виконання всієї системи в середовищі користувача.

12.5. Команда тестувальників

За функціональні і виконавчі тести відповідає розробник, а замовник більше впливає на складання випробувальних та інсталяційних тестів. Як правило, команда тестувальників не залежить від штату розробників програмних систем. Деякі члени цієї команди є досвідченими тестувальниками або навіть професіоналами в цій галузі. До них належать аналітики, програмісти, інженери-тестувальники, які використовують увесь час для вирішення проблем тестування систем. Вони працюють не лише зі специфікаціями, а і з методами та засобами тестування, організовують створення і виконання тестів на машині. Тестувальників включають у процес розроблення з початку створення проекту для складання тестових наборів та сценаріїв, а також графіків виконання тестів.

Фахові тестувальники розробляють методи і процедури тестування. До цієї команди залучаються додаткові люди, ознайомлені з вимогами системи або з підходами до її розроблення. Аналітиків включають як членів команди, оскільки вони розуміють проблеми визначення специфікацій замовників.

Проектувальники системи подають команді тестувальників проектні завдання, тому що їм відомі принципи декомпозиції системи на підсистеми і функції, а також принципи їх роботи. Після проектування тестів і тестових покриттів команда тестувальників залучає проектантів для аналізу можливостей системи.

До команди тестувальників входять також користувачі. Вони оцінюють отримані результати, зручність використання і людський фактор, а також висловлюють свою думку про принципи роботи системи на початкових етапах проекту.

Уповноважені замовника планують роботи для тих, хто буде використовувати і супроводжувати систему. При цьому вони можуть внести деякі зміни в проект у зв'язку з неповнотою заданих вимог і сформулювати системні вимоги для проведення верифікації системи і прийняття рішення про її готовність та корисність.

12.6. План тестування

Для проведення тестування створюється план (*test plan*), у якому описуються стратегії, ресурси і графік тестування окремих компонентів та системи в цілому. План стратегії містить визначення місця тесту в кожному процесі, ступінь покриття програми тестами і відсоток тестів, що виконуються зі спеціальними результатами. Тестові інженери створюють множину тестових сценаріїв (*test cases*), кожний з яких перевіряє результат взаємодії між актором і системою на підставі визначених передумов та постумов використання таких сценаріїв. Сценарії належать здебільшого до тестування за типом «білий ящик» і орієнтовані на перевірку структури та операції інтеграції компонентів системи.

Для проведення тестування тестові інженери пропонують процедури тестування (*test procedures*), що включають валідацію об'єктів та верифікацію тестових сценаріїв відповідно до планграфіка. Оцінювання тестів (*test evaluation*) полягає в оцінюванні результатів тестування та ступеня покриття програм сценаріями і статусу отриманих помилок. Відповідальність та обов'язки тестового інженера ілюструє рис. 12.9.

Тестувальник інтеграції системи проводить тестування інтерфейсів і оцінює результати виконання відповідних тестів. І, нарешті, тестувальник системи є відповідальним за виконання системних тестів і тестування, проведеного попередніми членами команди.

Під час виконання системних тестів, як правило, знаходять дефекти, що є результатом прихованих помилок у програмах, які виявляються під час тривалої прогонки системи за тестовими даними і сценаріями.

Рис. 12.9. Завдання та обов'язки інженера тестів

Засоби автоматизації процесу тестування. Дотепер розроблено безліч різноманітних засобів пошуку помилок, дефектів та відмов у програмах. За призначенням ці засоби поділяють відповідно до типів тестування (статичного, динамічного та ін.) ПЗ. Усі засоби об'єднуються базою даних проектування системи. Вона містить усі компоненти, налагоджувальні контрольні дані і тести, а також результати тестування та інформацію про документування системи та процесу тестування.

Засоби статичного тестування. До них належать засоби розрахунку часу виконання модулів та їх характеристик. Вони дають змогу отримувати середні значення і розподіляти розрахунки тривалостей аналітично без виконання програми на машині. За допомогою розрахунків виявляються компоненти програми, які потребують багато часу для виконання і перевірки виконуваності ПЗ у реальному часі. Ці дані дозволяють знайти деякі помилки порушення надійності функціонування через невідповідність часу виконання ПЗ потребам реального часу.

*Засоби динамічного налагодження і тестування***.** Їх можна поділити на два типи. Перший тип безпосередньо забезпечує виконання програм відповідно до тестових завдань, другий – допоміжні засоби, які обчислюють результати виконаного тестування і коригують програми. До другого типу належать засоби трансляції завдань, виконання програм і реєстрації даних про результати тестування. Тестові значення перетворюються у форму, придатну для виконання налагодженої програми. Оператори налагодженого

завдання об'єднуються з тестованою програмою або готуються для виконання в режимі інтерпретації.

Засоби керування виконанням. Ці засоби реалізують виконання програми за налагодженим завданням. У процесі оброблення завдань і тестів обираються результати відповідно до завдання.

Під час оброблення тестів проводиться селекція результатів тестування відповідно до визначених операторів, а також зіставлення їх з еталонними значеннями. Деякі результати зберігаються для подальшого їх використання.

У процесі виконання програм у режимі тестування результати відображаються на дисплеї, наприклад у графічній формі (шляхи проходження по графу програми), у вигляді послідовності діаграм *UML*, а також інформації про відмови та помилки або конкретні значення вихідних параметрів програми. Ці дані аналізуються розробниками для формулювання висновків про напрями подальшої перевірки правильності програм або завершення їх.

Засоби планування. Засоби автоматизації планування тестування призначені для підготовки інформації трьох видів:

1) про цикли;

2) про способи виконання програми;

3) про предикати, які керують виконанням маршрутів, і про межі змін параметрів та змінних.

Такі засоби спочатку будують стратегію і способи тестування відповідно до умов у предикатах за винятком виконання циклів. Перевіряються реальні значення імовірного розгалуження у вершинах графу та характеристик ітерації циклів. Цикли, як правило, перевіряються окремо з використанням вхідних даних про ітерації. Після перевірки вони вилучаються з виконання програми. Під час планування готуються тести, критерії і вхідні значення.

Наведені дані використовуються засобами автоматизації для багаторазового проходження тестів за виконуваною програмою.

Засоби документування результатів тестування. Відповідно до стандарту *IEEE Std* 829^{TM} результати налагодження і тестування програмних систем на всіх етапах життєвого циклу беруть зі спеціальних документів, які створюються автоматично і містять:

– загальний опис завдань, призначення та зміст програмної системи, а також опис її функцій відповідно до вимог замовника;

– опис технології розроблення системи;

– опис планів тестування різних об’єктів, необхідних ресурсів, відповідних спеціалістів для проведення тестування і технологічних засобів;

– специфікацію тестів, контрольних прикладів, критеріїв та обмежень оцінювання результатів програмного продукту, а також процесу тестування;

– звіт тестування, звіт про аномальні події, відмови та дефекти з підсумковими результатами тестування компонентів і системи в цілому.

Отже, у тестуванні використовується багато спеціальних засобів, які дають змогу поліпшити показники створюваної системи. Крім того, є окремі засоби, орієнтовані на тестування таких складних систем, як критичні системи та системи реального часу тощо.

12.7. Принципи розроблення тестових кейсів

В останні роки намітилася тенденція, коли зусилля фірм-розробників ПЗ спрямовані на підвищення якості своїх програмних продуктів. Поступово виробники відмовляються від «інтуїтивного» тестування програм і переходять до формального тестування з написанням тест-кейсів.

Повністю протестувати програму неможливо з таких причин:

– кількість можливих комбінацій вхідних даних надто велика, щоб її можна було перевірити повністю;

– кількість можливих послідовностей виконання коду програми також надто велика, щоб її можна було протестувати повністю.

– інтерфейс програми (що включає всі можливі комбінації дій користувача і його переміщень по програмі) зазвичай надто складний для повного тестування.

Характеристики якісного тесту:

– існує обґрунтована імовірність виявлення тестом помилки;

– набір тестів не повинен бути надмірним;

– тест має покривати базові маршрути і бути реалізованим.

Натепер спостерігається кілька методологій розроблення тест-кейсів. Вони відрізняються теоретичними підходами і практичною реалізацією.

Найбільш часто застосовується методологія розроблення тестових випадків – методологія, джерелами тестових випадків якої є варіанти використання.

12.7.1. Розроблення тестових випадків на підставі сценаріїв використання

Сценарій використання складається з певної кількості варіантів: нормального варіанта, розширеного і виняткових варіантів завершення. Для розроблення тест-кейсів на підставі одного випадку використання розробляються кілька сценаріїв. Сценарій використання являє собою оптимістичний сценарій, який вибирається найчастіше. У розділ «Альтернативні маршрути» можуть включатися кілька сценаріїв, відмінних від сценарію використання в різних аспектах, проте залишаються повноцінними маршрутами виконання. У розділ «виняткові маршрути» потрапляють ті сценарії, які призводять до виникнення помилок. Кожен сценарій передбачає дії діючого суб'єкта і потребує від системи відгуку, який відповідає основній частині тестового варіанта. Тестовий варіант складається з деякого набору передумов, стимулювального впливу (вхідні дані) і очікуваного відгуку.

У процесі розроблення потрібно визначити необхідну кількість тестових наборів з кожного варіанта використання і побудувати їх. Першим кроком визначення кількості тестових наборів, що припадають на один варіант використання, є побудова профілів використання.

Профіль використання системи – це упорядкування індивідуальних випадків використання, в основу якого покладено деяке поєднання значень частоти використання і критичності для окремих випадків використання.

Комбінація рейтингів частоти використання і критичності, що застосовується для упорядкування варіантів використання, забезпечує отримання певного критерію якості. Наприклад, можна зобразити емблему в правому нижньому кутку кожного вікна. Аналогічно з'єднання із сервером локальної бази даних відбувається вкрай рідко, однак невдале виконання цієї операції унеможливить успішне виконання множини інших функцій. Кількість тестових випадків, що припадають на один випадок використання, вибирається залежно від положення цього випадку використання в рейтинговій таблиці (чим частіше трапляється такий випадок використання і чим критичніше його неправильне виконання для системи, тим більше тест-кейсів має бути розроблено). На цьому етапі тестування підтримується проведення тестування програми в такому режимі, у якому воно буде використано на практиці.

Побудова профілів використання починається з визначення діючих суб'єктів на діаграмі випадків використання. Там, де є один діючий суб'єкт, значення профілю має відповідати значенню поля частоти використання у випадках використання. Але корисними є випадки, у яких участь беруть кілька діючих суб'єктів.

Дуже рідко всі ці діючі суб'єкти використовують систему одним і тим самим способом. Поле частоти випадку використання являє собою композицію частот використання окремих профілів діючих суб'єктів.

Цей підхід корисний для систем, які жодного разу не встановлювалися. Він більш точно оцінює, як діючий суб'єкт використовуватиме систему порівняно з простим угадуванням сукупного результату окремих випадків використання.

Випадок використання зазвичай містить численні сценарії, які можуть бути перетворені в тестові випадки.

Розроблення сценарію для випадку використання передбачає виконання таких дій:

– ідентифікування всіх значень, які вводяться діючими суб'єктами, що містяться в моделі випадку використання;

– виділення класів еквівалентності значень кожного типу вхідних даних;

– побудову таблиць, які містять список комбінацій значень з різних класів еквівалентності;

– побудову тестових варіантів, у яких поєднується одна перестановка значень із необхідними зовнішніми обмеженнями.

Як приклад можна розглянути певну систему керування персоналом; у ній використовуються три змінні. Кожен працівник поданий у системі іменем і змінними, що показують, чи є він новим працівником фірми, чи вже працює в ній протягом певного часу, і рівнем повноважень, санкціонованих системою безпеки.

12.7.2. Переставляння вхідних даних системи керування персоналом

На практиці кількість тестових випадків може бути обмежена, якщо брати до уваги важливість того чи іншого випадку використання або обсяг доступних системних ресурсів. Можна зробити спробу або скористатися відповідними статистичними даними для визначення потрібного обсягу ресурсів для виконання типового ви-

падку використання. Якщо відома кількість випадків використання, то можна отримати оцінку трудовитрат, необхідних для реалізації проекту в повному обсязі.

У тестуванні складних систем одне з найважчих завдань полягає у визначенні результатів, очікуваних від прогону конкретного тесту. Телекомунікаційні системи, ПЗ керування космічним кораблем, інформаційні системи багатонаціональних корпорацій – це випадки систем, для яких побудова тестових даних і тестових результатів обходиться дуже дорого. Деякі методи розроблення тест-кейсів можуть виявитися корисними для зниження витрат зусиль на розроблення і опис очікуваних результатів. Перший з них передбачає побудову результатів в інкрементальному режимі. За умовами цього підходу тестові випадки створюються для покриття деякого набору випадків використання системи, можливо, лише деяких процедур уведення даних. У наступних випадках покриття розширюються з метою перевірки використання системи в повному обсязі. Із розширенням тестових варіантів, тестові результати теж розширюються.

Тестові випадки розширюються в ітеративному режимі. Тобто починається написання тест-кейсів з опису невеликих тестових випадків, після чого поступово збільшуються розміри і підвищується складність тестових випадків. Цей процес триває доти, доки тести не стануть реалістичними з позицій промислового середовища. У системі керування базами даних можна почати з бази даних, що містить 50 записів, і поступово збільшувати їх кількість до кількох тисяч. Результати, очікувані на кожному новому рівні, повинні включати будь-які взаємодії, які виникають через появу нових випадків використання. Наприклад, наявність одного запису може перешкоджати вибору іншого, обраного у процесі виконання попереднього тесту.

Другий підхід полягає в розробленні тестових наборів великого циклу, у якому кожен тестовий випадок генерує дані, які є входом для наступного тестового випадку. За умовами такого підходу кожен тест переносить тестові дані через весь життєвий цикл. Отриманий при цьому стан бази даних використовується як вхідний для наступного тесту. Цей метод особливо ефективний для тестування життєвого циклу після того, як тестування нижнього рівня дозволило виявити більшу частину дефектів, спричинених відмовами. Якщо вибудувати тестові випадки у відповідну послідовність, то

після успішного виконання першого тестового варіанта встановлюється такий стан програми, який очікується як вхідний для другого тестового випадку. Очевидна проблема в умовах окресленого підходу полягає в тому, що невдале виконання першого тестового варіанта залишає програму в стані, який не очікувався, у результаті чого не можна виконувати прогін другого тестового випадку або навіть повернути програму в робочий стан.

Отже, розробляти тест-кейси на основі варіантів використання – процедура традиційна. Рекомендується проводити такі види тестування.

Тестування на відповідність функціональним вимогам. Перевірка якісних системних атрибутів. Організація розроблення ПЗ передбачає методи підтвердження всіх системних вимог, включаючи і претензії щодо надання програмному продукту особливих якостей. Є два види претензій, з якими може зіткнутися програма під час розроблення продукту. Перший вид претензій становить інтерес лише для організацій, які розробляють програмний продукт. Наприклад, твердження, що програмний код допускає багаторазове використання. Другий тип претензій цікавить користувачів системи. Наприклад, твердження про те, що система є більш повною, ніж інші системи подібного класу, що пропонуються на поточний момент на ринку програмного продукту. Цілком зрозуміло, що не всі ці претензії можуть підлягати перевірці через тестування. Однак на це слід звернути увагу.

Тестування механізму розгортання системи. Природне розширення тестування механізму розгортання системи полягає в додаванні в тестований програмний продукт функціональних засобів самоперевірки. Вважається, що система «зношується» в часі через зміни, що відбуваються під час взаємодії з оточенням, приблизно так само, як з часом зношується механічна система через тертя між її компонентами. У міру того, як встановлюються більш нові версії стандартних драйверів і бібліотек, невідповідності зростають разом зі збільшенням імовірності виникнення відмов. Кожна нова версія *DLL*-бібліотеки уможливлює появу нових областей нестикувань стандартних інтерфейсів або появу стану гонок між цією бібліотекою і додатком. Функціональні засоби самотестування мають забезпечувати виконання тестів, які досліджують роботу інтерфейсів між цими програмними продуктами.

Тестування системи безпеки. Розробляючи тест-кейси на підставі випадків використання, необхідно звернути увагу на всі ці аспекти функціонування ПЗ.

Контрольні запитання і завдання

1. Визначте поняття «тестування».
2. Що таке тест? Поясніть зміст процесу тестування.
3. Що таке вичерпне тестування?
4. Які завдання вирішує тестування?
5. Яких завдань не вирішує тестування?
6. Які принципи тестування існують? У чому полягає їх відмінність?
7. Назвіть основні методи доведення коректності програм і базис цих методів.
8. Визначте типи логічних операцій, використовуваних за логічного доведення коректності програм.
9. У чому полягає відмінність техніки формального доведення від символьного виконання програм?
10. У чому полягає відмінність між операціями верифікації та валідації програм ?
11. Назвіть об'єкти тестування і підходи до їх тестування.
12. Яка є класифікація типів помилок у програмах?
13. Визначте основні етапи життєвого циклу тестування ПЗ.
14. Наведіть класифікацію тестів для перевірки ПЗ.
15. Які завдання виконує група з тестування?

13. СТРУКТУРНЕ ТЕСТУВАННЯ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

13.1. Основні поняття і принципи тестування програмного забезпечення

Тестування – процес виконання програми для виявлення помилок. Кроки процесу задаються тестами.

Кожний тест визначає:

– свій набір вихідних даних і умов для запуску програми;

– набір очікуваних результатів роботи програми.

Повну перевірку програми гарантує вичерпне тестування. Воно потребує перевірки всіх наборів вихідних даних, усіх варіантів їх оброблення і включає велику кількість тестових варіантів. Вичерпне тестування в багатьох випадках провести неможливо через ресурсні обмеження (насамперед через обмеження часу).

Метою проектування тестових варіантів є систематичне виявлення різних класів помилок за мінімальних витрат часу і вартості. Тестування забезпечує:

– виявлення помилок;

– демонстрацію відповідності функцій програми її призначенню;

– демонстрацію реалізації вимог до характеристик програми;

– відображення надійності як індикатора якості програми.

Тестування не виявляє дефектів, указує лише на їх наявність.

Простий структурний аналіз орієнтований на аналіз структури програми (потоків керування і потоків даних), який можна виконати, наприклад, з використанням теорії графів. У цьому методі структура програми подається у формі графової моделі, кожна вершина якої є оператором, а дуга – передаванням керування між операторами. За цією моделлю визначається, чи досяжні вершини програми і чи є вихід з усіх потоків керування програми для її завершення. Цей підхід дозволяє виявити логічні помилки.

Для аналізу потоків даних граф розширюється змінними, їх значеннями і посиланнями для кожного оператора програми. Під час тестування потоків даних спочатку визначаються значення предикатів в операторах реалізації логічних рішень, за якими формуються маршрути виконання програми. Потім перевіряються обчислення за допомогою арифметичних операцій. Для відстеження маршрутів встановлюються точки, у яких є посилання на змінну до присвоєння їй значення, або змінній присвоюється значення без її опису, або наводиться повторний опис змінної, або до змінної немає звернення.

Розглянемо інформаційні потоки процесу тестування (рис.13.1). На вході процесу тестування три потоки:

1) текст програми;

2) вихідні дані для запуску програми;

3) очікувані результати.

Рис. 13.1. Інформаційні потоки процесу тестування

Виконуються тести, отримані результати оцінюються. Це означає, що реальні результати тестів порівнюються з очікуваними результатами. Коли виявляється розбіжність, фіксується помилка – починається відлагодження.

Після збирання й оцінювання результатів тестування починається відображення якості та надійності ПЗ. Якщо регулярними є істотні помилки, що потребують проектних змін, то якість і надійність ПЗ сумнівні, констатується необхідність підсилення тестування. Якщо ж функції ПЗ реалізовані правильно, а виявлені помилки легко виправляються, можна зробити один із двох висновків:

– якість і надійність ПЗ задовільні;

– тести не здатні виявляти істотні помилки.

Якщо тести не виявляють помилок, виникає сумнів у тому, що тестові варіанти достатньо продумані і що в ПЗ немає прихованих помилок. Такі помилки виявлятимуться користувачами і коригуватимуться розробниками на етапі супроводження (коли вартість виправлення зростає в 60–100 разів порівняно з етапом розроблення).

Результати, отримані в ході тестування, можна оцінювати і більш формальним способом. Для цього використовують моделі надійності ПЗ, що прогнозують надійність за реальним даними про інтенсивність помилок.

Існують два принципи тестування програм:

– функціональне тестування (тестування «чорного ящика»);

– структурне тестування (тестування «білого ящика»).

13.2. Тестування методом «чорного ящика»

Метод «чорного ящика» застосовується для тестування функцій і належить до функціонального тестування. Його метою є виявлення невідповідностей між реальною поведінкою реалізованих функцій і очікуваною поведінкою відповідно до специфікації та вихідних вимог. Функціональні тести мають охоплювати всі функції, реалізовані у ПЗ, з урахуванням найбільш імовірних типів помилок. Тестові сценарії, що об’єднують окремі тести, орієнтовані на перевірку якості розв’язання функціональних завдань.

У процесі тестування досліджується робота кожної функції на всій області визначення. Тестування «чорного ящика» доступне лише через інтерфейс ПЗ; формується множина вхідних даних x, аналізується множина результатів y, які отримуються під дією функції $F(x)$ (рис. 13.2).

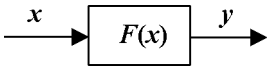


Рис. 13.2. Тестування методом «чорного ящика»

Тестування методом «чорного ящика» демонструє:

– виконання функції програм;
– приймання вихідних даних;
– вироблення результатів;
– збереження цілісності зовнішньої інформації.

Під час тестування методом «чорного ящика» розглядаються системні характеристики програм, ігнорується їх внутрішня логічна структура. Вичерпне тестування, як правило, неможливе. Наприклад, якщо в програмі 10 вхідних величин і кожна набуває по 10 значень, то буде потрібно 10^{10} тестових комбінацій. Тестування «чорного ящика» не реагує на багато особливостей програмних помилок.

13.3. Тестування методом «білого ящика»

У разі тестування цим методом повністю доступна інформація про внутрішню побудову програми. Під час тестування виявляється правильність функціонування внутрішніх елементів програми і зв'язків між ними. Об'єктом тестування тут є не зовнішнє, а внутрішнє поводження програми. Перевіряється коректність побудови всіх елементів програми і правильність їх взаємодії. Зазвичай аналізуються керувальні зв'язки елементів, рідше – інформаційні. Тестування за принципом «білого ящика» характеризується ступенем виконання чи покриття логіки (вихідний текст) програми. Розглянемо особливості цього принципу тестування.

Звичайне тестування «білого ящика» ґрунтується на аналізі керувальної структури програми. Програма вважається перевіреною, якщо проведено вичерпне тестування маршрутів її графа керування.

У цьому випадку формуються тестові варіанти, у яких:

– гарантується перевірка всіх незалежних маршрутів програми;
– проходять гілки *True*, *False* для всіх логічних рішень;
– виконуються всі цикли (у їх межах і діапазонах);
– аналізується правильність внутрішніх структур даних.

Недоліки методу.

1. Кількість незалежних маршрутів може бути дуже великою. Наприклад, якщо цикл у програмі виконується *k* разів, а всередині циклу є *n* розгалужень, то кількість маршрутів обчислюється за формулою

$$m = \sum_{i=1}^{k} n^i .$$

Якщо $n = 5$ і $k = 20$, кількість маршрутів $m = 12^{13}$. Припустімо, що на розроблення, виконання й оцінювання тесту по одному маршруту витрачається 1 мс. Тоді за роботи 24 год на добу, 365 днів у році на тестування витратиться 3170 років.

2. Вичерпне тестування маршрутів не гарантує відповідності програми вихідним вимогам до неї.

3. У програмі можуть бути випущені деякі маршрути.

4. Не можна виявити помилки, поява яких залежить від оброблюваних даних (це помилки, зумовлені виразами типу *if abs* (*a-b*) < < *eps...*, *if* (*a+b+c*)/3=*a...*).

Переваги тестування «білого ящика» полягають у тому, що цей принцип дозволяє враховувати особливості програмних помилок:

1. Кількість помилок мінімальна в «центрі» і максимальна на «периферії» програми.

2. Попередні припущення про ймовірність потоку керування або даних у програмі часто є некоректними. У результаті типовим може стати маршрут, модель обчислень за яким є недосконалою.

3. У разі запису алгоритму ПЗ у вигляді тексту мовою програмування можливе внесення типових помилок трансляції (синтаксичних і семантичних).

4. Деякі результати в програмі залежать не від вихідних даних, а від внутрішніх станів програми.

Кожна із цих причин є аргументом для тестування за принципом «білого ящика». Тести «чорного ящика» не зможуть реагувати на помилки таких типів.

Спосіб, що ґрунтується на принципі «білого ящика», – це тестування базового маршруту [19]. Він дає змогу:

– отримати оцінку комплексної складності програми;

– використовувати цю оцінку для визначення потрібної кількості тестових варіантів.

Тестові варіанти розробляються для перевірки базової множини маршрутів у програмі. Вони гарантують однократне виконання кожного оператора програми під час тестування.

Для подання програми використовується потоковий граф, який будується за відображенням керувальної структури програми. У ході відображення дужки умовних операторів і операторів циклів

(*end if*; *end loop*) розглядаються як окремі (фіктивні) оператори. Вузли (вершини) потокового графа відповідають лінійним ділянкам програми, включають один або декілька операторів програми. Дуги потокового графа відображають потік керування в програмі (передавання керування між операторами). Дуга – це орієнтоване ребро.

Розрізняють операторні та предикатні вузли. З операторного вузла виходить одна дуга, а із предикатного – дві дуги. Предикатні вузли відповідають простим умовам програми. Складна умова програми відображається в декількох предикатних вузлах. Складною називають умову, за якою використовується одна або декілька булевих операцій (*OR*, *AND*).

Приклад 13.1. Для фрагмента програми

if x OR y
then A
else B
end if.

ставиться завдання – побудувати потоковий граф.

Розв'язання. Правильним зображенням потокового графу є рис. 13.3, а пряме відображення потокового графу показано на рис. 13.4. Замкнуті ділянки, утворені дугами і вузлами, називають регіонами.

Зовнішній вигляд графу розглядається як додатковий регіон. Показаний на рис. 13.4 граф має три регіони – *R*1, *R*2, *R*3.

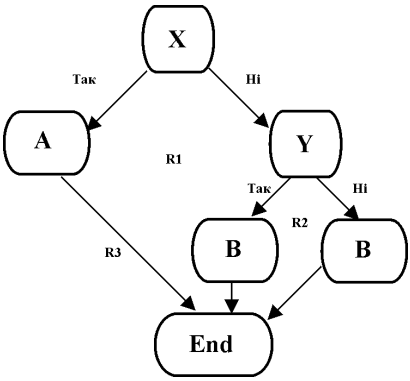


Рис.13.3. Перетворений потоковий граф

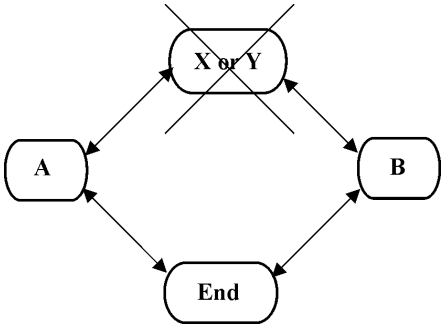


Рис. 13.4. Пряме відображення в потоковий граф

Приклад 13.2. Розглянемо процедуру стискання:

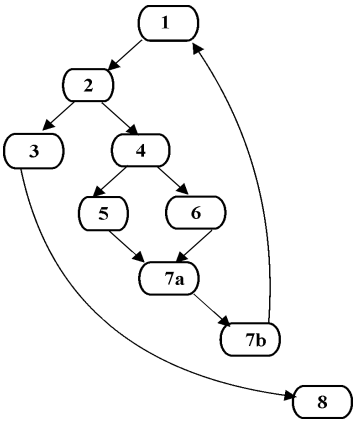


Рис. 13.5. Перетворений потоковий граф процедури стискання

```
процедура стискання
  1 виконувати доти, доки немає EOF
  1       читати запис;
  2       якщо запис порожній,
  3       то видалити запис:
  4       інакше, якщо поле a >= поля b,
  5               то видалити b;
  6               інакше видалити a;
 7a           кінець якщо;
 7a       кінець якщо;
 7b       кінець виконувати;
  8   кінець стискання;
```

Процедура відображається потоковим графом, показаним на рис. 13.5. Цей потоковий граф має чотири регіони.

13.4. Цикломатична складність

Цикломатична складність – метрика ПЗ, що забезпечує кількісну оцінку логічної складності програми. У способі тестування базового маршруту цикломатична складність визначає:

– кількість незалежних маршрутів у базовій множині програми;

– верхню оцінку кількості тестів, що гарантує однократне виконання всіх операторів.

Незалежним називається будь-який шлях, що вводить новий оператор оброблення або нову умову. У термінах потокового графа незалежний маршрут має містити дугу, що не входить у раніше визначені маршрути.

Маршрут починається в початковому вузлі, а закінчується в кінцевому вузлі графа. Незалежні маршрути формуються в порядку від самого короткого до самого довгого. Незалежні маршрути для потокового графа (приклад 13.2):

Маршрут 1: 1–8.

Маршрут 2: 2–3–7a–7 *b*–1–8.

Маршрут 3: 4–5–7a–7 *b*–1–8.

Маршрут 4: 4–6–7a–7 *b*–1–8.

Кожний новий маршрут включає нову дугу. Усі незалежні маршрути графу утворюють базову множину.

Властивості базової множині:

1) контрольні тести гарантують:

– однократне виконання кожного оператора;

– виконання кожної умови за гілками *True* і *False*;

2) потужність базової множини дорівнює цикломатичній складності потокового графу. Ця властивість дає апріорну оцінку кількості незалежних шляхів, які можна відшукати в графі.

Цикломатична складність обчислюється одним із трьох способів:

1) цикломатична складність дорівнює кількості регіонів потокового графу;

2) цикломатична складність визначається за формулою

$$V(G)= E-N+2,$$

де E – кількість дуг; N – кількість вузлів потокового графа;

3) цикломатична складність формується за виразом $V(G) =p+1$, де p – кількість предикатних вузлів у потоковому графі G.

Обчислимо цикломатичну складність графа із прикладу 13.1 кожним із трьох способів:

1) потоковий граф має 4 регіони;

2) $V(G)$ = 11 дуг – 9 вузлів + 2 = 4;

3) $V(G)$ = 3 предикатних вузлів +1=4.

Таким чином, цикломатична складність потокового графу із прикладу 13.1 дорівнює чотирьом.

Розглянемо цей спосіб для процедури обчислення середнього значення:

```
        процедура серед;
1         i := 1;
1     введено := 0;
1     кільк := 0; сум := 0;
1     вик пока
2     вел( i ) <> stop і
3     введено <=500
4     введено:= введено + 1;
5     якщо вел( i ) >= мін і
6     вел( i ) <= макс,
7                 то кільк := кільк + 1;
7                 сум := сум + вел( i );
```

```
8           кінець, якщо;
8           i := i + 1;
9   кінець вик;
10  якщо кільк > 0
11      то серед := сум / кільк;
12  інакше серед := stop;
13  кінець якщо;
13  кінець серед;
```

Процедури 1–13 містять складні умови (у заголовку циклу та умовному операторі). Елементи складних умов поміщено в рамки.

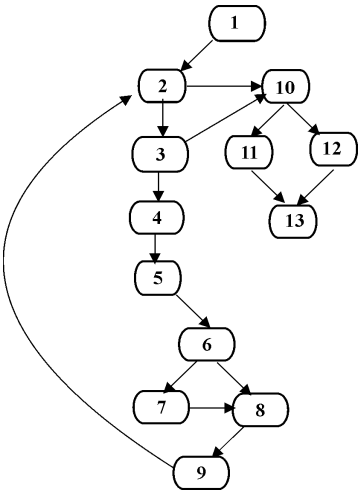


Рис. 13.6. Потоковий граф процедури обчислення середнього значення

Крок 1. На основі тексту програми формується потоковий граф:

– нумеруються оператори тексту (номери операторів показані в тексті процедури);

– пронумерований текст програми відображається у вузли і вершини потокового графу (рис. 13.6).

Крок 2. Визначається цикломатична складність потокового графу за кожною із трьох формул:

1) *V(G)* = 6 регіонів;

2) *V(G)* = 17 дуг – 13 вузлів + 2 = 6;

3) *V(G)* = 5 предикатних вузлів + + 1 = 6.

Крок 3. Визначається базова множина незалежних лінійних маршрутів:

Маршрут 1: 10–11–13; / вел=*s*top, кільк>0.
Маршрут 2: 10–12–13; / вел=*stop*, кільк=0.
Маршрут 3: 10–11–13; / спроба оброблення 501-й величини.
Маршрут 4: 8–9–2–... / вел < мін.
Маршрут 5: 8–9–2–... / вел > макс.
Маршрут 6: 8–9–2–... / режим нормального оброблення.

Для зручності подальшого аналізу по кожному маршруту позначені умови запуску. Крапки наприкінці шляхів 4, 5, 6 указують на їх продовження через залишок керувальної структури графу.

Крок 4. Підготовляються тестові набори, що ініціюють виконання кожного маршруту.

Кожний тестовий набір формується в такому вигляді:

Вихідні дані (*ВД*):

Очікувані результати (*ОЧ.РЕЗ.*):

Вихідні дані потрібно вибирати так, щоб предикатні вершини забезпечували потрібні перемикання – запуск тільки операторів на конкретному маршруті в певній послідовності.

Визначимо тестові варіанти, що задовольняють виявлену множину незалежних маршрутів.

Тестовий варіант для маршруту 1 ТВ1:

ВД: вел(*k*) = припустиме значення, де $k < i$; вел(*i*) = *stop*, де $2 < i < 500$.

ОЧ.РЕЗ.: коректне усереднення ґрунтується на *k* величинах і правильному розрахунку.

Маршрут не може тестуватися самостійно, а як частина маршрутів 4, 5, 6 (труднощі перевірки 11-го оператора).

Тестовий варіант для маршруту 2 ТВ2:

ВД: вел(1) = *st*op.

ОЧ.РЕЗ.: серед.=*st*op, інші величини мають початкові значення.

Тестовий варіант для маршруту 3 ТВ3:

ВД: спроба оброблення 501-ї величини, перші 500 величин мають бути правильними.

ОЧ.РЕЗ.: коректне усереднення ґрунтується на *k* величинах і правильному розрахунку.

Тестовий варіант для маршруту 4 ТВ4:

ВД: вел(*i*)=допустиме значення, де $i \le 500$; вел(*k*) < мін, де $k < i$.

ОЧ.РЕЗ.: коректне усереднення ґрунтується на *k* величинах і правильному розрахунку.

Тестовий варіант для маршруту 5 ТВ5:

ВД: вел(*i*)=допустиме значення, де $i \le 500$; вел(*k*) > макс, де $k < i$.

ОЧ.РЕЗ.: коректне усереднення ґрунтується на *n* величинах і правильному розрахунку.

Тестовий варіант для маршруту 6 ТВ6:

ВД: вел(*i*)=допустиме значення, де $i \le 500$.

ОЧ.РЕЗ.: коректне усереднення ґрунтується на *n* величинах і правильному розрахунку.

Реальні результати кожного тестового варіанта порівнюються з очікуваними результатами. Після виконання всіх тестових варіантів гарантується, що всі оператори програми виконано щонайменше один раз.

Важливо відзначити, що деякі незалежні маршрути не можуть перевірятися ізольовано. Їх потрібно перевіряти тестуванням іншого маршруту (як частину іншого тестового варіанта).

13.5. Способи тестування умов

Мета цієї сім'ї способів тестування – будувати тестові варіанти для перевірки логічних умов програми. При цьому бажано охопити операторів усіх гілок програми.

Уведемо позначення. Проста умова – булева змінна або вираз відношення. Вираз відношення має вигляд

$$E1 \text{ <оператор відношення> } E2, \qquad (13.1)$$

де $E1$, $E2$ – арифметичні вирази; як оператор відношення використовується один з таких операторів: $<$, $>$, $=$, $\neq$, $\leq$, $\geq$.

Складена умова містить декілька простих умов, булеві оператори і круглі дужки. Будемо застосовувати булеві оператори *OR*, *AND* (&), *NOT*. Умови, що не містять виразів відношення, називають булевими виразами.

Таким чином, елементами умови є: булевий оператор, булева змінна, пари дужок (проста або складена умови), оператор відношення, арифметичний вираз. Ці елементи визначають типи помилок в умовах.

Якщо умова некоректна, то некоректний щонайменше один з елементів умови. Отже, в умові можливі помилки:

– булевого оператора (наявність некоректних (відсутніх) надлишкових булевих операторів);

– булевої змінної;

– булевої дужки;

– оператора відношення;

– арифметичного виразу.

Спосіб тестування умов орієнтований на тестування кожної умови в програмі. Мета тестування умов – визначення не лише помилок в умовах, але й інших помилок у програмах. Якщо набір тестів для програми A ефективний для виявлення помилок в умовах, що утримуються в A, то ймовірно, що цей набір також ефективний для виявлення інших помилок в A. Крім того, якщо методика тестування ефективна для виявлення помилок в умові, то ймовірно,

що ця методика буде ефективною для виявлення помилок у програмі.

Існує кілька методик тестування умов. Найпростіша методика – тестування гілок. Тут для складної умови *C* перевіряються: кожна проста умова (що входить до неї); *True*-гілка; *False*-гілка.

Інша методика – тестування області визначення. У ній для виразу відношення потрібна генерація 3–4 тестів. Вирази вигляду (13.1) перевіряються трьома тестами, які формують значення E_1 більшим, ніж $E2$, що дорівнює $E2$ і меншим, ніж $E2$. Якщо оператор відношення неправильний, а $E1$ і $E2$ коректні, то ці три тести гарантують виявлення помилки оператора відношення.

Для визначення помилок в $E1$ і $E2$ тест має сформувати значення $E1$ більшим або меншим, ніж $E2$, причому досягнути якомога меншої різниці цих значень.

Для булевих виразів із n змінними має набір з 2^n тестів. Цей набір дозволяє виявити помилки булевих операторів, змінних і дужок, але практичний тільки за малого n. Якщо в булевому виразі кожна булева змінна входить лише один раз, то кількість тестів легко зменшується.

13.6. Тестування гілок і операторів відношень

Спосіб тестування гілок і операторів відношень виявляє помилки розгалуження і операторів відношення в умові, для якої виконуються такі обмеження [19]:

– усі булеві змінні і оператори відношення входять в умову лише по одному разу;

– в умові немає загальних змінних.

У цьому способі використовуються природні обмеження умов (обмеження результату). Для складеної умови *C*, що включає n простих умов, формується обмеження умови (ОУ):

$$\text{ОУ}_C = (d_1, d_2, d_3 \ldots . d_n),$$

де d_i – обмеження результату i-ї простої умови.

Обмеження результату фіксує можливі значення аргумента (змінної) простої умови (якщо він один) або співвідношення між значеннями аргументів (якщо їх декілька).

Якщо *i*-а проста умова є булевою змінною, то обмеження на її результат складається із двох значень і має вигляд

$$d_i = (true, false).$$

Якщо *j*-а проста умова є виразом відношення, то обмеження його результату складається із трьох значень і має такий вигляд:

$$d_j = (>, <, =).$$

Обмеження умови $ОУ_C$ (для умови *C*) покривається виконанням *C*, якщо в ході цього виконання результат кожної простої умови задовольняє відповідне обмеження в $ОУ_C$.

На підставі обмеження умови ОУ створюється обмежувальна множина (ОМ), елементи якої є об'єднанням усіх можливих значень d_1, d_2, d_3, ..., d_n. Обмежувальна множина – зручний інструмент для запису завдання тестування, оскільки воно складається з відомостей про значення змінних, які впливають на значення умови, що перевіряється.

Приклад 13.3. Перевірити умову, складену із трьох простих умов:

$$b\&(x>y)\&a.$$

Розв'язання. Умова набуває дійсного значення, якщо всі прості умови дійсні. У термінах значень простих умов це відповідає запису

(*true*, *true*, *true*),

а в термінах обмежень значень аргументів простих умов – запису

(*true*, >, *true*).

Другий запис є безпосереднім керівництвом для написання тесту. Він указує, що змінна *b* повинна мати дійсне значення, значення змінної *x* має бути більше від значення змінної *y*, і, нарешті, змінна *a* повинна мати дійсне значення.

Отже, кожний елемент ОМ задає окремий тестовий варіант. Вихідні дані тестового варіанта повинні забезпечити відповідну комбінацію значень простих умов, а очікуваний результат дорівнює значенню складеної умови.

Приклад 13.4. Розглянемо дві типові складові умови:

$$C_\& = a \ \& \ b, \ C_{or} = a \ or \ b,$$

де *a* і *b* – булеві змінні. Відповідні обмеження умов набувають вигляду

$$ОУ_\&=(d_1,d_2), \ ОУ_{or}=(d_1,d_2), \qquad (13.2)$$

де $d_1 = d_2 = (true, false)$.

Розв'язання. Побудуємо обмежувальні множини за допомогою таблиці істинності (табл. 13.1).

Таблиця задає в ОМ чотири елементи (і відповідно чотири тестові варіанти). Чи можна їх мінімізувати? Чи можна зменшити кількість їх елементів в ОМ?

Таблиця 13.1

Таблиця логічних операцій

Варіант	*a*	*b*	*a & b*	*a or b*
1	*false*	*false*	*false*	*false*
2	*false*	*true*	*false*	*true*
3	*true*	*false*	*false*	*true*
4	*true*	*true*	*true*	*true*

Із погляду тестування необхідно оцінити вплив складної умови на програму. Складна умова може набувати лише двох значень, кожне з яких залежить від великої кількості простих умов. Щоб позбутися впливу надлишкових простих умов, скористаємося ідеєю скороченої схеми обчислення: елементи виразу обчислюємо доти, доки вони впливатимуть на значення виразу (13.2). Під час тестування необхідно виявити помилки перемикання, тобто помилки через булевий оператор, оперуючи значеннями простих умов (булевих змінних). Справедливі такі висновки:

– для умови типу «І» *(a & b)* варіанти 2 і 3 поглинають варіант 1. Тому обмежувальна множина має вигляд:

$$\text{ОМ}_{\&} = \{(false, true), (true, false), (true, true)\};$$

– для умови типу «АБО» (*a or b*) варіанти 2 і 3 поглинають варіант 4, тому обмежувальна множина має вигляд:

$$\text{ОМ}_{or} = \{(false, false), (false, true), (true, false)\}.$$

Розглянемо кроки способу тестування гілок і операторів відношень.

Для кожної умови в програмі виконуються такі дії:

1) будується обмеження умов;

2) виявляються обмеження результату за кожною простою умовою;

3) будується обмежувальна множина. Побудова виконується підставлянням у константні формули $\text{ОМ}_{\&}$ або ОМ_{or} виявлених обмежень результату;

4) для кожного елемента ОМ розробляється тестовий варіант.

Приклад 13.5. Розглянемо складну умову *C*1 вигляду:

$$B_1 \,\&\, (E_1, E_2),$$

де B_1 – булевий вираз; E_1, E_2 – арифметичні вирази. Обмеження складної умови має вигляд

$$ОУ = (d_1, d_2),$$

де обмеження простих умов становлять

$$d_1 = (true, false),\ d_2 = (=, <, >).$$

Розв'язання. На основі аналогії між $C1$ і $C2$ (за винятком того, що в $C1$ друга проста умова – це вираз відношення), обмежувальна множина для $C1$ будується модифікуванням

$$ОМ_{\&} = \{(false, true), (true, false), (true, true)\}.$$

Зазначимо, що *true* для ($E_1 = E_2$) означає =, а *false* для ($E_1 = E_2$) означає або <, >. Замінюючи (*true*, *true*) і (*false*, *true*) обмеженнями (*true*, =) і (*false*, =) відповідно, а (*true*, *false*) – обмеженнями (*true*, <) і (*true*, >), отримуємо обмежувальну множину для $C1$:

$$ОМ_{C1} = \{(false,=),(true,<),(true,>),(true,=)\}.$$

Покриття цієї множини гарантує виявлення помилок булевих операторів і операторів відношення в $C1$.

Приклад 13.6. Задано складну умову $C2$ вигляду

$$(E_3 > E_4)\ \&\ (E_1 = E_2),$$

де E_1, E_2, E_3, E_4 – арифметичні вирази. Обмеження складної умови має вигляд

$$ОУ =(d_1, d_2),$$

де обмеження простих умов становлять

$$d_1 = (=,<,>),\ d_2 = (=,<,>).$$

Розв'язання. За аналогією із C_2 і C_1 (за винятком того, що в $C2$ перша проста умова – це вирази відношення) обмежувальна множина для $C2$ будується модифікуванням $ОМ_{C1}$:

$$ОМ_{C2} = \{(=, =), (<, =), (>, <),(>, >),(>, =)\}.$$

Покриття обмежувальною множиною гарантує виявлення помилок операторів відношення в $C2$.

13.7. Спосіб тестування потоків даних

У попередніх способах тести будувалися на підставі аналізу керувальної структури програми. У цьому способі аналізу підлягає інформаційна структура програми.

Роботу будь-якої програми можна розглядати як оброблення потоку даних, переданих від входу в програму до її виходу.

Приклад 13.7. Деяка програма має потоковий граф (рис. 13.7). Суцільними дугами показано зв'язки з керування між операторами в програмі. Пунктирні дуги відповідають інформаційним зв'язкам

(потокам даних). Позначені інформаційні зв'язки відповідають таким припущенням:

– у вершині 1 визначаються значення змінних *a*, *b*;

– значення змінної *a* використовується у вершині *4*;

– значення змінної *b* використовується у вершинах *3*, *6*;

– у вершині *4* визначається значення змінної *c*, що використовується у вершині *6*.

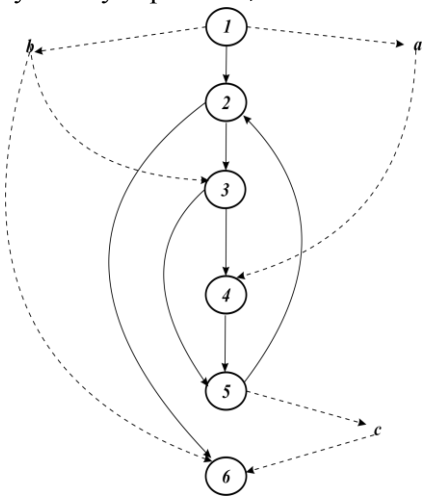


Рис. 13.7. Граф програми з керувальними та інформаційними зв'язками

Визначити кількість інформаційних потоків даних у програмі.

Розв'язання. Застосуємо механізм визначення–використання. Для кожної вершини графа введемо позначення:

– множини визначень даних: *DEF*(*i*) = { *x* | *i* -а вершина містить визначення *x*};

– множини використань даних: *USE* (*i*) = { *x* | *i* -а вершина використовує *x*}.

Під визначенням даних розуміють дії, що змінюють елемент даних. Ознака визначення – ім'я елемента, що міститься в лівій частині оператора присвоювання:

$$x := f(\ldots)\ldots \qquad (13.3)$$

Використання даних – це застосування елемента у виразі (13.3), де показано звернення до елемента даних, а не до змінної. Ознака використання (ім'я елемента) міститься у правій частині оператора присвоювання:

$$\square := f(x).$$

Прямокутником позначено місце підставлення іншого імені (прямокутник відіграє роль мітки-заповнювача).

Назвемо *DU-ланцюжком (ланцюжком визначення–використання)* конструкцію [*x*, *i*,*j*], де *i*,*j* – ім'я вершин; *x* – змінна, яка визначена в *i*-й вершині ($x \in DEF(i)$) і використовуєть-

ся в j -й вершині ($x \in USE(j)$).

Таким чином, маємо такі *DU*-ланцюжки:

$$[a, 1, 4],[b, 1, 3], [b, 1, 6], [c, 4, 6].$$

Спосіб *DU*-тестування охоплює всіх *DU*-ланцюжків програми. Отже, тести тут розробляються на підставі аналізу життя всіх даних програми.

Очевидно, що для підготовки тестів потрібно виділити маршрути виконання програми по керувальному графу. Критерій для вибору маршруту – покриття максимальної кількості *DU*-ланцюжків.

Кроки способу DU-тестування:

1) побудова керувального графу програми;
2) побудова інформаційного графу;
3) формування повного набору *DU*-ланцюжків;
4) формування повного набору відрізків шляхів по керувальному графу (відображенням набору *DU*-ланцюжків інформаційного графу (рис. 13.8);
5) побудова маршрутів – повних шляхів на керувальному графу, що покривають набір відрізків шляхів керувального графа;
6) підготовка тестових варіантів.

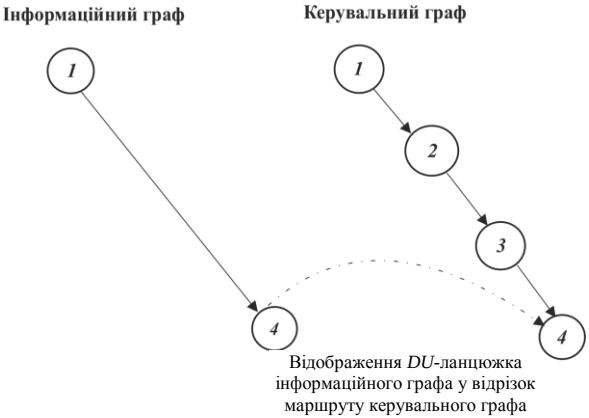


Рис. 13.8. Відображення *DU*-ланцюжка у відрізок шляху

Переваги DU-тестування:

– простота необхідного аналізу операційно-керувальної структури програми;

– простота автоматизації.

Недолік DU-тестування: труднощі вибору мінімальної кількості максимально ефективних тестів.

Область використання DU-тестування: програми із вкладеними умовними операторами і операторами циклу.

Тестування циклів. Цикл – найпоширеніша конструкція алгоритмів, реалізованих у ПЗ. Цикли тестуються за принципом «білого ящика»; під час перевірки циклів основна увага приділяється правильності конструкцій циклів. Розрізняють 4 типи циклів: прості, укладені, об'єднані, неструктуровані. Структуру циклів показано на рис. 13.9.

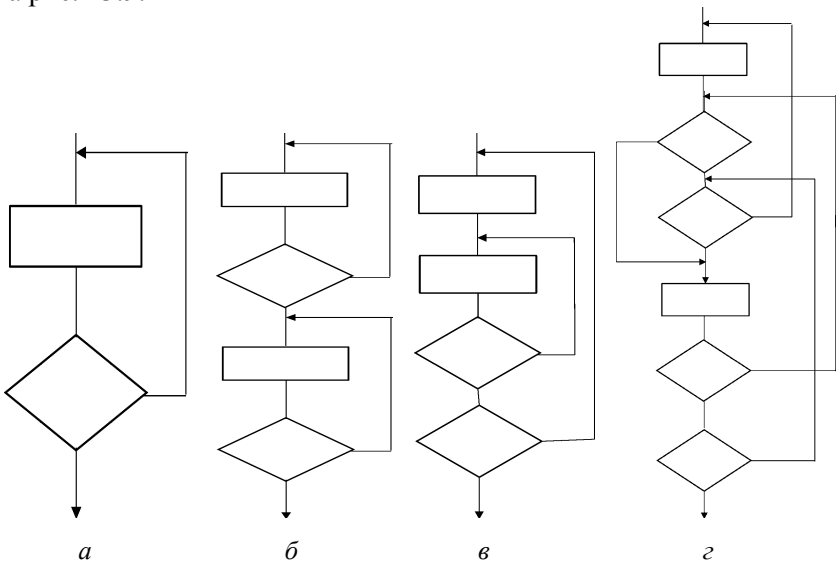


Рис.13.9. Типові структури циклів:
а – простого; *б* – складного; *в* – укладеного;
г – неструктурованого

Для перевірки простих циклів з кількістю повторень n можна використовувати один з таких наборів тестів:

1) прогін усього циклу;
2) лише один прохід циклу;
3) два проходи циклу;
4) m проходів циклу, де $m<n$;

5) $n - 1$, n, $n + 1$ проходів циклу.

Зі збільшенням рівня вкладеності циклів кількість можливих шляхів різко зростає. Це призводить до нереалізованої кількості тестів. Для скорочення кількості тестів застосовується спеціальна методика, у якій використовуються такі поняття, як об'єднаний і вкладений цикли (рис. 13.10).

Порядок тестування вкладених циклів ілюструє рис. 13.11.

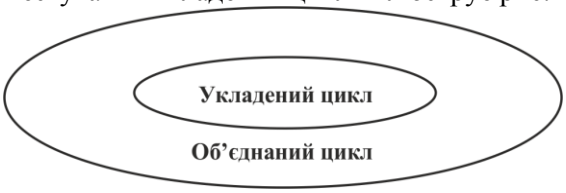


Рис. 13.10. Об'єднаний і вкладений цикли

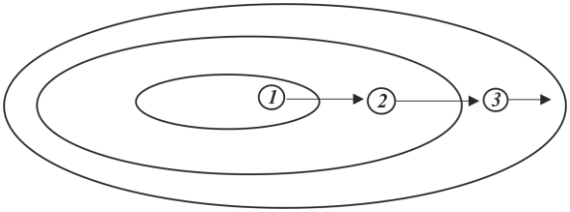


Рис. 13.11. Кроки тестування вкладених циклів

Кроки тестування.

1. Вибирається внутрішній цикл. Установлюються мінімальні значення параметрів усіх інших циклів.

2. Для внутрішнього циклу проводяться тести простого циклу. Додаються тести для вилучення значень і значень, що виходять за межі робочого діапазону.

3. Переходять до наступного циклу один за одним. Виконують його тестування. При цьому зберігаються мінімальні значення параметрів для всіх зовнішніх (об'єднаних) циклів і типові значення для всіх укладених циклів.

4. Робота триває доти, поки не будуть протестовані всі цикли.

Якщо кожний із циклів незалежний від інших, то використовується техніка тестування простих циклів, якщо залежний (напри-

клад, кінцеве значення лічильника першого циклу використовується як початкове значення лічильника другого циклу) – методика для вкладених циклів.

Неструктуровані цикли тестуванню не підлягають. Вони потребують перероблення за допомогою структурованих програмних конструкцій.

Контрольні запитання і завдання

1. У чому полягає суть тестування «чорного ящика»?
2. Які особливості тестування «білого ящика»?
3. Охарактеризуйте спосіб тестування базового шляху.
4. Які особливості має потоковий граф?
5. Поясніть поняття «цикломатична складність». Які є способи її розрахунку?
6. Продемонструйте застосування методу тестування базового шляху.
7. Назвіть типи помилок в умовах.
8. Поясніть спосіб тестування гілок і операторів відношень. Які він має обмеження?
9. Що таке обмеження результату, умови? Що являє собою обмежувальна множина? У чому полягає зручність її застосування?
10. Які підходи застосовують для мінімізації набору тестів?
11. Поясніть спосіб тестування потоків даних.
12. Поясніть відмінність множини визначення даних від множини використання даних.
13. Що таке ланцюжок визначення–використання?
14. Поясніть особливості тестування циклів.
15. Як протестувати простий цикл, укладений цикл?

14. МЕТРИКИ ОБ'ЄКТНО-ОРІЄНТОВАНИХ ПРОГРАМНИХ СИСТЕМ

Значна частина витрат у процесі конструювання об'єктно-орієнтованих програмних систем припадає на створення візуальних моделей. Важливим завданням стає визначення якості цих моделей у числовій формі, розв'язання якого полягає у введенні спеціального метричного апарату. Такий апарат розвиває ідеї класичного оцінювання складних програмних систем, що ґрунтується на метриках складності, зв'язності та з'єднання. Разом з тим він враховує специфічні особливості об'єктно-орієнтованих розв'язків.

14.1. Метричні особливості об'єктно-орієнтованих програмних систем

Об'єктно-орієнтовані метрики вводяться з метою:

– поліпшення розуміння якості продукту;

– оцінювання ефективності процесу конструювання;

– поліпшення якості роботи на етапі проектування.

Ця мета важлива, але для програмного інженера ставиться головна мета – підвищити якість продукту. Виділяють такі характеристики об'єктно-орієнтованих систем:

– локалізацію;

– інкапсуляцію;

– інформаційну закритість;

– спадкування;

– способи абстрагування об'єктів.

Локалізація фіксує спосіб угруповання інформації в програмі. У класичних методах, де використовується функціональна декомпозиція, інформація локалізується у функціях, у яких реалізуються процедурні модулі. У методах, керованих даними, інформація групується у структурах даних. В об'єктно-орієнтованому середовищі інформація групується всередині класів або об'єктів (інкапсуляція даних і процесів).

Оскільки основний механізм локалізації у класичних методах – функція, програмні метрики, орієнтовані на внутрішню структуру або складність функцій (довжину модуля, зв'язність, цикломатичну складність) або на спосіб, яким функції зв'язуються одна з одною (з'єднання модулів).

В об'єктно-орієнтованій системі базовим елементом є клас, тому локалізація тут ґрунтується на об'єктах. Метрики потрібно застосовувати до класу (об'єкту) як до комплексної сутності. Крім того, між операціями (функціями) і класами можуть бути відношення не лише «один-до-одного». Тому метрики, що відображають способи взаємодії класів, мають бути пристосовані до відношень «один-до-багатьох», «багато-до-багатьох».

Інкапсуляція – упаковування (зв'язування) сукупності елементів. Для класичних програмних систем прикладами низькорівневої інкапсуляції є записи і масиви. Механізми інкапсуляції середнього рівня є підпрограми (процедури, функції).

В об’єктно-орієнтованих системах інкапсулюються завдання класу, які ґрунтуються на його властивостях (а для агрегатів – додатково на властивостях інших класів), операціях і станах.

Інкапсуляція приводить до зсуву фокуса вимірів з одного модуля на групу властивостей і модулів оброблення (операцій). Крім того, інкапсуляція переводить виміри на більш високий рівень абстракції (наприклад – метрика «кількість операцій на клас»). Навпаки, класичні метрики орієнтовані на низький рівень – кількість булевих умов (цикломатична складність) і кількість рядків програми.

Інформаційна закритість робить невидимими операційні деталі програмного компонента. Іншим компонентам доступна лише необхідна інформація. Якісні об’єктно-орієнтовані системи підтримують високий рівень інформаційної закритості. Таким чином, метрики, що вимірюють ступінь досягнутої закритості, тим самим відображають якість об’єктно-орієнтованого проекту.

Спадкування – механізм, що забезпечує тиражування обов’язків одного класу в інші класи. Спадкування поширюється через всі рівні ієрархії класів. Стандартні програмні системи не підтримують цю характеристику. Оскільки спадкування – основна характеристика об’єктно-орієнтованих систем, на ній фокусуються багато об’єктно-орієнтованих метрик (кількість дітей – нащадків класу, кількість батьків, висота класу в ієрархії спадкування).

Абстракція – це механізм, що дозволяє проектувальнику виділяти головне в програмному компоненті (як властивості, так і операції) без урахування другорядних деталей. У міру переміщення на вищі рівні абстракції ігнорується велика кількість деталей, забезпечуючи більший рівень узагальненого подання поняття або елемента. У міру переміщення на нижчі рівні абстракції вводиться більша кількість деталей, забезпечуючи більш вдале подання поняття або елемента.

Клас – це абстракція, яку можна застосувати на різних рівнях деталізації і різними способами (наприклад, як список операцій, послідовність станів, послідовності взаємодій). Тому об’єктно-орієнтовані метрики мають виражати абстракції в термінах вимірів класу. Наприклад, кількість екземплярів класу в додатку, кількість родових класів у додатку, відношення кількості родових до кількості неродових класів.

14.2. Метрики зв'язності

Методи Л. Констентайна та Е. Йордана, які визнані класичними, визначають сім типів зв'язності.

1. *Зв'язність за збігом*. У модулі немає явно виражених внутрішніх зв'язків.
2. *Логічна зв'язність*. Частини модуля об'єднані за принципом функціональної подібності.
3. *Часова зв'язність*. Частини модуля не зв'язані, але потрібні в період роботи системи.
4. *Процедурна зв'язність*. Частини модуля зв'язані порядком виконуваних ними дій, що реалізовують деякий сценарій поводження.
5. *Комунікативна зв'язність*. Частини модуля зв'язані даними (працюють з однією і тією самою структурою даних).
6. *Інформаційна (послідовна) зв'язність*. Вихідні дані однієї частини використовуються як вхідні дані іншої частини модуля.
7. *Функціональна зв'язність*. Частини модуля разом реалізовують одну функцію.

Метод функціональний за своєю природою, тому найбільшою зв'язністю тут оголошена функціональна зв'язність. Разом з тим однією із принципових переваг об'єктно-орієнтованого підходу є природна зв'язність об'єктів.

Максимально зв'язаним є об'єкт, що являє собою одне ціле і який включає операції з ним. Наприклад, максимально зв'язаним є об'єкт, що містить таблицю символів компілятора, якщо в нього включені функції, такі як «додати символ», «пошук у таблиці» і т. ін.

Отже, останній тип зв'язності можна визначити як *об'єктну зв'язність*. Кожна операція забезпечує функціональність, яка передбачає, що всі властивості об'єкта будуть модифікуватися, відображатися і використовуватися як базис для надання послуг.

Висока зв'язність – бажана характеристика, оскільки вона означає, що об'єкт є однією частиною в проблемній галузі, існує в єдиному просторі. Зі зміною системи всі дії над частиною інкапсулюються в одному компоненті. Тому для зміни немає потреби модифікувати багато компонентів.

Якщо функціональність в об'єктно-орієнтованій системі забезпечується спадкуванням від батьківських класів, то зв'язність об'єкта, що успадковує властивості і операції, зменшується. У цьому випадку не можна розглядати об'єкт окремим модулем – повин-

ні враховуватися всі його батьківські класи. Системні засоби перегляду сприяють цьому. Однак розуміння елемента, що успадковує властивості від декількох батьківських класів, різко ускладнюється.

14.2.1. Метрики зв'язності за даними

Модель секціонування класу розробили Л. Отто і Б. Мехр [19]. Секціонування ґрунтується на екземплярних змінних класу. Для кожного методу класу отримують ряд секцій, а потім об'єднують всі секції класу. Вимір зв'язності ґрунтується на кількості лексем даних, які з'являються в декількох секціях і «склеюють» секції в модуль. Під лексемами даних розуміють визначення констант і змінних або посилання на константи і змінні.

Базовим поняттям методики є секція даних. Вона створюється для кожного вихідного параметра методу. Секція даних – це послідовність лексем даних в операторах, які потрібні для обчислення цього параметра.

Наприклад, на рис. 14.1 подано програмний текст методу *SumAndProduct*. Усі лексеми, що входять у секцію змінної *Sum*, виділено сірим кольором. Сама секція для *Sum* записується послідовністю лексем.

$$N_1 \cdot Sum_1 \cdot I_1 \cdot Sum_2 \cdot O_1 \cdot I_2 \cdot 1_2 \cdot N_2 \cdot Sum_3\ Sum_4 \cdot I_3.$$

```
Procedure SumAndProduct ( N: ineger;
    var SumN, ProdN: integer);
    var I: integer;
     begin
        SumN:=0;
        ProdN:=1;
          for I:=1 to N do begin
                    SumN:= SumN+I;
                    ProdN:= ProdN*I
                         end
end;
```

Рис. 14.1. Секція даних для змінної *Sum*

Індекс змінної «1_2» указує на друге входження лексеми «1» у текст методу. Аналогічно визначається секція для змінної *Prod*:

$$N_1 \cdot Prod_1 \cdot I_1 \cdot Prod_2 \cdot 1_1 \cdot I_2 \cdot 1_2 \cdot N_2 \cdot Prod_3 \cdot Prod_4 \cdot I_4.$$

Для визначення відношень між секціями даних можна показати профіль секцій даних у методі. Для розглянутого прикладу профіль секцій даних наведено в табл. 14.1.

Таблиця 14.1

Профіль секцій даних для методу *SumAndProduct*

Sum	*Prod*	Оператор
		procedure SumAndProduct
1	1	(*N: integer*;

Закінчення табл. 14.1

Sum	*Prod*	Оператор
1	1	*Var Sum, ProdN:integer*)
		var
1	1	*l*:*integer*;
		begin
2		*Sum*:=0
	2	*Prod*:=1
3	3	*for l*:=1 *to N do begin*
3		*Sum*:=*Sum*+*l*
	3	*Prod*:=*Prod***l*
		end
		end;

Видно, що змінні кожної секції вказують кількість лексем з *i*-го рядку методу, які включаються в секцію.

Ще одне базове поняття методики – секціонована абстракція.

Секціонована абстракція – це об'єднання всіх секцій даних методу. Наприклад, секціонована абстракція методу *SumAndProduct* має вигляд

$$SA(SumAndProduct) = \{N_1 \cdot Sum_1 \cdot I_1 \cdot Sum_2 \cdot 0_1 \cdot I_2 \cdot I_2 \cdot N_2 \cdot Sum_3 \cdot Sum_4 \cdot I_3, N_1 \cdot Prod_1 \cdot I_1 \cdot Prod_2 \cdot I_1 \cdot I_2 \cdot I_2 \cdot N_2 \cdot Prod_3 \cdot Prod_4 \cdot I_4\}.$$

Секціонованою абстракцією класу (*Class Slice Abstraction*) *CSA*(*C*) називають об'єднання секцій всіх екземплярних змінних класу. Повний набір секцій визначається обробленням усіх методів класу.

Склеєними лексемами називають ті лексеми даних, елементи яких складаються з більш ніж однієї секції даних.

Сильно склеєними лексемами називають ті склеєні лексеми, елементи яких складаються зі всіх секцій даних.

Сильна зв'язність за даними (*Strong Data Cohesion*) – метрика, яка ґрунтується на кількості лексем даних, що входять в усі секції даних для класу. Тобто сильна зв'язність за даними враховує кількість сильно склеєних лексем у класі *C*; вона обчислюється за формулою:

$$\mathrm{SDC}(C) = \frac{|SG(\mathrm{CSA}(C))|}{|\text{лексеми}(C)|},$$

де *SG*(*CSA*(*C*)) – об'єднання сильно склеєних лексем кожного з методів класу *C*; лексеми(*C*) – множина всіх лексем даних класу *C*. Таким чином, клас без сильно склеєних лексем має нульову сильну зв'язність за даними.

Слабка зв'язність за даними (*Weak Data Cohesion*) – метрика, що оцінює зв'язність, базуючись на склеєних лексемах. Склеєні лексеми не потребують зв'язування всіх секцій даних, тому метрика визначає більш слабкий тип зв'язності. Слабка зв'язність за даними обчислюється за формулою

$$\mathrm{WDC}(C) = \frac{|G(\mathrm{CSA}(C))|}{|\text{лексеми}(C)|},$$

де *G*(*CSA*(*C*)) – об'єднання склеєних лексем кожного з методів класу. Клас без склеєних лексем не має слабкої зв'язності за даними. Найбільш точною метрикою зв'язності між секціями даних є клейкість даних (*Data Adhesiveness*). Клейкість даних визначається як відношення суми з кількостей секцій, що містять кожну склеєну лексему, до добутку кількості лексем даних у класі на кількість секцій даних *CSA*(*C*). Метрика визначається як

$$\mathrm{DA}(C) = \frac{\sum d \in G(\mathrm{CSA}(C) \times \ d \in \text{секції}}{|\text{лексеми}(C)| \times |\mathrm{CSA}(C)|}.$$

Приклад 14.1. Застосуємо метрики до класу, профіль секцій якого наведено в табл. 14.2.

Таблиця 14.2

Профіль секцій даних для класу *Stack*

Array top size	Клас *Stack*
	class Stack {*int array, top, size*;
	public:

	Stack (*int s*) {
2 2	*size*=*s*;
2 2	*array*=*new int* [*size*];
2	*top*=0;}
	int IsEmpty () {
2	*return top*==0};
	int Size (){

Закінчення табл. 14.2

Array top size	Клас *Stack*
2	*return size*};
	int Vtop(){
3 3	*return array* [*top*-1]; }
	void Push (*int item*) {
2 2	*if* (*top*= =*size*)
	printf ("*Empty stack.* \n");
	else
3 3	*array* [*top*++]=*item*;}
	int Pop () {
1	*if* (*IsEmpty* ())
	printf ("*Full stack.* \n");
	else
1	*--top*;}
	};

Очевидно, що *CSA*(*Stack*) включає три секції з 19 лексемами, має 5 сильно склеєних лексем і 12 склеєних лексем. Розрахунки за розглянутими метриками дають такі значення:

$$SDC(CSA(Stack)) = 5/19 = 0{,}26;$$
$$WDC(CSA(Stack)) = 12/19 = 0{,}63;$$
$$DA(CSA(Stack)) = (7\cdot 2 + 5\cdot 3)/(19\cdot 3) = 0{,}51.$$

14.2.2. Метрики зв'язності за методами

Метрики зв'язності класу, які ґрунтуються на прямих і непрямих з'єднаннях між парами методів запропоновано Д. Бієменом і Б. Кенгом [44]. Якщо існують загальні екземплярні змінні (одна або

декілька), використовувані в методах, то ці методи з'єднані прямо. Пари методів можуть з'єднуватися побічно через інші прямо з'єднані методи.

Відношення між елементами класу *Stack* показано на рис. 14.2. Прямокутниками позначено методи класу, а овалами – екземплярні змінні. Зв'язки показують відношення використання між методами і змінними.

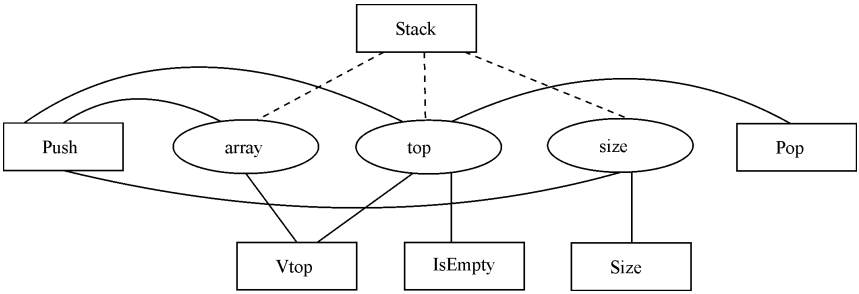


Рис. 14.2. Відношення між елементами класу *Stack*

Екземплярна змінна *top* використовується методами *Stack*, *Push*, *Pop*, *Vtop* і *IsEmpty*. Таким чином, всі ці методи попарно прямо з'єднані. Навпаки, методи *size* і *pop* з'єднані побічно: *size* з'єднаний прямо з *push*, що, у свою чергу, прямо з'єднаний з *pop*. Метод *stack* є конструктором класу, тобто функцією ініціалізації. Конструктору доступні всі екземплярні змінні класу, він використовує ці змінні спільно з усіма іншими методами. Отже, конструктори створюють з'єднання і між такими методами, які ніяк не зв'язані один з одним. Тому ні конструктори, ні деструктори тут не враховуються. Зв'язки між конструктором і екземплярними змінними на рис. 14.2 показано пунктирними лініями.

Для формалізації моделі вводяться поняття абстрактного методу і абстрактного класу.

Абстрактний метод AM(*M*) – це подання реального методу *M* у вигляді множини екземплярних змінних, які прямо або побічно використовуються методом. Екземплярна змінна прямо використовується методом *M*, якщо вона з'являється в методі як лексема даних. Екземплярна змінна може бути визначена в тому ж класі, що і *M*, або у батьківському класі цього класу. Множину екземплярних

змінних, прямо використовувану методом *M*, позначимо як *DU*(*M*). Екземплярна змінна побічно використовується методом *M*, якщо:

1) екземплярна змінна прямо використовується іншим методом *M*, що викликається (прямо або побічно) з методу *M*;

2) екземплярна змінна, прямо використовувана методом *M*, якщо перебуває в тому ж об'єкті, що і *M*.

Множин екземплярних змінних, побічно використовуваних методом *M*, позначимо як *IU*(*M*).

Кількісно абстрактний метод формується за виразом

$$AM\ (M) = DU\ (M)\ \cup IU\ (M).$$

Абстрактний клас AC(*C*) – це подання реального класу *C* у вигляді сукупності абстрактних методів, причому кожний абстрактний метод відповідає видимому методу класу *C*. Кількісно абстрактний клас формується за виразом

$$AC\ (C) = [[AM\ (M) \mid M \in V\ (C)]],$$

де *V*(*C*) – множина всіх видимих методів у класі *C* і в класах-батьках для *C*.

Зауважимо, що *AM*-подання різних методів може збігатися, тому в *AC* можуть бути дубльовані елементи. Тому *AC* записується у формі мультимножини (подвійні квадратні дужки розглядаються як його позначення).

Локальний абстрактний клас LAC(*C*) – це сукупність абстрактних методів, де кожний абстрактний метод відповідає видимому методу, визначеному лише всередині класу *C*. Кількісно абстрактний клас формується за виразом

$$LAC(C)=[[AM(M)|M \in LV(C)]],$$

де *LV*(*C*) – множина всіх видимих методів, визначених у класі *C*.

Абстрактний клас для стека, наведеного в табл. 14.2, має вигляд:

$$AC\ (Stack) = [[\{top\}, \{size\}, \{array, top\}, \{array, top, size\}, \{pop\}]].$$

Оскільки клас *Stack* не має суперкласу, то справедливим є вираз

$$AC\ (Stack) = LAC\ (Stack).$$

Нехай *NP*(*C*) – загальна кількість пар абстрактних методів в *AC*(*C*); *NP* визначає максимально можливу кількість прямих або непрямих з'єднань у класі. Якщо в класі *C* є *N* методів, тоді

$$NP(C) = N*(\ N\text{-}N\text{-}\ 1)/2.$$

Уведемо позначення: *NDC*(*C*) – кількість прямих з'єднань *AC*(*C*); *NIC*(*C*) – кількість непрямих з'єднань в *AC*(*C*).

Тоді метрики зв'язності класу можна подати у вигляді:

$$TCC\ (C) = NDC\ (C) / NP\ (C),$$

де *TCC* – сильна зв'язність класу (*Tight Class Cohesion*);

$$LCC\ (C) = (NDC\ (C) + NIC\ (C)) / NP\ (C),$$

де *LCC* – слабка зв'язність класу (*Loose Class Cohesion*).

Очевидно, що завжди справедливо така нерівність:

$$LCC(C) \geq TCC(C).$$

Для класу *Stack* метрики зв'язності набувають таких значень:

$$TCC(Stack) = 7/10 = 0{,}7; \qquad LCC(Stack) = 10/10 = l.$$

Метрика *TCC* показує, що 70 % видимих методів класу *Stack* з'єднані прямо, а метрика *LCC* показує, що всі видимі методи класу *Stack* з'єднані прямо або непрямо.

Метрики *TCC* і *LCC* індикують ступінь зв'язаності між видимими методами класу. Видимі методи або визначені в класі, або успадковані ним. Корисними є метрики зв'язності для видимих методів, які визначені лише всередині класу; тут виключається вплив зв'язності суперкласу. Очевидно, що метрики локальної зв'язності класу визначаються на основі локального абстрактного класу. Зазначимо, що для локальної зв'язності екземплярні змінні і викликувані методи можуть включати успадковані змінні.

З'єднання об'єктів. Класичний метод Л. Констентайна і Е. Йордана визначає шість типів з'єднань, які орієнтовані на процедурне проектування. Принципова перевага об'єктно-орієнтованого проектування полягає в тому, що природа об'єктів приводить до створення слабкоз'єднаних систем. Фундаментальна властивість об'єктно-орієнтованого проектування полягає в прихованості змісту об'єкта. Як правило, зміст об'єкта невидимий зовнішнім елементам. Ступінь автономності об'єкта досить висока. Будь-який об'єкт може бути заміщений іншим об'єктом з таким же інтерфейсом.

Проте спадкування в об'єктно-орієнтованих системах приводить до іншої форми зчеплення. Об'єкти, які успадковують властивості і операції, з'єднані з їх батьківськими класами. Зміни в батьківських класах потрібно проводити обережно, оскільки ці зміни поширюються на усі класи, які успадковують їх характеристики.

Таким чином, самі по собі об'єктно-орієнтовані механізми не гарантують мінімального з'єднання. Класи – потужний засіб абстракції даних. Їх уведення зменшило потік даних між модулями, а, отже, послабило з'єднання усередині системи. Однак кількість типів залежностей між модулями збільшилася. Виникли відношення спадкування, делегування, реалізації і т. ін. Більш різноманітним став склад модулів у системі (класи, об'єкти, вільні функції і процедури, пакети). Звідси висновок: вимірювання і регулювання з'єднання в об'єктно-орієнтованих системах є необхідними діями.

14.3. Залежність зміни між класами

Залежність зміни між класами CDBC (*Change Dependency Between Classes*) визначає потенційну кількість змін, необхідних після модифікації класу-сервера *SC* (*server class*) на етапі супроводження. Доти, поки реальна кількість необхідних змін класу-клієнта *CC* (*client class*) невідомі, *CDBC* указує кількість методів, на які впливає зміна *SC*.

Значення *CDBC* залежать від:

– ділянки видимості змінюваного класу-сервера всередині класу-клієнта (визначається типом відношень між *CS* і *CC*);

– типу доступу *CC* до *CS* (інтерфейсний доступ або доступ до реалізації).

Можливі типи відношень наведено в табл. 14.3, де n — кількість методів класу *CC*; α – кількість методів *CC*, з'єднаних потенційною зміною.

Таблиця 14.3

Відношення «клієнт–сервер» і залежність зміни

Тип відношення	α
SC не використовується класом *CC*.	0
SC – клас екземплярної змінної у класі *CC*.	n
Локальні змінні типу *SC*, використовуються усередині методів класу *CC*.	j
SC є батьківським класом *CC*.	n
SC є типом параметра для методів класу *CC*.	j
CC має доступ до глобальної змінної класу *SC*	n

Елементи класу-сервера *SC*, які доступні класу-клієнту *CC*, є причиною змін. Якщо клас *SC* є «зрілою» абстракцією, то передба-

чається, що його інтерфейс більш стабільний, ніж його реалізація. Таким чином, багато змін у реалізації *SC* можуть виконуватися без впливу на його інтерфейс. Тому вводиться фактор стабільності інтерфейсу для класу-сервера; позначається як k ($0 < k < 1$). Внесок доступу до інтерфейсу в залежність зміни можна врахувати множенням на $(1 - k)$.

Метрика для обчислення ступеня *CDBC* має вигляд:

$$; A = \sum_{\substack{\text{Доступ} \\ \text{до } i\text{-ії реалізації}}} \alpha_i + (1-k) \sum_{\substack{\text{Доступ} \\ \text{до } i\text{-го інтерфейсу}}} \alpha_i$$

CDBC(*CC*, *SC*) = min(*n*, *A*). Способи мінімізації *CDBC*:

1) обмеження доступу до інтерфейсу класу-сервера;

2) обмеження видимості класів-серверів (специфікаторами доступу *public*, *protected*, *private*).

Локальність даних LD (*Locality of Data*) – метрика, що відображає якість абстракції, реалізованої класом. Чим вища локальність даних, тим вища самодостатність класу. Ця характеристика впливає на такі зовнішні характеристики, як повторна використовуваність і тестованість класу.

Метрика *LD* – відношення кількості локальних даних у класі до загальної кількості даних, використовуваних цим класом.

Будемо використовувати термінологію мови *C*++. Позначимо $M_i (1 \le i \le n)$ методами класу. До них не будуть включатися методи зчитування / запису екземплярних змінних. Тоді формулу для обчислення локальності даних можна записати у вигляді

$$LD = \frac{\sum_{i=1}^{a} |L_i|}{\sum_{i=1}^{a} |T_i|},$$

де $L_i (1 \le i \le n)$ – множина локальних змінних, до яких мають доступ методи M_i (прямо або за допомогою методів зчитування / запису). Такими змінними є: непублічні екземплярні змінні класу, успадковані захищені екземплярні змінні їх батьківських класів, статичні змінні, локально визначені в M_i; $T_i (1 \le i \le n)$ – множина всіх змінних, використовуваних у M_i, крім динамічних локальних змінних, визначених у M_i.

Для забезпечення надійності оцінки вилучено всі допоміжні змінні, визначені в M_i, – вони не є значущими для проектування.

Захищена екземплярна змінна, успадкована класом *C*, є локальною змінною для його екземпляра (і отже, є елементом L_i), навіть якщо вона не оголошена в класі *C*. Використання такої змінної методами класу не шкодить локальності даних, однак це небажано, якщо є зацікавленість у зменшенні значення *CDBC*.

14.4. Метрики Чидамбера і Кемерера

У 1994 р. С. Чидамбер і К. Кемерер запропонували шість проектних метрик, орієнтованих на класи [45]. Набір Чидамбера–Кемерера найчастіше цитується в програмній індустрії і наукових дослідженнях. Розглянемо їх.

Метрика 1. Зважені методи, що припадають на клас *WMC* (*Weighted Methods Per Class*). Припустімо, що в класі *C* визначено *n* методів зі складністю c_1, ..., c_2, ..., c_n. Для оцінювання складності можна обрати будь-яку метрику складності (наприклад, цикломатичну складність). Головне – нормалізувати цю метрику так, щоб номінальна складність для методу набула значення 1. У цьому випадку

$$WMC = \sum_{i=1}^{n} C_i \ .$$

Кількість методів і їх складність є індикатором витрат на реалізацію і тестування класів. Крім того, чим більше методів, тим складніше дерево спадкування (усі підкласи успадковують методи їх батьків). Зі збільшенням кількості методів у класі його застосування стає дедалі більш специфічним і тим самим обмежується можливість багаторазового використання. Тому метрика *WMC* повинна мати визначене низьке значення. Часто застосовують спрощену версію метрики. Вважають, що $C_i = 1$, тоді *WMC* – кількість методів у класі.

Виявляється, що розраховувати кількість методів у класі досить складно. Можливі два протилежні варіанти розрахунку.

1. Розраховуються лише методи поточного класу. Успадковані методи ігноруються, оскільки вони вже підраховані в тих класах, де вони визначалися. Таким чином, інкрементність класу – прийнятний показник його функціональних можливостей. Найважливішим джерелом інформації для розуміння того, що робить клас, є його операції. Якщо клас не може відреагувати на повідомлення (наприклад, не має власного методу), тоді він відправляє повідомлення

батьківському.

2. Розраховуються методи, визначені в поточному класі, і всі успадковані методи. Цей підхід підкреслює важливість простору станів у розумінні класу (а не інкрементність класу).

Існує ряд проміжних варіантів. Наприклад, розраховуються поточні методи та прямо успадковані від батьків. Аргумент на користь цього підходу – на поводження дочірнього класу найбільше впливає спеціалізація батьківських класів.

На практиці прийнятний кожен з описаних варіантів. Головне – не змінювати варіанти розрахунків для забезпечення коректного збирання метричних даних.

Метрика *WMC* дає відносний ступінь складності класу. Якщо вважати, що всі методи мають однакову складність, то це буде лише кількість методів у класі. Існують рекомендації щодо складності методів. Наприклад, М. Лоренц вважає, що середня довжина методу повинна обмежуватися 8 рядками для *Smalltalk* і 24 рядками для *C++* [46]. Взагалі клас, що має максимальну кількість методів серед класів одного з ним рівня, є найскладнішим; імовірно, він специфічний для цього додатка і містить найбільшу кількість помилок.

Метрика 2. Висота дерева спадкування *DIT* (*Depth of Inheritance Tree*), що визначається як максимальна довжина шляху від листка до кореня дерева спадкування класів. Для показаної на рис. 14.3 ієрархії класів метрика *DIT* дорівнює 3.

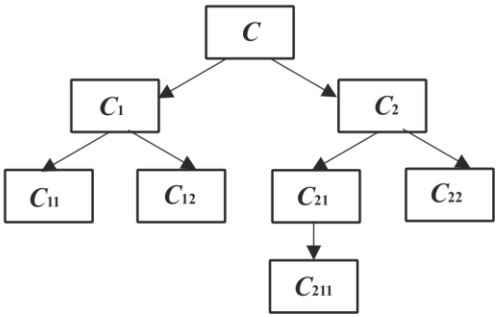


Рис. 14.3. Дерево спадкування класів

Відповідно, для окремого класу *DIT* – це довжина максимального шляху від заданого класу до кореневого класу в ієрархії класів.

Зі зростанням *DIT* імовірно, що класи нижнього рівня будуть успадковувати багато методів. Це утруднює пророкування поводження класу. Висока ієрархія класів (велике значення *DIT*) призводить до більшої складності проекту, оскільки залучає більшу кількість методів і класів.

Разом з тим, велике значення *DIT* означає, що методи можуть використовуватися багаторазово.

Метрика 3. Кількість дітей *NOC* (*Number of children*). Підкласи, які безпосередньо підлеглі суперкласу, називаються його дітьми. Значення *NOC* дорівнює кількості дітей, тобто кількості безпосередніх спадкоємців класу в ієрархії класів. На рис. 14.3 клас C_2 має двох дітей – підкласи C_{21} і C_{22}. Зі збільшенням *NOC* зростає багаторазовість використання, оскільки спадкування – це форма повторного використання.

Однак зі зростанням *NOC* послаблюється абстракція батьківського класу. Це означає, що в дійсності деякі з дітей уже не є членами батьківського класу і можуть бути неправильно використані. Кількість дітей характеризує потенційний вплив класу на проект. Зі зростанням *NOC* збільшується також кількість тестів, необхідних для перевірки кожної дитини.

Метрики *DIT* і *NOC* – кількісні характеристики форми і розміру структури класів. Добре структурована об'єктно-орієнтована система частіше буває організованою як ліс класів, ніж як надвисоке дерево. На думку Г. Буча, варто будувати збалансовані за висотою і шириною структури спадкування: зазвичай не вище і не ширше, ніж 7 ± 2 рівні чи гілки [22].

Метрика 4. З'єднання між класами об'єктів *CBO* (*Coupling Between Object Classes*) – кількість кооперацій, передбачених для класу, тобто кількість класів, з якими він з'єднаний. З'єднання означає, що методи цього класу використовують методи або екземплярні змінні іншого класу.

Інше визначення метрики: *CBO* дорівнює кількості з'єднань класу; з'єднання утворює виклик методу або властивості в іншому класі.

Ця метрика характеризує статичну складову зовнішніх зв'язків класів. Зі зростанням *CBO* багаторазовість використання класу, імовірно, зменшується. Чим більша незалежність класу, тим легше його повторно використовувати в іншому додатку.

Високе значення *CBO* ускладнює модифікацію і тестування, яке потрібне для виконання модифікації. Зрозуміло, що, чим більша кількість з'єднань, тим вища чутливість усього проекту до змін в окремих його частинах. Мінімізація міжоб'єктних з'єднань поліпшує модульність і сприяє інкапсуляції проекту.

Метрика *CBO* для кожного класу повинна мати низьке значення. Це узгоджується з рекомендаціями зі зменшення з'єднання стандартного ПЗ.

Метрика 5. Відгук для класу *RFC* (*Response For a Class*). Уводиться допоміжне визначення. Множина відгуків класу *RS* – це множина методів, які можуть виконуватися у відповідь на надходження повідомлень в об'єкт цього класу. Формула для визначення *RS* має вигляд

$$RS = \{M\} \cup_{all\_i} \{R_i\},$$

де $\{R_i\}$ – множина методів, що викликана методом *i*; $\{M\}$ – множина методів у класі. Метрика *RFC* дорівнює кількості методів у множині відгуку, тобто дорівнює потужності цієї множини:

$$RFC = \{RS\}.$$

Інакше *RFC* – це кількість методів тестованого класу та кількість методів інших класів, що викликані з цього класу. Метрика *RFC* є мірою потенційної взаємодії цього класу з іншими класами, дозволяє оцінювати динаміку поводження відповідного об'єкта в системі. Метрика *RFC* характеризує динамічну складову зовнішніх зв'язків класів.

Якщо у відповідь на повідомлення може бути викликана велика кількість методів, то тестування і налагодження класу ускладнюються, оскільки від розробника тестів потрібен більший рівень розуміння класу, збільшується довжина тестової послідовності. Зі зростанням *RFC* збільшується складність класу. Найбільша величина відгуку за часом може використовуватися для визначення часу тестування.

Метрика 6. Недолік зв'язності в методах *LCOM* (*Lack of Cohesion in Methods*). Кожний метод усередині класу звертається до одного або декількох властивостей (екземплярних змінних). Метрика *LCOM* показує, наскільки методи не зв'язані один з одним через властивості (змінні). Якщо всі методи звертаються до однакових властивостей, то *LCOM* = 0.

Уведемо позначення: незв'язані – кількість пар методів без загальних екземплярних змінних; зв'язані – кількість пар методів із загальними екземплярними змінними; I_j – набір екземплярних змінних, використовуваних методом M_j.

Очевидно, що

$$\text{незв'язані} = \{I_{ij} \mid I_i \cap I_j = 0\};$$
$$\text{зв'язані} = \{I_{ij} \mid I_i \cap I_j \neq 0\}.$$

Тоді формула для визначення недоліку зв'язності в методах набуває вигляду

$$LCOM = \begin{cases} \text{í åç'âÿçàí ³ – çâ'ÿçàí ³, ÿêù î (í åçâ'ÿçàí ³ } > \text{ çâ'ÿçàí ³;} \\ \text{0 â ³í ø î ì ó âèï àäêó.} \end{cases}$$

Можна визначити метрику по-іншому: *LCOM* – це кількість пар методів, незв'язаних через властивості класу, без кількості пар методів, що мають такий зв'язок.

Розглянемо приклади застосування метрики *LCOM*.

Приклад 14.1. У класі є методи: *M*1, *M*2, *M*3, *M*4. Кожний метод працює зі своїм набором екземплярних змінних:

$$I_1 = \{a, b\};\ I_2 = \{a, c\};\ I_3 = \{x, y\};\ I_4 = \{m, n\}.$$

У цьому випадку

$$\text{незв'язані} = \{I_{13}, I_{14}, I_{23}, I_{24}, I_{34}\} = 5;$$
$$\text{зв'язані} = card\ \{I_{12}\} = 1.$$

Розв'язання: *LCOM* = 5 – 1 = 4.

Приклад 14.2. У класі використовуються методи: *M*1, *M*2, *M*3. Для кожного методу задано набір екземплярних змінних:

$$I_1 = \{a, b\};\ I_2 = \{a, c\};\ I_3 = \{x, y\},$$
$$\text{незв'язані} = (I_{13}, I_{23}) = 2;\ \text{зв'язані} = (I_{12}) = 1,$$

Розв'язання: *LCOM* = 2 – 1 = 1.

Методи всередині класу повинні бути сильнозв'язаними, оскільки це сприяє інкапсуляції. Якщо *LCOM* має велике значення, то методи слабкозв'язані один з одним через властивості. Це збільшує складність, у зв'язку із чим зростає ймовірність помилок під час проектування.

Великі значення *LCOM* означають, що клас треба спроектувати краще (поділом на два або більше окремих класи). Будь-яке визначення *LCOM* допомагає виявити недоліки в проектуванні класів,

оскільки ця метрика характеризує якість упакування даних і методів в оболонку класу.

Висновок: зв'язність у класі бажано зберігати високою, тобто варто домагатися низького значення *LCOM*.

Набір метрик Чидамбера–Кемерера – одна з піонерських робіт з комплексної оцінки якості об'єктно-орієнтованого проектування. Відомі численні пропозиції щодо вдосконалення, розвитку цього набору. Так, недоліком метрики *WMC* є залежність від реалізації.

Приклад 14.3. Розглянемо клас, що пропонує операцію інтегрування.

Розв'язання. Можливі дві реалізації:

1) кілька простих операцій: *Set_interval* (*min*. *max*), *Setjnethod* (*method*), *Set_precision* (*precision*), *Set_function_to_integrate* (*function*), *Integrate*;

2) одна складна операція: *Integrate* (*function*, *min*, *max*. *method*, *precision*)

Для забезпечення незалежності від цих реалізацій можна визначити метрику *WMC*2:

$$WMC2 = \sum_{i=1}^{n} (\text{ê}^3\text{ë}\text{üê}^3\text{ñòü ï àðàì åòð}^3\text{â } i\text{-ãî ì åòî äó}) .$$

У наведеному прикладі *WMC*2 = 5 і для першої, і для другої реалізації. Зазначимо, що для першої реалізації *WMC* = 5, а для другої реалізації *WMC* = 1.

Додатково можна визначити метрику *середня кількість аргументів методу ANAM* (*Average Number of Arguments per Method*):

$$ANAM = WMC2 / WMC.$$

Метрика *ANAM* орієнтована на прийняті в об'єктно-орієнтованому проектуванні рішення – застосовувати прості операції з малою кількістю аргументів, а нескладні операції – із числовими
аргументами.

Ще одна пропозиція – увести метрику, симетричну метриці *LCOM*, яка має вигляд

$$LCOM = max \text{ (0, незв'язані – зв'язані),}$$

нормалізована NLCOM обчислюється за формулою:

$$NLCOM = \text{зв'язані / (незв'язані + зв'язані).}$$

Діапазон значень цієї метрики: $0 \le NLCOM \le 1$, причому чим ближче *NLCOM* до 1, тим більша зв'язність класу.

Набір Чидамбера–Кемерера не містить метрики для безпосереднього вимірювання інформаційної закритості класу, тому запропоновано метрику поведінкової закритості інформації *BIH* (*Behaviourial Information Hiding*):

$$BIH = (WEOC/WIEOC),$$

де *WEOC* – зважені зовнішні операції на клас (фактично це *WMC*); *WIEOC* – зважені внутрішні та зовнішні операції на клас; *WIEOC* обчислюється так само, як і *WMC*, але враховує повний набір операцій, реалізованих класом. Якщо *BIH* = 1, клас показує іншим класам всі свої можливості. Чим менше значення *BIH*, тим менш видиме поводження класу; *BIH* може розглядатися і як міра складності. Складні класи, імовірно, будуть мати малі значення *BIH*, а прості класи – значення, близькі до 1. Якщо клас з великим значенням *WMC* має значення *BIH*, близьке до 1, варто з'ясувати, чому він настільки видимий ззовні.

Використання метрик Чидамбера–Кемерера. Оскільки основу логічного подання ПЗ утворює структура класів, для оцінювання її якості зручно використовувати метрики Чидамбера–Кемерера. Приклад розрахунку метрик для структури, показаний на рис. 14.4, наведено в табл. 14.4.

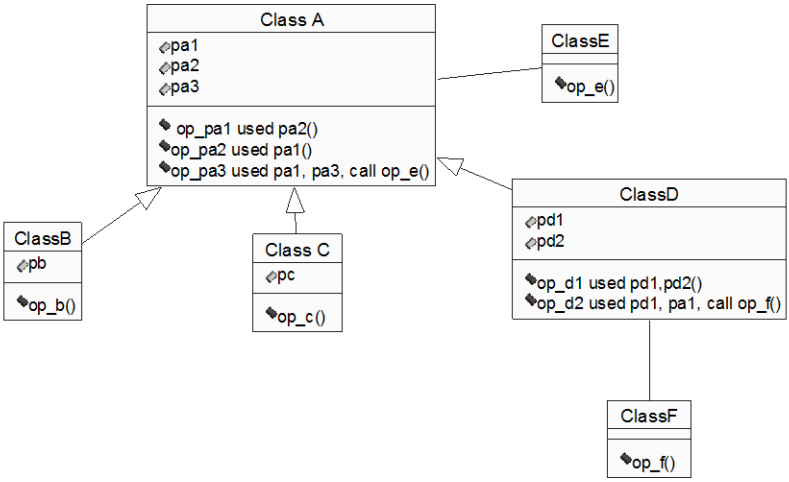


Рис. 14.4. Структура класів для розрахунку метрик Чидамбера–Кемерера

Таблиця 14.4

Розрахунок метрик Чидамбера-Кемерера

Ім'я класу	*WMC*	*DIT*	*NOC*	*CBO*	*RFC*
Class A	3	0	3	1	4
Class B	1	1	0	0	1
Class C	1	1	0	0	1
Class D	2	1	0	2	3

Прокоментуємо результати розрахунку. Клас *A* має три методи (*op_al*(), op_a2(), op_a3()), три дочірні класи (*B*, *C*, *D*) і є кореневим класом. Тому метрики *WMC*, *NOC* і *DIT* мають відповідно значення 3, 3 і 0.

Метрика *CBO* для класу *A* дорівнює 1, оскільки він використовує один метод з іншого класу (метод op_e() із класу *E*, він викликається з методу op_a3()). Метрика *RFC* для класу *A* дорівнює 4, тому що у відповідь на прибуття в цей клас повідомлень можливе виконання чотирьох методів (три оголошені в цьому класі, а четвертий метод op_e() викликається з op_a3()).

Для обчислення метрики *LCOM* визначається кількість пар методів класу. Вони розраховуються за формулою

$$C_m^2 = m!/(2(m-2)!),$$

де *m* – кількість методів класу.

Оскільки в класі три методи, можливі три пари: *op_al*()&op_a2(), *op_al*( )&op_a3() і op_a2( )&op_a3(). Перша і друга пари не мають загальних властивостей, третя пара має загальну властивість (*pal*). Таким чином, кількість незв'язаних пар дорівнює 2, кількість зв'язаних пар дорівнює 1 і *LCOM* = 2–1 = 1.

Відзначимо також, що для класу *D* метрика *CBO* дорівнює 2 – тут використовуються властивість *pal* і метод *op_f*() з інших класів. Метрика *LCOM* у цьому класі дорівнює 0, оскільки методи *op_dl*() і *op_d*2() зв'язані через властивості *pdl*, а від'ємне значення заборонено.

14.5. Метрики Лоренца і Кідда

Колекція метрик Лоренца і Кідда – результат практичного, промислового підходу до оцінювання проектів [45]. М. Лоренц і Д. Кідд поділяють метрики, які орієнтовані на класи, на чотири ка-

тегорії: *метрики розміру*, *метрики спадкування*, *внутрішні* та *зовнішні метрики*. Розмірно-орієнтовані метрики ґрунтуються на розрахунках властивостей і операцій для окремих класів, а також їх середніх значень для всієї об'єктно-орієнтованої системи. Метрики спадкування застосовують спосіб повторного використання операцій в ієрархії класів. Внутрішні метрики класів спрямовані на зв'язність і кодування. Зовнішні метрики досліджують з'єднання і повторне використання.

Метрика 1. Розмір класу *CS* (*Class Size*). Загальний розмір класу визначається вимірюваннями:

– загальною *кількістю операцій* (разом із власними та наслідуваними екземплярними), які інкапсулюються всередині класу;

– *кількості властивостей* (разом із власними та наслідуваними екземплярними), які інкапсулюються класом.

Метрика *WMC* Чидамбера–Кемерера також є зваженою метрикою розміру класу. Великі значення *CS* указують, що клас має надто багато обов'язків. Вони зменшують можливість повторного використання класу, ускладнюють його реалізацію і тестування. Визнаючи розмір класу, успадкованим (відкритим) операціям і властивостям надають більшої значущості. Причина – операції та властивості, які забезпечують спеціалізацію і більш локалізовані в проекті. Можуть обчислюватися середні кількості властивостей і операцій класу. Чим менший середній розмір, тим більша ймовірність повторного використання класу. Рекомендоване значення: $CS \leq 20$ методів.

Метрика 2. Кількість операцій, переобумовлених підкласом, *NOO* (*Number of Operations Overridden by a Subclass*). Перевизначенням називають випадок, коли підклас заміщує операцію, успадковану від батьківського класу, власною версією. Більші значення *NOO* вказують на проблеми проектування. Зрозуміло, що підклас повинен розширювати операції батьківського класу. Розширення набуває вигляду нових імен операцій. Якщо ж *NOO* велике, то розробник порушує абстракцію батьківського класу. Це послабляє ієрархію класів, ускладнює тестування і модифікацію ПЗ. Значення, що рекомендується: $NOO \leq 3$ методів.

Метрика 3. Кількість операцій, доданих підкласом, *NOA* (*Number of Operations Added by a Subclass*). Підкласи спеціалізуються через додавання власних операцій і властивостей. Зі зростанням *NOA* підклас віддаляється від абстракції батьківського кла-

су. Зі збільшенням висоти ієрархії класів (збільшенням *DIT*) має зменшуватися значення *NOA* на нижніх рівнях ієрархії.

Для значень *CS* = 20 і *DIT* = 6 рекомендується *NOA* ≤ 4 методів.

Метрика 4. Індекс спеціалізації *SI* (*Specialization Index*). Забезпечує грубу оцінку ступеня спеціалізації кожного підкласу. Спеціалізація досягається додаванням, видаленням або перевизначенням операцій:

$$SI = (NOO \times рівень) / M_{заг},$$

де *рівень* – номер рівня в ієрархії, на якому перебуває підклас; $M_{заг}$ – загальна кількість методів класу. Приклад розрахунку індексів спеціалізації показано на рис. 14.5.

Чим більше значення *SI*, тим більша ймовірність того, що в ієрархії класів є класи, які порушують абстракцію батьківського класу. Рекомендоване значення *SI* ≤ 0,15.

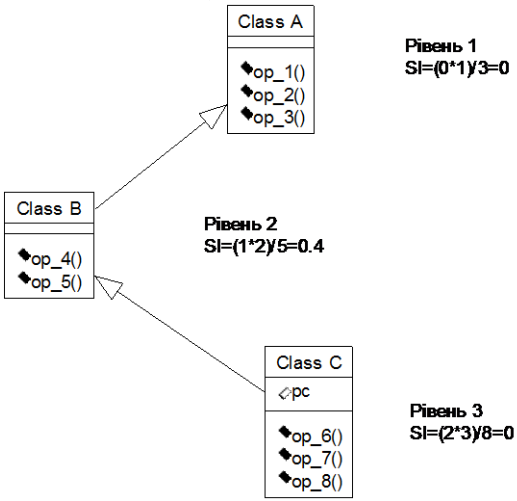


Рис. 14.5. Розрахунок індексів спеціалізації класів

Операційно-орієнтовані метрики орієнтовані на оцінку операцій у класах. Методи мають тенденцію бути невеликими як за розміром, так і за логічною складністю. Проте реальні характеристики операцій можуть бути корисні для глибокого розуміння системи.

Метрика 5. Середній розмір операції OS_{AVG} (*Average Operation Size*). Як індикатор розміру може використовуватися кількість рядків програми, однак *LOC*-оцінки призводять до відомих проблем.

Альтернативний варіант – *кількість повідомлень, посланих операцією.* Зростання значення показника означає, що обов'язки розміщені в класі не дуже вдало. Рекомендоване значення $OS_{AVG} \leq 9$.

Метрика 6. Складність операції *OC* (*Operation Complexity*). Складність операції можна визначати за стандартними метриками складності, тобто за допомогою *LOC* або *FP*-оцінок, метрики цикломатичної складності, метрики Холстеда.

М. Лоренц і Д. Кідд пропонують обчислювати *OC* підсумовуванням оцінок з ваговими коефіцієнтами, наведеними в табл. 14.5.

Таблиця 14.5

Вагові коефіцієнти для метрики ОС

Параметр	Ваговий коефіцієнт
Виклики функцій API	5,0
Присвоювання	0,5

Закінчення табл. 14.5

Параметр	Ваговий коефіцієнт
Арифметичні операції	2,0
Повідомлення з параметрами	3,0
Вкладені вирази	0,5
Локальні параметри	0,3
Прості виклики	7,0
Тимчасові змінні	0,5
Повідомлення без параметрів	1,0

Оскільки операція обмежується конкретним обов'язком, бажано зменшувати *OC*. Для запропонованого підсумовування рекомендується значення $OC \leq 65$.

Метрика 7. Середня кількість параметрів на операцію NP_{AVG} (*Average Number of Parameters per operation*). Чим більше параметрів в операції, тим складніша кооперація між об'єктами. Тому значення NP_{AVG} має бути якомога меншим. Рекомендоване значення $NP_{AVG} = 0{,}7$.

Основними завданнями менеджера проекту є планування, координація, відстеження робіт і керування програмним проектом. Одним із ключових питань планування є оцінювання розміру програмного продукту. Прогноз розміру продукту забезпечують об'єктно-орієнтовані метрики.

Метрика 8. Кількість описів сценаріїв *NSS* (*Number of Scenario Scripts*). Ця кількість прямо пропорційна кількості класів, необхідних для реалізації вимог, кількості станів для кожного класу, а також кількості методів, властивостей і кооперацій. Метрика *NSS* – ефективний індикатор розміру програми.

Рекомендоване значення *NSS* – не менше одного сценарію на публічний протокол підсистеми, що відображає основні функціональні вимоги до підсистеми.

Метрика 9. Кількість ключових класів *NKC* (*Number of Key Classes*). Ключовий клас безпосередньо пов'язаний з комерційною проблемною галуззю, для якої призначена система. Малоймовірно, що ключовий клас може виникнути в результаті повторного використання існуючого класу. Тому значення *NKC* імовірно відображає майбутній обсяг проекту. М. Лоренц і Д. Кідд припускають, що в типові об'єктно-орієнтованій системі на частку ключових класів припадає 20–40 % від загальної кількості класів. Як правило, класи, що залишилися, реалізують загальну інфраструктуру (*GUI*, комунікації, бази даних).

Рекомендоване значення: якщо *NKC* < 0,2 від загальної кількості класів системи, варто поглибити дослідження проблемної галузі (для виявлення найважливіших абстракцій, які потрібно реалізувати).

Метрика 10. Кількість підсистем *NSUB* (*Number of SUBsystem*). Кількість підсистем дає змогу з'ясувати: розміщення ресурсів, планування (з акцентом на одночасне розроблення), загальні витрати на інтеграцію.

Рекомендоване значення: *NSUB* > 3. Значення метрик *NSS*, *NKC*, *NSUB* корисно нагромаджувати як результат кожного виконаного об'єктно-орієнтованого проекту. Так формується метричний базис, у який також включаються метричні значення за класами і операціями. Ці дані можна використовувати для обчислення метрик продуктивності (середню кількість класів на розробника або середню кількість методів на людино-місяць). Спільне застосування метрик дозволяє оцінювати витрати, тривалість, персонал і інші характеристики поточного проекту.

14.6. Набір метрик Фернандо Абреу

Набір метрик *MOOD* (*Metrics for Object Oriented Design*), запропонований Ф. Абреу в 1994 р., – інший приклад академічного під-

ходу до оцінювання якості об'єктно-орієнтованого проектування [47]. Основними завданнями *MOOD* набору є:

1) покриття базових механізмів об'єктно-орієнтованої парадигми, таких як інкапсуляція, спадкування, поліморфізм, відправлення повідомлень;

2) формальне визначення метрик, що дозволяє уникнути суб'єктивності виміру;

3) незалежність від розміру оцінюваного програмного продукту;

4) незалежність від мови програмування, якою виконаний оцінюваний продукт.

Набір *MOOD* містить у собі такі (фактори) метрики:

– закритість методу (*MH*);

– закритість властивості (*AHF*);

– спадкування методу (*MIF*);

– спадкування властивості (*AIF*);

– поліморфізму (*POF*);

– з'єднання (*CO*).

Кожна із цих метрик належить до основного механізму об'єктно-орієнтованої парадигми: інкапсуляції (*MH* і *AH*), спадкування (*MIF* і *AIF*), поліморфізму (*POF*) і відправлення повідомлень (*CO*). У визначеннях *MOOD* не використовуються специфічні конструкції мов програмування.

Метрика 1. Фактор закритості методу *MHF* (*Method Hiding Factor*). Уводяться позначення: $M_v(C_i)$ – кількість видимих методів у класі C_i (інтерфейс класу); $M_h(C_i)$ – кількість прихованих методів у класі C_i (реалізація класу); $M_d(C_i)$ – загальна кількість методів, наявних у класі C (успадковані методи не враховуються) $M_d(C_i) = M_v(C_i) + M_h(C_i)$.

Тоді формула метрики *MH* набуває вигляду

$$MHF = \frac{\sum_{i=1}^{TC} M_h(C_i)}{\sum_{i=1}^{TC} M_d(C_i)},$$

де *TC* – кількість класів у системі. Якщо видимість *m*-го методу *i*-го класу з *j*-го класу обчислювати виразом:

$$is\_visible(M_{mi}, C_j) = \begin{cases} 1, \text{if} \begin{cases} j \neq 1, \\ C_j \text{ може викликати } M_{mi}, \end{cases} \\ 0, \text{else}, \end{cases}$$

а відсоткову кількість класів, які належать до *m*-го методу *i*-го класу, за співвідношенням

$$V(M_{mi}) = \frac{\sum_{i=1}^{TC} is\_visible(M_{mi}, C_j)}{TC - 1},$$

то формулу метрики *MH* можна подати у вигляді

$$MHF = \frac{\sum_{i=1}^{TC} \sum_{m=1}^{M_d(C_i)} (1 - V(M_{mi}))}{\sum_{i=1}^{TC} M_d(C_i)}. \qquad (14.1)$$

У чисельнику формули (14.1) подано суму закритості всіх методів у всіх класах. Закритість методу – відсоткова кількість класів, для яких метод не доступний. Знаменник – це загальна кількість методів, наявних у розглянутій системі.

Зі збільшенням *MH* зменшуються щільність дефектів у системі і витрати на їх усунення. Розроблення класу являє собою покроковий процес, за якого до класу додається дедалі більше деталей (прихованих методів). Така схема розроблення сприяє зростанню як значення *MH*, так і якості класу.

Метрика 2. Фактор закритості властивості *AHF (Attribute Hiding Factor)*. Уводяться позначення: $A_v(C_i)$ – кількість видимих властивостей у класі C_i (інтерфейс класу); $A_h(C_i)$ – кількість схованих властивостей у класі C_i (реалізація класу); $A_d(C_i)$ – загальна кількість властивостей, наявних у класі C_i (успадковані властивості не враховуються); $A_d(C_i) = A_v(C_i) + A_h(C_i)$. Тоді формула метрики *AHF* набуває вигляду:

$$AHF = \frac{\sum_{i=1}^{TC} A_h(C_i)}{\sum_{i=1}^{TC} A_d(C_i)},$$

де *TC* – кількість класів у системі. Якщо видимість *m*-ї властивості *i*-го класу з *j*-го класу обчислювати за виразом

$$is\_visible(A_{mi}, C_j) = \begin{cases} 1, \text{if} \begin{cases} j \neq 1, \\ C_j \text{ може викликати } A_{mi}, \end{cases} \\ 0, \text{else}, \end{cases}$$

а відсоткову кількість класів, які бачать *m*-у властивість *i*-го класу, за співвідношенням

$$V(A_{mi}) = \frac{\sum_{i=1}^{TC} is\_visible(A_{mi}, C_j)}{TC - 1},$$

то формулу метрики *AHF* можна подати у вигляді

$$AHF = \frac{\sum_{i=1}^{TC} \sum_{m=1}^{A_d(C_i)} (1 - V(A_{mi}))}{\sum_{i=1}^{TC} A_d(C_i)}. \qquad (14.2)$$

У чисельнику формули (14.2) подано суму закритості всіх властивостей у всіх класах. Закритість властивості – відсоткова кількість класів, для яких ця властивість недоступна. Знаменник – загальна кількість властивостей, наявних у розглянутій системі.

В ідеальному випадку всі властивості повинні бути приховані і доступні лише для методів відповідного класу (*AHF* = 100 %).

Метрика 3. Фактор спадкування методу *MIF* (*Method Inheritance Factor*). Уводяться позначення: $M_i(C_i)$ – кількість успадкованих і неперевизначених методів у класі C_i; $M_0(C_i)$ – кількість успадкованих і перевизначених методів у класі C_i; $M_n(C_i)$ – кількість нових (неуспадкованих і перевизначених) методів у класі C_i; $M_d(C_i)$ – кількість методів, визначених у класі C_i, $M_d(C_i) = M_n(C_i) + M_0(C_i)$; $M_a(C_i)$ – загальна кількість методів, доступних у класі C_i, $M_a(C_i) = M_d(C_i) + M_i(C_i)$. Тоді формула метрики *MIF* набуває вигляду

$$MIF = \frac{\sum_{i=1}^{TC} M_i(C_i)}{\sum_{i=1}^{TC} M_i(C_i)}. \qquad (14.3)$$

У чисельнику формули (14.3) подано суму успадкованих (і неперевизначених) методів у всіх класах розглянутої системи. Знаменник – загальна кількість доступних методів (локально визначених і успадкованих) для всіх класів.

Значення *MIF* = 0 указує, що в системі не відбувається ефективного спадкування, наприклад, усі успадковані методи перевизначені. Зі збільшенням *MIF* зменшуються щільність дефектів і витрати на виправлення помилок. Великі значення *MIF* (70–80 %) призводять до зворотного ефекту. Таким чином, помірне використання спадкування – засіб, придатний для зниження щільності дефектів і витрат на доробку.

Метрика 4. Фактор спадкування властивості *AIF* (*Attribute Inheritance Factor*). Уведемо позначення кількості: $A_i(C_i)$ – успад-

кованих і неперевизначених властивостей у класі C_i; $A_0(C_i)$ – успадкованих і перевизначених властивостей у класі C_i; $A_n(C_i)$ – нових (неуспадкованих і перевизначених) властивостей у класі C_i; $A_d(C_i)$ – властивостей, певних у класі C_i, $A_d(C_i) = A_n(C_i) + A_0(C_i)$; $A_a(C_i)$ – кількість властивостей, доступних у класі C_i, $A_a(C_i) = A_d(C_i) + A_i(C_i)$.

Тоді формула для *AIF* набуде вигляду

$$AIF = \frac{\sum_{i=1}^{TC} A_i(C_i)}{\sum_{i=1}^{TC} A_a(C_i)}. \qquad (14.4)$$

У чисельнику формули (14.4) подано суму успадкованих (і неперевизначених) властивостей у всіх класах розглянутої системи. Знаменник – загальна кількість доступних властивостей (локально визначених і успадкованих) для всіх класів.

Метрика 5. Фактор поліморфізму *POF* (*Polymorphism Factor*). Уведемо позначення: $M_0(C_i)$ – кількість успадкованих і перевизначених методів у класі C_i; $M_n(C_i)$ – кількість нових (неуспадкованих і перевизначених) методів у класі C_i; $DC(C_i)$ – кількість нащадків класу C_i; $M_d(C_i)$ – кількість методів, визначених у класі C_i, $M_d(C_i) = M_n(C_i) + M_0(C_i)$.

Тоді формула для *POF* набуває вигляду

$$POF = \frac{\sum_{i=1}^{TC} M_0(C_i)}{\sum_{i=1}^{TC} [M_n(C_i) DC(C_i)]}. \qquad (14.5)$$

У чисельнику формули (14.5) фіксується реальна кількість можливих поліморфних ситуацій. Очевидно, що повідомлення, відправлене в клас C_i, зв'язується (статично або динамічно) з реалізацією іменованого методу. Цей метод, у свою чергу, може або подаватися декількома формами, або перевизначатися (у нащадках C_i). Знаменник – максимальна кількість можливих поліморфних ситуацій для класу C_i. Тобто, коли всі нові методи, визначені в C_i, перевизначаються у всіх його нащадках.

Помірне використання поліморфізму зменшує як щільність дефектів, так і витрати на доробку. Однак, якщо *POF* > 10 %, можливий зворотний ефект.

Метрика 6. Фактор з'єднання *COF* (*Coupling Factor*). У цьому наборі з'єднання фіксує наявність між класами відношення «клі-

єнт–постачальник» (*client–supplier*). Відношення «клієнт–постачальник» (C_c =>C_s) тут означає, що клас-клієнт містить щонайменше одне неуспадковане посилання на властивість або метод класу-постачальника. Якщо відношення «клієнт–постачальник» визначати за виразом

$$is\_client(C_c, C_s) = \begin{cases} 1, if\ C_c => C_s \cap C_c \neq C_s, \\ 0, else, \end{cases}$$

то формула для обчислення метрики *COF* набуває вигляду

$$COF = \frac{\sum_{i=1}^{TC}[\sum_{j=1}^{TC} is\_client(C_i, C_j)]}{TC^2 - TC}. \qquad (14.6)$$

Знаменник у формулі (14.6) відповідає максимально можливій кількості з'єднань у системі із *TC*-класами (потенційно кожний клас може бути постачальником для інших класів). Рефлексивні відношення не розглядаються, коли клас є власним постачальником.

Чисельник – фіксує реальну кількість з'єднань, що не належать до спадкування.

Із посиленням з'єднань класів щільність дефектів і витрати на доробку також зростають. З'єднання негативно впливає на якість ПЗ, їх потрібно зводити до мінімуму. Практичне застосування цієї метрики доводить, що з'єднання збільшує складність, зменшує інкапсуляцію і можливості повторного використання, утруднює розуміння й ускладнює супроводження ПЗ.

14.7. Метрики для об'єктно-орієнтованого тестування

Розглянемо проектні метрики, які, на думку Р. Байндера, безпосередньо впливають на тестованість об'єктно-орієнтованих систем [48]. Р. Байндер згрупував ці метрики в три категорії, що відображають найважливіші проектні характеристики.

Метрики інкапсуляції. Недостатня зв'язність у методах *LCOM*, відсоток публічних і захищених методів *PAP* (*Percent Public and Protected*) і публічний доступ до компонентних даних *PAD* (*Public Access to Data members*).

Метрика 1. Недостатня зв'язність у методах *LCOM*. Чим більше значення *LCOM*, тим більше станів треба тестувати, щоб гарантувати відсутність побічних ефектів під час роботи методів.

Метрика 2. Відсоток відкритих і захищених методів *PAP*. Публічні властивості успадковуються від інших класів і тому видимі для цих класів. Захищені властивості є спеціалізацією і належать до певного підкласу. Ця метрика показує відсоток публічних властивостей класу. Великі значення *PAP* підвищують імовірність побічних ефектів у класах. Тести повинні гарантувати виявлення побічних ефектів.

Метрика 3. Відкритий доступ до компонентних даних *PAD*. Метрика показує кількість класів (або методів), які мають доступ до властивостей інших класів, тобто порушують їх інкапсуляцію. Великі значення зумовлюють виникнення побічних ефектів у класах. Тести повинні гарантувати виявлення таких побічних ефектів.

14.7.1. Метрики спадкування

До метрик спадкування належать і кількість кореневих класів *NOR* (*Number Of Root classes*), коефіцієнт об'єднання по входу *FIN*, кількість дітей *NOC* і висота дерева спадкування *DIT*.

Метрика 1. Кількість кореневих класів *NOR* (*Number Of Root classes*). Ця метрика підраховує кількість дерев спадкування в проектній моделі. Для кожного кореневого класу та дерева спадкування потрібно розробляти набір тестів. Зі збільшенням *NOR* зростають витрати на тестування.

Метрика 2. Коефіцієнт об'єднання на вході *FIN*. У контексті об'єктно-орієнтованих систем *FIN* фіксує множинне спадкування. Значення *FIN* > 1 указує, що клас успадковує свої властивості і операції від декількох кореневих класів. Варто уникати *FIN* > 1 скрізь, де це можливо.

Метрика 3. Кількість дітей *NOC*. Метрику запозичено з набору Чидамбера–Кемерера.

Метрика 4. Висота дерева спадкування *DIT*. Метрику запозичено з набору Чидамбера–Кемерера. Методи батьківського класу потрібно повторно тестувати для кожного підкласу.

14.7.2. Метрики поліморфізму

До метрик поліморфізму належать: відсоткова кількість неперевизначених запитів *OVR*, відсоткова кількість динамічних запитів *DYN*, стрибок класу *Bounce-C* і стрибок системи *Bounce-S*.

Метрика 1. Відсоткова кількість неперевизначених запитів *OVR*. Відсоткова кількість від усіх запитів у тестованій системі, які не призводять до перекриття модулів. Перекриття може зумовлювати непередбачену зв'язність. Велике значення *OVR* збільшує можливості виникнення помилок.

Метрика 2. Відсоткова кількість динамічних запитів *DYN*. Відсоткова кількість від усіх повідомлень у тестованій системі, приймачі повідомлень визначаються в період виконання. Динамічна зв'язність може зумовлювати непередбачене зв'язування. Велике значення *DYN* означає, що для перевірки всіх варіантів зв'язності методу потрібно багато тестів.

Метрика 3. Стрибок класу *Bounce-C*, кількість скачкоподібних маршрутів видимих тестованому класу. Скачкоподібний – це маршрут, що в ході динамічного зв'язування перетинає кілька ієрархій класів-постачальників. Стрибок може призводити до непередбаченої зв'язності. Велике значення *Bounce-C* збільшує можливості виникнення помилок.

Метрика 4. Стрибок системи *Bounce-S*. Кількість скачкоподібних маршрутів у тестованій системі. У цій метриці підсумовується кількість маршрутів, що перебігають по кожному класу системи. Велике значення *Bounce-S* збільшує можливості виникнення помилок.

Контрольні запитання і завдання

1. Які фактори об'єктно-орієнтованих систем впливають на метрики для їх оцінювання, як проявляється цей вплив?
2. Як впливає спадкування на зв'язність класів?
3. Які характеристики об'єктно-орієнтованих систем погіршують зв'язність класів?
4. Поясніть, як визначити зв'язність класів за допомогою метрики «залежність зміни між класами».
5. Які метрики входять у набір Чидамбера–Кемерера? Які завдання вони вирішують?
6. Як можна підраховувати кількість методів у класі?
7. Які метрики Чидамбера–Кемерера оцінюють зв'язність класів? Поясніть їх зміст.
8. Яка метрика Чидамбера–Кемерера оцінює зв'язність класу? Поясніть її зміст.
9. Як домогтися незалежності метрики *WMC* від реалізації?

10. Як оцінити інформаційну закритість класу?
11. Порівняйте набори Чидамбера–Кемерера та Лоренца–Кідда. У чому полягає їх розбіжність та подібність?
12. На які цілі орієнтований набір метрик Фернандо Абреу?
13. Порівняйте набори Чидамбера–Кемерера і Фернандо Абреу. У чому полягає їх розбіжність та подібність?
14. Порівняйте набори Лоренца–Кідда і Фернандо Абреу. У чому полягає їх розбіжність та подібність?
15. Охарактеризуйте метрики об'єктно-орієнтованого тестування.

ДОДАТОК

Приклад контрольного запитальника для перевірки зручності застосування *web*-додатків

Архітектура і навігація сайту
Чи відповідає структура сайту цілям, для досягнення яких він призначений?
Чи зрозуміла схема навігації?
Чи можна визначити, у якому місці сайту ви перебуваєте?
Чи легко знайти на сайті потрібну інформацію?
Чи прийнятною є кількість елементів у панелях навігації?
Чи логічно відсортовані елементи?
Чи відповідають назви гіперпосилань назвам сторінки?
Чи чітко виділено гіперпосилання ?
Чи чітко виділено посилання на головну сторінку?
Чи існує можливість пошуку інформації на сайті?
Чи існує карта сайту?
Чи кожна сторінка дозволяє зрозуміти, на якому сайті ви перебуваєте?
Чи можна керувати навігацією по сайту?
Планування і дизайн сайту
Чи відповідає розмір сторінки розміру вікна?
Чи повторюється схема планування на всіх сторінках?
Чи існує виразний фокус на кожній сторінці?
Чи видно візуально планування?
Чи ефективно використовується вирівнювання та угруповання?
Чи сайт естетичний?
Зміст сайту
Чи зрозумілі та лаконічні тексти на сайті?
Чи організований текст у вигляді невеликих блоків?
Чи трапляються в тексті граматичні й орфографічні помилки?
Чи містять сторінки вступний текст?
Чи підтримують мультимедійні компоненти завдання користувача?
Чи є одиниці вимірювання, використовувані на сайті, зрозумілими?
Чи зазначені на сайті час і дата створення сторінок?

Чи зазначені на сайті номери контактних телефонів?
Чи подані на сайті адреси з поштовими індексами?
Форми і взаємодія
Чи відповідають форми завданням користувача?
Чи підтримують діалоги логічну послідовність кроків?
Чи видно кнопку або посилання для переходу до наступного кроку діалогу?
Чи всі елементи форм використовуються за призначенням?
Чи згруповані елементи форми за значеннєвим призначенням?
Чи зрозумілим є вигляд кнопки відправлення форми?
Графіка
Чи прийнятна якість використовуваної графіки ?
Чи всі графічні елементи мають альтернативні текстові написи?
Чи містять графічні елементи інформацію про розмір файла?
Чи оптимізовані графічні елементи для передавання по Інтернету?
Чи реагують графічні елементи на рухи миші?
Чи використовується анімація?
Чи прийнятний обсяг графічних файлів?
Кольори
Чи прийнятний вибір кольорів для сайту?
Чи не багато кольорів?
Чи використовуються кольори логічно і послідовно?
Чи розрізняються кольори в чорно-білому режимі?
Оформлення тексту
Чи зрозумілі тексти?
Чи прийнятний розмір шрифту?
Чи має шрифт придатний колір і чи достатньо він контрастний?
Чи відформатований текст так, щоб у рядку було від 10 до 12 слів?
Чи достатня ширини поля навколо тексту?
Стійкість до помилок
Чи користувач здатний запам’ятати зміст сторінки?
Чи з’являється попередження у разі спроби вчинити незворотні дії?
Чи можна скасувати ризиковані дії?
Чи перехоплюються виникаючі помилки локально без звертання до сервера?

Закінчення додатка

Чи містять сторінки з повідомленням про виниклі помилки корисну інформацію?
Чи є сторінки з порожніми результатами пошуку?
Чи існує система контекстної допомоги (довідки)?
Чи структурована довідкова інформація користувачу? Чи зрозумілі дії користувачу?
Платформа та особливості реалізації
Чи швидко завантажуються сторінки (від 3 до 15 с)?
Чи всі гіперпосилання працюють правильно?
Чи є пошкоджені графічні елементи?
Чи написаний текст на сторінках так, щоб його могли знайти пошукові системи?
Чи працює сайт із різними браузерами користувача?
Чи працює сайт на моніторах високої і низької роздільної здатності?
Чи використовуються нестандартні *plug-in*? Чи потрібні вони і корисні?

ПРЕДМЕТНИЙ ПОКАЖЧИК

СПИСОК ЛІТЕРАТУРИ

1. *Сидоров М. О.* Вступ до інженерії ПЗ / М. О. Сидоров. – К. : Вид-во НАУ «НАУ-друк», 2010. – 112 с.

2. *Програмна* інженерія. – [Електронний ресурс]. – Режим доступу: *http://www.programsfactory.univ.kiev.ua/content/books*/2.

3. *Guide* to the Software Engineering Body of Knowledge (SWEBOK) – [Електронний ресурс]. – Режим доступа: http://www.webcitation.org/6I9hPdr8K

4. *Сидоров М. О.* Методи і засоби створення і супроводження ПЗ великих розподілених комп'ютеризованих інформаційних систем / М. О. Сидоров // Інженерія ПЗ. – Т.20. – №4. – 2014. – С. 30–38.

5. *Соммервил И.* Инженерия программного обеспечения / И. Соммервил. – М. : Вильямс, 2002. – 624 с.

6. *Система* розроблення та поставлення продукції на виробництво. Правила виконання науково-дослідних робіт. Загальні положення : ДСТУ 3973–2000. – [Чинний від 2001–07–01]. – К. : Держстандарт, 2001. – 20 с.

7. *Система* розроблення та поставлення продукції на виробництво. Правила виконання дослідно-конструкторських робіт. Загальні положення: ДСТУ 3974-2000. – [Чинний від 2000–07–01]. – К. : Держстандарт, 2001. – 38 с.

8. *Systems* and software engineering – Sistems and software quality requirements and evaluation (SQuaRE) – Planning and management: ISO / IEC 25001:2014. – [Publication 2014–03–15]. – 13 p. – [Електронний ресурс]. – Режим доступу: https://www.iso.org/obp/ui/#iso:std:iso-iec:25001:ed-2:v1:en.

9. *Systems* and software engineering – Sistems and software quality requirements and evaluation (SQuaRE) – Evaluation for developers, acquirers and independent evaluators: ISO / IEC 25041:2012. – [Publication 2012–10–15]. – 52 p. – [Електронний ресурс]. – Режим доступу: https://www.iso.org/obp/ui/#iso:std:iso-iec:25041:ed-1:v1:en.

10. *Software* engineering – Product evaluation – Part 6: Documentation of evaluation modules: ISO / IEC 14598 – 6: 2001. – [Publication 2001–06–01]. – 31 p. – [Електронний ресурс]. – Режим доступу: https://www.iso.org/obp/ui/#iso:std:iso-iec:14598:-6:ed-1:v1:en.

11. *Stallings W.* High-Speed Networks and Інтернетs. Perfomance and Qaulity of Service / Stallings W. – N.-J: Prentice Hall PTR. – 2002. – 783 p.

12. *Растригин Л. А.* Адаптация сложных систем: Методы и приложения / Л. А. Растригин. – Рига : Зинатне, 1981. – 375 с.

13. *Буч Г.* Объектно-ориентированный анализ и проектирование с примерами приложений / Г. Буч. – М. : Вильямс, 2008. – 720 с.

14. *Бабенко Л. П.* Основи програмної інженерії / Л. П. Бабенко. – К. : Т-во «Знання», КОО, 2001. – 269 с.

15. *Буч Г.* Унифицированный процесс разработки программного обеспечения / Г. Буч, А. Якобсон, Дж. Рамбо. – СПб. : Питер, 2002. – 496 с.

16. *Кобер А.* Современные методы описания функциональных требований к системам / А. Кобер. – М. : Лори, 2002. – 670 с.

17. *Константайн Л.* Разработка программного обеспечения / Л. Константайн, Л. Локвуд. – СПб. : Питер, 2004. – 592 с.

18. *Иванова Г.С.* Технология программирования: учеб. для вузов / Г.С. Иванова. – М. : Изд-во МГТУ им. Н.Э. Баумана, 2002. – 320 с.

19. *Липаев В. В.* Проектирование программных средств / В. В. Липаев. – М. : Высш. шк., 1990. – 452 с.

20. *Опалева Э. А.* Технология разработки программного обеспечения / Э. А. Опалева, В. П. Самойленко. – Л. : ЛЭТИ, 1988. – 358 с.

21. *Брауде Э. Дж.* Технологии разработки программного обеспечения / Э. Дж. Брауде. – СПб. : Питер, 2004. – 389 с.

22. *Орлов С. А.* Программная инженерия [Електронний ресурс] / С. А. Орлов. – СПб. : Питер, 2016. – 640 с. – Режим доступу : https://books.google.com.ua/books?id=_9ZLDAAAQBAJ&pg.

23. *Трофимов С. А.* C*ase*-технологии. Практическая работа в Rational Rose / С. А. Трофимов. – М. : Бином-Пресс, 2002. – 288 с.

24. *IBM Rational Rose*. – [Електронний ресурс]. – Режим доступу: ftp://public.dhe.ibm.com/software/rational/web/datasheets/rose_ds.pdf

25. *Systems* and software engineering – Vocabulary. ISO/IEC/IEEE 24765. [Електронний ресурс]. – Режим доступу: https://webstore.iec.ch/ preview/info_isoiecieee24765 %7Bed1.0 %7Den.pdf.

26. *Куликов С.* Тестирование программного обеспечения : базовый курс [Електронний ресурс] / С. Куликов. – *EPAM Systems, RD. Dep.*, 2016. – 289 с. – Режим доступа : svyatoslav.biz/ software_testing_book_download/

27. *Програмні* засоби ЕОМ. Забезпечення якості. Терміни та визначення: ДСТУ 2844:94. – [Чинний від 1996–01–01]. – К. : Держстандарт, 1996. – 12 с.

28. *Програмні* засоби ЕОМ. Підготовлення і проведення випробувань: ДСТУ 2853:94. – [Чинний від 1996–01–01]. – К. : Держстандарт, 1996. – [Чинний від 1996–01–01]. – 20 с.

29. *Systems* and software engineering – Software life cycle processes: ISO/IEC 12207:2008 [Електронний ресурс]. – Режим доступу: http://www.iso.org/iso/iso_catalogue/catalogue_tc/catalogue_detail.htm?csnumber=43447.

30. *SWEBOK V*3 [Електронний ресурс]. – Режим доступу: *https*://*www.computer.org*/*web*/*swebok*/*v*3

31. *Тамре Л.* Введение в тестирование программного обеспечения / Л. Тамре. – М. : Вильямс, 2003. – 368 с.

32. *IEEE Standard* for software verification and validation: IEEE Std 1012–2004. – [Чинний від 2008–06–30] // IEEE Computer Society. – 19 p.).

33. *Forsberg K.* The Relationship of system engineering to the project cycle / K. Forsberg, H. Mooz // CSM. – 1996. – P. 1–12.

34. *System* and software engineering – System and software quality requirements and evaluation (SQuaRE) – System and software quality models: ISO / IEC 2510:2011. – [Publication 2011–03–11]. – ISO / IEC JTC. – 34 p. – [Електронний ресурс]. – Режим доступу: https://www.iso.org/obp/ui/#iso:std:iso-iec:25010:ed-1:v1:en

35. *Ergonomics* of human system interaction – Part 210: Human-centred design for interactive systems : *ISO* 9241-210: 2010. – [Publication 2010–03–15]. – ISO / TC 159/SC4. – 32 p. – [Електронний ресурс]. – Режим доступу до ресурсу: http://www.iso.org/iso/catalogue_detail.htm?csnumber=52075.

36. *Information* technology – Open systems interconnection – Conformance testing methodology and framework. – Part 1: General concepts: ISO 9646: 1994. – [Publication 1994–12–22]. – ISO / IEC JTC 1. – 46 p. – [Електронний ресурс]. – Режим доступу: https://www.iso.org/ obp/ui/#iso:std:iso-iec:9646:-1:ed-2:v1:en.

37. *Анализ* видов, последствий и причин потенциальных несоответствий [Електронний ресурс]. – ООО «Новое качество». – Режим доступу: *http://www.new-quality.ru/lib/FMEA_new-quality.pdf*

38. *Шевченко Д. Н.* Анализ динамического дерева отказов / Д. Н. Шевченко // Електромагнітна сумісність та безпека на залізничному транспорті. – 2011. – №2. – С. 142–148.

39. *Криспин Л.* Гибкое тестирование: практическое руководство для тестировщиков ПО и гибких команд / Л. Криспин, Д. Грегори. – М. : Вильямс, 2010. – 464 с.

40. *Бейзер Б.* Тестирование черного ящика. Технологии функционального тестирования программного обеспечения и систем / Б. Бейзер. – СПб. : Питер, 2004. – 318 с.

41. *Model-Based Testing*. – Microsoft [Електронний ресурс]. – Режим доступу: *https://msdn.microsoft.com/en-us/library/ee*620469.*aspx*.

42. *Model-Based Testing*. – *Wikipedia* [Електронний ресурс]. – Режим доступу: *https://en.wikipedia.org/wiki/Model-based_testing*.

43. *Рогов С.* Тестирование производительности *Web*-серверов [Електронний ресурс] / С. Рогов, Д. Намиот // Открытые системы. СУБД. – 2002. – № 12. – Режим доступу: *http://www.osp.ru/os/* 2002/12/182266/.

44. *Под предельной* нагрузкой : обзор программ нагрузочного тестирования веб-серверов [Електронний ресурс] // Журнал «Хакер». – 2008. – Апр. 21. – Режим доступу: *https://xakep.ru/* 2008/04/21/43327/.

45. *Bourdonov I.* UniTesK Test Suite Architecture / I. Bourdonov, A. Kossatchev, V. Kuliamin, and A. Petrenko // Proc. of FME 2002. LNCS 2391, pp. 77-88, Springer-Verlag, 2002.

46. *Systems* and software engineering – Sistems and software quality requirements and evaluation (SQuaRE) – Measurements of system and software product quality: ISO / IEC 25023:2016. – [Publication 2016–06–15]. – 45 p. – [Електронний ресурс]. – Режим доступу: https://www.iso.org/obp/ui/#iso:std:iso-iec:25023:ed-1:v1:en.

47. *Systems* and software engineering – Sistems and software quality requirements and evaluation (SQuaRE) – Measurements of quality in use: ISO / IEC 25022:2016. – [Publication 2016–06–15]. – 41 p. – [Електронний ресурс]. – Режим доступу: https://www.iso.org/obp/ui/#iso:std:iso-iec:25022:ed-1:v1:en.

48. *IEEE Standard* for software and system test documentation: IEEE Std 829™ – 2008. – [Publication 2008–07–18]. – ISO / IEC JTC. – 34 p. – [Електронний ресурс]. – Режим доступу: http://dis.unal.edu.co/~icasta/ggs/Documentos/Normas/829-2008.pdf.

49. *Bieman J. M*. Cohesion and reuse in an object-oriented system / B-K. Kang, J. M. Bieman // Proc. ACM Symposium on Software Reusability (SSR'95), April 1995. – P. 259–262.

50. *Chidamber S. R.* A metrics suite for object oriented design / S. R. Chidamber, C. F. Kemerer // IEEE Transactions on Software Engineering. – 1994. – Vol. 20. – № 6. – P. 476–493.

51. *Lorenz M*. Object-oriented software metrics: a practical guide / M. Lorenz, J. Kidd. – Prentice Hall, 1994. – 146 p.

52. *Abreu F. B*. The Design of eiffel programs: quantitative evaluation using the MOOD metrics / F. B. Abreu, R. Esteves, M. Goulao // Proceedings of the TOOLS'96. Santa Barbara, California. – July 1996. – 20 p.

53. *Binder R. V.* Design for Testability in Object-Oriented Systems / R. V. Binder // Communications of the ACM. – 1994. – Vol. 37. – № 9. – P. 87–101.

54. *Дишлевий О. П.* Засіб моніторингу та аналіз метрик дефектів якості програмного коду / О. П. Дишлевий, Ю. В. Драпушко // Інженерія ПЗ. – 2013. – Т.14. – № 2. – С. 49–54.

Навчальне видання

КУЧЕРОВ Дмитро Павлович
АРТАМОНОВ Євген Борисович

ІНЖЕНЕРІЯ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

Навчальний посібник

Редактор *Р. М. Шульженко*
Технічний редактор *А. І. Лавринович*
Коректор *О. О. Крусь*
Комп'ютерна верстка *Н. В. Чорної*

Підп. до друку Формат 60х84/16. Папір офс.
Офс. друк. Ум. друк. арк. 22,55. Обл.-вид. арк. 24,25.
Тираж 100 прим. Замовлення №

Видавець і виготівник
Національний авіаційний університет
03680. Київ-58, проспект Космонавта Комарова, 1.

Свідоцтво про внесення до Державного реєстру ДК № 977 від 05.07.2002