МІНІСТЕРСТВО ОСВІТИ И НАУКИ УКРАЇНИ
Національний аерокосмічний університет ім. М. Є. Жуковського
«Харківський авіаційний інститут»

ТЕХНОЛОГІЯ СТВОРЕННЯ ПРОГРАМНИХ ПРОДУКТІВ

Навчальний посібник

Харків «ХАІ» 2015

УДК 004.9:658.012.34 (075.8)
ББК 32.973я73
Т38

Авторський колектив:
О. К. Погудіна, Д. М. Крицький, І. М. Бабак, Є. А. Дружинін, В. М. Овсяннік

Рецензенти: д-р техн. наук, проф. Г. А. Кучук,
д-р техн. наук, проф. І. В. Рубан

Технологія створення програмних продуктів [Текст] : навч. посіб. / О. К. Погудіна, Д. М. Крицький, І. М. Бабак та ін. – Х. : Нац. аерокосм. ун-т ім. М. Є. Жуковського «Харк. авіац. ін-т», 2015. – 160 с.

Т38

ISBN 978-966-662-437-9

Розглянуто принципи й методи технології розроблення програмних продуктів, їх опис та оптимізацію; принципи кодування даних і створення алгоритмів на одній із сучасних мов програмування (с++); технології та інструментальні засоби використання об'єктно-орієнтованої моделі програмування; методи тестування й супроводження програмного забезпечення.

Для студентів напрямку «Комп'ютерні науки» при самостійній підготовці до лекцій, повторенні й вивченні основного матеріалу курсу «Технологія створення програмних продуктів».

Іл. 59. Табл. 14. Бібліогр.: 22 назви

УДК 004.9:658.012.34 (075.8)
ББК 32.973я73

ISBN 978-966-662-437-9

ЗМІСТ

1. ЖИТТЄВИЙ ЦИКЛ І СТАНДАРТИ ПРОГРАМНОЇ ІНЖЕНЕРІЇ

1.1. Поняття програмного забезпечення і проблеми розроблення складного ПЗ

Програмне забезпечення (Software) – набір комп'ютерних програм, процедур і пов'язаної з ними документації і даних (ISO/IEC 12207).

Таким чином, програмне забезпечення – це не просто програма. Це ще й документація і посібник користувача. Замість словосполучення «програмне забезпечення» часто використовують інше – «програмний продукт». Одна з головних властивостей програмного продукту – продаваність.

Державною службою статистики України встановлено, що у сфері інформатизації працює 3119 підприємств, основним видом діяльності яких є надання послуг у цій сфері. З них, за оцінками профільних асоціацій, майже 2000 компаній спеціалізуються на розробленні програмної продукції.

Для того щоб збільшити обсяги ринку, необхідно збільшити показник успішності проектів, а отже, запровадити нові технології і залучити грамотних фахівців, здатних ці технології застосувати.

Технологія програмування – сукупність методів, способів і засобів для зменшення вартості й підвищення якості розроблення програмних систем.

У будь-якій компанії, що розробляє програмне забезпечення, на кожному етапі процесу розробленя застосовується велика кількість різних технологій. Над створенням програмного продукту працюють аналітики, керівники (менеджери), тестувальники, кодувальники (програмісти), технічні письменники, системні адміністратори, фахівці з повторного використання, дизайнери, фахівці з ергономіки й ін.

Розглянемо технології, які стосуються загальних питань аналізу, проектування і розроблення: структурне, модульне, об'єктно-орієнтоване і компонентне програмування.

Структурне програмування. Виникнення концепції структурного програмування [7, 9] пов'язують з ім'ям відомого голландського вченого Е. Дейкстри – у 60-х роках минулого століття він сформулював основні її положення.

Принцип, на якому ґрунтується технологія структурного програмування, фундаментальна наукова й технічна ідея про виокремлення багатьох базисних елементів, з допомогою яких можна виразити (з яких можна зібрати) будь-який об'єкт із деякого широкого набору.

Отже, основний принцип технології структурного програмування формулюється так: для будь-якої простої програми можна побудувати функ-

ціонально еквівалентну їй структурну програму, тобто програму, сформовану на основі фіксованої базисної множини, що містить структуру послідовної дії, структуру вибору однієї із двох дій і структуру циклу, тобто багаторазового повторення деякої дії з перевіркою умови припиненя повторення.

На рис. 1.1 зображено алгоритмічні конструкції у вигляді блок-схем. Тут прямокутник означає узагальнену дію, ромб – перевірку умови, стрілки – перехід від однієї дії до іншої.

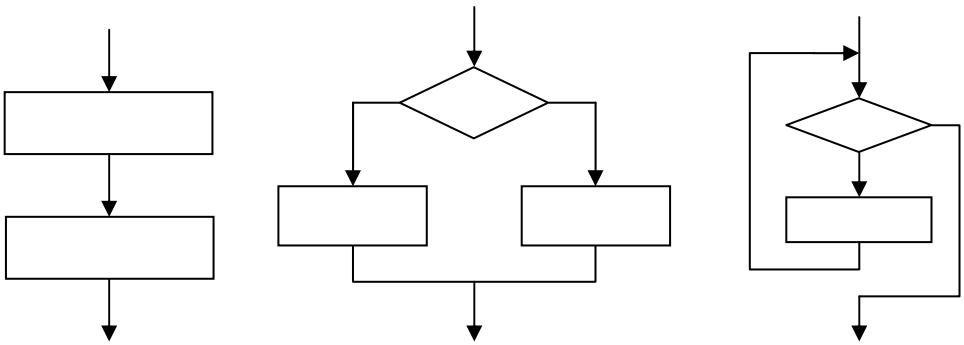


Рис. 1.1. Блок-схеми базисних алгоритмічних конструкцій

Під простою програмою в цьому випадку розуміють програму, що має тільки один вхід і один вихід по керуванню, і через всі її функціональні блоки проходить шлях від входу до виходу. Викладений принцип являє собою теорему про структурування. Базисні алгоритмічні конструкції абсолютно відповідають означенню простої програми, тобто мають один вхід і один вихід, що забезпечує можливість здійснювати їх суперпозицію. Кожну із трьох структур можна підставити в інші або в саму себе.

При становлені концепції структурного програмування відбувалися важливі процеси, пов'язані з поділом глобальних і локальних даних, для чого було використано підпрограми (рис. 1.2).

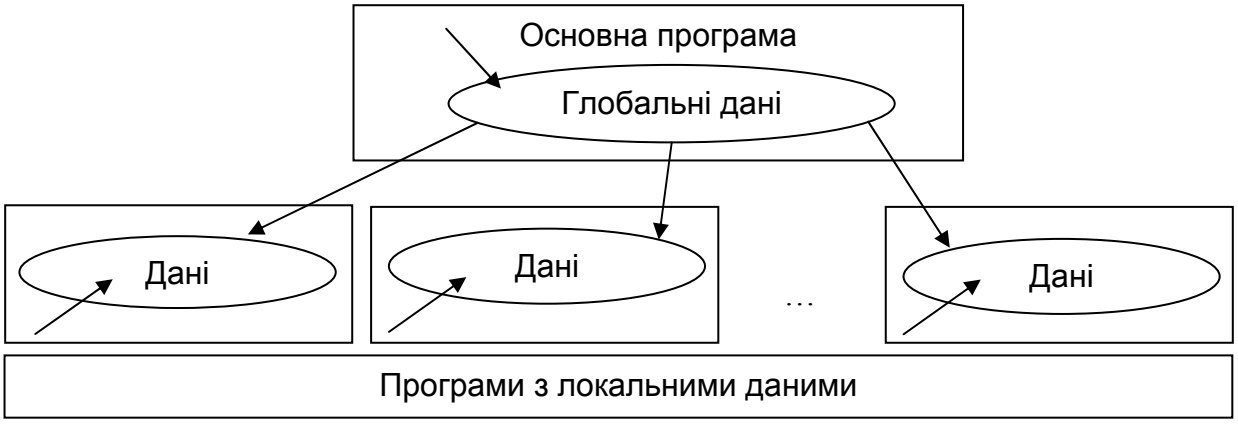


Рис. 1.2. Архітектура програми з підпрограмами й локальними даними

Наприкінці 1960-х років було переборено «кризу програмування», що полягала у збільшенні вартості й незавершеності проектів розроблення.

На початку етапу розвитку концепції структурного програмування стихійно використовувалося розроблення «знизу вверх» – підхід, при якому спочатку проектували й реалізовували порівняно прості підпрограми, з яких потім намагалися побудувати складну програму. За відсутності чітких моделей опису підпрограм і методів їх проектування створення кожної підпрограми перетворювалося на непросте завдання, інтерфейси підпрограм були складними, і при складанні програмного продукту виявлялася велика кількість помилок узгодження. Виправлення таких помилок зазвичай потребувало серйозного змінення вже розроблених підпрограм. Як наслідок, було поширено технологію проектування «зверху вниз», що припускає реалізацію загальної ідеї, забезпечуючи пророблення інтерфейсів підпрограм. Одночасно було введено обмеження на конструкції алгоритмів, рекомендовано формальні моделі їх опису, а також метод проектування алгоритмів – метод покрокової деталізації.

Таким чином, центральний технологічний принцип структурного програмування полягає в тому, що формулювання алґоритму і його запис у вигляді програми рекомендується виконувати на основі базису із трьох алґоритмічних конструкцій, застосовуючи за необхідності їхню суперпозицію. Результатом послідовного застосування цього принципу буде більш виразна структура програми, що, безсумнівно, полегшить пошук помилок і спростить її модифікацію.

Модульне програмування. Технологія модульного програмування, що сформувалася на початку 70-х років XX ст., містить ідею розроблення великих програмних систем [8]. Це фундаментальна концепція, яка є основою всіх сучасних підходів до проектування й реалізації. Водночас суть її є простою. Ця технологія є відображенням широко відомих наукових і технічних методів, що полягають у пошуку і реалізації деякого базового набору елементів, комбінація яких буде рішенням усіх завдань певного кола.

Якщо структурне програмування полягає в деякому універсальному алгоритмічному базисі, то модульне програмування – у розробленні під конкретне завдання або коло завдань (предметну область) власного базису у вигляді набору модулів, що дає змогу за багатьма критеріями найбільш ефективно побудувати програмний комплекс. Модулі, що належать до базису, це великі програми (на відміну від примітивів структурного програмування), які вирішують деякі підзадачи основних завдань (рис. 1.3).

Для того щоб забезпечити максимальну незалежність модулів, треба чітко відокремити процедури, які будуть викликатися іншими модулями (відкриті процедури), від допоміжних, що обробляють дані, занесені в цей модуль (закриті процедури). Дані, занесені в модуль, також поділяються на відкриті й закриті.

Із застосуванням модульного програмування виникають можливості колективного розроблення програм як набору «незалежних» частин, послідовного зменшення складності методом розбиття складного завдання на

більш прості підзадачи, а також можливості повторного використання створеного раніше коду.

«Вузьким» місцем модульного програмування є те, що помилка в інтерфейсі під час виклику підпрограми виявляється тільки при виконанні програми (через роздільну компіляцію модулів виявити ці помилки раніше неможливо). При збільшенні розміру програми зазвичай зростає складність міжмодульних інтерфейсів, і з деякого моменту передбачити взаємовплив окремих частин програми стає майже неможливо.

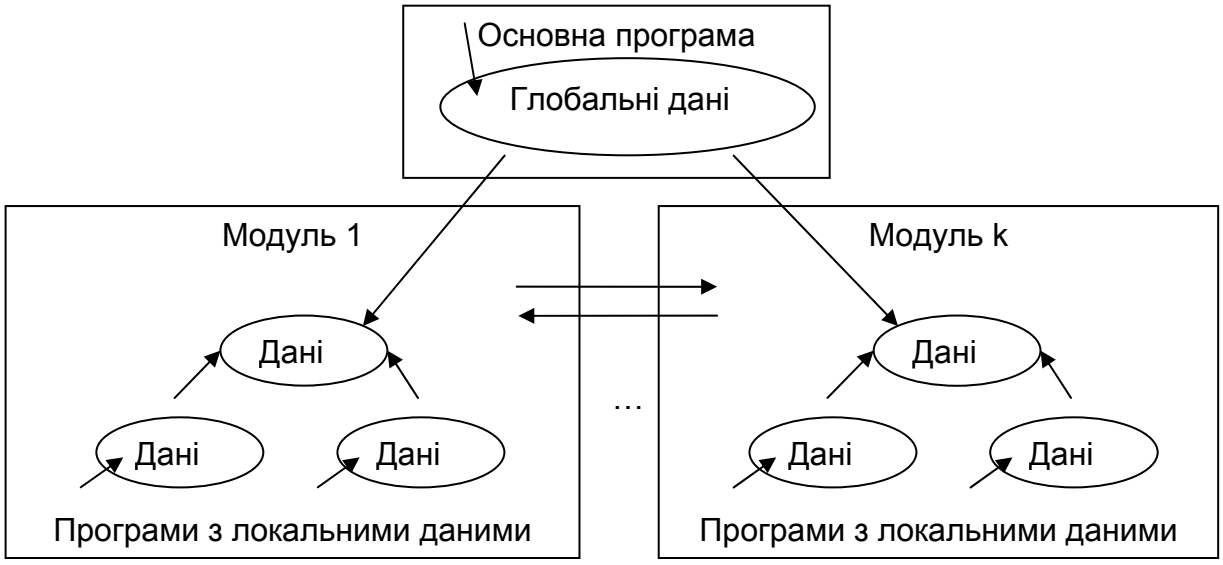


Рис. 1.3. Архітектура програми з використанням модулів

Об’єктно-орієнтоване програмування. Об’єктно-орієнтована технологія деякою мірою вирішила більшість проблем модульного програмування. На відміну від розглянутих раніше технологій об’єктно-орієнтована технологія застосовується на стадіях аналізу, проектування й програмування. Основою технології є об'єктна модель і об'єктна декомпозиція [3].

До основних принципів об'єктної моделі часто зараховують такі: абстракція; інкапсуляція; спадкування; поліморфізм; модульність.

Суть об'єктної декомпозиції полягає у виокремленні в предметній області класів і об'єктів, а також зв'язків між ними, і лише потім даних і алгоритмів, якими характеризується кожен клас. Таким чином, саме класи стають основним «будівельним блоком» в ООП, тоді як раніше такими блоками були функції (рис. 1.4).

Бурхливий розвиток технологій програмування, основаних на об'єктному підході, дало змогу вирішити багато проблем. Так, було створено середовища, що підтримують візуальне програмування, наприклад Delphi, C++ Builder, Visual C++ і т. д. При використанні візуального середовища програміст може проектувати деяку частину, наприклад інтерфейси майбутнього продукту, із застосуванням візуальних засобів додавання і настроювання спеціальних бібліотечних компонентів. Результатом візуаль-

ного проектування є заготовка (каркас) майбутньої програми, до якої вже внесено відповідні коди.

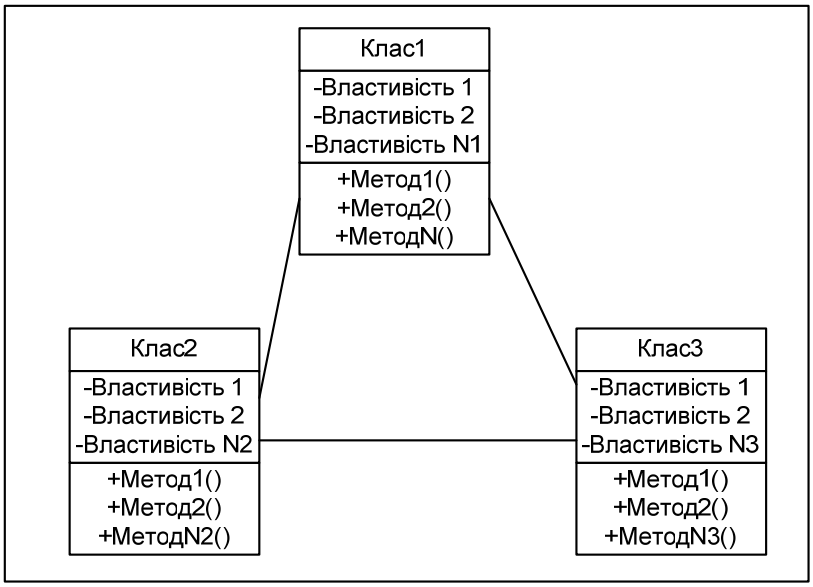


Рис. 1.4. Архітектура програми ООП

Компонентне програмування є вдосконаленням об'єктно-орієнтованної технології. На відміну від ООП в компонентному підході введено наступний рівень абстракції, де класи поєднуються в компоненти.

При компонентному підході CBSE (Component-Based Software Engineering) припускається побудова ПЗ з окремих компонентів – фізично окремо існуючих частин ПЗ, які взаємодіють між собою через стандартизовані двійкові інтерфейси.

На відміну від звичайних об'єкти-компоненти можна зібрати в динамічно приєднувані бібліотеки або виконувані файли, поширювати у двійковому вигляді (без початкових текстів) і використовувати їх у будь-якій мові програмування, що підтримує відповідну технологію (рис. 1.5).

Компонент є ізольованим від зовнішнього світу своїм інтерфейсом – набором методів (їхніми сигнатурами). Компонентна програма – набір незалежних компонентів, зв'язаних один з одним з допомогою інтерфейсів.

Компонентний підхід є основою технологій, розроблених на базі COM і CORBA.

З допомогою технології COM визначають загальний принцип взаємодії програм будь-яких типів: бібліотек, програм, операційної системи, тобто дає змогу одній частині ПЗ використати функції (служби), які надано іншій, незалежно від того, чи функціонують ці частини в межах одного процесу, у різних процесах на одному комп'ютері або на різних комп'ютерах. Модифікація COM, що забезпечує передачу викликів між комп'ютерами, має назву DCOM.

За технологією COM програма надає свої служби, використовуючи

об'єкти COM, які є екземплярами класів COM. Об'єкт COM може реалізовувати кілька інтерфейсів. На базі технології COM було розроблено компонентні технології, що вирішують різні завдання розроблення ПЗ.

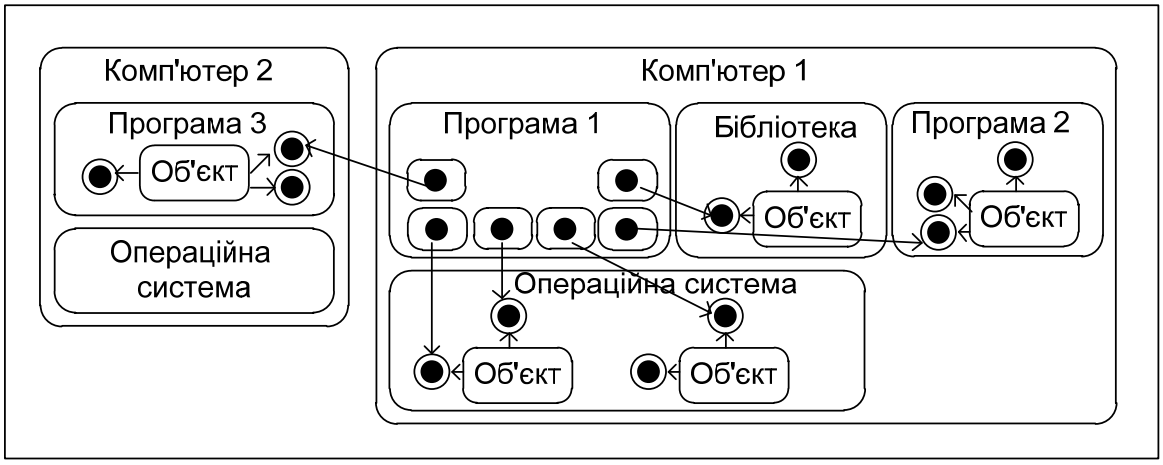


Рис. 1.5. Архітектура програми компонентного підходу

OLE-automation – технологія створення програм, що забезпечує доступ до їхніх внутрішніх служб. Її, наприклад, підтримує Microsoft Excel, надаючи іншим програмам свої служби.

ActiveX – технологія, яку побудовано на базі OLE-automation і призначено для створення як розподіленого в мережі, так і зосередженого на одному комп'ютері ПЗ. Допускається використання візуального програмування для створення компонентів – елементів керування ActiveX. Отримані таким чином елементи керування можна встановлювати на комп'ютер дистанційно з віддаленого сервера, причому встановлюваний код залежить від використовуваної операційної системи.

MTS (Microsoft Transaction Server – сервер керування транзакціями) – технологія, з допомогою якої забезпечується безпечна і стабільна робота розподілених програм при велких обсягах переданих даних.

MIDAS (Multilier Distributed Application Server – сервер багатоланкових розподілених програм) – технологія, з допомогою якої організовується доступ до даних різних комп'ютерів з урахуванням балансування навантаження мережі.

Зазначені технології реалізують компонентний підхід технології COM.

Технологія CORBA, яку було розроблено групою компаній OMG, реалізує підхід, аналогічний COM, на базі об'єктів і інтерфейсів CORBA. Програмне ядро CORBA реалізоване для всіх основних апаратних і програмних платформ, тому цю технологію можна використати для створення розподіленого ПЗ у різноманітному обчислювальному середовищі.

Взаємодія між об'єктами клієнта й сервера в CORBA організовується з допомогою спеціального посередника, що має назву VisiBroker, та іншого спеціалізованого ПЗ.

COTS (component off the shelf) – методологія, за якою розроблювачам пропонується використовувати сторонні компоненти.

Основна мета розробленя – це отримання ПЗ з потрібним набором функціональності. Звичайно, що відразу отримати готову програму неможливо. Тому доводиться робити все поступово, при цьому хочеться бути впевненим, що результат буде якісним, замовлення буде оплачено або успішно здано.

Наведемо найбільш розповсюджені проблеми, що виникають під час розроблення ПЗ.

Брак прозорості. У будь-який момент часу складно сказати, у якому стані перебуває проект і яким є відсоток його завершення. Ця проблема виникає при недостатньому плануванні структури (або архітектури) майбутнього ПЗ, тому необхідно врахувати, скільки часу забере розроблення, якими будуть етапи, чи можна якісь етапи виключити.

Недолік контролю. Без точного оцінювання процесу розроблення можна зірвати графіки виконання робіт і перевищити встановлені бюджети.

Недолік моніторингу. Неможливість спостерігати процес розвитку проекту не дає змоги контролювати процес розроблення в реальному часі.

Неконтрольовані зміни. У замовника постійно виникають нові ідеї щодо розроблюваного ПЗ. Вплив змін може бути суттєвим для успіху проекту, тому важливо оцінювати пропоновані зміни і реалізовувати тільки ті, що схвалив виконавець. Ця проблема виникає внаслідок того, що замовник на етапі проектування не продумав добре програму, її функціональне призначення (виконувані операції і т. д.).

Найскладніший етап – пошук і виправлення помилок у програмах. Цей етап має назву налагодження програми. Ця проблема виникає, якщо виконавець припустився помилок при розробленні програми, а замовник недбало проаналізував проміжні результати розроблюваної програми, які виконавець регулярно має надавати для перевірки.

Контрольні запитання

1. Що розуміють під терміном «технологія програмування»?
2. Що називають підходом? Чим підхід відрізняється від методу?
3. Назвіть основні періоди історії розвитку технології програмування. Чим характеризуються ці періоди?
4. Назвіть основні принципи об’єктної моделі.
5. У чому полягає різниця між технологіями OLE-automation і ActiveX?
6. Які особливості має технологія CORBA?
7. У чому полягає різниця між простими об’єктами і об’єктами-компонентами?
8. Які проблеми розроблення ПЗ Ви знаєте? Опишіть причину їх виникнення.

1.2. Життєвий цикл і процеси розроблення ПЗ

Модель життєвого циклу програмного забезпечення (Software Life Cycle Model, SLCM) – це концептуальна структура, що містить процеси, дії і завдання, які стосуються розроблення, експлуатації і супроводу програмного продукту, та охоплює життєвий цикл системи, починаючи з визначення вимог до неї і закінчуючи припиненням її використання.

У моделі SLCM визначено точні інструкції, які розробник може використовувати для створення високоякісних програмних систем.

Конкретні моделі життєвого циклу програмного забезпечення визначаються особливістю завдань, обмеженнями ресурсів, досвідом розробників і т. д.

Модель життєвого циклу ПЗ складається з набору фаз (етапів, стадій) проекту зі створення ПЗ, у яких виконуються окремі процеси, розбиті на дії і завдання.

Фаза проекту – об'єднання логічно зв'язаних процесів, що зазвичай завершуються досягненням одного з основних результатів.

Фазу можна розбити на етапи. На відміну від процесу фаза проекту завершується отриманням одного з основних результатів, а процес – просто значущого результату. Слід зазначити, що склад, кількість і порядок виконання фаз визначається особливістю проекту

Відомими моделями життєвого циклу є такі [4]:

- каскадна;
- V-подібна;
- прототипування;
- RAD;
- інкрементна;
- спіральна.

Вибір конкретної моделі ЖЦ ПЗ відбувається за схемою, яку зображено на рис. 1.6.

1.2.1. Каскадна модель

Каскадну модель вперше чітко сформулював американський вчений У.В. Ройс 1970 р. Ця модель відображає послідовність і зміст фаз розроблення проекту. Це була перша модель, у якій формалізувалася структура етапів розроблення ПЗ і ставилися вимоги до програмного продукту, процесів проектування і документування на всіх етапах розроблення проекту. У 70–80-х роках минулого століття цю модель було прийнято за стандарт Міністерства оборони США.

Каскадна модель (від англ. waterfall – водоспад) відповідно до стандарту IEEE 1074 містить такі фази (рис. 1.7):

– *дослідження концепції* – дослідження вимог, розроблення продукту й оцінювання можливості його реалізації;

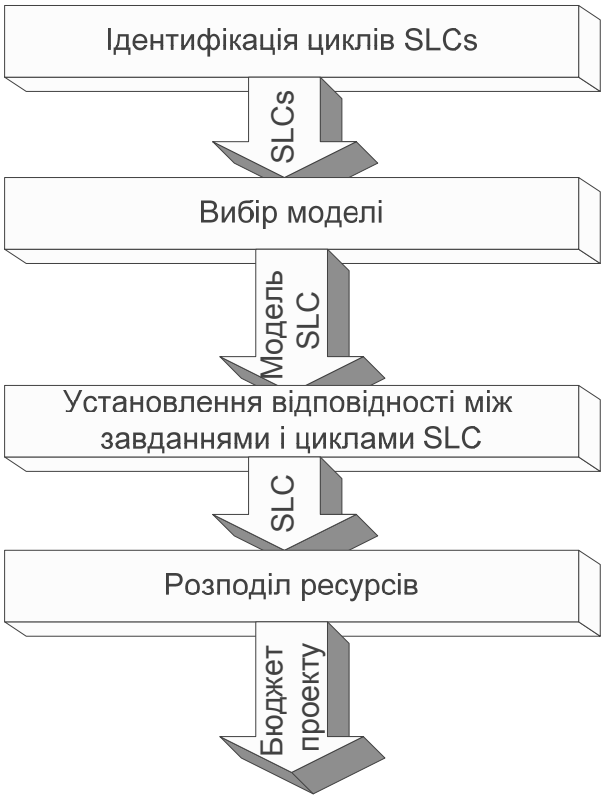


Рис. 1.6. Схема вибору моделі життєвого циклу ПЗ

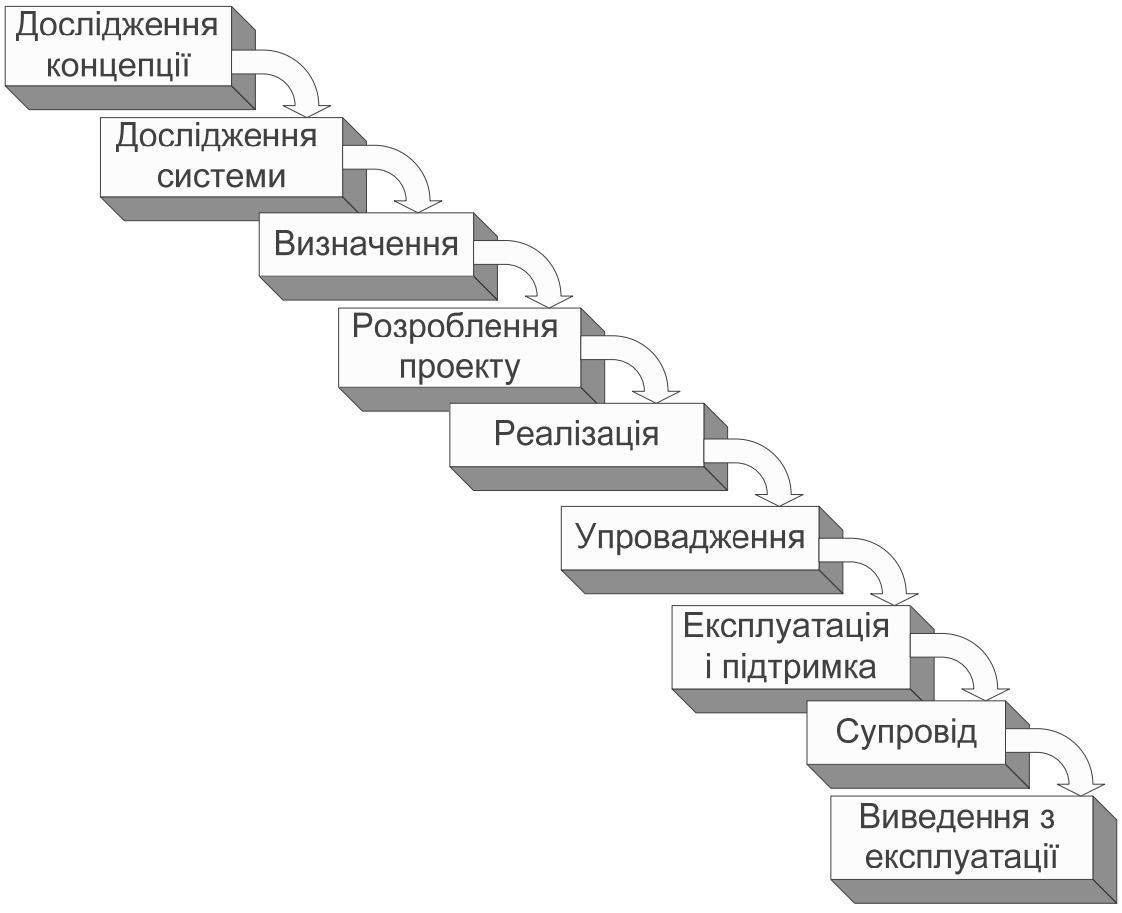


Рис. 1.7. Каскадна модель життєвого циклу ПЗ

– *дослідження системи,* що виконується для систем, у яких необхідно розробити не тільки програмне, але й апаратне забезпечення; функції, що розробляються, розподіляються для ПЗ і устаткування відповідно до архітектури системи;

– *визначення вимог* – специфікація програмних вимог для предметної області системи, визначення цілей, функцій, інтерфейсів і продуктивності продукту;

– *розроблення проекту* – логічне розроблення технічних характеристик програмної системи, а також структури даних, *архітектури програмного забезпечення*, інтерфейсного подання і процесної (алгоритмічної) деталізації;

– *реалізація***,** під час якої логічний опис ПЗ, виконаний у попередній фазі, перетворюється на повноцінний програмний продукт, унаслідок чого отримують початковий код, базу даних і документацію; реалізація зазвичай складається з двох етапів: реалізації компонент ПЗ та інтеграції компонент у готовий продукт, на обох етапах виконується кодування і тестування, які інколи розглядають як два підетапи;

– *установлення* ПЗ, його перевірка і офіційне прийняття замовником;

– *експлуатація і підтримка* – запуск і поточне забезпечення програми, у тому числі надання технічної допомоги, обговорення питань, що постають, з користувачем, реєстрація запитів користувача на модернізацію і внесення змін, а також коригування або усунення помилок;

– *супровід* – усунення програмних помилок, несправностей, збоїв, модернізація і внесення змін;

– *виведення з експлуатації* – виведення системи з активного використання шляхом припинення роботи або її заміни новою, або модернізованою системою.

При проектуванні на основі каскадної моделі необхідно дотримуватися таких **принципів**:

– послідовне виконання фаз, тобто кожна наступна фаза починається, коли цілком завершено виконання попередньої фази;

– кожна фаза повинна мати певні вхідні й вихідні дані;

– кожна фаза має повністю документуватися;

– перехід від однієї фази до іншої має здійснюватися з допомогою формального огляду за участі замовника;

– основу моделі мають складати вимоги, сформульовані в документі SRS «Специфікація вимог до програмного забезпечення» (або «Технічне завдання»), які не повинні змінюватися;

– критерій якості проекту – відповідність створеного продукту вимогам, установленим у SRS або ТЗ.

Каскадна модель має такі **переваги:**

– є простою і зрозумілою для замовників, оскільки її часто використовують організації для відстеження проектів, не пов'язаних із розробленням ПЗ;

– є простою і зручною в застосуванні, тому що процес розроблення виконується поетапно;

– її структурою може користуватися навіть слабко підготовлений або недосвідчений персонал;

– з її допомогою можна чітко контролювати керування проектом;

– кожну стадію можуть виконувати незалежні команди, тому що процес розроблення обов'язково документується;

– полегшує планування термінів і витрат на розроблення.

Для каскадної моделі життєвого циклу характерною є автоматизація окремих незв'язаних завдань, які не потребують виконання інформаційної інтеграції і сумісності, програмного, технічного й організаційного поєднання. Каскадна модель життєвого циклу за термінами розроблення й надійністю виправдала себе для вирішення окремих завдань.

Застосування каскадної моделі життєвого циклу до великих і складних проектів унаслідок великої тривалості процесу проектування і змінюваності вимог за цей час призводить до неможливості їх практичної реалізації.

Каскадна модель має такі основні **недоліки:**

– значне збільшення витрат і збій в графіку при спробі повернутися на одну або дві фази назад, щоб виправити яку-небудь помилку або недолік;

– збільшення вартості усунення помилок через інтеграцію компонент, під час якої зазвичай виявляється велика частина помилок, що виникають наприкінці розроблення;

– запізнення з отриманням результатів через змінення вимог під час виконання проекту.

Каскадна модель з можливістю повернення на попередній крок, щоб переглянути результати, стає **ітераційною** (рис. 1.8).

Через недоліки каскадна модель сьогодні мало застосовується, тільки у випадках, коли вимоги і їх реалізація максимально чітко визначено і зрозуміло, наприклад завдання таких типів:

– науково-обчислювальні (пакети й бібліотеки наукових програм розрахункового характеру в будівництві, автобудуванні й т. д.);

– проектування операційних систем і компіляторів;

– проектування систем керування конкретними об'єктами в реальному часі.

Слід особливо зазначити, що принципи каскадної моделі є основою для елементів моделей інших типів.

Рис. 1.8. Каскадна модель життєвого циклу ПЗ зі зворотними зв'язками

1.2.2. V-подібна модель

V-подібну модель було розроблено як різновид каскадної моделі. У ній більше уваги приділяється плануванню проекту з можливістю паралельного тестування системи. Особливе значення надається верифікації та атестації продукту, тому що тестування продукту обговорюється, проектується і планується на ранніх етапах життєвого циклу розроблення. План випробування (приймання) замовником розробляється на етапі планування, а компонувального випробування системи – на фазах аналізу, розроблення проекту і т. д. Цей процес позначено пунктирною лінією між прямокутниками V-подібної моделі (рис. 1.9).

V-подібна модель має таку ж послідовну структуру, як і каскадна: кожна наступна фаза починається після завершення попередньої фази й отримання результату. Модель є комплексним підходом до визначення фаз процесу розроблення ПЗ. Між аналітичними фазами й фазами проектування існують взаємозв'язки, які передують кодуванню й тестуванню. Пунктирні лінії означають, що фази необхідно розглядати паралельно.

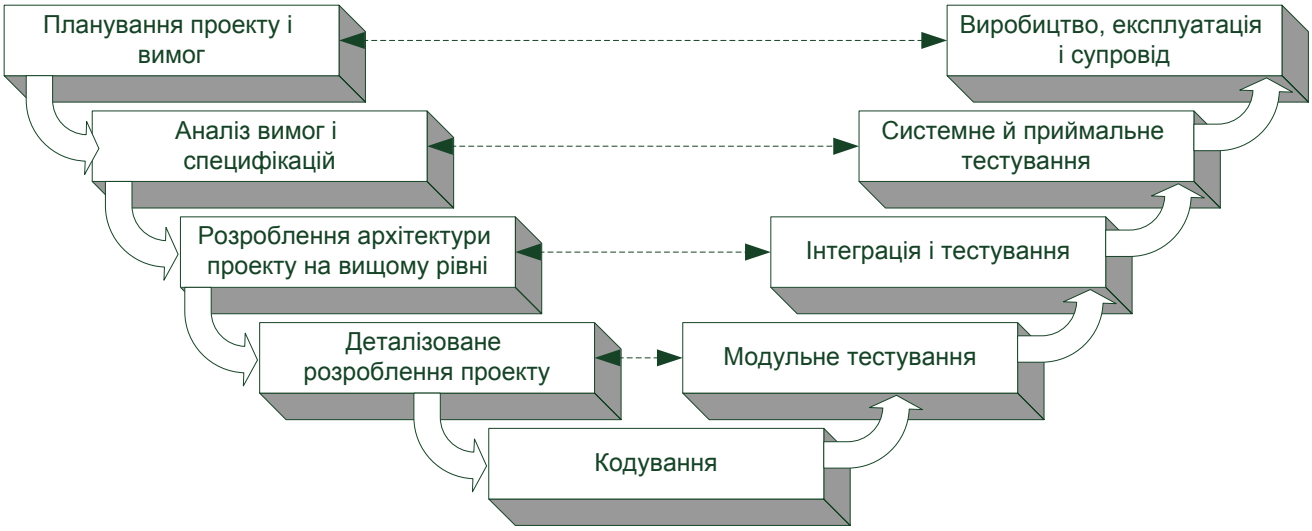


Рис. 1.9. V-подібна модель життєвого циклу

У V-подібній моделі передбачено виконання таких фаз:

– *планування проекту і вимог* – визначення системних вимог, розподіл ресурсів організації з метою їх відповідності поставленим вимогам;

– *аналіз вимог до продукту і його специфікація*;

– *високорівневе розроблення архітектури* – визначення функцій ПЗ, які буде реалізовано в проекті;

– *деталізоване розроблення проекту* – знаходження алгоритмів для кожного компонента, який було визначено на фазі будування архітектури (ці алгоритми надалі буде перетворено на код);

– *розроблення програмного коду* – перетворення алгоритмів, визначених на етапі деталізованого проектування, на готове ПЗ;

– *модульне тестування* – перевірка кожного закодованого модуля на наявність помилок;

– *інтеграція і тестування* – перевірка на наявність помилок в інтегрованій групі модулів;

– *системне й приймальне тестування* – перевірка функціонування програмної системи в цілому (повністю інтегрованої системи) після розміщення її в апаратному середовищі згідно зі специфікацією вимог до ПЗ;

– *виробництво, приймальні випробування, експлуатація і супровід*.

Переваги V-подібної моделі:

– забезпечує високу якість результату: у моделі особливе значення надається плануванню, спрямованому на верифікацію й атестацію розроблюваного продукту, на ранніх стадіях його розроблення; фаза модульного тестування підтверджує правильність деталізованого проектування; на фазах інтеграції і тестування реалізується архітектурне проектування або проектування на вищому рівні; на фазі тестування системи підтверджується правильність виконання етапу вимог до продукту і його специфікації; у

моделі передбачено атестацію і верифікацію всіх зовнішніх і внутрішніх отриманих даних, а не лише самого програмного продукту;

– має чітку послідовність одержання проміжних результатів (визначення вимог перед розробленням проекту системи, проектування ПЗ перед розробленням компонентів);

– дає змогу отримати обґрунтований результат: визначення продуктів (у тому числі супровідних), які має бути отримано й перевірено під час розроблення;

– можна відстежити процес розроблення, оскільки використовується часова шкала, а завершення кожної фази є контрольною точкою;

– є простою у використанні.

Недоліки V-подібної моделі:

– труднощі під час роботи з паралельними подіями;

– не враховано ітерації між фазами;

– не передбачено внесення динамічних змін на різних етапах життєвого циклу;

– тестування вимог в життєвому циклі відбувається надто пізно, унаслідок чого неможливо внести зміни без змінення графіка виконання проекту;

– не містить дій, спрямованих на аналіз ризиків.

V-подібну модель, як і каскадну, краще за все використовувати тоді, коли вся інформація про вимоги є доступною. Використання моделі є ефективним, коли доступними є інформація про метод реалізації ПЗ і технологія, а персонал має необхідні навички й досвід роботи з цією технологією. V-подібну модель зазвичай застосовують, коли потрібна висока надійність систем, пов’язана з ризиком для життя.

1.2.3. Модель швидкого прототипування

Фред Брукс (Fred Brooks) 1975 р. у своїй книзі «Легендарна людина-місяць» («The Mythical Man-Month») написав: «У більшості проектів перша побудована система навряд чи придатна до використання. Вона може бути занадто повільною, надто об’ємною, незручною у використанні або мати всі три перелічені недоліки. Але немає іншого вибору, окрім як почати із самого початку, доклавши всіх зусиль, і побудувати модернізовану версію, у якій розв’язувалися б усі три проблеми...». Саме ця концепція будування експериментальної, або прототипної, системи привела до виникнення «структурної», «еволюційної» моделі швидкого прототипування (RAD) і спіральної моделі.

Прототипування – це процес будування робочої моделі системи. Прототип – еквівалент експериментальної моделі, або «макету», у середовищі апаратного забезпечення. Еволюційні програми виконуються в ме-

жах контексту плану, спрямованого на досягнення гранично високої продуктивності. За цим методом передбачається також, що розроблення інкрементів програми є очевидним для користувача, який бере участь у всьому процесі розроблення.

Модель швидкого прототипування (рис. 1.10) призначено для швидкого створення прототипів продукту з метою уточнення вимог і поетапного впровадження прототипів у кінцевий продукт. Швидкість (висока продуктивність) виконання проекту забезпечується плануванням розроблення прототипів і участю замовника в процесі розроблення.

Рис. 1.10. Модель швидкого прототипування

Перший рівень. Початок життєвого циклу розроблення знаходиться в центрі еліпса. Спільно з користувачем розробляється план проекту на основі попередніх вимог. Результат початкового планування – документ, де описано в загальних рисах приблизні графіки й результівні дані.

Другий рівень – створення на основі швидкого аналізу прототипу бази даних призначеного для користувача інтерфейсу.

Третій рівень. Починається ітераційний цикл швидкого прототипування. Розробник проекту демонструє черговий прототип, користувач оцінює його функціонування і вони спільно визначають проблеми і шляхи їх подолання для переходу до наступного прототипу. Цей процес триває доти, доки користувач не погодиться, що в черговому прототипі точно відображено всі вимоги.

Четвертий рівень. Отримавши схвалення користувача, швидкий

прототип перетворюють на детальний проект, а систему налаштовують на виробниче використання. Саме на цьому етапі налаштування прискорений прототип стає системою, що повністю діє.

Під час розроблення виробничої версії програми може знадобитися вищий рівень функціональних можливостей, різні системні ресурси, необхідні для забезпечення повного робочого навантаження, або обмеження в часі. Після цього прототип тестують на граничних режимах, визначають вимірювальні критерії і налаштування, а потім, як завжди, функціональний супровід.

Переваги моделі швидкого прототипування:

– кінцевий користувач може спостерігати реалізацію системних вимог під час їх упровадження командою розробників; таким чином, взаємодія замовника із системою починається на перших етапах розроблення; виходячи з реакції замовників на демонстрації продукту, розробники отримують відомості про один або декілька аспектів поводження системи, завдяки чому зводиться до мінімуму кількість неточностей у вимогах;

– у процес розроблення можна внести нові або неочікувані вимоги користувача; можна виконати гнучке проектування й розроблення, у тому числі декілька ітерацій на всіх фазах життєвого циклу;

– при використанні моделі спостерігаються видимі ознаки прогресу у виконанні проекту, завдяки чому замовники почуваються впевнено;

– можливість виникнення розбіжностей під час спілкування замовників з розробниками є мінімальною;

– очікувана якість продукту визначається за активної участі користувача на перших етапах розроблення;

– можливість спостерігати ту чи іншу функцію в дії сприяє розробленню додаткових функціональних можливостей;

– завдяки меншому обсягу доробок зменшуються витрати на розроблення;

– завдяки тому, що проблема виявляється до залучення додаткових ресурсів, зменшуються загальні витрати;

– забезпечується керування ризиками;

– документацію сконцентровано на кінцевому продукті, а не на його розробленні.

Недоліки моделі швидкого прототипування:

– швидко створені прототипи або мають багато зайвої документації, або її бракує;

– якщо цілі прототипування не узгоджено заздалегідь, то процес проектування може стати некерованим;

– під час створення робочого прототипу може бути недостатньо враховано якість усього ПЗ або довгострокову експлуатаційну надійність;

– іноді при використанні моделі отримують систему з низькою робо-

чою характеристикою, особливо якщо під час її створення передбачається етап підгонки;

– під час використання моделі вирішення складних проблем може відкладатися на майбутнє, що призведе до того, що наступні отримані продукти можуть не відповідати прототипу;

– якщо користувачі не можуть брати участі в проекті на ітераційній фазі швидкого прототипування життєвого циклу, то це відбивається на якості кінцевого продукту;

– на ітераційному етапі прототипування швидкий прототип є частковою системою; якщо виконання проекту завершується достроково, то у кінцевого користувача залишиться тільки часткова система;

– замовник може вважати за краще отримати прототип, ніж чекати повної, добре продуманої версії;

– і розробники, і користувачі не завжди розуміють, що коли прототип перетворюється на кінцевий продукт, необхідне створення традиційної документації; якщо її немає, то модифікувати модель на пізніших етапах може виявитися дорожче, ніж просто не скористатися створеним прототипом;

– на розроблення системи може бути витрачено надто багато часу, тому що ітераційний процес демонстрації прототипу і його переглядання можуть тривати нескінченно довго без належного керування процесом.

Структурну еволюційну модель швидкого прототипування застосовують у таких випадках:

– вимоги є невідомими заздалегідь;

– вимоги є непостійними або можуть неправильно тлумачитися чи бути невдало сформульованими;

– вимоги слід уточняти;

– потребується розроблення призначених для користувача інтерфейсів;

– потребується перевірка концепції;

– здійснюються тимчасові демонстрації;

– розробляється новий, який не має аналогів, продукт;

– потребується зменшення неточностей у визначенні вимог, тобто зменшення ризику створення системи, яка не має ніякої цінності для замовника;

– замовник неохоче погоджується на фіксований набір вимог або про прикладну програму немає чіткого уявлення і тоді вимоги можна швидко змінити;

– розробники не впевнені в тому, яку оптимальну архітектуру або алгоритм слід застосовувати;

– алгоритми або системні інтерфейси є ускладненими;

– потрібно продемонструвати технічну здійсненність в умовах висо-

кого технічного ризику.

Швидке прототипування найкраще застосовувати при розробленні інтенсивно використовуваних систем (інтерактивних), нових продуктів, а також систем забезпечення прийняття рішень, наприклад систем керування, медичної діагностики, тощо.

1.2.4. Модель швидкого розроблення ПЗ (RAD)

Метод швидкого розроблення додатків (Rapid Application Development, RAD) було використано у 80-х роках XX ст. фірмою IBM. Цей метод до уваги розробників ПЗ уперше запропонував Джеймс Мартін. Завдяки методу RAD користувач бере участь у всіх фазах життєвого циклу проекту не лише під час визначення вимог, але й під час проектування, розроблення, тестування, а також кінцевого постачання програмного продукту. Участь користувача стає такою активною завдяки використанню засобів розроблення й середовища, що дає змогу оцінити продукт на всіх стадіях його розроблення. Це забезпечується наявністю засобів розроблення призначених для користувача графічного інтерфейсу і кодогенераторів.

Характерним для RAD є короткий період від визначення вимог до створення цілісної системи. Метод ґрунтується на послідовній ітерації еволюційної системи або прототипів, критичний аналіз яких обговорюється із замовником. Під час такого аналізу формуються вимоги до продукту. Розроблення кожного інтегрованого продукту обмежується чітко окресленим періодом часу, який зазвичай становить 60 днів і має назву часового блоку.

Чинники, з допомогою яких можна створити систему за 60 днів без шкоди для якості:

– потужні інструментальні засоби розроблення;

– високий рівень повторного використання елементів системи;

– осмислені й виділені ресурси.

Кінцевий користувач безпосередньо бере участь в переміщенні процесу роботи від програмування й тестування до планування й проектування. Користувачам доводиться виконувати великий обсяг роботи на початку життєвого циклу, але завдяки цьому утворюється система, побудована за коротший інтервал часу.

Фази моделі ЖЦ RAD і участь користувача на всіх фазах процесу проектування (рис. 1.11):

– планування вимог – вимоги збирають під час використання методу спільного планування вимог (Joint requirements planning, JRP), який є структурним аналізом і обговоренням наявних комерційних завдань;

– складання опису для користувача – спільне проектування додатка

(Joint application design, JAD) використовується з метою залучення користувачів; на цій фазі проектування системи, що не є промисловою, команда, яка працює над проектом, часто використовує інструментальні засоби, що забезпечують збір призначеної для користувача інформації;

– конструювання – ця фаза об'єднує в собі деталізоване проектування, будування (кодування й тестування), а також постачання програмного продукту замовникові за певний час. Терміни виконання цієї фази значною мірою залежать від використання генераторів коду, екранних форм та інших типів інструментальних засобів;

– переведення на нову систему експлуатації – ця фаза містить проведення користувачами приймальних випробувань, установлення системи й навчання користувачів.

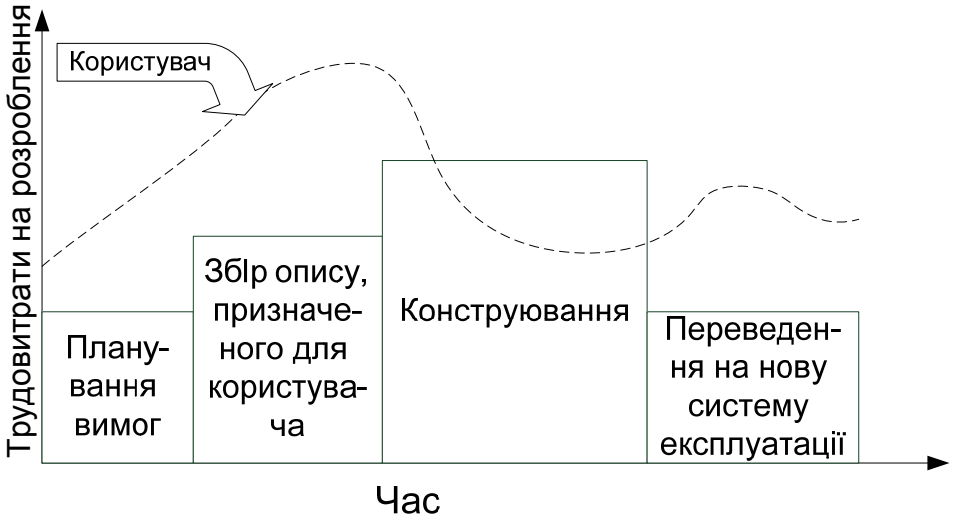


Рис. 1.11. Модель швидкого розроблення додатків

Переваги моделі RAD:

– наявність потужних інструментальних засобів, завдяки чому можна зменшити час циклу розроблення для всього проекту;

– потребується менша кількість фахівців, оскільки розроблення системи виконується зусиллями команди, інформованої в предметній області;

– існує можливість здійснити швидкий початковий перегляд продукту;

– зменшення витрат завдяки зменшенню часу циклу і вдосконаленій технології, а також меншій кількості задіяних у процесі розробників;

– мінімальний ризик незадоволення замовниками розробленим продуктом у разі його залучення на постійній основі;

– основним є код, а не документація, при цьому правильним є принцип «отримуєте те, що бачите» (What You See Is What You Get, WYSIWYG);

– у моделі можна повторно використовувати компоненти вже існую-

чих програм.

Недоліки моделі RAD:

– користувачі не можуть постійно брати участь у процесі розроблення впродовж всього життєвого циклу, це може негативно позначитися на кінцевому продукті;

– використання моделі може виявитися невдалим у випадку, якщо немає придатних для повторного використання компонентів;

– для реалізації моделі потрібні розробники й замовники, які готові до швидкого виконання дій, незважаючи на жорсткі часові обмеження.

Застосування моделі RAD:

– у системах, які підлягають моделюванню (базуються на використанні компонентних об'єктів), а також у масштабованих системах;

– у системах, вимоги для яких є добре відомими;

– у випадках, коли кінцевий користувач може брати участь у процесі розроблення впродовж усього життєвого циклу;

– коли команда, що працює над проектом, добре знає предметну область;

– у процесі виконання проектів, розробити які необхідно в скорочені терміни, зазвичай не більш ніж за два місяці;

– коли необхідно отримати придатні до повторного використання частини програмних продуктів;

– у системах, які призначено для концептуальної перевірки, є критичними або мають невеликий розмір;

– у системах, які не потребують досягнення високої продуктивності через невисокий ступінь технічних ризиків.

1.2.5. Інкрементна (покрокова) модель

Інкрементне розроблення є процесом поетапної реалізації всієї системи й поетапного збільшення функціональних можливостей.

Цей підхід дає змогу зменшити витрати, завдані до моменту досягнення рівня вихідної продуктивності. З допомогою цієї моделі прискорюється процес створення функціональної системи. Цьому сприяє використовуваний принцип компонування зі стандартних блоків, завдяки якому забезпечується контроль над процесом розроблення змінюванних вимог.

Інкрементна модель реалізується таким чином (рис. 1.12).

Перший крок. Формулюється повний, наперед сформульований набір вимог, які поділяються за певними ознаками на групи.

Другий крок. Вибирається перша група вимог і виконується повне проходження по каскадній моделі. Після того як перший варіант системи,

що виконує першу групу вимог, здано замовнику, розробники переходять до наступного варіанта (другого інкремента), який виконує другу групу вимог, і т. д.

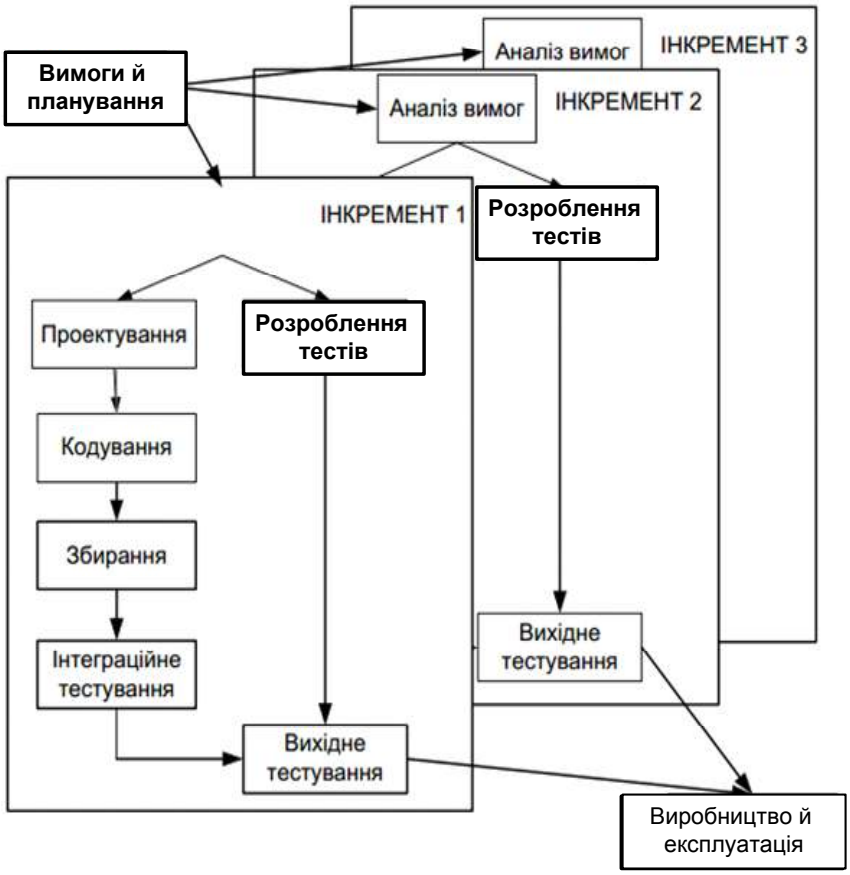


Рис. 1.12. Інкрементна модель

В інкрементній моделі описується процес, під час виконання якого увага приділяється перш за все системним вимогам, а потім – їх реалізації в групах розробників. Зазвичай з часом інкременти зменшуються і реалізують щоразу меншу кількість вимог. Кожна наступна версія системи додає до попередньої певні функціональні можливості доти, доки не буде реалізовано всі заплановані можливості. У цьому випадку можна зменшити витрати, контролювати вплив змінюваних вимог і прискорити створення системи, яка функціонує завдяки використанню методу компонування зі стандартних блоків.

Передбачається, що на ранніх етапах життєвого циклу (планування, аналізу й розроблення проекту) виконується конструювання системи в цілому. На цих етапах визначаються інкременти, що належать до них, і функції. Кожен інкремент проходить потім наступні фази життєвого циклу: кодування, тестування і постачання.

Спочатку виконується конструювання, тестування і реалізація набору функцій, які формують основу продукту, або вимог першорядного значення

для успішного виконання проекту або зменшення ступеня ризику. Наступні ітерації поширюються на ядро системи, поступово покращуючи її функціональні можливості або робочі характеристикі. Додавання функцій здійснюється з допомогою виконання суттєвих інкрементів з метою комплексного задоволення потреб користувача. Кожна додаткова функція атестується відповідно до великої кількості вимог.

Особливістю інкрементної моделі є розроблення приймальних тестів на етапі аналізу вимог, що спрощує приймання варіанта замовником і дає змогу встановити чіткі цілі при розробленні чергового варіанта системи. Інкрементна модель є особливо ефективною, коли завдання розбивається на декілька відносно незалежних підзавдань (розроблення підсистем «Персонал», «Касовий зал», «Склад», «Постачальники»). Крім того, в інкрементній моделі для внутрішньої ітерації можуть використовуватися каскадна, спіральна й V-подібна моделі, що дає змогу зменшити витрати й ризики під час розроблення системи.

Лінійну послідовність виконання інкрементів можна розподілити згідно з календарним графіком, причому внаслідок виконання кожної послідовності може створюватися результативний інкремент програмного продукту.

Переваги інкрементної моделі:

− не потрібно наперед витрачати кошти на весь проект, оскільки виконуються розроблення й реалізація основної функції ПЗ з групи високого ризику;

− унаслідок виконання кожного інкремента одержується готовий продукт;

− проблема, що виникла, розбивається на керовані частини, а це запобігає формуванню громіздких переліків вимог, які ставляться перед командою розробників;

− наприкінці постачання кожного інкремента існує можливість переглянути ризики, пов’язані з витратами і дотриманням установленого графіка;

− потреби клієнта краще задовольняються, оскільки час розроблення кожного інкремента є дуже незначним;

− замовники можуть розпізнавати найважливіші й найкорисніші функціональні можливості продукту на найперших етапах розроблення;

− ризик поділяється на декілька менших за розміром інкрементів і його не зосереджено в одному великому проекті розроблення;

− вимоги стабілізуються (з допомогою залучення до процесу користувачів) на момент створення певного інкремента, оскільки вимоги, що не є особливо важливими, відстрочують до моменту створення наступних інкрементів;

− поліпшується розуміння вимог для пізніших інкрементів, що забезпечується завдяки можливості користувача отримати уявлення про раніше

отримані інкременти на практичному рівні.

Недоліки інкрементної моделі:

− не передбачено ітерації в межах кожного інкремента;

− визначення повної функціональної системи має здійснюватися на початку життєвого циклу, щоб забезпечити визначення інкрементів;

− виникає необхідність у чітко визначених інтерфейсах, оскільки створення деяких модулів буде завершено значно раніше за інші;

− формальний критичний аналіз і перевірку набагато важче виконати для інкрементів, ніж для системи в цілому;

− може виникнути тенденція до відстрочення вирішення важких проблем на майбутнє з метою продемонструвати керівництву успіх, досягнутий на найперших етапах розроблення.

Інкрементна модель застосовується в таких випадках:

− якщо більшість вимог можна сформулювати заздалегідь, але їх поява очікується через певний період часу;

− якщо ринкове «вікно» надто «вузьке» і є потреба у швидкому постачанні на ринок продукту, що має функціональні базові властивості;

− для проектів, на виконання яких передбачено великий термін часу розроблення, зазвичай один рік;

− під час розроблення програм, пов’язаних з низьким або середнім ступенем ризику;

− під час виконання проекту із застосуванням нової технології, що дає змогу користувачеві адаптуватися до системи шляхом виконання незначних інкрементних кроків, без різкого переходу до застосування основного нового продукту;

− коли однопрохідне розроблення системи пов’язане з великим ступенем ризику.

1.2.6. Спіральна модель

Зазвичай розроблення ПЗ має циклічний характер, коли після виконання певних фаз доводиться повертатися до попередніх. Можна навести дві основні причини таких повернень:

− помилки, яких припустилися розробники на ранніх стадіях і які було виявлено пізніше (помилки аналізу вимог, проектування, кодування, що виявляються зазвичай на стадії тестування);

− помилки замовників, що спричиняють змінення вимог під час розроблення.

У спіральній моделі, вперше представленій Баррі Боемом 1988 року, враховано недоліки каскадної моделі, з допомогою якої не можна адекватно вирішити проблеми внесення змін. У моделі наведено відносно стандартну й впорядковану послідовність стадій розроблення, але не передбачено використання методів прискореного прототипування або мов високого

рівня. Спіральна модель втілює в собі переваги каскадної моделі, при цьому до її складу також включено:

− аналіз ризиків, керування ними, а також процеси підтримки й керування;

− розроблення програмного продукту під час використання методу прототипування або швидкого розроблення додатків з допомогою мов програмування і вдосконалених засобів розроблення.

Спіральна модель життєвого циклу ПЗ реалізується таким чином (рис. 1.13):

− розроблення варіантів продукту подається як набір циклів спіралі, що розкручується;

− кожному циклу спіралі відповідає така ж кількість стадій, як і в каскадній моделі, при цьому початкові стадії, пов'язані з аналізом і плануванням вимог, наведено детальніше із додаванням нових елементів.

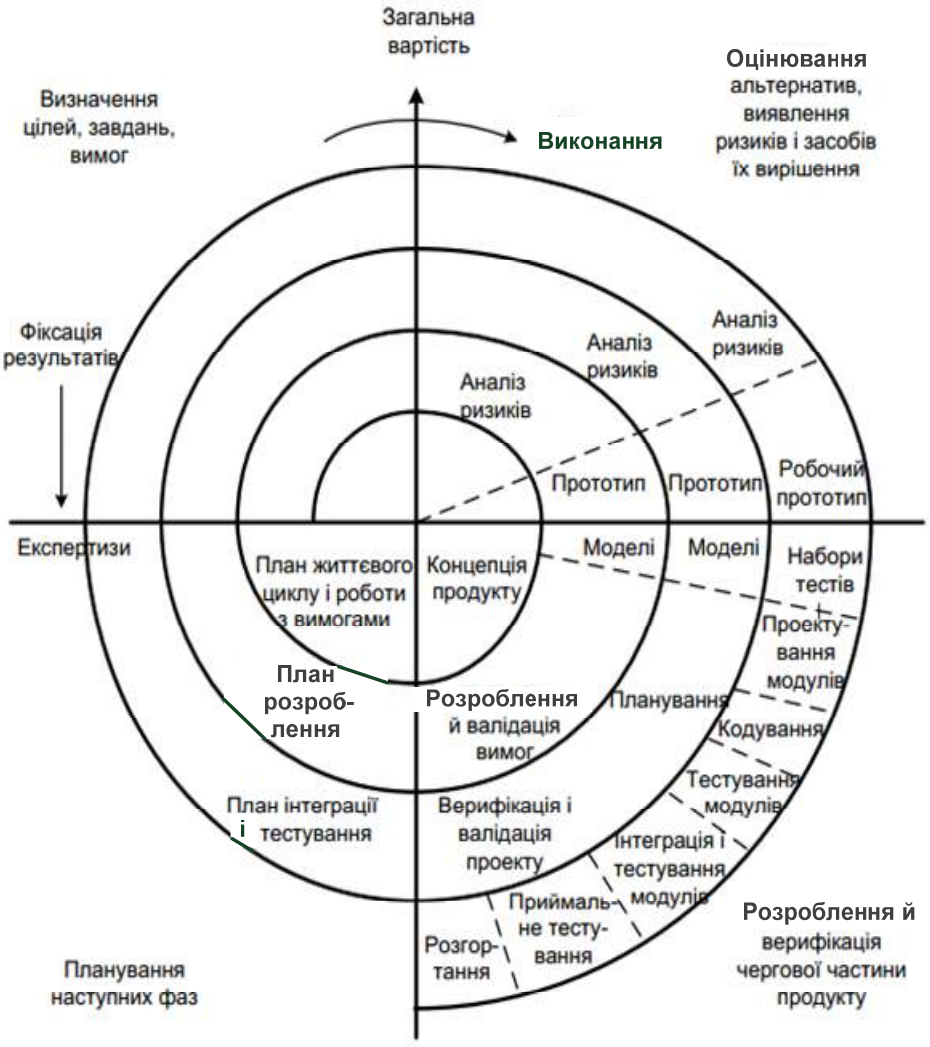


Рис. 1.13. Спіральна модель

У кожному циклі моделі вирізняють чотири базові фази: визначення цілей, альтернативних варіантів і обмежень; оцінювання альтернативних варіантів, ідентифікація і вирішення ризиків; розроблення продукту наступного рівня; планування наступної фази.

До початку розроблення ПЗ є декілька повних циклів аналізу вимог і проектування. Кількість циклів моделі (як у частині аналізу й проектування, так і в частині реалізації) є необмеженою і визначається складністю й обсягом завдання. У моделі передбачено повернення до залишених варіантів у разі змінення вартості ризиків.

Перший цикл – створення бачення продукту: визначаються загальні цілі, попередні обмеження, можливі альтернативи підходів до вирішення завдання. Далі оцінюються підходи, встановлюються їх ризики. На етапі розроблення створюється концепція (бачення) продукту і шляхів її створення.

Другий цикл – аналіз вимог – починається з планування вимог і деталей ЖЦ продукту для оцінювання витрат. На етапі визначення цілей встановлюються альтернативні варіанти вимог, пов’язані з їх ранжуванням за важливістю й вартістю виконання. На етапі оцінювання встановлюються ризики варіантів вимог, на етапі розроблення – специфікація вимог (із зазначенням ризиків і вартості), готується демоверсія ПЗ для аналізу вимог замовником.

Третій цикл – розроблення проекту – починається з планування процесу розроблення.

На етапі визначення цілей установлюються обмеження проекту (за термінами, обсягом фінансування, ресурсами і т. д.), визначаються альтернативи проектування, пов’язані з альтернативами вимог, використовуваними технологіями проектування, залученням субпідрядників тощо. На етапі оцінювання альтернатив встановлюються ризики варіантів і здійснюється вибір варіанта для подальшої реалізації. На етапі розроблення виконується проектування і створюється демоверсія, що відображає основні проектні рішення.

Четвертий цикл – реалізація ПЗ – починається з планування реалізацій. Альтернативними варіантами реалізації можуть бути використовувані технології реалізації, залучені ресурси. Альтернативи і пов’язані з ними ризики у цьому циклі оцінюються за ступенем «відпрацьованості» технологій і «якістю» наявних ресурсів. Етап розроблення виконується за каскадною моделлю, результатом якої є діючий варіант (прототип) продукту.

Основні **принципи спіральної моделі** можна сформулювати таким чином:

− розроблення варіантів продукту, що відповідають різним варіантам вимог, з можливістю повернутися до попередніх варіантів;

− створення прототипів ПЗ як засобів спілкування із замовником для уточнення й виявлення вимог;

− планування наступних варіантів з оцінюванням альтернатив і аналізом ризиків, пов'язаних з переходом до наступного варіанта;

− перехід до розроблення наступного варіанта до завершення попереднього у разі, коли ризик завершення чергового варіанта (прототипу) стає невиправдано високим;

− використання каскадної моделі як схеми розроблення чергового варіанта;

− активне залучення замовника до роботи над проектом; замовник бере участь в оцінюванні чергового прототипу ПЗ, уточненні вимог у разі переходу до наступного варіанта, оцінюванні запропонованих альтернатив чергового варіанта й оцінюванні ризиків.

Переваги спіральної моделі:

− користувач може побачити систему на найперших етапах завдяки швидкому прототипуванню;

− забезпечується визначення нездоланних ризиків без додаткових витрат; користувач може брати активну участь у плануванні, аналізуванні ризиків, розробленні й оцінюванні;

− забезпечується поділ великого потенційного обсягу роботи з розробленням продукту на невеликі частини, у яких спочатку реалізуються вирішальні функції з високим ступенем ризику, тому в разі потреби можна припинити роботу над проектом і тим самим зменшити витрати;

− передбачено можливість гнучкого проектування, оскільки існують переваги каскадної моделі, а також дозволені ітерації для всіх фаз цієї моделі;

− реалізовано переваги інкрементної моделі (випуск інкрементів, скорочення графіка з допомогою перекривання інкрементів, розсортованих за версіями, і незмінність ресурсів при поступовому зростанні системи);

− зворотний зв'язок у напрямку від користувача до розробника виконується з високою частотою і на найперших етапах моделі, що забезпечує створення продукту високої якості;

− удосконалення адміністративного керування процесом забезпечення якості, процесом розроблення, витратами, графіком і кадровим забезпеченням, що досягається шляхом виконання огляду наприкінці кожної ітерації;

− підвищення продуктивності завдяки застосуванню придатних для повторного використання властивостей;

− підвищення ймовірності передбаченої поведінки системи з допомогою уточнення поставлених цілей;

− немає потрібності у розподілі заздалегідь усіх необхідних для вико-

нання проекту фінансових ресурсів;

− можна часто оцінювати сукупні витрати.

Недоліки спіральної моделі:

− складність аналізу й оцінювання ризиків під час вибору варіантів; якщо проект має низький ступінь ризику або невеликі розміри, то модель може виявитися дорогою;

− складність структури моделі, що може ускладнити її застосування розробниками, менеджерами й замовниками;

− складність підтримки версій продукту (зберігання версій, повернення до найперших версій, комбінація версій, документування версій);

− нескінченність моделі – після кожної створеної версії замовник може ставити нові вимоги, які приводять до наступного циклу розроблення.

Спіральну модель застосовують у таких випадках:

− коли користувачі не впевнені у своїх потребах;

− коли вимоги є надто складними й можуть змінюватися під час виконання проекту і необхідно мати прототипування для аналізу й оцінювання вимог; під час розроблення нової функції або нової серії продуктів;

− коли досягнення успіху не є гарантованим і необхідно оцінити ризик продовження проекту;

− коли проект є складним, дорогим і його фінансування можливе лише під час виконання;

− коли йдеться про застосування нових технологій, що пов’язано з ризиком їх освоєння і досягнення очікуваного результату;

− під час виконання дуже великих проектів, які через обмеженість ресурсів можна здійснювати лише частинами;

− коли очікуються суттєві зміни, наприклад, у процесі вивчення або дослідної роботи;

− для організацій, які не можуть собі дозволити зазделегіть виділити усі необхідні для виконання проекту грошові кошти, і коли для процесу розроблення немає фінансової підтримки;

− коли переваги розроблення неможливо точно визначити, а досягнення успіху не є гарантованим;

− під час розроблення систем, що потребують великого обсягу обчислень, забезпечують прийняття рішень, під час виконання бізнес-проектів, а також проектів у галузі оборони, інжинірингу і т. д.

На рис 1.14 показано інструментальні засоби керування проектами й стандарти ЖЦ ПЗ. До цієї класифікації увійшли методології з керування розробленням ПЗ і методології впровадження ПЗ. Методології містять рекомендації з використання окремих інструментів: метрик, технічних стандартів, мов графічного моделювання.

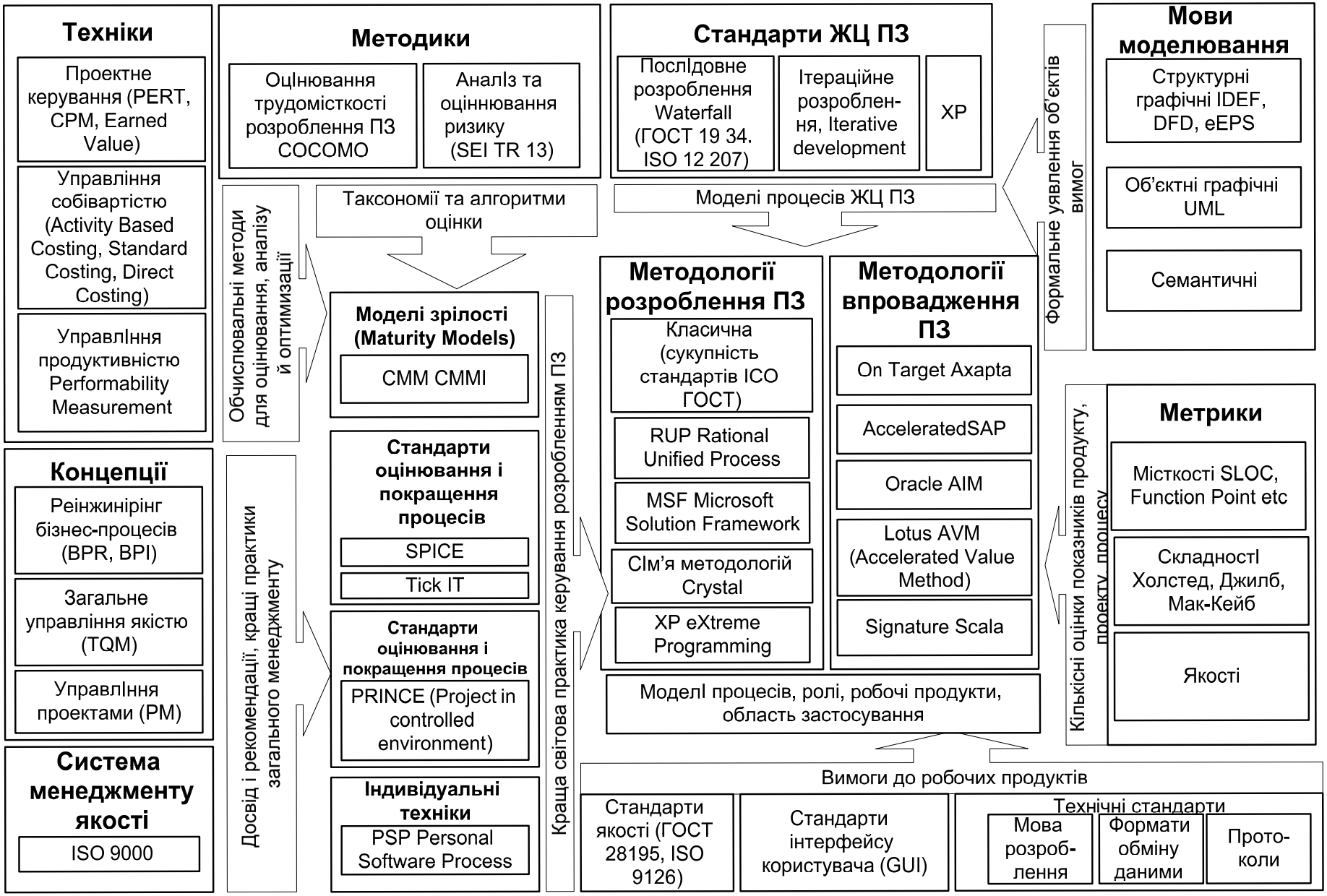


Рис. 1.14. Інструментальні засоби в проектах ПЗ

Контрольні запитання

1. Охарактеризуйте стани життєвого циклу продукту.
2. Що таке життєвий цикл програмного продукту?
3. Назвіть основні характеристики ЖЦ ПЗ.

1.3. Міжнародні й національні стандарти розроблення складних програмних продуктів

Перш за все програміст повинен добре знати закони України, за якими регламентують інформаційні роботи, а саме:

– про стандартизацію від 17 травня 2001 р. № 2408-III;

– про інформацію від 2 жовтня 1992 року № 2657-XII;

– про Національну програму інформатизації від 4 лютого 1998 року № 75/98-ВР, зі змінами від 13 вересня 2001 року 2684-III;

– про захист інформації в інформаційно-телекомунікаційних системах від 5 липня 1994 р. № 80/94-ВР;

– про авторське право і суміжні права, зі змінами і доповненнями від 28 лютого 1995 року № 75/95-ВР.

Існуючі стандарти прийнято поділяти на корпоративні, галузеві, державні й міжнародні [4].

Корпоративні стандарти розробляють великі фірми (корпорації) з метою підвищення якості своєї продукції. Такі стандарти базуються на власному досвіді фірм і в них ураховано вимоги світових стандартів. Корпоративні стандарти не сертифікуються, але є обов'язковими для застосування всередині корпорації. В умовах ринкової конкуренції стандарти можуть мати закритий характер. У сфері IT-технологій застосовують відомі стандарти, розроблені фірмами IBM, Intel, Microsoft тощо.

Галузеві стандарти діють у межах організацій певної галузі, міністерства (наприклад, стандарти спеціальностей, яким навчають у навчальних закладах Міністерства освіти і науки України). Ці стандарти розробляються з урахуванням вимог світового досвіду й специфіки галузі, є зазвичай обов'язковими й підлягають сертифікації.

Державні стандарти (ГОСТ, ДСТУ) приймаються державними органами і мають силу закону. Розробляються з урахуванням світового досвіду або на основі галузевих стандартів. Можуть мати як рекомендаційний, так і обов'язковий характер, як, наприклад, стандарти пожежної безпеки. Для сертифікації створюються державні органи або органи ліцензування. Національні стандарти України щодо розроблення програмного статку і системної документації наведено в табл. Д.1.1.

Міжнародні стандарти розробляють зазвичай спеціальні міжнародні організації на основі світового досвіду і кращих корпоративних стандартів. Такі стандарти мають суто рекомендаційний характер. Право сертифікації отримують організації (державні й приватні), що пройшли ліцензування в

міжнародних організаціях. Основні організації-розробники міжнародних стандартів у сфері програмної інженерії наведено в табл. Д.1.2.

Під час розроблення й практичного застосування стандартів стосовно ЖЦ ПЗ виникло декілька проблем: вкладення значних коштів для застосування стандартів, що не завжди окупалося; не було ясності, які процеси треба виконувати і в якому обсязі; до різних типів програмних систем (ІС організаційного керування, керування технологічними процесами і т. д.) ставилися різні вимоги; динамічність ІТ-галузі призвела до швидкого «старіння» стандартів; виникла термінологічна невідповідність різних корпоративних стандартів. Тому світовими організаціями було розроблено декілька стандартів у галузі програмної інженерії, у яких вони намагалися вирішити ці проблеми.

Серед найбільш відомих стандартів програмної інженерії можна виокремити такі:

а) ISO/IEC 12207 – Information Technology – Software Life Cycle Processes – процеси життєвого циклу програмних засобів;

б) ISO/IEC 15504 – Software Process Assessment – оцінювання й атестація зрілості процесів створення й супроводу ПЗ;

в) SEI CMM – Capability Maturity Model (for Software) – модель зрілості процесів розроблення програмного забезпечення;

г) SEI CMMI (Capability Maturity Model Integration) – інтеграція моделей зрілості процесів розроблення програмного забезпечення;

д) PMBOK – Project Management Body of Knowledge – зведення знань з керування проектами. Стандарт PMBOK, розроблений PMI, містить описи дев'яти розділів (галузей знань) керування:

– інтеграцією – Project Integration Management;
– обмеженнями – Project Scope Management;
– часом – Project Time Management;
– витратами – Project Cost Management;
– ризиками – Project Risk Management;
– персоналом – Project Personnel Management;
– комунікаціями – Project Communication Management;
– закупівлями – Project Procurement Management;
– якістю – Project Quality Management;

е) IEEE SWEBOK – IEEE Computer Society Software Engineering Body of Knowledge Software Engineering Body of Knowledge – зведення знань з програмної інженерії – проект IEEE Computer Society. Основна ідея проекту аналогічна PMBOK і полягає у створенні певного базового набору загальновживаних знань, необхідних будь-якому професійному програмістові. Проект містить описи 10 розділів (галузей знань) програмної інженерії:

– Software Requirements – вимоги до ПЗ;
– Software Design – проектування ПЗ;
– Software Construction – конструювання ПЗ;
– Software Testing – тестування ПЗ;

– Software Maintenance – супровід ПЗ;
– Software Configuration Management – керування конфігураціями;
– Software Engineering Management – керування ІТ-проектом;
– Software Engineering Process – процес програмної інженерії;
– Software Engineering Tools and Methods – методи й інструменти;
– Software Quality – якість ПЗ.

Стандарти є сумою досвіду, накопиченого експертами в інженерії ПЗ на основі великої кількості проектів, що проводились у межах комерційних структур США і Європи, а також військових контрактів. Велика частина стандартів створювалася як набір критеріїв для відбору постачальників програмного забезпечення для Міністерства оборони США, і це завдання вони вирішують досить успішно.

1.3.1. Стандарт ISO/IEC 12207

У стандарті ISO/IEC 12207 визначається організація життєвого циклу програмного продукту як сукупність процесів, кожен із яких розбито на дії, що складаються з окремих завдань, встановлюється структура (архітектура) життєвого циклу програмного продукту у вигляді переліку процесів, дій і завдань (рис. 1.15).

Процес – це велика кількість взаємозв'язаних дій і ресурсів, які перетворюють входи на виходи.

Дія – це елемент процесу проекту. Дії зазвичай мають очікувану тривалість, потребу в ресурсах, вартість і можуть поділятися на завдання.

Відповідно до стандарту ISO/IEC 12207 процеси організації ЖЦ ПЗ поділяються на три групи: основні процеси, процеси підтримки, організаційні процеси.

Рис. 1.15. Організація ЖЦ ПЗ згідно зі стандартом ДСТУ 3918−1999 (ISO/IEC 12207)

До **основних процесів** життєвого циклу належать такі (рис. 1.16):

– замовлення – визначаються дії замовника, тобто організації, яка замовляє систему, програмний продукт або програмну послугу;

– постачання – визначаються дії постачальника, тобто організації, яка надає систему, програмний продукт або програмну послугу замовнику;

– розроблення – визначаються дії розробника, тобто організації, яка визначає і проектує програмний продукт;

– експлуатація – визначаються дії оператора, тобто організації, що надає послуги із експлуатації комп'ютерної системи в її наявному середовищі для її користувачів;

– супровід – визначаються дії супроводжувача, тобто організації, яка надає послуги із супроводу програмного продукту, тобто керує внесенням змін до програмного продукту з метою підтримання його в належному й працездатному стані; цей процес охоплює перенесення програмного продукту й вилучення його з експлуатації.

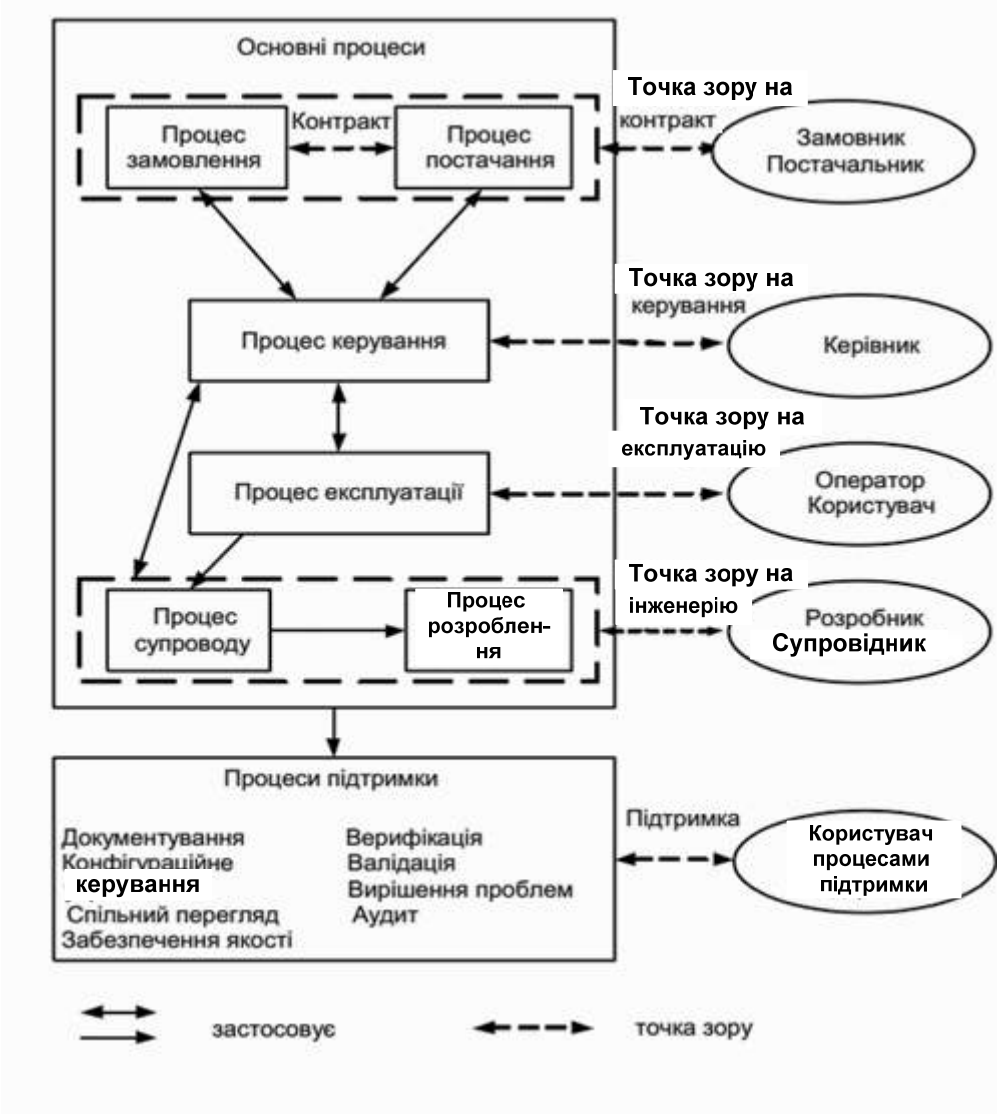


Рис. 1.16. Взаємозв'язки між процесами життєвого циклу:
застосування, точка зору

До **процесів підтримки** життєвого циклу належать:

– документування – реєстрація інформації, виробленої під час життєвого циклу;

– конфігурування – керування конфігурацією;

– забезпечення якості – набуття об'єктивної впевненості в тому, що програмні продукти й процеси відповідають заданим для них вимогам і

встановленим планам; як методи забезпечення якості можна використовувати процеси спільного перегляду, аудиту, верифікації і валідації;

– верифікація – проведення контролю програмного продукту з різним ступенем глибини залежно від програмного проекту;

– валідація – оцінювання програмного продукту, одержаного в межах програмного проекту;

– спільний перегляд – оцінювання стану й результатів певної дії, цей процес можуть застосовувати будь-які два учасники, один із яких здійснює перегляд дій іншого учасника в межах спільного обговорення;

– аудит – визначення відповідності вимогам, планам і контракту, цей процес можуть застосовувати будь-які два учасники, один із яких проводить аудит програмного продукту або дій іншого учасника;

– вирішення проблем – аналіз і зняття проблем (включно з невідповідностями) незалежно від їхньої природи і причин, виявлених під час розроблення, експлуатації, супроводу або виконання інших процесів.

До **організаційних процесів** життєвого циклу належать:

– керування проектом;

– створення інфраструктури;

– удосконалення – створення, вимірювання, контроль і вдосконалення процесів життєвого циклу;

– навчання персоналу.

Динамічність ІТ-галузі потребувала подальшого опрацювання стандарту ISO 12207, тому 1998 р. вийшов новий стандарт ISO/IEC TR 15504: Information Technology – Software Process Assessment – оцінювання процесів розроблення ПЗ. У цьому документі розглянуто питання атестації, визначення зрілості й удосконалення процесів життєвого циклу ПЗ.

У стандарті ISO 15504 запроваджено нову класифікацію процесів:

1) основні процеси: замовник-постачальник (CUS); інженерні (ENG);

2) допоміжні процеси: допоміжні (SUP);

3) організаційні процеси: управлінські (MAN); організаційні (ORG).

1.3.2. Модель CMM

У моделі CMM запропоновано уніфікований підхід до оцінювання можливостей організації виконання завдання різного рівня. Для цього визначаються три рівні елементів:

– рівні зрілості організації (maturity levels);

– ключові області процесу (key process areas);

– ключові практики (key practices).

Найчастіше під моделлю CMM розуміється саме модель рівнів зрілості. Професіоналізм організації визначається з допомогою зрілості процесу. У стандарті вирізняють п’ять рівнів зрілості процесу, що є визначенням ступіня готовності організації виконати великий проект:

– *початковий (initial)* – технологія є повністю імпровізованою, іноді навіть хаотичною; успіх цілком залежить від зусиль окремих співробітників; до цього рівня належать організації, що розробляють ПЗ, але не мають усвідомленого процесу розроблення, не планують його і не оцінюють своїх можливостей;

– *повторюваний (repeatable)* – базові процеси керування проектом ПЗ установлено; є дисципліна дотримання, що забезпечує можливість повторення успіху попередніх проектів у тій самій прикладній області; в організаціях, що відповідають цьому рівню, ведеться облік витрат ресурсів і відстежується розвиток проектів, встановлюються правила керування проектами, основані на отриманому досвіді;

– *визначений (defined)* – процеси задокументовано, стандартизовано й інтегровано в єдину для всієї організації технологію створення ПЗ;

– *керований (manageable)* – збираються й накопичуються метрики (об'єктивні дані) про якість виконання процесів і вихідної продукції; керування процесами і вихідною продукцією здійснюється за кількісними оцінками. В організаціях такого рівня крім встановленого й описаного процесу використовуються вимірні показники якості продуктів і результативності процесів, що дає змогу досить точно передбачити обсяг ресурсів (часу, грошей, персоналу), необхідний для розроблення продукту з певною якістю;

– у*досконалюваний (optimizing)* – технологія створення ПЗ удосконалюється безперервно на основі кількісного зворотного зв'язку від процесів і пілотного впровадження інноваційних ідей; у таких організаціях крім процесів і методів їх оцінювання є методи визначення слабких місць, визначено процедури пошуку й оцінювання нових методів і техніки розроблення, навчання персоналу роботі з ними і їх включення до загального процесу організації в разі підвищення ними ефективності виробництва.

Ключові області процесу. Згідно із CMM, рівні зрілості організації можна визначати за використанням в організації чітко встановленої техніки і процедур, що належать до різних ключових галузей процесу.

Кожна така область є набором пов'язаних загальними цілями видів діяльності, суттєвих для оцінювання результативності технологічного процесу в цілому. Усього вирізняють 18 областей. Велика кількість ділянок, які мають підтримуватися організацією, збільшується під час переходу до вищих рівнів зрілості.

До першого рівня не ставиться ніяких вимог. Організації другого рівня зрілості мають певним чином підтримувати керування вимогами, планування проектів, нагляд за виконанням проекту, керування підрядниками, забезпечення якості ПЗ, керування конфігурацією. Організації третього рівня повинні крім діяльностей другого рівня підтримувати проведення експертиз, координацію діяльності окремих груп, розроблення програмного продукту, інтегроване керування розробленням і супроводом, навчання персоналу, виробляти і підтримувати технологічний процес організації, ко-

нтролювати дотримання технологічного процесу організації. До діяльності організацій четвертого рівня додаються керування якістю ПЗ і процесом. Організації п’ятого рівня зрілості мають додатково підтримувати керування змінами процесу й використовуваних технологій і запобігати дефектам.

1.3.3. Модель UML

Уніфікована мова моделювання (UML – Unified Modeling Language) – мова графічного опису для об'єктного моделювання в області розроблення ПЗ. UML є мовою широкого профілю, це – відкритий стандарт, у якому використовуються графічні позначення для створення абстрактної моделі системи, що має назву UML-моделі (ISO / IEC 19505-1, 19505-2). UML було створено для визначення, візуалізації, проектування й документування, в основному програмних систем. UML не є мовою програмування, але на основі UML-моделей можливою є генерація коду.

Можна сказати, що модель UML – це граф (точніше, навантажений мульти-псевдо-гіперорграф), у якому вершини й ребра навантажено додатковою інформацією і можуть мати складну внутрішню структуру. Вершини цього графа називають сутностями, а ребра – відносинами.

Структурні сутності призначено для опису структури. Зазвичай до структурних сутностей належать такі:

– клас – опис багатьох об'єктів із загальними атрибутами й операціями;

– інтерфейс – велика кількість операцій для визначення набору послуг, що надаються класом або компонентом;

– дійова особа – сутність, яка перебуває поза модельованою системою і безпосередньо взаємодіє з нею;

– варіант використання – опис послідовності вироблених системою дій, що надає значущого для деякої діючої особи результату;

– компонент – фізично замінюваний артефакт, який реалізує деякий набір інтерфейсів;

– вузол – фізичний обчислювальний ресурс.

Поведінкові сутності призначено для опису поведінки. Основних поведінкових сутностей усього дві:

– стан – період в життєвому циклі об'єкта, у якому об'єкт задовольняє деяку умову, виконує діяльність або очікує подію.

– діяльність – стан, при якому виконується робота, а не просто пасивно очікується настання події (подія – атомарний неподільний елемент роботи).

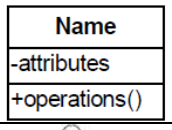
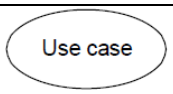
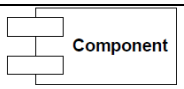
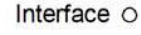
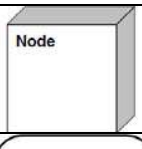
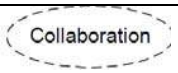
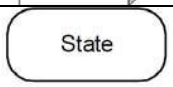
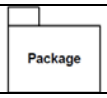
Крім того, при моделюванні поведінки використовується ще кілька допоміжних сутностей (які тут не наведено), тому що співіснують тільки разом із зазначеними основними.

В UML групується одна універсальна сутність – **пакет** – група елеме-

нтів моделі (у тому числі пакетів).

У табл. 1.1 наведено стандартне графічне позначення для сутностей.

Таблиця 1.1

Найменування	Позначення	Найменування	Позначення	Найменування	Позначення
Клас		Варіант використання		Діяльність	
Дійова особа		Компонент		Розвилка / злиття	
Інтерфейс		Вузол		Розгалуження / з'єднання	
Кооперація		Стан		Пакет	

Анотаційна сутність теж одна – **примітка** – зате в неї можна помістити все що завгодно, оскільки зміст примітки в UML не обмежується (рис. 1.17).

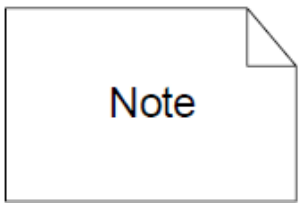


Рис. 1.17. Примітка

В UML використовуються чотири основних **типи відношень**: залежність, асоціація, узагальнення, реалізація.

Залежність – це найбільш загальний тип відношень між двома сутностями. Відношення залежності свідчить про те, що змінення незалежної сутності якимось чином впливає на залежну. Зазвичай семантика конкретної залежності уточнюється в моделі з допомогою додаткової інформації. Наприклад, залежність зі стереотипом «use» означає, що залежна сутність використовує незалежну (наприклад, викликає операцію).

Асоціація – це найбільш часто використовуваний тип відношень між сутностями. Відношення асоціації має місце, якщо одна сутність безпосередньо пов'язана з іншою (або з іншими, оскільки асоціація може бути не тільки бінарною). Графічно асоціація зображується у вигляді суцільної лінії з різними доповненнями, що з'єднує пов'язані сутності. На програмному рівні безпосередній зв'язок може бути реалізовано різним чином, головне, що асоційовані сутності «знають» один про одного. Наприклад, відношення часть – ціле є окремим випадком асоціації.

Агрегація – це асоціація між цілим і його частиною (або частинами). Агрегацію замість асоціації наводять, якщо відношення ціле – частина в конкретному випадку є істотним. Наприклад, якщо колесо для людей є

тільки частиною автомобіля, то між відповідними класами доцільно вказати відношення агрегації, а якщо колесо – товар, як і автомобіль, то зв'язок ціле – частина є неістотним.

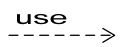
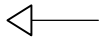
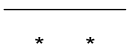
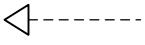
Композиція – сильніший різновид агрегації, де припускається, що об'єкт-частина може належати тільки єдиному цілому і при цьому створюватися і знищуватися тільки разом зі своїм цілим.

Узагальнення – це відношення між двома сутностями, одна з яких є приватним (спеціалізованим) випадком іншої. В UML відношення узагальнення виконується за принципом підстановки: якщо сутність А (загальне) є узагальненням сутності Б (окремого), то Б можна підставити замість А в будь-якому контексті. Принцип підстановки є правомірним для використання в об’єктно-орієнтованному програмуванні, оскільки існує узагальнення, що сутність класу Б має всі властивості й поведінку сутності классу А. Графічно узагальнення зображується у вигляді суцільної стрілки з трикутником на кінці, спрямованої від окремого до загального. Відношення спадкування між класами в об'єктно-орієнтованих мовах програмування є типовим прикладом узагальнення.

Відношення **реалізації** застосовується дещо менше, ніж попередні три типи відношень, оскільки часто використовуються за замовчуванням. Відношення реалізації свідчить про те, що одна сутність є реалізацією іншої. Наприклад, клас є реалізацією інтерфейсу. Графічно реалізація зображується пунктирною стрілкою з трикутником на кінці, спрямованою від реалізаційної сутності до реалізованої.

Перелічені типи відношень є основними, їх позначення наведено в табл. 1.2.

Таблиця 1.2

Найменування	Позначення	Найменування	Позначення	Найменування	Позначення
Залежність		Композиція		Узагальнення	
Асоціація		Агрегація		Реалізація	

Контрольні запитання

1. Що таке стандарт? Які види стандартів існують? Наведіть приклади по кожному виду стандартів.
2. Охарактеризуйте основні організації-розробники стандартів у галузі програмної інженерії.
3. Назвіть найбільш відомі стандарти в галузі програмної інженерії.
4. Назвіть взаємозв’язки ПЗ, код якого наведено у дод. 3.

2. МЕТОДИ І ЗАСОБИ РОЗРОБЛЕННЯ ПЗ

2.1. Методології розроблення ПЗ (RUP, MSF, XP, DSDM, RAD)

Хорошу методологію ніколи не буде обмежено популяризацією якої-небудь ідеї, методу або інструментарію, її буде запропановано як комплексну підтримку діяльності виконавців проекту. Однак часто замість методології фактично пропонується набір інструментів, здатних тою чи іншою мірою підтримувати деякі методології без урахування меж застосовності. Це дає змогу підтримувати різні методології. Однак від застосування інструментів до реальної підтримки – довгий шлях, набагато більш невизначений, ніж процес розроблення інструментарію. І це, на жаль, зазвичай замовчується, що цілком відповідає прагненням не обмежувати сферу застосування пропонованих засобів межами конкретної методології. Це, мабуть, є основною причиною утруднень при застосуванні методологій у конкретних умовах.

У цьому розділі буде розглянуто практичне застосування засобів підтримки ведення програмних проектів, виходячи з прикладів популярних методологій.

2.1.1. Модель RUP

Сьогодні 51 % програмних розробок застосовують у CASE-системах, що підтримують RUP (Rational Unified Processing). Уже одного цього досить, щоб звернути увагу на цей підхід і дослідити, завдяки чому забезпечується відчутна користь для практичних розробок. Безумовно, перевагою RUP є запропонований інструментарій моделювання різних аспектів реалізованого ПО, і саме цей інструментарій застосовується найчастіше. Він приваблює розроблювачів виразністю, підтримкою узгодженого використання систем моделей, що зв'язані загальною системою понять.

У межах цього розділу акцентується увага на можливості підтримки діяльності менеджера проекту в RUP. Для цього розглянемо модель життєвого циклу як основи методичних підходів, які можуть розвиватися на базі RUP.

RUP може стати універсальною основою будь-яких програмних розробок за єдиною («раціональною») схемою. Тому природно, що автори пропонують загальну модель життєвого циклу, яка буде придатною для всіх проектів, що розвиваються «раціонально». Відповідно до орієнтації на стратегію ітеративного розвитку модель має підтримувати ведення ітеративних програмних проектів, етапи, що виконуються під час ітерації, і виробничі функції.

Модель задається у вигляді матриці интенсивностей функцій, виконуваних на етапах (фазах), які проектуються на ітерації (рис. 2.1).

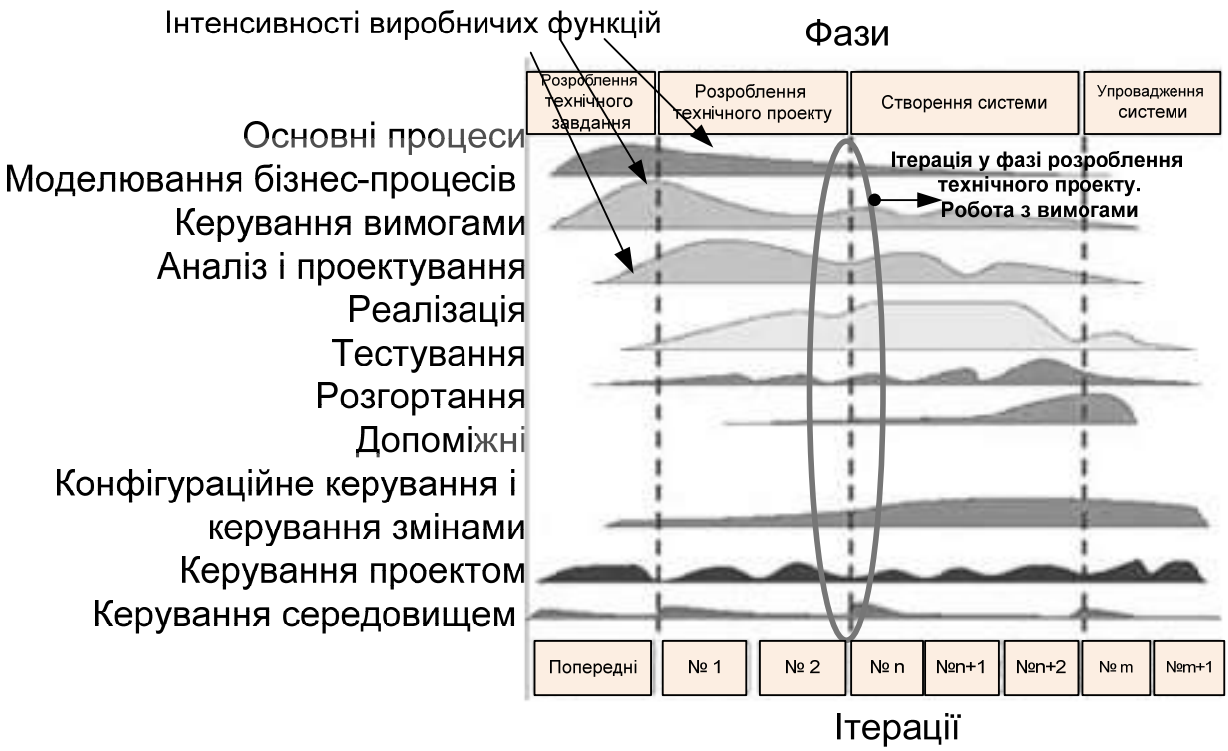


Рис. 2.1. Модель RUP

Автори CASE-систем, що підтримують RUP, незмінно підкреслюють ілюстративний стиль зображення інтенсивностей. Для кожної ітерації можна зазначити фазу, у якій вона перебуває в цей момент (сірий овал на рис. 2.1), а також функцію яка розглядається (точка зі стрілкою, напрямленою у бік наступної фази).

За межами моделі залишаються способи, за допомогою яких можна було б переходити до функцій від фаз життєвого циклу. У моделі не конкретизуються види робіт на етапах — це залишається в описовій частині документа про процес розроблення. Час також є досить умовним: дослідження окремої функції дає уявлення про горизонтальний розвиток подій (переходів від фази до фази), але про можливості спільного виконання деяких виробничих функцій не йдеться (тому, мабуть, функції досліджуються у великому плані, а схеми моделі, що ілюструють опис RUP, змінюються під час дослідження різних аспектів).

Модель має вигляд універсальної схеми: з точністю до найменувань у ній відображається те, що додається до будь-якого виробництва програмного забезпечення. Однак її не можна включити до схеми додаткових етапів і функцій, що відображають специфіку конкретного процесу або роботи колективу розроблювачів. Це порушило б фіксований зв'язок між життєвим циклом за RUP з моделями рівня проектування, яким у досліджуваному підході приділено особливу увагу.

У RUP наведено багато детально пророблених операційних маршру-

тів, але стосуються вони не діяльності користувача, а конкретних інструментів різного застосування (які використовуються переважно для моделювання). Засоби моделювання є елементами мови UML і, як наслідок, колекцією, а не комплексом підтримки процесів конкретних методологій. Це, однак, не означає заперечення інструментів CASE-систем RUP, які виявляються дуже корисними для реалізації різних методологічних підходів, чим і пояснюється їх популярність.

2.1.2. Модель процесів MSF

Пропозиція MSF (Microsoft Solutions Framework), що стосується життєвого циклу, ґрунтується на ідеї механічного «схрещування» каскадної моделі MSF і найпримітивнішої спиралеподібної моделі (рис. 2.2).

Рис. 2.2. Модель MSF

На жаль, автори пропозиції MSF не зовсім точно оцінили модель. Як і модель RUP, її можна зробити універсальною, що приведе до необхідності словесного доповнення схеми. Модель MSF має ті самі недоліки, що й модель з розкручуваною спіраллю: неможливість відстеження часових співвідношень між термінами виконання робіт, труднощі доповнення специфічних етапів.

Якщо звернутися до опису процесів, які відображають модель MSF, то стає помітним прагнення авторів зробити методологію гнучкою. Наприклад, автори пишуть про співробітництво із замовником, що залучення замовника до проекту є необхідною умовою його успішності. Модель процесів MSF надає замовникові широкий спектр можливостей для уточнення й модифікації проектних вимог і установки контрольних точок (віх) для моніторингу роботи над проектом. Зі свого боку, це потребує витрат часу замовника та перейняття ним на себе певних зобов'язань [22]. Однак далі гово-

риться про те, що «MSF визнає першорядну важливість договірних і юридичних відносин між замовником, його постачальниками і проектною командою та необхідність керування цими відносинами». Тільки на схемі ні перше, ні друге твердження побачити не можна. І це приклад словесного доповнення, яке доводиться вживати при незадовільному схематичному поданні моделі.

Вивчення методології, прописаної в MSF, дає змогу зробити висновок про те, що її автори досить ретельно проробили процеси менеджменту, коли основою організації колективу розроблювачів зробили проектну групу з розподіленими ролями. Однак в моделі життєвого циклу це ніяк не відображено, що цілком обумовлено прагненням до загальності. Звідси виходить, що обговорювана пропозиція в конкретних проектах завжди має бути адаптованою. Зробити це простіше, ніж, наприклад, у моделі RUP або в жорстких схемах, які тільки й наводяться в стандарті CMM. Однак стратегіями швидкого розвитку пропозиції MSF не стають і мають проміжне положення між жорсткими й гнучкими методологіями.

2.1.3. Життєвий цикл у методологіях RAD

Методолгія RAD (від англ. Rapid Application Development — швидке розроблення проектів) — концепція створення засобів розроблення програмних продуктів, коли приділяється особлива увага швидкості й зручності програмування, створенню технологічного процесу, що дає змогу програмісту максимально швидко створювати комп’ютерні програми. Концепцію RAD також часто пов'язують з концепцією візуального програмування.

За технологією в концепції RAD передбачено активне залучення замовника вже на найперших стадіях — обстеження організації, розроблення вимог до системи, тобто повне виконання функціональних і не функціональних вимог замовника, з урахуванням їх можливих змін під час розроблення системи, а також отримання якісної документації, що забезпечує зручність експлуатації і супроводу системи. Таким чином, терміни й витрати проекту фіксуються, а вимоги замовника можуть змінюватися (рис. 2.3).

Як стверджують прихильники швидкого розвитку, їхні методології не мають потреби в чіткому фіксуванні етапів розроблення програмного проекту. Однак вони розуміють, що життєвий цикл корисно враховувати під час розроблення ПЗ в концептуальному плані. Менеджер у своїй діяльності має дотримуватися самодисципліни й співробітництва замість дисципліни й підпорядкування; планування, контрольні й інші функції мають такий характер, що дає менеджеру змогу більшою мірою зосередитися на керуванні командою, а не процесом. Унаслідок цього відстеження процесу не потребує, наприклад, спеціальних документів про проблеми й досягнуті ре-

зультати, для яких потрібна спеціальна підтримка. З цієї причини моделі життєвого циклу швидкого розвитку не претендують на інструментальність, і з такого погляду їх розглядати не має сенсу. Проте поняття контрольних точок і контрольних заходів, розподілу ресурсів, оцінювання залишаються, хоча їхній зміст стає менш формалізованим.

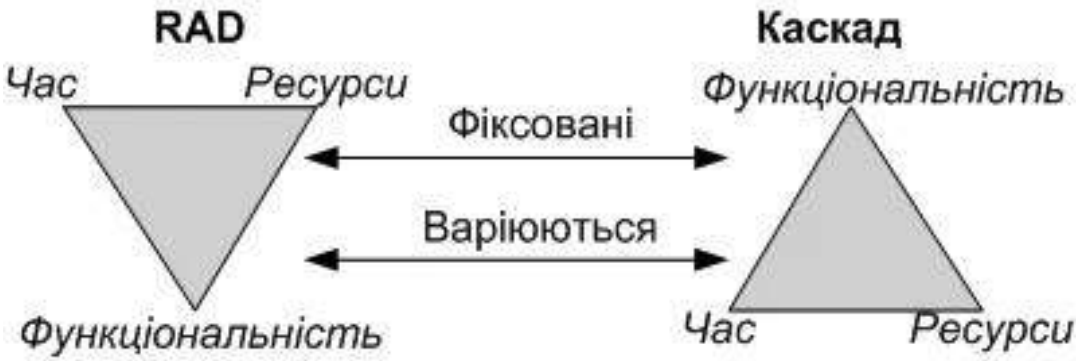


Рис. 2.3. Відмінність методології RAD від каскадного ЖЦ ПЗ

Життєвий цикл будь-якої методології швидкої розробки можна описати таким чином:

– початкова фаза, яку виокремлено, оскільки доводиться виконувати роботи, що не є характерними для основного процесу;

– кілька максимально коротких ітерацій, що складаються із кроків: вибір реалізованих вимог (в екстремальному програмуванні – історій користувача); реалізація тільки певних вимог; передання результату для практичного використання; короткий період оцінювання досягнутого результату (залежно від обсягу робіт цей період можна назвати етапом або контрольним заходом);

– фаза заключного оцінювання розроблення проекту.

З допомогою реальних швидких методологій конкретизують цю схему, доповнюють її тими або іншими методиками. До останнього часу швидкі процеси було не прийнято формалізовувати настільки, щоб пропонувати їх як стандарт. Прагнення до сертифікації стає причиною перенесення межі між гнучкими й жорсткими методологіями ближче до жорстких, і може бути таке, що внаслідок цього швидкі підходи стануть формалізованими настільки, що їх не можна вже буде називати гнучкими.

2.1.4. Модель життєвого циклу екстремального програмування (XP)

Кент Бек у своїй монографії описує життєвий цикл екстремального програмування, не наводячи схеми, тому розглянемо цю модель з допомогою рис. 2.4. Зі схеми видно, що початкова фаза розвитку проекту містить деяку кількість ітерацій без випуску релізу. Це період, коли просто неможливо надати систему користувачам. Як завдання вивчимо застосування

інструментів, постановку експериментів з метою визначення і наступного будування стартового варіанта архітектурного каркаса системи, а також, можливо, освоєння командою методик екстремального програмування.

Серія ітерацій – це стаціонарний період розвитку проекту, пов'язаний з обслуговуванням і підтримкою. Як тільки в проекті вичерпується постійно поповнюваний пул історій користувача, які замовляють для реалізації, зникає стимул для розвитку проекту. Цю ситуацію К. Бек називає «смертю» проекту. Однак використання побудованої системи не припиняється. Навпаки, відсутність нових вимог до неї свідчить про те, що потреби користувача забезпечено адекватною підтримкою. Можливими є й інші, менш оптимістичні причини для «смерті». Це або застосування користувачами конкурентної розробки, що виявилася більш придатною, або неможливість у розумний термін реалізувати необхідні можливості системи. У всіх випадках «смерть» проекту є подією його закінчення. У моделі позначено контрольні точки, пов'язані з подіями проекту (0—11): 0 – початок проекту, 1 − фомування історій користувача, 2 – вибор історії користувача для реалізації, 3 – структуризація історії користувача для формуваня завдань розробникам, 4 − випуск ПЗ редакції першого релізу, 5 – результати тестування ПЗ редакції першого релізу, 6 – оцінка якості ПЗ першого релізу, 7 – вибір історії користувача для наступної версії ПЗ, 8 − структуризація історії користувача для формуваня завдань розробникам, 9 − випуск ПЗ редакції наступного релізу, 10 − результати тестування ПЗ, 11 – оцінка якості ПЗ.

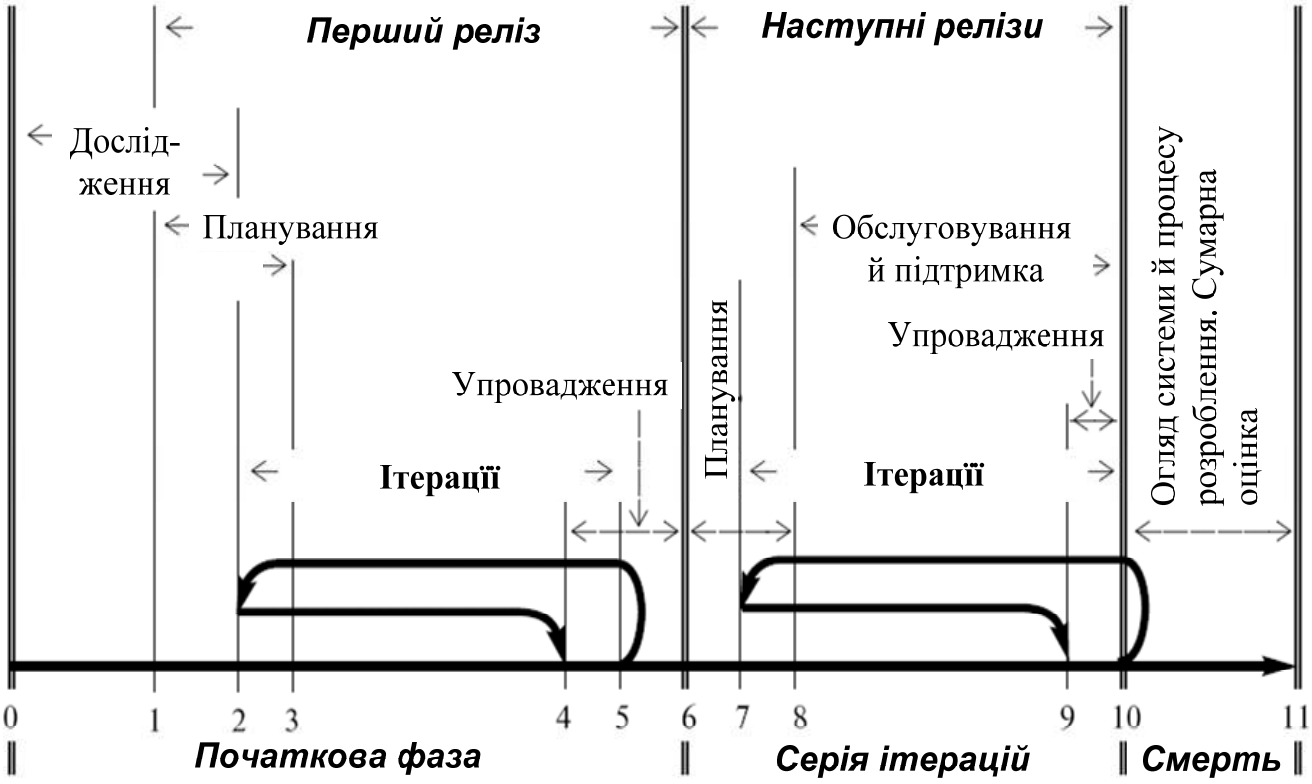


Рис. 2.4. Модель життєвого циклу в екстремальному програмуванні

Умови застосування підходу: замовник погано уявляє, що фактично потрібно для підтримки діяльності користувача, а для аналітичного дослідження немає можливості.

2.1.5. Адаптивне розроблення за Хайсмітом

Модель ASD належить до області швидких методологій, у якій увагу зосереджено на необхідності адаптивного розроблення (рис. 2.5).

Основу ASD становлять три нелінійні фази, які перекривають одна одну: обмірковування, співробітництво і навчання, що належать до кожного періоду розроблення, який завершується випуском релізу. Підкреслюється, що планування в оточенні, яке потребує адаптивності, є парадоксом, оскільки результати в цьому випадку завжди будуть непередбаченими. При звичайному плануванні відхилення від плану є помилками, які потрібно виправляти. При адаптивному розробленні відхилення ведуть до рішень, які є об'єктивно обумовленими, а тому їх слід уважати правильними. Невизначеність у настільки непередбачуваному середовищі усувається завдяки активному співробітництву розроблювачів. При цьому увага керівництва спрямована не стільки на пояснення, що саме потрібно робити, скільки на забезпечення комунікації, коли розроблювачі самі знаходять відповіді на питання, що постають перед ними. Це обумовлює необхідність підвищеної уваги до навчання, значення якого в передбачуваних методологіях часто применшується: усе розписується заздалегідь, так що потім залишається тільки додержуватися плану.

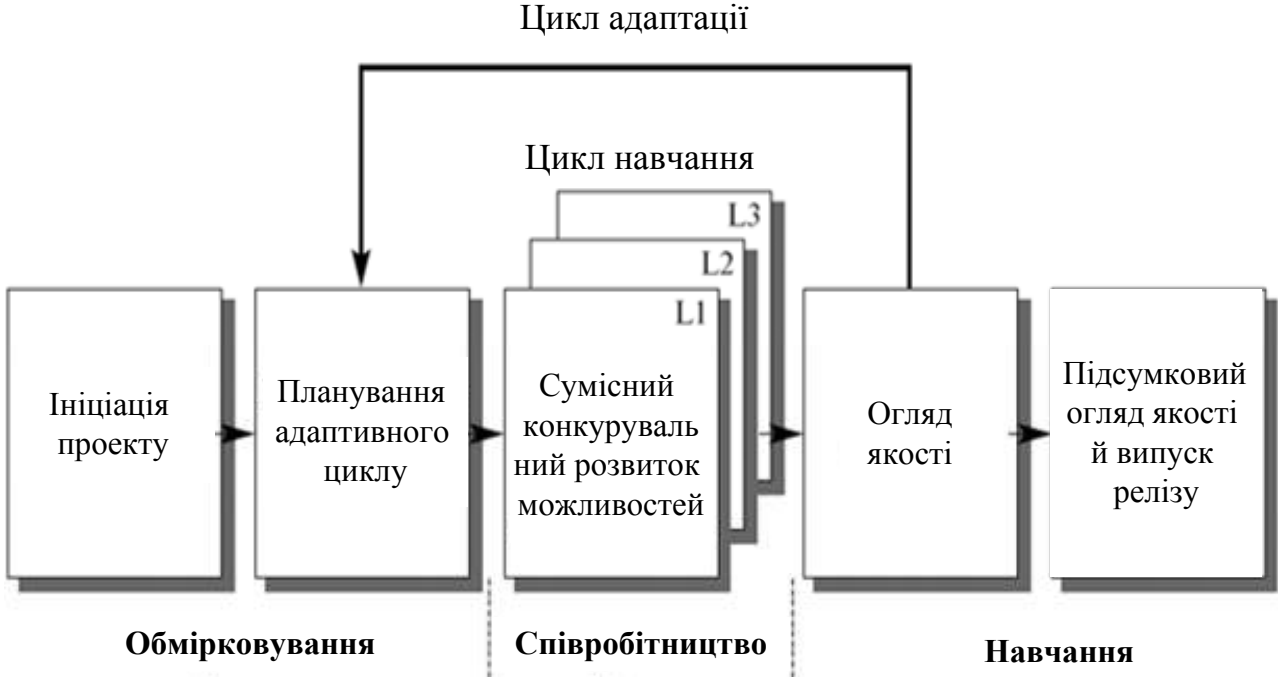


Рис. 2.5. Модель життєвого циклу ASD

З наведеної схеми не ясно, як пропонується організовувати процес розроблення ПЗ, що активно створюється, без сформованої загальної архітектурної бази. Автор ASD висвітлює тільки окремі моменти адаптивного розроблення: питання забезпечення співробітництва й навчання під час виконання проекту та ін.

ASD − це не готова методологія, а базова концепція для різних адаптивних розробок. Схема життєвого циклу не є фактором побудови конкретних методологій, які цілком можуть належати до найрізноманітніших стратегій, містити ті чи інші методики.

Підхід Дж. Хайсміта схожий ще з однією методологією швидкого розроблення, запропонованою Алістером Коуберном, − Crystal. А. Коуберн називає їх сім'єю, оскільки переконаний, що різним проектам потрібні різні методології. Він вводить таку градацію проектів: на одній осі відкладається кількість зайнятих у проекті людей, на іншій − критичність помилок. Отже, проект, у якому зайнято 40 чоловік і на якому компанія може дозволити собі втратити деяку суму, буде працювати за іншою методологією, ніж проект для шести розробників, від якого залежить існування компанії.

Таким чином, можна сказати, що ASD – це базова концепція для різних адаптивних розробок.

Контрольні запитання

1. У чому полягає основна відмінність «живих» технологій проектування ПЗ від «важких»?
2. Якими є основні характеристики, підхід до розроблення й принципи методології RUP?
3. Охарактеризуйте модель ЖЦ RUP. Що таке динамічна і статична структури моделі ЖЦ RUP?
4. Охарактеризуйте основні концепції і принципи методології MSF.
5. Які фази і віхи містить модель ЖЦ MSF?
6. Якими є особливості, переваги й недоліки екстремального програмування? Де застосовують екстремальне програмування?
7. Охарактеризуйте модель ЖЦ XP.
8. У чому полягає суть RAD-технології?
9. Якими є принципи RAD-технології?
10. У чому полягає суть ASD-технології?
11. Якими є обмеження ASD-технології?
12. Проаналізуйте відмінність найменувань фаз створення ПЗ при використанні розглянутих методологій.
13. Проаналізуйте рис. 2.1 і поясніть призначення інтенсивностей виробничих функцій.
14. Назвіть авторів розглянутих методологій.

2.2. Архітектура ПЗ, стандарти опису архітектур ПЗ

Архітектура – сукупність принципів структурування, за якими систему можна скласти з декількох більш простих систем, кожна з яких у своєму локальному контексті не залежить, але й не суперечить контексту всієї системи як єдиного цілого.

В архітектурі системи необхідно враховувати і функціональні (ділові), і нефункціональні (сервісні) вимоги. На початковому етапі розроблювач повинен визначити рівень сервісного вимірювання для кожної із сервісних вимог. Наприклад, для досягнення кращої розширюваності, розроблювач має використати модульну архітектуру з більшою кількістю об'єктів, і таким чином підвищити вимоги до пам'яті, що, зі свого боку, впливає на продуктивність. Найбільш важливими сервісними вимогами є масштабованість, зручність обслуговування, надійність, доступність, розширюваність, ефективність, керованість, захищеність

Архітектура ПЗ зазвичай містить кілька видів, які є аналогічними різним типам креслень у будівництві будинків. В онтології, установленій ANSI / IEEE 1471-2000, види є екземплярами точок зору різних проектувальників, які існують для опису архітектури заданої множини зацікавлених осіб. Приклади видів:

- функціональний / логічний вид;
- вид розроблення (development) / структурний вид;
- вид паралельності виконання / процес / потік;
- фізичний вид / вид розгортання;
- вид з огляду на дії користувача;
- вид з огляду на дані.

Хоча було розроблено кілька мов для опису архітектури програмного забезпечення, сьогодні немає згоди щодо того, який набір видів необхідно взяти за еталон. Розглянемо подання архітектури з допомогою мови UML.

Канонічні діаграми UML можна проілюструвати умовною класифікацією діаграм, наведеною на рис. 2.6.

Діаграма використання — найбільш загальне подання функціонального призначення системи.

На діаграмі використання застосовуються два типи основних сутностей: варіанти використання і дійові особи, між якими встановлюються такі основні типи відношень:

- асоціація між дійовою особою і варіантом використання;
- узагальнення між дійовими особами;
- узагальнення між варіантами використання;
- залежності (різних типів) між варіантами використання.

Рис. 2.6. Ієрархія типів діаграм

Крім того, на діаграмі використання можна застосовувати спеціальний графічний коментар – позначити межу системи (дійові особи знаходяться зовні, а варіанти використання – усередині), якщо це з якоїсь причини не є очевидним. Основні елементи нотації, які застосовують на діаграмі використання, показано на рис. 2.7.

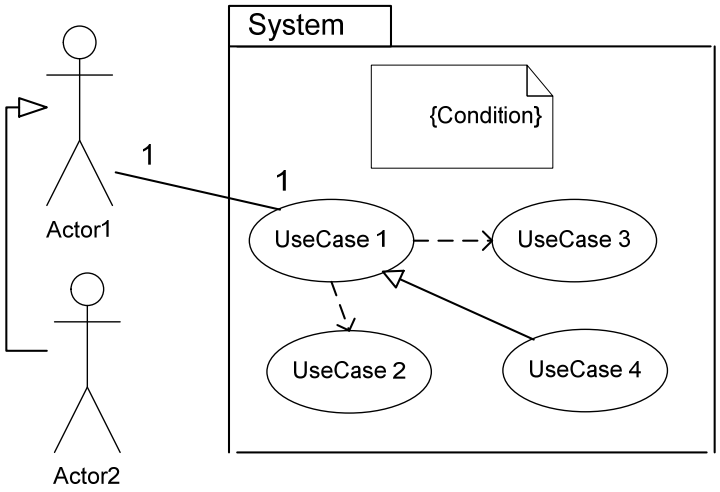


Рис. 2.7. Нотація діаграми використання

Діаграма класів – основний спосіб опису структури системи, оскільки UML є дуже об’єктно-орієнтованою мовою, класи є основним «будівельним матеріалом» системи.

На діаграмі класів застосовують один основний тип сутностей: класи (у тому числі часткові випадки класів: інтерфейси, типи, класи-асоціації та ін.), між якими встановлюються такі основні типи відношень:

− асоціація між класами (з багатьма додатковими подробицями);

− узагальнення між класами;

− залежності (різних типів) між класами й інтерфейсами.

Основні елементи нотації, що застосовуються на діаграмі класів, показано на рис. 2.8.

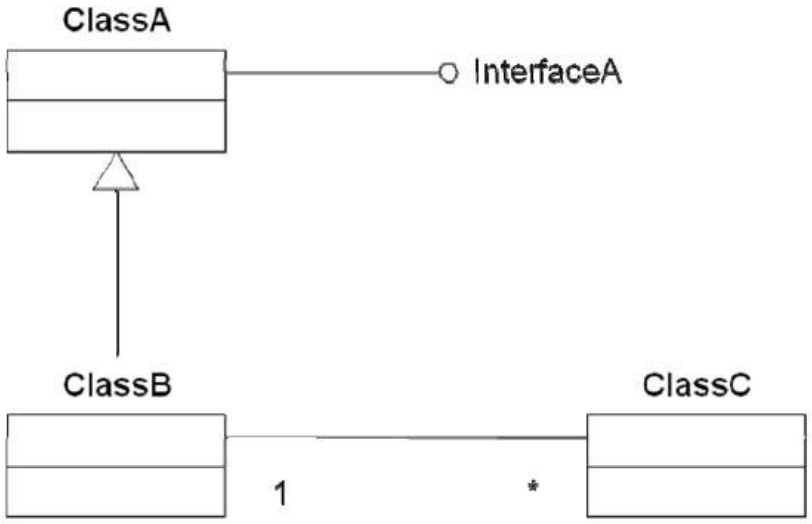


Рис. 2.8. Нотація діаграми класів

Опис класу може містити безліч різних елементів, і щоб їх не переплутати, у мові передбачено групування елементів опису класу за розділами.

Розділ імені – нарівні з обов'язковим ім'ям може містити також стереотип, кратність і список властивостей:

«стереотип» ІМ'Я {властивості} кратність.

У табл. 2.1 наведено стандартні стереотипи класів.

Таблиця 2.1

Стереотип	Опис	Стереотип	Опис
Actor	Дійова особа	Powertype	Метакласс, екземплярами якого є всі спадкоємці цього класу
Enumeration	Перелічуваний тип даних	Process, thread	Активні класи
Exception	Сигнал, що розповсюджуэться за ієрархією узагальнень	Signal	Клас, екземплярами якого є повідомлення
Implementation Class	Реалізація класу	Stereotype	Стереотип
Interface	Немає атрибутів, всі операції є абстрактними	Type (datatype)	Тип даних
Metaclass	Екземпляри є класами	Utility	Немає екземплярів - служба

Обов'язкове ім'я класу може бути виділено курсивом, у цьому випадку клас є абстрактним, тобто не може мати безпосередніх екземплярів. Якщо ім'я підкреслено, то це вже не ім'я класу, а ім'я об'єкта. Клас, а також окремі елементи його опису можуть мати задані користувачем довільні обмеження та іменовані значення.

Розділ атрибутів – у загальному випадку опис атрибута має такий синтаксис:

видимість ІМ'Я кратність : тип = початкове_значення {властивості}.

Видимість позначається або знаками (+ (відкритий), – (закритий), # (захищений)), або ключовими словами (public, protected, private). Якщо видимість не зазначено, то ніякого значення видимості за замовчуванням не мається на увазі. Підкреслення опису атрибута відповідає описувачу static у мові C++, Java.

Відкритий елемент моделі є видимим скрізь, де є видимим контейнер, який його утримує. Наприклад, відкритий атрибут класу є видимим скрізь, де є видимим сам клас. Захищений елемент моделі є видимим у своєму контейнері й у всіх контейнерах, для яких цей елемент є узагальненням. Наприклад, захищений атрибут класу є видимим у цьому класі й у всіх класах, що успадковуються. Закритий елемент моделі є видимим тільки у своєму контейнері. Наприклад, закритий атрибут класу є видимим тільки в цьому класі.

Розділ операцій містить список описів операцій класу в такому форматі:

видимість ІМ'Я (параметри) : тип {властивості}.

Як і всі основні сутності UML, клас обов'язково має ім'я, а отже, розділ імені не можна пропускати. Інші розділи можуть бути порожніми.

Розглянемо приклад ієрархії субординації в організації і взаємозв'язок класу Position з інтерфейсами та іншими класами

Кожна посада може мати два значення. З одного боку, посада може розглядатися як посада начальника (chief), і в цьому випадку вона надає інтерфейс IChief, який має операцію petition (начальникові можна подати службову записку). З іншого боку, посада може розглядатися як посада підлеглого (subordinate), і в цьому випадку вона надає інтерфейс ISubordinate, що має операцію report (від підлеглого можна запросити звіт). У начальника може бути довільна кількість підлеглих, у тому числі й нуль, у підлеглого може бути не більше одного начальника.

На рис. 2.9 зображено асоціацію класу Position із самим собою, а також використано специфікацію інтерфейсів і відношеня реалізації.

Клас Department для реалізації операцій, пов'язаних з рухом кадрів, використовує операції класу Position, що дають змогу обіймати і звільняти посаду, а інші операції класу Position для класу Department не потрібні. Для цього можна визначити відповідний інтерфейс IPosition і зв'язати його відношеннями з цими класами.

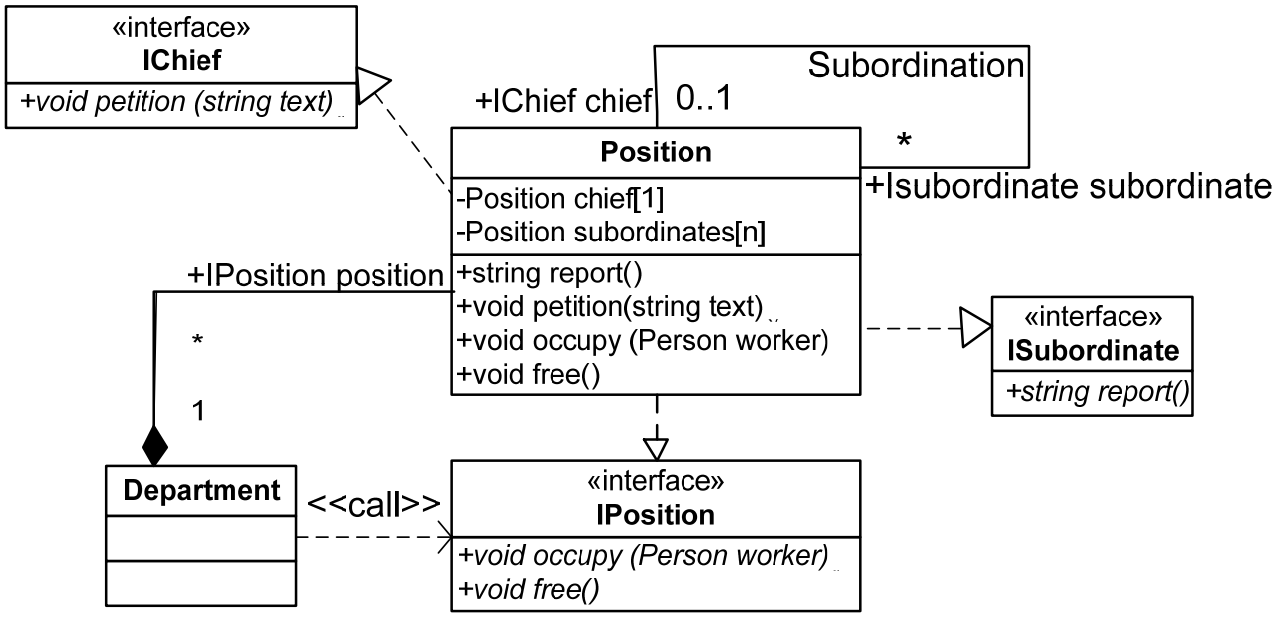


Рис. 2.9. Асоціація класу Position

Діаграма об'єктів − це окремий випадок діаграми класів. Діаграми об'єктів мають допоміжний характер – по суті це приклади (можна сказати, дампи пам'яті) об'єктів і зв'язків між ними в деякий конкретний момент функціонування системи.

На діаграмі об'єктів застосовують один основний тип сутностей: об'єкти (екземпляри класів), між якими наведено конкретні зв'язки (екземпляри асоціацій).

Основні елементи нотації, що застосовуються на діаграмі об'єктів, показано на рис. 2.10.

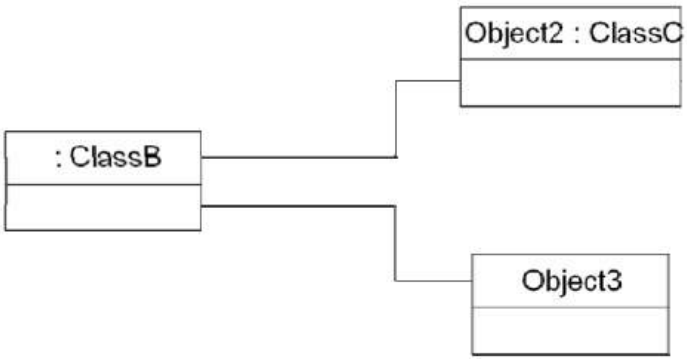


Рис. 2.10. Нотація діаграми об'єктів

Діаграма станів – це основний спосіб детального опису поводження в UML. По суті діаграми станів являють собою граф станів (state) і переходів кінцевого автомата, навантажений множиною додаткових деталей і подробиць.

На діаграмі станів застосовують один основний тип сутностей – стани, і один тип відносин – переходи, але і для тих, і для інших визначено багато різновидів, спеціальних випадків і додаткових позначень.

На рис. 2.11 показано елементи нотації, що застосовуються на діаграмі станів. Чорне зафарбоване коло означає початковий стан, обведене зафарбоване коло – закінчення розгляду станів.

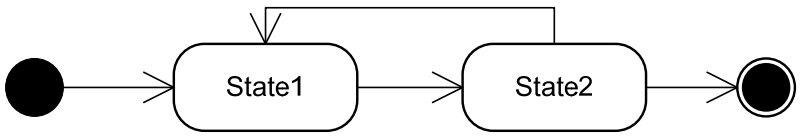


Рис. 2.11. Нотація діаграми станів

Складений стан – це стан, який складається з машини станів. Якщо вкладено тільки одну машину, то стан називають послідовним, якщо декілька – то паралельним. Глибина вкладаня в UML є необмеженою, тобто стани вкладеної машини станів можуть бути складеними.

Семантику складених станів і переходів можна пояснити з допомогою еквівалентних конструкцій, що не використовують складених станів. На рис. 2.12 стан A – складений, а стани B і C – будь-які, тобто прості або складені.

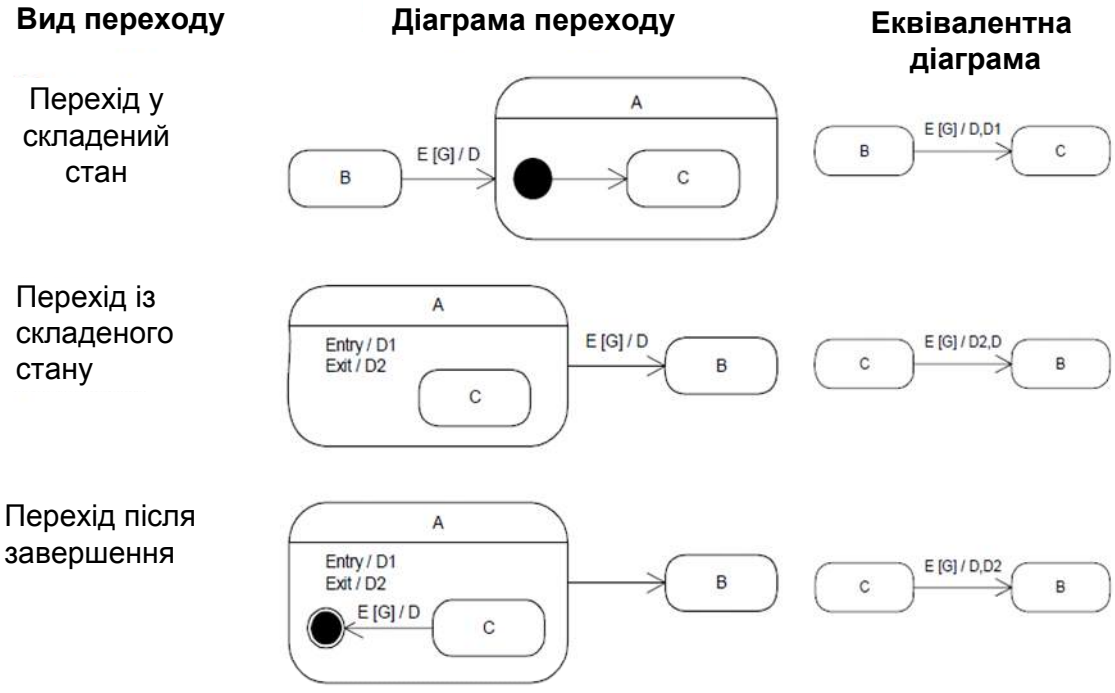


Рис. 2.12. Переходи між складеними станами

Історичний стан може використовуватися у вкладеній машині станів усередині складеного стану. При першому запуску машини станів історичний стан означає те ж саме, що й початковий, тобто такий, у якому машина

перебувала на початку роботи. Якщо в машині станів використовується історичний стан, то при виході зі складеного стану запам'ятовується той стан, у якому перебувала вкладена машина при виході. При повторному вході в складений стан як поточний відновлюється той стан, у якому машина перебувала при виході. Історичний стан змушує машину пам'ятати, у якому стані її перервали минулого разу, і «продовжувати те, що було почато». Позначається такий стан буквою H, обведеною кружком.

Діаграма діяльності — це фактично блок-схема алгоритму, у якій модернізовано позначення, а семантику узгоджено із сучасним об’єктно-орієнтованим підходом, що дало змогу органічно додати діаграми діяльності до методу UML.

На діаграмі діяльності застосовують один основний тип сутності – діяльність, і один тип відношеня – перехід (передача керування), а також використовують графічні позначення (розвилки, злиття й розгалуження).

Крім потоку керування на діаграмі діяльності можна показати й потік даних, використовуючи такі сутності, як об'єкт (у певному стані) і відповідну залежність. Можна також застосувати спеціальний графічний коментар – так звані доріжки, які підкреслюють, що деякі діяльності відрізняються одна від одної, наприклад виконуються в різних місцях. Основні елементи нотації, що застосовуються на діаграмі діяльності, показано на рис. 2.13.

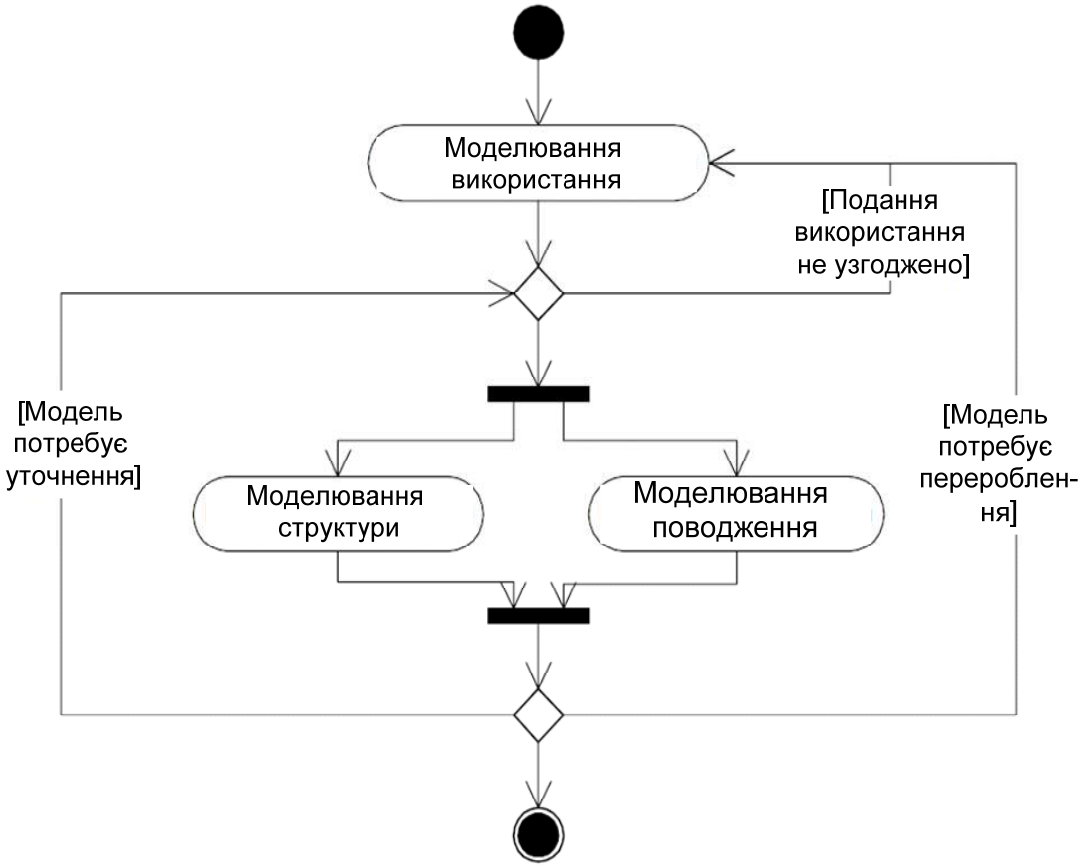


Рис. 2.13. Нотація діаграми діяльності

Діаграма послідовності – це спосіб описати поводження системи «на прикладах». Фактично діаграма послідовності – це запис протоколу конкретного сеансу роботи системи (або фрагмента такого протоколу). В об’єктно-орієнтованому програмуванні найбільш істотним під час виконання є посиланя повідомлень об'єктами, що взаємодіють. Саме послідовність посиланя повідомлень відображається на цій діаграмі, що й дало її назву. На діаграмі послідовності застосовують один основний тип сутності – об'єкти (екземпляри взаємодійних класів і дійових осіб) і один тип відношення – повідомлення, якими обмінюються взаємодійні об'єкти. Передбачено кілька типів повідомлень, які в графічній нотації різняться видом стрілки, що відповідає певному відношенню.

Важливим аспектом діаграми послідовності є явне відображення перебігу часу. На відміну від усіх інших типів на діаграмі послідовності має значення не тільки наявність графічних зв'язків між елементами, але й взаємне положення елементів на діаграмі. Уявимо, що є (невидима) вісь часу, за замовчуванням спрямована зверху вниз, і те повідомлення, яке було відправлено пізніше, буде розташовано нижче.

Вісь часу може бути горизонтальною, у цьому випадку вважається, що час змінюється в напрямку зліва направо.

На рис. 2.14 показано основні елементи нотації, які використовуються на діаграмі послідовності. Для позначення взаємодійних об'єктів застосовується стандартна нотація — прямокутник з підкресленим ім'ям об'єкта. Пунктирну лінію, що виходить з об'єкта, називають ***лінією життя***. Це не позначення відношення в моделі, а графічний коментар, призначений направити погляд читача діаграми в правильному напрямку.

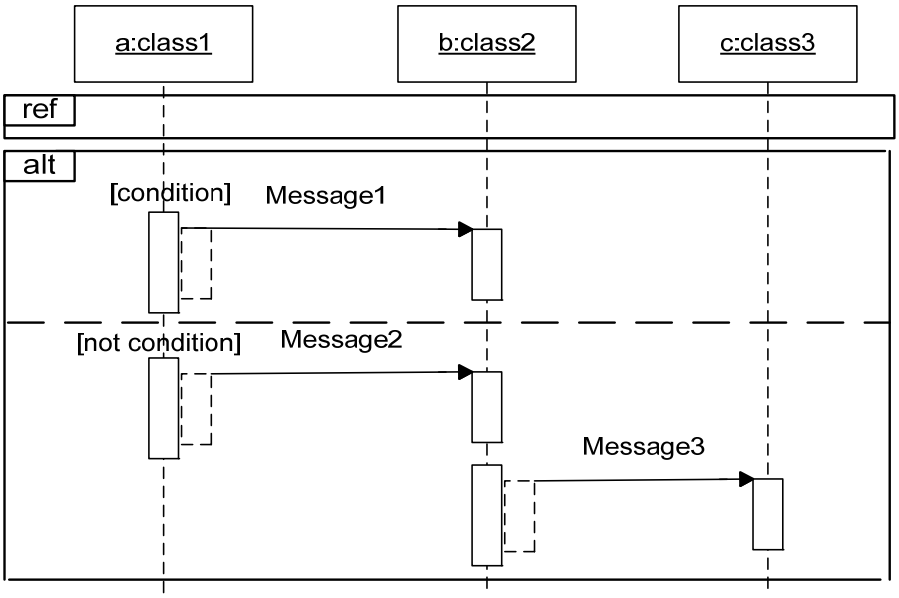


Рис. 2.14. Нотація діаграми послідовності (UML 2.0)

Фігури у вигляді вузьких смужок, накладених на лінію життя, також не є зображеннями сутностей, що моделюються. Це графічний коментар, який показує відрізки часу, під час яких об'єкт має керування (як кажуть, має місце активація об'єкта). Створення об'єкта під час взаємодії визначається тим, що значок об'єкта розташовують нижче (тобто об'єкт виникає пізніше). Видалення об'єкта позначається великим косим хрестом, і лінія життя припиняється. Типи передання повідомлень між об'єктами наведено в табл. 2.2.

Таблиця 2.2

Вид стрілки	Тип передання повідомлень
	Вкладений потік керування. Відправник може відправити наступне повідомлення тільки після того, як завершиться виконання всіх дій, ініційованих попереднім повідомленням. Зазвичай застосовується під час виклику операцій
	Простий потік керування. Керування передається від відправника повідомлення до одержувача (можливо, безповоротно). Зазвичай застосовується під час моделювання поводження на рівні дійових осіб і варіантів використання
	Асинхронний потік керування. Повідомлення асинхронно передається від відправника до одержувача, при цьому у відправника зберігається свій потік керування, що не залежить від потоку керування одержувача. Зазвичай застосовується при відправленні сигналів
	Повернення керування. Повернення керування після виконання всіх дій, ініційованих переданням повідомлення із вкладеним потоком керування. Цей тип передання повідомлення можна не відображати на діаграмі, оскільки його припускають за замовчуванням під час виклику операцій
Не визначається	Допускається використання при моделюванні інших, не обумовлених в UML типів передання керування, наприклад передання керування після закінчення часу

Діаграма кооперації (діаграма комунікації) є семантично еквівалентною діаграмі послідовності. Фактично це такий самий опис послідовності обміну повідомленнями взаємодійних об'єктів, тільки виражений іншими графічними засобами. Крім того, більшість інструментів може автоматично перетворювати діаграми послідовності на діаграми кооперації і навпаки. Таке перетворення є взаємооднозначним.

Таким чином, на діаграмі кооперації також застосовують один основний тип сутностей – об'єкт (екземпляри взаємодійних класів і дійових осіб) і один тип відношення – повідомлення, якими обмінюються взаємодійні об'єкти. Однак тут акцент робиться не на часі, а на зв'язках між конкретни-

ми об'єктами.

На рис. 2.15 показано основні елементи нотації, які застосовуються на діаграмі кооперації. Для позначення взаємодійних об'єктів застосовується стандартна нотація – прямокутник з підкресленим ім'ям об'єкта. Взаємне положення об'єктів на діаграмі кооперації не має значення – важливими є тільки зв'язки (екземпляри асоціацій), уздовж яких передаються повідомлення. Для відображення впорядкованості повідомлень у часі застосовується ієрархічна десяткова нумерація.

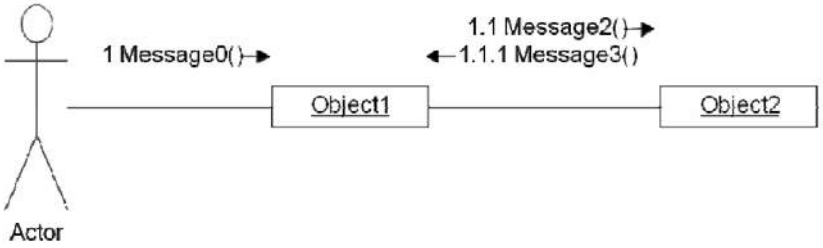


Рис. 2.15. Нотація діаграми кооперації

Діаграма компонентів − це фактично список артефактів, з яких складається система, що моделюється, із зазначенням деяких відношень між артефактами. Найбільш поширеним типом артефактів програмних систем є програми. Таким чином, на діаграмі компонентів основний тип сутностей – це компонент (як здійсненні модулі, так і інші артефакти), а також інтерфейси (щоб указувати взаємозв'язок між компонентами) і об'єкти (що входять до складу компонентів). На діаграмі компонентів застосовуються такі відношення:

– реалізації між компонентами й інтерфейсами (компонент реалізує інтерфейс);

– залежності між компонентами й інтерфейсами (компонент використовує інтерфейс);

– залежності між об'єктами й компонентами (об'єкт належить компоненту).

На рис. 2.16 показано основні елементи нотації, що застосовуються на діаграмі компонентів. Відношення залежності, що відповідає належності (наприклад, об'єкта компоненту), часто зображують, поміщаючи фігуру однієї сутності усередину фігури іншої.

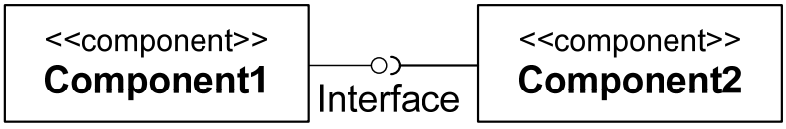
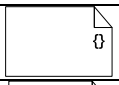
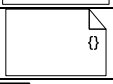
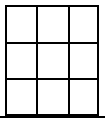


Рис. 2.16. Нотація діаграми компонентів

Для того щоб якось відображати таку різноманітність типів артефактів, що є компонентами, в UML передбачено стандартні стереотипи компонентів (табл. 2.3).

Таблиця 2.3

Стереотип	Опис	Позначення
Executable	Виконувана програма будь-якого виду за замовчуванням, якщо ніякого стереотипу не зазначено	
Document	Документ будь-якого типу, наприклад файл із документацією до програми	
File	Файл із вихідним кодом програми або з даними, які використовує програма	
Library	Статична або динамічна бібліотека	
Table	Таблиця бази даних	

Діаграма розміщення дещо відрізняється від діаграми компонентів. Фактично нарівні з відображенням складу і зв'язків компонентів тут показано, як фізично розміщено компоненти на обчислювальних ресурсах під час виконання. Таким чином, на діаграмі розміщення порівняно з діаграмою компонентів додається один тип сутностей – вузол Node (може бути як класифікатор, що описує тип вузла, так і конкретний екземпляр), а також відношення асоціації між вузлами, що є відображенням їх фізичного зв'язку під час виконання.

На рис. 2.17 показано основні елементи нотації, які використовуються на діаграмі розміщення. Включення фігури однієї сутності усередину фігури іншої тут застосовується особливо часто.

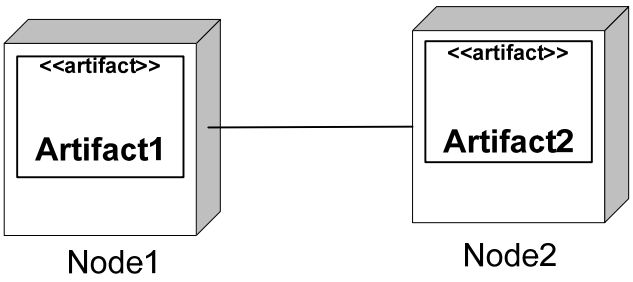


Рис. 2.17. Нотація діаграми розміщення

Розглянемо діаграми розміщення системних архітектур. Системна архітектура має два види: файл-сервер і клієнт-сервер.

Архітектура «Файл-сервер». Історично – це перша архітектура ін-

формаційних систем. Виконувані модулі й дані розміщуються в окремих файлах операційної системи (рис. 2.18). Доступ до даних здійснюється шляхом зазначення шляху (path) і використання файлових операцій («Відкрити», «Прочитати», «Записати»). Для зберігання даних використовується виділений сервер (окремий комп'ютер), який і є файловим сервером. Виконувані модулі зберігаються або на робочих станціях, або на файловому сервері. В останньому випадку спрощується процедура їх адміністрування, але при цьому зростають вимоги до надійності мережі.

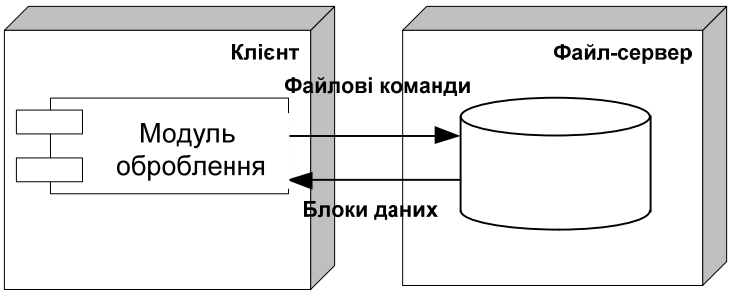


Рис. 2.18. Архітектура «Файл-сервер»

Архітектура «Клієнт-сервер». Суть цієї архітектури полягає в тому, що клієнт (виконуваний модуль) запитує ті чи інші сервіси відповідно до певного протоколу обміну даними. При цьому немає необхідності у використанні прямих шляхів операційної системи: клієнт їх «не знає», йому «відомі» лише ім'я джерела даних та інші спеціальні відомості, які використовуються для авторизації клієнта на сервері. Сервер, що фізично може знаходитися на тому самому комп'ютері, а може – на іншому боці земної кулі, обробляє запит клієнта і, зробивши відповідні маніпуляції з даними, передає клієнтові запитувані дані. В архітектурі «Клієнт-сервер» існують два основних «діалекти»: «тонкий» і «товстий» клієнти.

У системах на основі «тонкого» клієнта використовується потужний сервер баз даних. Це – високопродуктивний комп'ютер і бібліотека так званих збережених процедур, які дають змогу робити обчислення, що реалізують основну логіку оброблення даних, безпосередньо на сервері. Клієнтський додаток відповідно ставить невисокі вимоги до апаратного забезпечення робочої станції.

Основна перевага таких систем – відносна дешевина клієнтських станцій.

Системи з «товстим» клієнтом, навпаки, реалізують основну логіку оброблення на клієнті, а сервер являє собою в чистому виді сервер баз даних, що забезпечує виконання тільки стандартизованих запитів на маніпуляцію з даними (зазвичай читання, запис, модифікація даних у таблицях реляційної бази даних). У системах такого класу вимоги до робочої станції є вищими, а до сервера – нижчими.

Перевага архітектури – можливість перенесення серверного компонента на сервери різних виробників: усі промислові сервери баз даних реляційного типу підтримують роботу зі стандартизованою мовою маніпулю-

вання даними SQL, але внутрішня вбудована мова оброблення даних, необхідна для реалізації логіки оброблення на сервері, у кожного із серверів своя.

Розглянемо приклад тришарової архітектури (рис. 2.19).

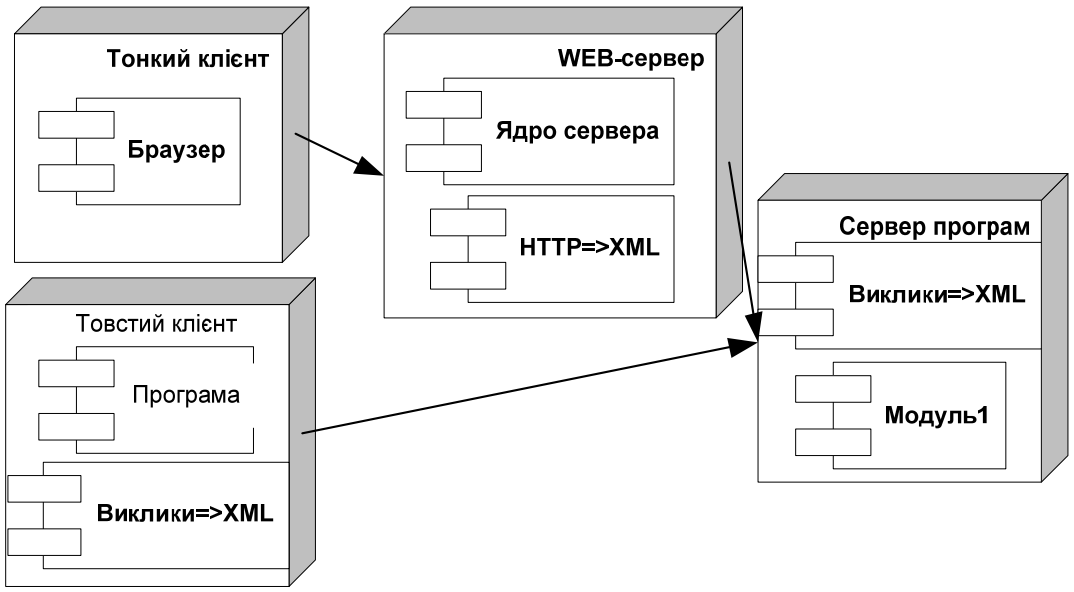


Рис 2.19. Приклад архітектури «Клієнт-сервер»

Модулі, що реалізують змістовну функціональність (бізнес-логіку), реалізуються на сервері програм. На веб-сервері розташовано дизайн сторінок, але немає ніяких програм. При надходженні HTTP-запиту до якого-небудь модуля виконується вхідне перетворення на XML-подання параметрів, потім запит передається серверу програм, результат виконання функції перетворюється на HTML-код результату з використанням шаблонів дизайну.

Контрольні запитання

1. Побудуйте діаграму послідовності дій для об'єктів будь-яких запропонованих пакетів. Якими повідомленнями обмінюються об'єкти? Яку інформацію програміст отримає, аналізуючи цю діаграму?
2. Яку діаграму використовують при уточненні взаємодії об'єктів?
3. Перелічіть основні компоненти класів. Охарактеризуйте їх.
4. У яких випадках використовують діаграми станів об'єкта? Побудуйте діаграму станів для будь-якого керувального об'єкта.
5. Побудуйте уточнену діаграму класів за результатами дослідження взаємодії об'єктів. Яка ще інформація необхідна для реалізації цих класів?
6. Що розуміють під діаграмою компонентів? Яка інформація в ній міститься? У яких випадках доцільно будувати діаграми компонентів?
7. Яку інформацію містить діаграма розміщення? У яких випадках доцільно використовувати такий тип діаграми?

2.3. Патерни проектування ПЗ

Патерни (шаблони, зразки) проектування (Design Patterns) – це форма обміну досвідом і вдалими проектними рішеннями. Патерни різняться ступенем деталізації і рівнем абстракції. Пропонується така загальна класифікація патернів за категоріями їх застосування: архітектурні патерни, патерни проектування, аналізу, тестування і реалізації.

Архітектурні патерни (Architectural Patterns) – велика кількість попередньо визначених підсистем зі специфікацією їх відповідальності, правил і базових принципів установлення відношень між ними.

Архітектурні патерни призначено для специфікації фундаментальних схем структуризації програмних систем. Найбільш відомими є патерни GRASP (General Responsibility Assignment Software Pattern), які належать до рівня систем і підсистем, але не до рівня класів. Зазвичай формулюються в узагальненій формі, не залежать від області програми і в них використовується звичайна термінологія. Патерни цієї категорії систематизував та описав К. Ларман.

Патерни проектування (Design Patterns) – спеціальні схеми для уточнення структури підсистем або компонентів програмної системи й відношень між ними.

Патерни проектування описують загальну структуру взаємодії елементів програмної системи, які реалізують вихідну проблему проектування в конкретному контексті. Найбільш відомими є патерни GoF (Gang of Four), названі на честь Є. Гами, Р. Хелма, Р. Джонсона й Дж. Уліссидеса, які систематизували й описали їх. Патерни GoF містять 23 патерни і не залежать від мови реалізації, але їх реалізація залежить від області програми.

Патерни аналізу (Analysis Patterns) – спеціальні схеми для подання загальної організації процесу моделювання.

Патерни аналізу належать до однієї або декількох предметних областей, їх застосовують в термінах предметної області. Найбільш відомими є патерни бізнес-моделювання ARIS (Architecture of Integrated Information Systems), які характеризують абстрактний рівень подання бізнес-процесів. Патерни аналізу конкретизуються в типових моделях для аналітичного оцінювання або імітаційного моделювання бізнес-процесів.

Патерни тестування (Test Patterns) – спеціальні схеми для подання загальної організації процесу тестування програмних систем.

До цієї категорії належать такі патерни, як тестування чорного й білого ящиків, окремих класів, системи. Патерни цієї категорії систематизував та описав М. Гранд. Деякі з них реалізовано в інструментальних засобах, найбільш відомими з яких є IBM Test Studio. У зв'язку із цим патерни тестування іноді називають стратегіями, або схемами тестування.

Патерни реалізації (Implementation Patterns) – сукупність компонентів та інших елементів реалізації, які використовують у структурі моделі

при написанні програмного коду.

Цю категорію патернів поділяють на такі підкатегорії: організації програмного коду, оптимізації програмного коду, стійкості коду, розроблення графічного інтерфейсу користувача та ін. Патерни цієї категорії описано в роботах М. Гранда, К. Бека, Дж. Тидвелла та ін. Деякі з них реалізовано в популярних інтегрованих середовищах програмування у вигляді шаблонів створюваних проектів. У цьому випадку вибір шаблону програмного забезпечення дає змогу одержати деяку заготовку програмного коду.

Розглянемо структуру, приклад і класифікацію патернів проектування.

Опис патерна складається із чотирьох основних елементів зразка.

Ім'я. Посилаючись на ім'я зразка, можна стисло описати проблему проектування, її вирішення та наслідки використання ПЗ. Це дає змогу проектувати на більш високому рівні абстракції. Словник загальновідомих імен зразків дає змогу ефективно вести обговорення з колегами, лаконічно документувати прийняті архітектурні рішення. Добір імені – одне з найважливіших завдань при складанні опису зразка.

Завдання – опис контексту застосування зразка проектування, тобто конкретної проблеми проектування й переліку умов, при виконанні яких має сенс застосовувати певний зразок.

Вирішення – опис елементів проектування, відношень між ними, функції кожного елемента, а також абстрактний опис завдання проектування і його узагальнене вирішення.

Результати. Тут описується вплив програми на ступінь ефективності, гнучкості, розширюваності й кросплатформеності системи.

Розглянемо докладніше поняття патерна проектування стосовно UML. Синтаксично в UML патерн – це параметрична кооперація класів (тобто шаблон кооперації).

Застосовуваний зразок зображується у вигляді пунктирного овалу, усередині якого написано ім'я кооперації. Цей овал з'єднується пунктирними лініями з класами, які є фактичними аргументами, причому на лінії вказується ім'я ролі, яку клас відіграє в застосовуваній кооперації.

Як приклад розглянемо класичний зразок проектування, описаного під ім'ям **Observer** (в інших джерелах – **Publish-Subscribe)**.

Завдання. Поводження деяких об'єктів системи (підписників-екземплярів класу Subscriber) має залежати від змінення стану (події) іншого об'єкта (видавця-екземпляра класу Publisher). Однак видавець не повинен прямо взаємодіяти з підписниками.

Вирішення. Створимо службу повідомлення про події, для того щоб видавець міг опосередковано повідомляти підписників про настання події. Для цього введемо єдиний об'єкт класу EventManager, що реалізує цю службу. Клас EventManager має метод subscribe, викликаючи який, підписник підписується на повідомленні про настання події, і метод signalEvent, з

допомогою якого видавець повідомляє про настання події. Викликаючи метод signalEvent, об'єкт EventManager посилає повідомлення про подію всім підписникам і викликає метод notify, переданий як параметр під час підписки. На рис. 2.20 наведено діаграму кооперації, де описано цей шаблон взаємодії.

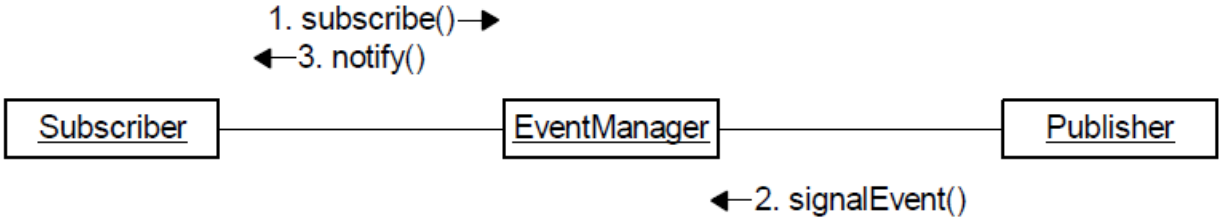


Рис. 2.20. Діаграма кооперації для зразка проектування Publish-Subscribe

Результат. Клас Publisher не залежить від класу Subscriber. Можливі різні модифікації зразка: якщо необхідно одержувати повідомлення про різні події, то потрібно додати відповідний параметр (наприклад, ім'я події); для збільшення ефективності можна обов'язки ведення служби повідомлень про події передоручити прямо класу Publisher, але в цьому випадку зменшується гнучкість, оскільки реалізовувати цю службу доведеться в кожному класі-видавці.

Розглянемо застосування зразка на прикладі інформаційної системи відділу кадрів за умови, що моделювання ведеться засобами UML (рис. 2.21).

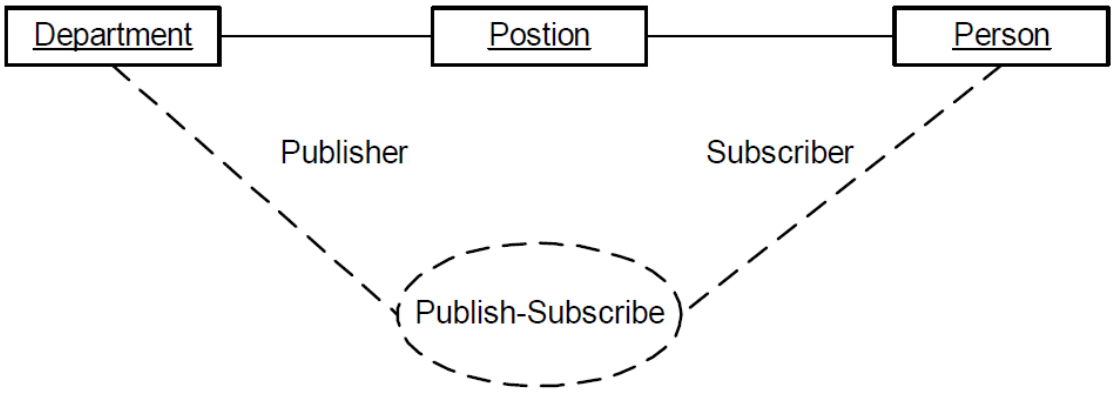


Рис. 2.21. Застосування зразка проектування

Нехай потрібно повідомляти співробітників про змінення стану підрозділу. У цьому випадку на діаграмі кооперації можна показати, що до класів Department і Person слід застосувати зразок проектування Publish-Subscribe, причому клас Department відіграє роль Publisher, а клас Person – роль Subscriber.

З допомогою наведеної діаграми знаходимо кооперацію, у якій містяться набагато більше елементів, ніж зображено на діаграмі. Зокрема, мається на увазі, що в моделі визначено клас EventManager (хоча його немає на діаграмі), між класами визначено відповідні асоціації для виклику методів, а класи мають необхідні методи для застосування зразка і т. д.

Будь-який розроблювач добре розуміє, що саме, чому і як потрібно запрограмувати в інформаційній системі відділу кадрів, щоб забезпечити необхідне оповіщення об'єктів класу Person про змінення стану об'єкта класу Department. Однак було б надмірним сподіватися на те, що інструменти моделювання розуміють зразки проектування так само добре, як люди. Цілком імовірно, що у використовуваному інструменті навіть не знайдеться такої фігури, як пунктирний овал для позначення зразка, але з тією самою імовірністю знайдеться бібліотека готових зразків. Зазвичай інструменти дають змогу додати зразок до моделі (у вигляді заготовки діаграми кооперації) і вручну зв'язати (ототожнити або перевизначити) елементи зразка з елементами моделі.

Далі розглянемо класифікацію патернів проектування (рис. 2.22). Існують кілька типів патернів проектування, кожний з яких призначено для вирішення свого кола завдань:

− твірні (Creational) – застосовують для створення нових об'єктів у системі;

− структурні (Structural) – вирішують завдання компонування системи на основі класів і об'єктів;

− поведінки (Behavioral) – застосовують для розподілу обов'язків між об'єктами в системі.

Creational						**Structural**	
107 **FM** Factory Method							139 **A** Adapter
117 **PT** Prototype	127 **S** Singleton	**Behavioral**			223 **CR** Chain of Responsibility	163 **CP** Composite	175 **D** Decorator
87 **AF** Abstract Factory	325 **TM** Template Method	233 **CD** Command	273 **MD** Mediator	293 **O** Observer	243 **IN** Interpreter	207 **PX** Proxy	185 **FA** Facade
97 **BU** Boider	315 **SR** Strategy	283 **MM** Memento	305 **ST** State	257 **IT** Iterator	331 **V** Visitor	195 **FL** Flyweight	151 **BR** Bridge

Рис. 2.22. Класифікація патернів проектування

Зведену інформацію зі всіма патернами (шаблонами) об’єктно-орієнтованого проектування наведено в табл. 2.4.

Таблиця 2.4

Назва	Переклад	Тип	Стислий опис
Abstarct Factory	Абстрактна фабрика	Твірний	Створює сім'ю взаємозалежних об'єктів
Adapter	Адаптер	Структурний	Перетворює інтерфейс існуючого класу на такий, що є придатним для використання
Bridge	Міст	Структурний	Робить абстракцію й реалізацію незалежними
Builder	Будівельник	Твірний	Поетапне створення складного об'єкта
Chain of Responsibi-lity	Ланцюжок обов'язків	Поводження	Надає спосіб передання запиту по ланцюжку одержувачів
Command	Команда	Поводження	Інкапсулює запит у вигляді об'єкта
Composite	Компоновник	Структурний	Групує схожі об'єкти в дерево подібні структури
Decorator	Декоратор	Структурний	Динамічно додає об'єкту нову функціональність
Facade	Фасад	Структурний	Надає уніфікований інтерфейс замість набору інтерфейсів деякої системи
Factory Method	Фабричний метод	Твірний	Визначає інтерфейс для створення об'єкта, при цьому його тип визначається підкласами
Flyweight	Пристосованець	Структурний	Використовує поділ для підтримки багатьох дрібних об'єктів
Interpreter	Інтерпретатор	Поводження	Визначає граматику мови й інтерпретатор, що використовує цю граматику
Iterator	Ітератор	Поводження	Надає механізм обходу елементів колекції
Mediator	Посередник	Поводження	Інкапсулює взаємодію між кількома об'єктами в об'єкті-посереднику
Memento	Хоронитель	Поводження	Зберігає й відновлює стан об'єкта
Object Pool	Пул об'єктів	Твірний	Створює «витратні» об'єкти шляхом їх багаторазового використання

Закінчення табл. 2.4

Назва	Переклад	Тип	Стислий опис
Observer	Спостерігач	Поводження	При зміненні об'єкта сповіщає всі залежні об'єкти для їх відновлення
Prototype	Прототип	Твірний	Створює об'єкти на основі прототипів
Proxy	Заступник	Структурний	Замінює інший об'єкт для контролю доступу до нього
Singleton	Одинак	Твірний	Створює єдиний екземпляр деякого класу й надає до нього доступ
State	Стан	Поводження	Змінює поводження об'єкта при зміненні його стану
Strategy	Стратегія	Поводження	Переносить алгоритми до окремої ієрархії класів, роблячи їх взаємозамінними
Template Method	Шаблонний метод	Поводження	Визначає кроки алгоритму, даючи змогу підкласам змінити деякі з них
Visitor	Відвідувач	Поводження	Визначає нову операцію в класі без його змінення

Контрольні запитання

1. Що таке патерни проектування? Назвіть їх призначення й загальні характеристики.
2. Як описуються патерни проектування?
3. Для чого призначено твірні патерни?
4. Призначення і випадки застосування патерна «абстрактна фабрика».
5. Назвіть переваги й недоліки патерна «абстрактна фабрика».
6. Яким чином можна реалізувати патерн «абстрактна фабрика»?
7. Для чого призначено патерн «прототип»?
8. У яких випадках застосовується патерн «прототип»?
9. Назвіть основні відмінності патерна «прототип» від інших твірних патернів.
10. Якими є особливості реалізації прототипу?
11. Проаналізуйте структуру патерна Abstract Factory. Наведіть приклад його застосування.
12. Назвіть відомих авторів патернів. До якого класу вони належать?
13. Який патерн використовується для реалізації служби повідомлення відділу кадрів підприємства?

2.4. Засоби автоматизації розроблення програмних продуктів

Програмний продукт є детальним і закінченим описом архітектури й алгоритмів за допомогою обраної мови програмування. Виконавцем ПЗ є комп'ютер. Для виконання комп'ютером ПЗ його необхідно подати у машинному коді – послідовності чисел, які процесор розуміє. Написати ПЗ у машинних кодах вручну досить складно, тому сьогодні майже всі ПЗ створюються з допомогою мов програмування, які за своїми синтаксисом і семантикою наближено до людської мови. Це зменшує трудомісткість програмування. Однак, текст ПЗ, записаний з допомогою мови програмування, необхідно перетворити на машинний код. Ця операція виконується автоматично з допомогою спеціальної службової ПЗ, яку називають транслятором.

Транслятори поділяються на два типи: інтерпретатори й компілятори**.**

Інтерпретатори перетворюють текст ПЗ на машинний код і виконують почергово оператори (команди) програми. Якщо команда повторюється, то інтерпретатор розглядає її так само, як і першу (нову).

Компілятор перетворює вихідний текст програми цілком на машинний код. Тому перевага компіляторів – швидкодія і автономність отриманих програм, а інтерпретаторів – їх компактність, можливість зупинити виконання програми в будь-який момент, виконати різні перетворення даних і продовжити роботу програми.

У загальному випадку для створення програм потрібно мати такі компоненти:

− текстовий редактор – для набору початкового тексту програми;

− компілятор та (або) інтерпретатор – для перетворення тексту програми на машинний код;

− редактор зв'язків (компоновник, лінкер) – для збору декількох компільованих модулів в одну програму;

− бібліотеки функцій – для введення стандартних функцій до програми.

Сучасні системи програмування містять усі зазначені компоненти і мають назву **інтегрованих середовищ розроблення (Integrated Development Environment, IDE).**

Microsoft Visual Studio (MVS) – лінійка продуктів компанії Майкрософт, що містять інтегроване середовище розроблення програмного забезпечення та деякі інші інструментальні засоби.

У Visual Studio реалізовано контейнери: рішення й проекти, щоб зробити можливим використання в IDE всього діапазону засобів, конструкторів, шаблонів і параметрів. Visual Studio також надає папки рішень для того, щоб структурувати зв'язані проекти за групами і потім виконувати дії над ними.

Проект – це група файлів і настройок, з яких збирається остаточна програма або вихідні файли.

Проект містить набір файлів джерел і метаданих, наприклад посилання на компонент і інструкції будування (рис. 2.23). Зазвичай при будуванні проектів створюється один або кілька вихідних файлів. Рішення містить один або декілька проектів, а також файли і метадані, необхідні для опису рішення в цілому:

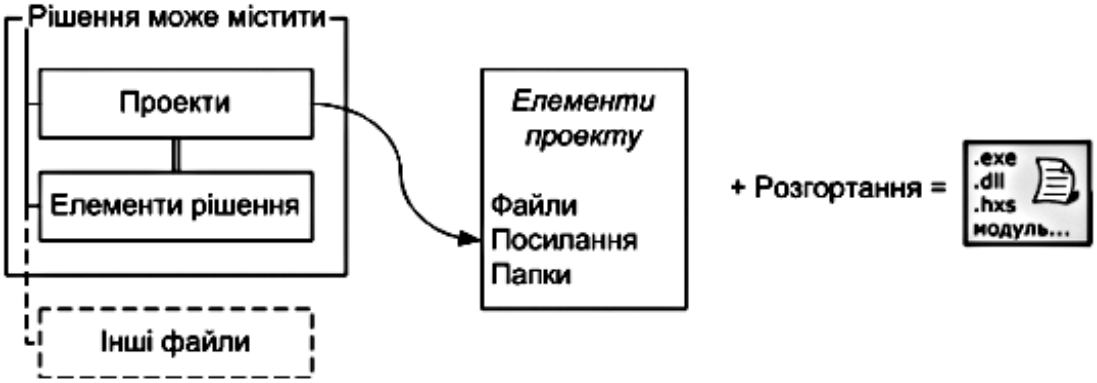


Рис. 2.23. Формування рішення

Щоб допомогти користувачам організовувати й виконувати стандартні завдання із застосуванням розроблюванних елементів, проекти Visual Studio використовуються як контейнери в межах рішення. Це дає змогу логічно керувати, будувати й відлагоджувати елементи, що утворюють програму. На виході проект зазвичай являє собою виконувану програму (EXE), файл бібліотеки динамічного компонування (DLL) або модуль.

Проект може бути простим або складним залежно від конкретних вимог. Простий проект містить файли вихідного коду і файл проекту, більш складні – ці ж елементи і сценарії баз даних, збережені процедури й посилання на існуючі XML (веб-служби).

Visual Studio надає шаблони для проектів найбільш поширених типів (рис. 2.24).

У середовищі Visual Studio передбачено потужний набір засобів будування й відлагодження. З допомогою конфігурацій будування можна вибирати компоненти для будування, виключати компоненти, які не потрібні для будування, а також визначати, як буде побудовано вибрані проекти і для якої платформи. Розглянемо вікна середовища (рис. 2.25).

IDE MVS-2008 має багато вікон. Вікна викликаються і приховуються з допомогою меню View. На рис. 2.25 цифрами позначено:

- 1 – вікно рішень, ресурсів і класів;
- 2 – вікно перегляду і редагування коду;
- 3 – вікно властивостей проекту (Properties Window);
- 4 – вікно виведення (Output Window).

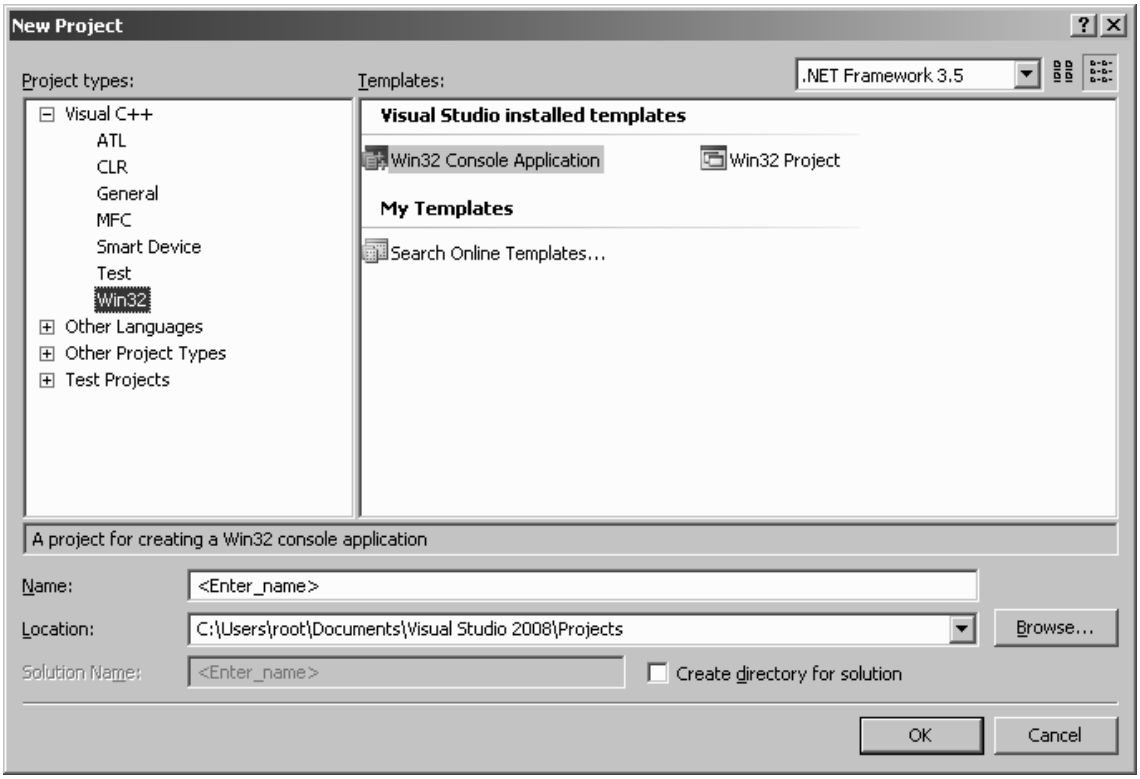


Рис. 2.24. Вікно вибору шаблону проекта

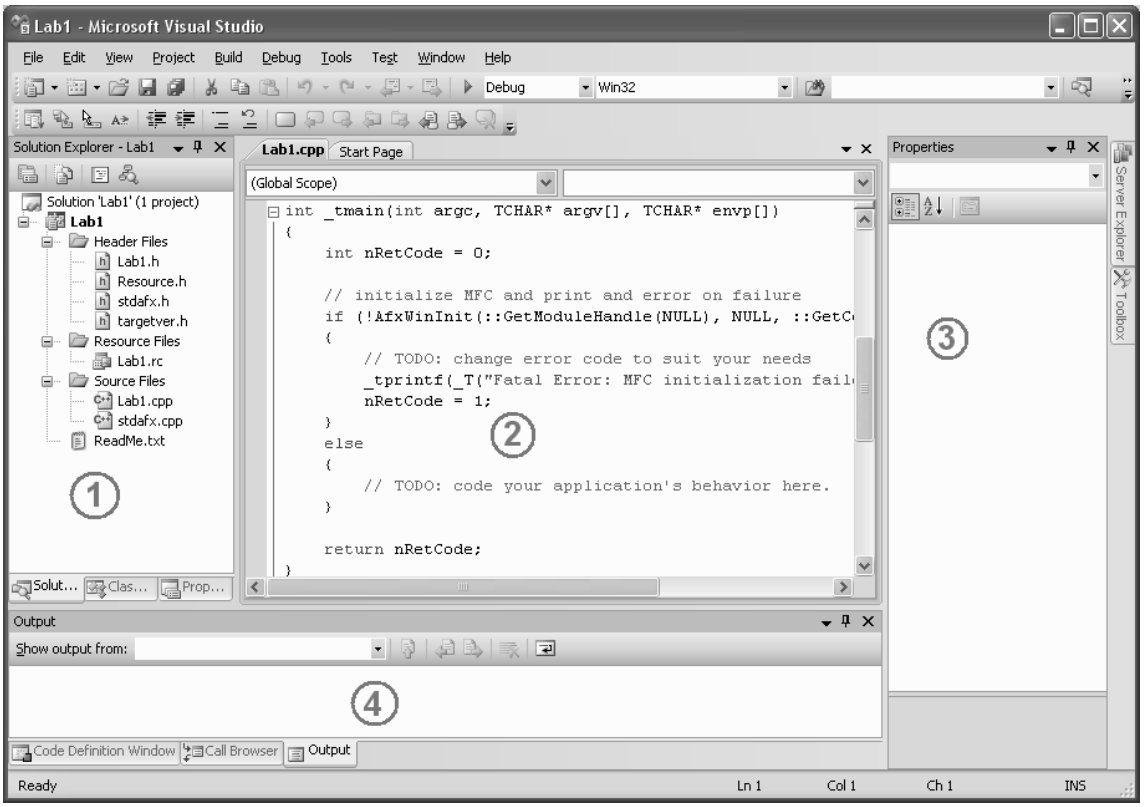


Рис. 2.25. Основні вікна IDE

Усі перелічені вікна, крім вікна перегляду і редагування коду, можна викликати з допомогою відповідних команд меню View; можна керувати розміщенням і розмірами вікон і можна їх пристиковувати до кордонів головного вікна (docking) загальноприйнятими способами.

Вікно рішень, ресурсів і класів у версії Microsoft Visual Studio 6.0 мало спеціальну назву Workspace (робочий простір), що більш точно відображало його призначення. У версії IC MVS-2008 це вікно має такі вкладки:

– перелік файлів проекту (Solution Explorer);

– перелік класів проекту (Class View);

– перелік ресурсів (Resource View).

На вкладці Solution Explorer відображаються імена майже всіх файлів проекту. Клацання по імені файла, наприклад Lab1.cpp, викликає вікно перегляду і редагування вмісту цього файла.

На вкладці Class View (рис. 2.26) відображаються імена класів, що входять до складу проекту, їхні члени-функції і члени-данні, а також глобальні функції і описи. На цій вкладці є тільки ім'я проекту Lab1, а в нижній частині вкладки – єдина поки функція _tmain () і глобальний об'єкт theApp, який є додатоком. По суті вкладка Class View є альтернативним способом навігації за вихідним кодом порівняно з вкладкою Solution Explorer.

Вкладка Resource View містить перелік файлів ресурсів, діалогових вікон, рядкових та інших ресурсів.

Зауваження. Перелічені вкладки часто об'єднують («докірують») в одне вікно для зручності (див. рис. 2.25).

Рис. 2.26. Вкладка Class View вікна робочого простору

Створений майстром IDE каркас є працездатним додатком. Для перевірки цього факту необхідно виконати команду Debug->Start Debugging (F5), яка компілює файли проекту, збірає їх (будування exe-файла) і запускає програму на виконання в режимі настроювання. При її виконанні у вікні Output відображається проведення компіляції і збірки проекту, виводяться попередження Warnings і про помилки. З допомогою списку Show output from у цьому вікні можна вибирати вкладки Debug і Build. На вкладці Build відображається проведення компіляції і збірки, а також повідомлення про помилки. На вкладці Debug відображається виконання програми: завантаження бібліотек, повідомлення про помилки часу виконання, виводиться код завершення програми.

В IDE MVS-2008 є підтримка діаграм класів (розширення файлів .cd). Діаграми класів дають змогу швидко отримати ілюстрацію, редагувати клас просто на діаграмі, з моментальним оновленням коду. Однак частіше IDE MVS-2008 використовують для формування діаграм за існуючим кодом ПЗ.

Для етапів розроблення й моделювання ПЗ використовуються інші системи, найпопулярнішою серед яких є система **Rational Rose**. Робочий інтерфейс програми IBM Rational Rose 2003 складається з різних елементів, основними з яких є головне меню, стандартна панель інструментів, спеціальна панель інструментів, вікно браузера проекту, робоча область зображення діаграми (вікно діаграми), вікно документації, вікно журналу (рис. 2.27).

Розглянемо призначення й основні функції кожного з цих елементів.

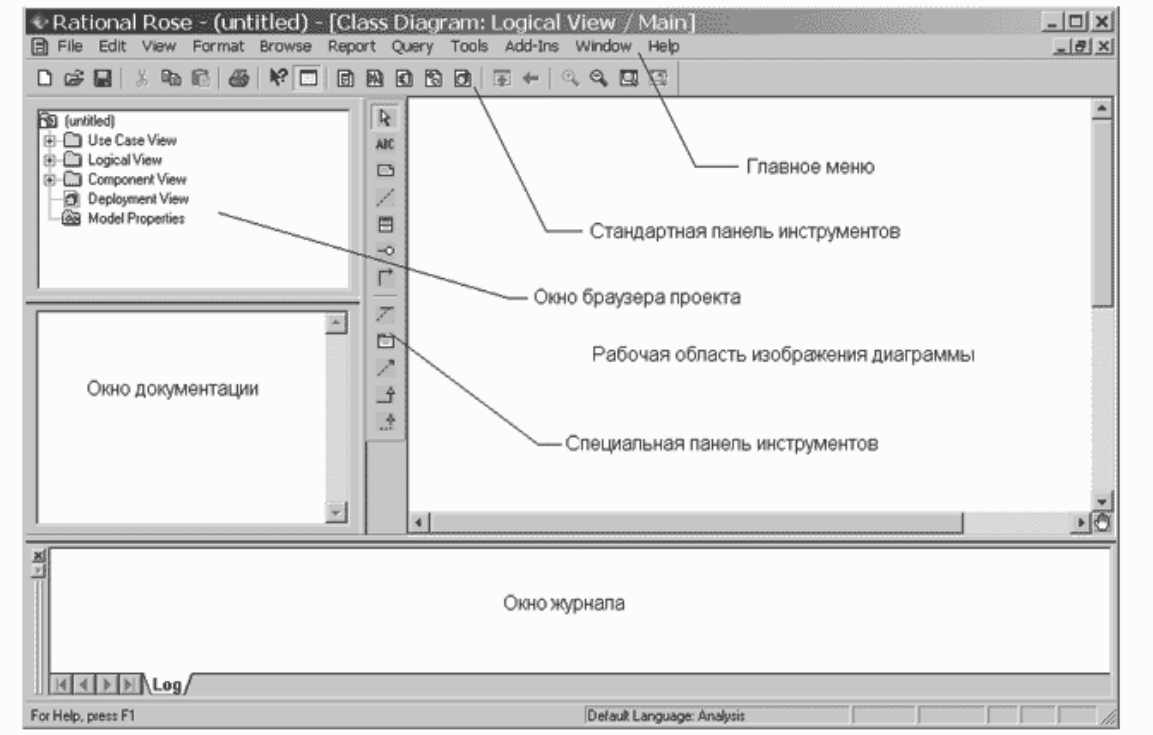


Рис. 2.27. Загальний вигляд інтерфейсу IBM Rational Rose 2003

Вікно браузера проекту за замовчуванням розташовано в лівій частині робочого інтерфейсу нижче стандартної панелі інструментів і має вигляд, показаний на рис. 2.28.

Браузер проекту організовує подання моделі у вигляді ієрархічної структури. При цьому верхній рядок браузера проекту містить ім'я розроблюваного проекту. Ієрархічне подання структури кожного розроблюваного проекту організовано у вигляді чотирьох уявлень:

− Use Case View – подання варіантів використання, у якому містяться діаграми варіантів використання та їх реалізація у вигляді варіантів взаємодії;

− Logical View – логічне подання, у якому містяться діаграми класів, станів і діяльності;

− Component View – подання компонентів, у якому містяться діаграми компонентів розроблюваної моделі;

− Deployment View – подання розгортання, у якому міститься єдина діаграма розгортання розроблюванної моделі.

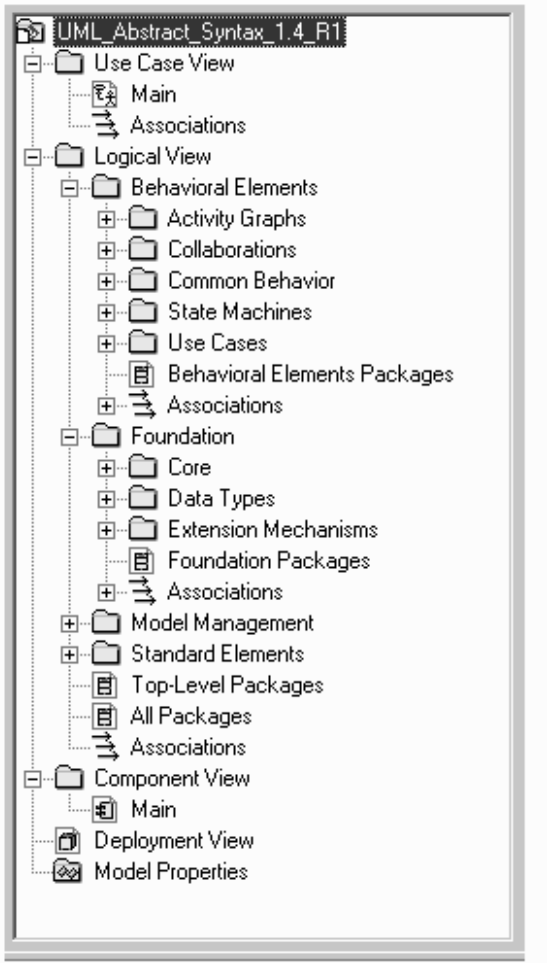


Рис. 2.28. Зовнішній вигляд браузера проекту з ієрархічним поданням його структури

При створенні нового проекту зазначена ієрархічна структура формується програмою автоматично.

Спеціальну панель інструментів розташовано між вікном браузера і вікном діаграми в середній частині робочого інтерфейсу. За замовчуванням панель інструментів пропонується для будування діаграми класів моделі.

Зовнішній вигляд спеціальної панелі інструментів залежить не тільки від вибору типу розроблюваної діаграми, але й від вибору графічної нотації для зображення елементів цих діаграм. В IBM Rational Rose 2003 реалізовано три таких нотації: UML, OMT і Booch (останні дві нотації майже не використовуються на практиці).

Вікно діаграми є основною графічною областю програми IBM Rational Rose 2003, у якій візуалізуються різні уявлення моделі проекту.

Головне меню програми IBM Rational Rose 2003 містить стандартні операції:

− *Report (Звіт)* – дає змогу відображати різну інформацію про елементи розроблюваної моделі й викликати діалогове вікно вибору шаблону для генерації звіту про модель;

− *Query (Запит)* – дає змогу додавати існуючі елементи розроблюваної моделі на діаграму, а також налаштовувати спеціальний фільтр відображення відносин між окремими елементами моделі;

− склад операцій *Tools (Інструменти)* головного меню залежить від установлених у програмі IBM Rational Rose 2003 конкретних розширень, наприклад: перевірка розроблюваної моделі на наявність помилок, інформація про які відображається у вікні журналу; дозвіл на виконання настроювання властивостей і специфікацію моделі для генерації програмного коду IDE MVS C++, вибраного як мова реалізації окремих елементів моделі.

Контрольні запитання

1. Перелічіть типи трансляторів. У чому полягає принципова відмінність інтерпретаторів і компіляторів?
2. Які основні компоненти зазвичай містить сучасне середовище розроблення?
3. Перелічіть засоби відлагодження коду ПЗ у середовищі розроблення Visual Studio.
4. Які подання проекту розроблення ПЗ існують в IBM Rational Rose 2003?
5. Які існують шаблони проектів Visual Studio? Назвіть ПЗ, де їх застосовують.
6. Які ще моделі крім моделі в нотації UML можна використовувати в IBM Rational Rose 2003?

3. ВИМОГИ ЗАМОВНИКА І ЯКІСТЬ ПЗ

3.1. Аналіз вимог замовника до ПЗ

Експлуатаційні вимоги визначають деякі характеристики розроблюваного ПЗ, що виявляються під час його функціонування:

– *правильність* – функціонування відповідно до технічного завдання;

– *універсальність* – забезпечення правильної роботи з будь-якими допустимими даними і захист від неправильних даних;

– *надійність* – забезпечення повної повторюваності результатів, тобто забезпечення їх правильності за наявності різного роду збоїв;

– *перевірність* – можливість перевірки одержуваних результатів;

– *точність результатів* – забезпечення похибки результатів не вище від заданої;

– *захищеність* – забезпечення конфіденційності інформації;

– *програмна сумісність* – можливість спільного функціонування з іншим ПЗ;

– *апаратна сумісність* – можливість спільного функціонування з деяким устаткуванням;

– *ефективність* – використання мінімально можливої кількості ресурсів технічних коштів, наприклад часу мікропроцесора або місткості оперативної пам'яті;

– *адаптивність* – можливість швидкої модифікації ПЗ з метою пристосування до змінюваних умов функціонування;

– *повторне входження* – можливість повторного виконання ПЗ без перезавантаження з диска;

– *реєнтерабельність* – можливість «паралельного» використання ПЗ декількома процесами.

Правильність є обов'язковою вимогою для будь-якого ПЗ: все, що зазначено в технічному завданні, неодмінно має бути реалізовано. Однак слід розуміти, що ні тестування, ні верифікація не доводять правильності створеного програмного продукту. У зв’язку з цим зазвичай говорять про певну ймовірність наявності помилок. Звичайно, чим більша відповідальність перекладається на комп'ютерну систему, тим меншою має бути ймовірність як програмного, так і апаратного збою. Наприклад, очевидно, що ймовірність неправильної роботи для системи керування атомною електростанцією має дорівнювати майже нулю.

Вимога ***універсальності*** також зазвичай належить до групи обов'язкових. Погано, якщо розроблена система буде видавати результат для некоректних даних або аварійно завершить свою роботу на деяких наборах даних. Однак довести універсальність порівняно складної програми, як і її правильність, неможливо, тому має сенс говорити про ступінь універсаль-

ності програми.

Чим вище вимоги до правильності й універсальності ПЗ, тим вище й вимоги до його ***надійності***. Джерелами перешкод можуть бути всі учасники обчислювального процесу: технічні й ПЗ та люди.

У технічних засобах можуть бути збої, наприклад, через різкі стрибки напруги живлення або перешкоди при переданні інформації з мереж. ПЗ може містити помилки, а люди можуть помилятися при введенні вихідних даних.

Сучасні обчислювальні пристрої вже є досить надійними. Збої технічних засобів зазвичай реєструються апаратно, відповідно результати обчислень відновлюються. У разі тривалих обчислень проміжні результати зберігають (прийом отримав назву «створення контрольних точок»), що дає змогу при виникненні збою продовжити обчислення з використанням даних, записаних в останній контрольній точці.

Передання інформації з мереж також апаратно контролюється, крім того, зазвичай застосовується спеціальне кодування, яке дає змогу знаходити й виправляти помилки передання даних. Однак повністю виключити помилки технічних засобів неможливо, тому в тих випадках, коли вимоги до надійності є високими, зазвичай використовують дублювання систем, при якому дві системи вирішують одне і те саме завдання періодично звіряючи отримані результати.

Часто «найненадійнішим елементом» сучасних систем є люди. Уже добре відомо, що в умовах монотонної роботи за пультом обчислювальної установки оператори припускаються великої кількості помилок. Відомими є й засоби, що дають змогу зменшити кількість помилок у конкретних ситуаціях. Так, там, де це можливо, уводять надлишкову інформацію, що дає змогу перевіряти правильність уведених даних. Крім того, часто використовують різні підказки, коли інформацію необхідно не вводити, а вибирати з деякого списку, і т. ін.

Підвищені вимоги до надійності ставять при розробленні систем керування, що функціонують у режимі реального часу, коли обчислення виконуються паралельно з технологічними процесами. Зазвичай цю вимогу ставлять і до науково-технічних систем і баз даних.

Для забезпечення ***перевірки*** слід документально фіксувати вихідні дані, установлені режими та іншу інформацію, яка впливає на одержувані результати.

Особливо це є істотним у випадках, коли дані надходять безпосередньо від датчиків. Якщо такі дані не виводити разом з результатами, то останні не можна буде перевірити.

Точність, або величина похибки, результатів залежить від точності вихідних даних, ступеня адекватності використовуваної моделі, точності вибраного методу й похибки виконання операцій у комп'ютері. Вимоги до точності результатів зазвичай є найбільш жорсткими для систем керуван-

ня технологічними процесами (наприклад, хімічними) і систем навігації (наприклад, система керування стикуванням космічних апаратів).

Забезпечення ***захищеності*** (конфіденційності) інформації, що використовується проектованою системою, зокрема і в умовах наявності мереж, є досить складним завданням. Крім чисто ПЗ захисту, таких, як кодування інформації та ідентифікація користувача, для забезпечення захищеності використовують також спеціальні організаційні прийоми. Найбільш жорсткі вимоги ставляться до систем, у яких зберігається інформація, пов'язана з державною і комерційною таємницею.

Вимога ***програмної сумісності*** може варіюватися від можливості спільного установлення зазначеного ПЗ з іншим для забезпечення взаємодії з ним, наприклад обміну даними тощо. Найчастіше доводиться забезпечувати функціонування ПЗ під керуванням заданої операційної системи. Однак може знадобитися отримання даних з якоїсь програми або передання деяких даних до неї. У цьому випадку необхідно точно визначити формати переданих даних.

Вимогу ***апаратної сумісності*** в основному формулюють у вигляді мінімально можливої конфігурації обладнання, на якому працюватиме ПЗ. Якщо передбачається використання нестандартного обладнання, то для нього має бути вказано інтерфейси або протоколи обміну інформацією. При цьому для операційних систем класу Windows нестандартними вважаються пристрої, для яких у системі немає драйверів – програм, що забезпечують взаємодію пристрою з операційною системою.

Ефективність системи зазвичай оцінюється окремо за кожним ресурсом обчислювальної установки. Часто використовують такі критерії:

– *час відповіді системи* (зазвичай цей критерій належить до швидкодії використовуваного обладнання) – для систем, що взаємодіють з користувачем в інтерактивному режимі, і систем реального часу;

– *місткість оперативної пам'яті* – для продуктів, що працюють в системах з обмеженою місткістю оперативної пам'яті, наприклад MS DOS;

– *місткість зовнішньої пам'яті* – для продуктів, що інтенсивно використовують зовнішню пам'ять, наприклад баз даних;

– *кількість обслуговуваних зовнішніх пристроїв* – для продуктів, що інтенсивно взаємодіють із зовнішніми пристроями, наприклад датчиками.

Вимоги ефективності можуть суперечити одна одній. Наприклад, щоб зменшити час виконання деякого фрагмента програми, може знадобитися додаткова місткість оперативної пам'яті.

Адаптивність є визначенням технологічної якості ПЗ, тому оцінити цю характеристику кількісно майже неможливо. Можна тільки констатувати, що при створенні продукту використовують технології і спеціальні прийоми, що полегшують його модернізацію.

Вимогу ***повторного входження*** зазвичай ставлять до ПЗ, резидентно завантаженому в оперативну пам'ять, наприклад до драйверів. Для

забезпечення цієї вимоги необхідно так організувати програму, щоб ніякі її вихідні дані не затиралися під час виконання або відновлювалися на початку або після завершення кожного виклику.

Вимога ***реєнтерабельності*** є більш жорсткою, ніж повторне входження, оскільки в цьому випадку всі дані, які програма змінює під час виконання, має бути виділено в спеціальний блок, копія якого створюється для кожного процесу під час виклику програми.

Складність багатьох програмних систем не дає змоги відразу сформулювати чіткі вимоги до них. Зазвичай для переходу від ідеї створення деякого ПЗ до чіткого формулювання вимог, які можна занести в технічне завдання, необхідно виконати передпроектні дослідження в галузі розроблення.

Технічне завдання *(ТЗ) являє собою документ, у якому сформульовано основні цілі розроблення, вимоги до програмного продукту, визначено терміни й етапи розроблення і регламентований процес приймально-здавальних випробувань.*

У розробленні технічного завдання беруть участь представники як замовника, так і виконавця. Основою цього документа є вихідні вимоги замовника, аналіз передових досягнень техніки, результати виконання науково-дослідних робіт, передпроектних дослідів, наукового прогнозування і т. ін.

Основні чинники, що характеризують розроблюване ПЗ:

– вихідні дані й необхідні результати, які визначають функції програми або системи;

– середа функціонування (програмна й апаратна), яку можна задавати, а можна вибирати для забезпечення параметрів, зазначених у технічному завданні;

– можлива взаємодія з іншим ПЗ (або спеціальними технічними засобами) – також можна визначати або вибирати з кількох виконуваних функцій.

Технічне завдання розробляють у такій послідовності. Перш за все встановлюють набір виконуваних функцій, перелік і характеристики вихідних даних і результатів, а також способи їх подання. Потім уточнюють середовище функціонування програмного забезпечення: конкретну комплектацію і параметри технічних засобів, версію операційної системи і, можливо, версії і параметри іншого встановленого програмного забезпечення, з яким належить взаємодіяти майбутньому програмному продукту.

У випадках, коли розроблюване ПЗ збирає і зберігає деяку інформацію або використовується в керуванні будь-яким технічним процесом, необхідно також чітко регламентувати дії програми при збоях устаткування й енергопостачання.

На технічне завдання існують стандарти: *ГОСТ 19.201–78 «Технічне завдання. Вимоги до змісту та оформлення», ГОСТ 34.602−89 «Інфор-*

маційна технологія. Комплекс стандартів на автоматизовані системи. Технічне завдання на створення автоматизованої системи», ГОСТ 25123−82 «Машини обчислювальні і системи обробки даних. Технічне завдання. Порядок побудови, викладення та оформлення».

Відповідно до *ГОСТ 19.201–78* технічне завдання має містити такі розділи:

– вступ;
– підстави для розроблення;
– призначення розробки;
– вимоги до програми або програмного виробу;
– вимоги до програмної документації;
– техніко-економічні показники;
– стадії та етапи розроблення;
– порядок контролю і приймання.

За необхідності допускається в технічне завдання заносити додатки.

Розглянемо більш детально зміст кожного розділу.

Вступ має містити найменування і коротку характеристику області застосування програми або програмного продукту, а також об'єкта (наприклад, системи), у якому передбачається їх використовувати. Основне призначення вступу – продемонструвати актуальність цієї розробки й показати, яке місце їй належить серед подібних.

Розділ *«Підстави для розроблення»* має містити найменування документа, на основі якого ведеться розроблення, організації, яка затвердила цей документ, і найменування або умовне позначення теми розроблення. Таким документом може бути план, наказ, договір тощо.

Розділ *«Призначення розроблення»* має містити опис функціонального й експлуатаційного призначення програмного продукту із зазначенням категорій користувачів.

Розділ *«Вимоги до програми або програмного виробу»* має містити такі підрозділи:

– вимоги до функціональних характеристик;
– вимоги до надійності;
– умови експлуатації;
– вимоги до складу і параметрів технічних засобів;
– вимоги до інформаційної і програмної сумісності;
– вимоги до маркування й упаковки;
– вимоги до транспортування і зберігання;
– спеціальні вимоги.

Найбільш важливим є підрозділ «Вимоги до функціональних характеристик», де має бути перелічено виконувані функції та описано склад, характеристики й форми подання вихідних даних і результатів. Тут же за необхідності зазначають критерії ефективності: максимально допустимий час відповіді системи, максимальну місткість використовуваної оператив-

ної та (або) зовнішньої пам'яті тощо.

Примітка. Якщо розроблене ПЗ не буде виконувати зазначених в технічному завданні функцій, то його вважають таким, що не відповідає технічним завданням, тобто неправильним з огляду на критерії якості. Універсальність майбутнього продукту зазвичай спеціально не обговорюється, але вважається.

У підрозділі «*Вимоги до надійності*» зазначають рівень надійності, який має бути забезпечено розроблюваної системою, і час відновлення системи після збою. Для систем із звичайними вимогами до надійності іноді регламентують дії розроблюваного продукту зі збільшення надійності результатів (контроль вхідної і вихідної інформації, створення резервних копій проміжних результатів тощо).

У підрозділі *«Умови експлуатації»* зазначають особливі вимоги до умов експлуатації: температури навколишнього середовища, відносної вологості повітря і т. ін. Зазвичай такі вимоги формулюють, якщо розроблювана система буде експлуатуватися в нестандартних умовах або використовувати спеціальні зовнішні пристрої, наприклад, для зберігання інформації. Тут же зазначають вид обслуговування, необхідну кількість і кваліфікацію персоналу. В іншому випадку допускається зазначати, що вимоги не ставляться.

У підрозділі «*Вимоги до складу і параметрів технічних засобів*» зазначають необхідний склад технічних засобів і їхні основні технічні характеристики: тип мікропроцесора, місткість пам'яті, наявність зовнішніх пристроїв тощо. При цьому часто зазначають два варіанти конфігурації: мінімальний і рекомендований.

У підрозділі *«Вимоги до інформаційної і програмної сумісності»* за необхідності можна визначити мову або середу програмування для розроблення, а також операційну систему та інші системні й призначені для користувача ПЗ, з якими має взаємодіяти ПЗ, що розробляється. У цьому ж підрозділі за необхідності зазначають, яким має бути ступінь захисту інформації.

У розділі *«Вимоги до програмної документації»* зазначають необхідність наявності посібника програміста, посібника користувача, посібника системного програміста, пояснювальної записки і т. ін. На всі ці типи документів існують ДСТУ.

У розділі *«Техніко-економічні показники»* рекомендується зазначати орієнтовну економічну ефективність, передбачену річну потребу й економічні переваги порівняно з існуючими аналогами.

У розділі *«Стадії та етапи розроблення»* зазначають стадії розроблення, етапи й зміст робіт із зазначенням терміну розроблення й виконавців.

У розділі *«Порядок контролю і приймання»* зазначають види випробувань і загальні вимоги до прийняття роботи.

У *додатках* за необхідності наводять перелік науково-дослідних робіт; схеми алгоритмів, таблиці, описи, обгрунтування, розрахунки та інші документи, які слід використовувати під час розроблення.

Залежно від особливостей розроблюваного продукту можна уточнювати зміст розділів, використовувати підрозділи, уводити нові розділи або об'єднувати їх. У випадках, якщо замовник не ставить ніяких вимог, передбачених у певному розділі технічного завдання, то слід у відповідному місці зазначити: «Вимог не поставлено».

Розроблення технічного завдання – процес трудомісткий, потребує певних навичок. Найбільш складним зазвичай є чітке формулювання основних розділів: вступу, призначення розроблення та вимог до програми або програмного виробу. При складанні розділу*«Вимоги до програми або програмного виробу»* до питання потрібно підходити формально. Інакше кажучи, "відкривати нове вікно", "редагувати поточний файл за допомогою команд з користувальницьких консолей", і "зберігати зміни при закритті головного вікна програми" - це чіткий і формальний підхід.

Імовірні недоліки, які бувають при складані технічного завдання: нечіткість постановки, недосяжність вимог, неповність специфікацій, недоступність ПЗ для більшої кількості користувачів, нереальні плани (строки).

Приклад технічного завдання на виконання домашнього завдання, складеного за скороченою схемою, наведено в дод. 2.

Контрольні запитання

1. Що містить розділ технічного завдання «Вимоги до програмної документації»?
2. Що містить розділ технічного завдання «Порядок контролю і приймання системи»?
3. Назвіть основні експлуатаційні вимоги.
4. Яке ПЗ відповідає вимозі «реєнтерабельність»?
5. Що розуміється під надійністю до ПЗ?
6. У який розділ технічного завдання необхідно внести вимоги до діапазону вхідних даних?
7. У якому розділі технічного завдання знаходяться дані про вартість ПЗ?
8. Якщо у ПЗ необхідно передбачити автозбереження даних через зазначений час, то до якого розділу необхідно внести цю вимогу?
9. Якщо ПЗ буде використовуватися тількі в операційній системі Windows, то до якого розділу необхідно внести цю вимогу?
10. Які документи наводять у розділі «Вимоги до програмної документації»? Знайдіть стандарти, що регламентують написання цих документів.

3.2. Якість ПЗ, метрики якості, стандарти якості ПЗ

Розглянемо основні означення ПЗ і його якостей за стандартами (табл. Д.1.1), які використовуються як в наукових, так і в прикладних сферах.

Помилка (Error) *– це хибне значення величини на виході системи (або підсистеми), спричинене несправностями або збоями, яке, зі свого боку, може спричинити відмову.*

З огляду на надійність ПЗ помилку можна розглядати як недолік або неточність, яких припустилися проектувальники ПЗ, програмісти, аналітики й тестувальники. Наприклад, проектувальник може неправильно зрозуміти завдання, а програміст – неправильно описати змінну та ін.

Несправність, дефект (Fault) *– це визначена або передбачувана причина помилки; наслідок відмови деякої системи, що обслуговувала або обслуговує в певний момент часу досліджувану систему.*

Дефекти також часто називають багами (від англ. bugs — жучки). Цей термін використовують, якщо вплив дефекту на роботу програми є невеликим. Якщо ж помилка пов'язана зі специфікаціями або архітектурою програми, то використовують термін «дефект».

Збій (Malfunction) *– це виявлення несправності, зазвичай у роботі устаткування.*

Відмова (Failure) *– це порушення нормального функціонування системи, повна або часткова втрата роботоздатності системи (або підсистеми).*

Під час виконання програми або роботи всієї системи тестувальник, розробник або користувач може не отримати очікуваних результатів. У деяких випадках така поведінка є симптомом помилки. Досвідчений розробник завжди зберігає базу даних помилок, які йому траплялися.

Некоректне поводження ПЗ може означати, що вихідні значення, відклик пристрою або зображення на екрані є неправильними. Під час розроблення відмови й баги зазвичай виявляють тестувальники, а дефекти знаходять і виправляють самі розробки. Якщо відмову виявив користувач, то звіт про помилку він надсилає розробнику з метою її усунення.

Несправність у коді не завжди призводить до відмови. Насправді, неправильна частина програми може функціонувати довгий час без виявлення яких-небудь недоліків. Несправність може спричинити відмову за таких умов:

– вхідні дані програми мають використовувати частину коду, яка містить несправність;

– наслідком несправності має стати некоректний внутрішній стан програми;

– некоректний внутрішній стан програми має впливати на вихідні дані так, щоб результат помилки можна було спостерігати.

Якщо за наявності несправності у коді ПЗ відповідає переліченим умовам, то його можна вважати придатним для тестування (testable).

Контрольний приклад (test case) для тестування – це записи, що належать до тесту і містять:

– вхідні дані тесту – інформація, яку програма отримує із зовнішнього джерела, наприклад з пристрою, іншої програми або людини;

– умови виконання – вимоги для проведення тесту, наприклад, певний стан бази даних або конфігурація пристрою;

– очікувані вихідні дані – передбачуваний результат роботи коду.

Цей перелік визначає мінімальну необхідну інформацію для контрольного прикладу відповідно до стандартів. Компанія-розробник може визначити додаткову інформацію як таку, що є необхідною для внесення до звіту з метою її повторного використання в майбутньому або надання докладніших даних іншим тестувальникам і розробникам.

Наприклад, мету тесту можна додати до контрольного прикладу для більшої зрозумілості результатів тесту. До визначення точного формату звіту контрольного прикладу слід залучати розробників, тестувальників і (або) відділ забезпечення якості ПЗ.

Якість (Quality) *– це ступінь відповідності системи, компонента або процесу заданим вимогам, потребам або очікуванням користувача.*

З метою визначення добротності системи, компонента або процесу використовують так звані атрибути якості – характеристики, що відображають цю властивість.

Метрика (Metrics) *– це кількість наявних атрибутів системи, компонента або процесу.*

Приклади атрибутів якості:

– надійність (reliability) – безперервність коректного перебігу процесу;

– готовність (availability) – постійна готовність системи;

– безпека (safety) – відсутність катастрофічних наслідків в системному середовищі;

– конфіденційність (confidentiality) – запобігання несанкціонованому доступу до інформації;

– цілісність (integrity) – відсутність появи в системі невідповідних змін інформації;

– ремонтопридатність (maintainability) – здатність системи піддаватися ремонту й розвитку;

– зручність роботи (usability) – здатність до навчання, роботи, підготовки вхідних і оброблення вихідних даних ПЗ, тобто рівень зусиль, необхідних для цього.

Кількісні методи оцінювання успішно застосовуються в інших сферах науки, а тому багато теоретиків і практиків у галузі інформаційних технологій спробували перенести цей підхід у процес розроблення ПЗ. У загальному випадку застосування метрик дає змогу визначити складність розро-

бленного ПЗ, однак метрики можуть бути лише рекомендаційними характеристиками.

До кількісних метрик належать, наприклад, **метрики Холстеда**. Їх розрахунок базується на таких показниках:

– n_1 – кількість унікальних операторів програми, у тому числі символів-роздільників, імен процедур і знаків операцій;

– n_2 – кількість унікальних операндів програми;

– N_1 – загальна кількість операторів у програмі;

– N_2 – загальна кількість операндів у програмі;

– n'_1 – теоретична кількість унікальних операторів;

– n'_2 – теоретична кількість унікальних операндів.

Використовуючи ці показники, можна визначити:

– словник програми $n = n_1 + n_2$;

– довжину програми $N = N_1 + N_2$;

– теоретичний словник програми $n' = n'_1 + n'_2$;

– теоретичну довжину програми $N' = n_1\log_2(n_1) + n_2\log_2(n_2)$;

– місткість програми $V = N\log_2(n)$;

– теоретичну місткість програми $V' = N'\log_2(n')$;

– рівень якості програмування $L' = (2\ n_2) / (n_1 N_2)$;

– складність розуміння програми $Ec = V/(L')^2$;

– трудомісткість кодування програми $D = 1/L'$;

– інформаційну місткість програми $I = V/D$;

а також оцінити необхідні інтелектуальні зусилля при розробленні програми $E = N'\log_2(n/L)$.

Серед інших метрик, які дають змогу кількісно оцінювання складність коду, найбільш відомими є такі:

− розрахунок об'єктно-орієнтованої складності на основі оцінювання характеристик класів і їх взаємозв’язків;

− розрахунок цикломатичної складності на основі графа керувальної логіки ПЗ;

− метрика Чепіна на основі інформаційної міцності ПЗ.

Контрольні запитання

1. Назвіть метрики вимог будь-якого програмного продукту.

2. Поняття метрик вимог і підходи до вимірювання вимог. Міри і метрики у програмній інженерії.

3. Які існують підходи до вимірювання вимог?

4. Як можна кількісно оцінити місткість ПЗ?

5. Як можна оцінити необхідні інтелектуальні зусилля ПЗ з допомогою кількісних метрик Халстеда?

6. Яким чином можна кількісно оцінити надійність ПЗ? Запропонуйте свій варіант.

3.3 Верифікація, валідація і тестування. Стандарти тестування ПЗ

Тестування – це процес керованого експериментування з продуктом з допомогою тестів для виявлення в ньому помилок, неточностей, яких припустилися розробники ПЗ.

Тест – контрольне завдання для перевірки коректності функціонування системи та (або) її ПЗ. Основна ідея тестування – запустити ПЗ і спостерігати за його роботою та наслідками цієї роботи. Якщо збій у роботі ПЗ відбувся, то аналізується звіт з метою виявлення місцезнаходження помилки, яка його спричинила.

Тест є вдалим тоді, коли виконання програми закінчилося з помилкою і навпаки. Метою тестування є демонстрація якості функціонування ПЗ і виявлення й усунення помилок в ПЗ [6].

Головною метою тестування є збільшення ймовірності того, що ПЗ буде відповідати своїм специфікаціям.

Тестування – процес ітераційний. Після виявлення й виправлення кожної помилки обов'язково слід повторити тести для перевірки працездатності програми. Більш того, для ідентифікації причини виявленої проблеми може потребуватися спеціальне додаткове тестування. При цьому завжди потрібно пам'ятати фундаментальний висновок, зроблений професором Едсжером Дейкстром 1972 р: «Тестування програм може бути доказом наявності помилок, але ніколи не доведе їх відсутність!» [3].

Тестування можна здійснити вручну або автоматично. Автоматичне тестування, якщо його виконати коректно, забезпечує більш швидке, якісне й ефективне тестування, що зменшує кількість тестів, час тестування, підвищує стійкість системи.

Тестування супроводжує весь життєвий цикл ПЗ, починаючи від проектування і закінчуючи невизначено довгим етапом експлуатації. Тестування безпосередньо пов'язане з керуванням вимогами і змінами, адже метою тестування є можливість переконатися у відповідності програм заявленим вимогам [6].

Як було зазначено вище, результати тестування є підтвердженням, що ПЗ працює згідно зі специфікацією. Вважається, що програма працює коректно, якщо задовольняє таким критеріям:

– отримавши коректні дані, програма дає правильний результат;

– отримавши некоректні дані, програма відхиляє їх;

– програма не «зависає» і не «вилітає», приймаючи як коректні, так і некоректні дані;

– програма функціонує нормально стільки часу, скільки потрібно;

– програма працює без збоїв і виконує всі необхідні функції в повному обсязі.

Перевірка на валідність (Validation) *– це процес, що дає змогу ви-*

значити, наскільки точно з позицій потенційного користувача деяка модель є відображенням заданої суті реального світу.

***Верифікація (Verification)** – це процес, який дає змогу визначити, що розроблене ПЗ точно реалізує концептуальний опис певної системи, або, як ще кажуть, процес перевірки відповідності системи заданим стандартам.*

Верифікація є успішною, якщо отримані дані збігаються з очікуваними, заздалегідь визначеними як правильні. Зазначимо, що верифікація може бути неформальною, тобто тестувальник визначає успішність на основі своїх знань.

Верифікацію і перевірку на валідність часто використовують разом, але їх по-різному визначають. Різниця між ними є важливою для тестування ПЗ. Верифікація є підтвердженням того, що продукт відповідає специфікаціям, а перевірка на валідність – вимогам користувача. Може здатися, що означення є дуже схожими, але на прикладі опису проблем орбітального телескопа Хаббла видно різницю.

У квітні 1990 року було створено орбітальний телескоп Хаббла з великим дзеркалом для збільшення об'єктів. Конструкція телескопа потребувала неймовірної точності, а тестування було майже неможливим, тому його перевірили лише на відповідність специфікаціям.

На жаль, після введення його в експлуатацію виявилося, що надіслані зображення є розфокусованими. З досліджень стало зрозуміло, що було виготовлено неправильне дзеркало. Незважаючи на те, що, можливо, це було найбільш точно розраховане дзеркало з коли-небудь створених, а допуск становив не більше 1/20 довжини хвилі видимого світла, виявилося, що краї були дуже плоскими. Відхилення від потрібної форми поверхні становило лише 2 мкм, але результат виявився катастрофічним — сильна сферична аберація, оптичний дефект, при якому світло, відбите від країв дзеркала, фокусується в точці, відмінній від тієї, у якій фокусується світло, відбите від центра дзеркала. Тестування показало, що дзеркало пройшло верифікацію, але не пройшло перевірку на валідність. 1993 року експедиція на космічному кораблі відремонтувала телескоп, установивши коригувальну лінзу для відновлення фокуса.

Лише повна перевірка дасть змогу уникнути проблем, які постали перед ученими у випадку з телескопом Хаббла.

Методи контролю якості дають змогу переконатися, що певні характеристики якості ПЗ досягнуто. Самі по собі вони не можуть допомогти їх досягненню, а лише дають змогу визначити, чи вдалося отримати необхідний результат, а також знайти помилки, дефекти й відхилення від вимог.

Методи й техніки контролю якості ПЗ можна класифікувати таким чином:

– для аналізу властивостей ПЗ під час його роботи (усі види тестування, а також вимірювання кількісних показників якості, які можна визна-

чити за висновками роботи ПЗ: ефективність за часом та іншими ресурсами, надійність, доступність та ін.);

– для визначення показників якості на основі симуляції роботи ПЗ з допомогою моделей різного типу (перевірка на моделях (model checking), прототипування (макетування), що використовується для оцінювання якості прийнятих рішень);

– для виявлення порушень формалізованих правил будування початкового коду ПЗ, проектних моделей і документації (інспекція коду, що полягає в цілеспрямованому пошуку певних дефектів і порушень вимог у коді на основі набору шаблонів; автоматизовані методи пошуку помилок у коді, що не базуються на його виконанні; методи перевірки документації на узгодженість і відповідність стандартам);

– звичайного або формалізованого аналізу проектної документації і початкового коду для виявлення їх властивостей (числові методи аналізу архітектури ПЗ, формального доказу властивостей ПЗ, формального аналізу ефективності вживаних алгоритмів).

3.3.1. Рівні і види тестування

Під час розроблення ПЗ зазвичай тестується на декількох рівнях інтеграції: поблочне тестування, перевірка взаємодії (інтеграційне тестування) і системне тестування.

Відповідно до етапів розроблення ПЗ вирізняють три фази тестування: модульне, інтеграційне і системне.

Модульне, або автономне, тестування (module testing, unit testing) – контроль окремого програмного модуля, зазвичай в ізольованому середовищі (тобто ізольовано від усіх інших модулів). Під модулем розуміється логічно замкнений фрагмент програми, який можна викликати через його інтерфейс. Модуль перевіряється на відповідність своїм специфікаціям і внутрішню логіку.

Інтеграційне тестування, або тестування взаємодій (integration testing), – контроль взаємодії між частинами системи (модулями, компонентами, підсистемами).

Системне, або комплексне, тестування (system testing), – контроль та (або) випробування всього програмного забезпечення як повної системи в цільовому середовищі, тобто підтвердження того, що доступ до всіх компонентів системи і взаємодія з ними є несуперечливими і їх передбачено згідно зі специфікаціями системи.

Очевидно, що фази не є взаємозамінними, і, наприклад, проведення модульного тестування не гарантує правильності інтеграційного, оскільки правильність функціонування окремих компонент не гарантує правильності їх взаємодії як між собою, так і з системою в цілому.

Альфа- і бета-тестування. Зазвичай перед завершенням ПЗ прохо-

дить дві стадії тестування.

Перша стадія – альфа-тестування (alpha testing) – використання майже готової версії продукту (зазвичай програмного або апаратного забезпечення) штатними програмістами (розробниками і тестувальниками) з метою виявлення помилок в його роботі для їх подальшого усунення перед бета-тестуванням.

Другою стадією є бета-тестування (beta testing) – інтенсивне використання ПЗ з метою виявлення максимальної кількості помилок в його роботі для їх подальшого усунення перед остаточним виходом (випуском) продукту на ринок до масового споживача.

На відміну від альфа-тестування, що проводиться силами штатних розробників або тестувальників, для бета-тестування залучають добровольців з-поміж майбутніх користувачів продукту, яким розсилається попередня версія продукту (так звана бета-версія). Такими добровольцями (їх називають бета-тестувальниками) зазвичай керує цікавість до нового продукту, вони згодні його випробувати, щоб знайти ще невиправлені помилки.

Крім того, бета-тестування можна використовувати як частину стратегії просування продукту на ринок (наприклад, безкоштовне роздавання бета-версій дає змогу привернути увагу споживачів до дорогої версії продукту), а також для отримання попередніх відгуків про нього від широкого кола майбутніх користувачів.

Тестування, що базується на вимогах (Requirement based testing) – це тестування кожної функціональної вимоги з певного документа (документи технічної підтримки, посібники та інша документація користувача).

Регресійне тестування (regression testing, від лат. regressio — рух назад) – це спільна назва для всіх видів тестування ПЗ, метою яких є виявлення помилок у вже протестованих ділянках початкового коду. Такі помилки (коли після внесення змін програма перестає працювати) називають регресійними помилками (regression bugs).

Зазвичай використовувані методи регресійного тестування містять повторні прогони попередніх тестів, а також перевірки стосовно потрапляння регресійних помилок у чергову версію внаслідок злиття коду.

З досвіду розроблення ПЗ відомо, що повторне виникнення одних і тих самих помилок — випадок досить поширений. Іноді це відбувається унаслідок слабкої техніки керування версіями або людської помилки при роботі з системою керування версіями, але вирішити проблему надовго неможливо: після наступного змінення в програмі треба переписувати якунебудь частину коду, при цьому часто виникають ті самі помилки, що й у попередній версії. Тому найкращим рішенням є створення тесту на виявлення помилок з метою його використання при подальших зміненнях програми. Хоча регресійне тестування можна виконати й вручну, але найчастіше це робиться з допомогою спеціальних програм, що дають змогу виконувати всі регресійні тести автоматично. У деяких проектах використову-

ються навіть інструменти для автоматичного прогону регресійних тестів через заданий інтервал часу. Зазвичай це виконується після кожної вдалої компіляції (у невеликих проектах), або щоночі, або кожного тижня.

При тестуванні **чорного ящика (black-box testing)** тестувальник має доступ до ПЗ тільки через ті самі інтерфейси, що й замовник чи користувач, або через зовнішні інтерфейси, які дають змогу іншому комп'ютеру чи іншому процесу під'єднатися до системи для тестування.

Можна виділити кілька видів тестування чорного ящика.

Тестування за класами еквівалентності (Equivalence class testing). Клас еквівалентності – це множина значень змінної, які за припущенням є еквівалентними. Контрольні приклади є еквівалентними, якщо виконуються такі вимоги:

– всі тести перевіряють один об'єкт;

– якщо один із тестів «має» дефект, то інший також його «має»;

– якщо один із тестів не виявляє дефект, то інший, скоріше за все, також його не виявить.

Якщо є такий клас еквівалентності, то треба використовувати для тестування його компонентів лише один із тестів.

За способами вибору представників класу еквівалентності визначають вид тестування.

Граничне тестування (Boundary testing). Граничні значення є найбільшими й найменшими значеннями класів еквівалентності.

При тестуванні граничних значень тестуються також значення менше за найменше та більше за найбільше.

Тестування кращих представників (Best representative testing). Тестування значень, що найбільш імовірно приведуть до виявлення дефекту. Значення завжди можна змінити різними шляхами. Треба покрити всі можливі варіанти.

Приклад 1. Специфікація: програму призначено для додавання двох цілих чисел. Кожен з доданків – не більше за двозначне ціле число. Програма запитує у користувача два числа, після чого виводить результат. Виділимо класи еквівалентності (табл. 3.1).

Приклад 2. Специфікація: програма читає з консолі три цілих числа й визначає тип трикутника (рівносторонній, рівнобедрений або різнобічний). Розглянемо тести стосовно мети й набору вхідних даних:

– коректний різнобічний трикутник {5, 6, 7};

– коректний рівносторонній трикутник {15, 15, 15};

– три коректних рівнобедрених трикутники (a = b, b = c, a = c) {3, 3, 4; 5, 6, 6; 7, 8, 7};

– одна, дві або три сторони дорівнюють нулю (7 тестів) {0, 1, 1; 2, 0, 2; 3, 2, 0; 0, 0, 9; 0, 8, 0; 11, 0, 0; 0, 0, 0};

– одна сторона є від'ємною {3, 4, -6}, {3, -6, 4}, {-6, 3, 4};

– майже виконується правило трикутника (три варіанти: a + b = c, a + c = b, b + c = a) {1, 2, 3; 2, 5, 3; 7, 4, 3};

– не виконується правило трикутника (три варіанти: a + b < c,

a + c < b, b + c < a) {1, 2, 4; 2, 6, 2; 8, 4, 2};

– неціле число або нечисловий символ {Q, 2, 3} {2, Q, 3} {2, 3, Q};

– неправильна кількість аргументів {2, 4; 4, 5, 5, 6};

– переповнення під час перевірки правила трикутника {16384, 16384, 16384}.

Таблиця 3.1

Компонент	Класи коректних даних	Класи некоректних даних	Граничні й спеціальні значення
Перший доданок	−99…−10 −9…−1 0 1…9 10…99	Понад 99 Меньше −99 1,5 Q	0, 1, −1, 9, −9 10, −10 99, −99 100, −100
Другий доданок	Те саме	Те саме	Те саме
Сума	−198…−100 −99…−1 0 1…99 100…198	Понад 198, Меньше −198, Некоректні дані	(−99, −99) (−49, −51) (99, −99) (49, 51)

Таким чином, для здійснення тестування чорного ящика необхідно виокремити правильні й неправильні класи еквівалентності та визначити граничні значення.

При тестуванні **білого (white-box testing),** або **прозорого, ящика**, розробник тесту має доступ до початкового коду і може писати код, який пов'язаний з бібліотеками тестованого ПЗ. Це є типовим для модульного тестування (англ. unit testing), коли тестуються тільки окремі частини системи, при цьому забезпечується роботоздатність і стійкість компонентів конструкції.

Тестування білого ящика складається з кількох його видів.

Тестування логіки (Logic testing). Багато програм мають логіку типу «якщо, то». При тестуванні логіки тестуються можливі сценарії і комбінації.

Тестування станів (State-bases testing). Робота кожної програми – це переходи з одного стану в інший. Тестування станів проводиться для перевірки коректності роботи програми у кожному її стані.

Тестування покриття операторів і гілок (Statement and branch coverage). Коли покрито всі рядки і всі твердження коду − це стовідсоткове покриття, в інших випадках покриття береться у відсотковому відношені.

Тестування шляхів (Path testing) – тестування набору шляхів (підшляхів), які проходить програма.

Тестування методом білого ящика іноді називають структурним тестуванням.

3.3.2. Структурне тестування

Тестування потоків керування програми. Для тестування на основі потоку керування існує кілька критеріїв: покриття операторів, рішень, шляхів, циклів, умов і т. д. Найпростішим є критерій покриття операторів.

Критерій покриття операторів (C0). Кожен оператор програми має бути виконано (покрито) хоча б один раз. Цей критерій є найбільш слабким з тих, що використовуються в структурному тестуванні, тому що проходження всіх операторів не є гарантією перевірки правильності послідовності попарних переходів між ними.

Критерій покриття рішень(C1). Кожна гілка алгоритму (кожний перехід між вершинами) має бути пройденою хоча б один раз. Виконання цього критерію забезпечує й покриття операторів, проте критерій C1 не є ідеальним (наприклад, не забезпечується перевірка правильності оброблення операторів логічних переходів і циклів).

Критерій покриття шляхів (C∞). Кожен шлях в алгоритмі (де шлях – це послідовність вершин (nstart, n1,..., nm, nfinal)) має бути протестовано хоча б один раз. Два шляхи вважаються ідентичними, якщо послідовності вершин є ідентичними. Це найбільш повний критерій, проте його реалізація ускладнена через велику кількість необхідних тестів. Так, наприклад, проблемою є тестування циклічних структур через необхідність їх багаторазового повторення. Для спрощення цієї процедури було запропоновано два критерії.

Граничне тестування циклів. Відповідно до цього критерію має бути виконано вхід у кожний цикл (проте виконання ітерацій не потребується).

Внутрішнє тестування циклів. Відповідно до цього критерію має бути виконано вхід у кожний цикл і щонайменше одну ітерацію.

Очевидно, що виконання другого критерію забезпечує виконання критерію граничного тестування циклів, але разом з тим потребує більшої кількості тестів.

Вище було зазначено, що часто виникають проблеми з тестуванням логічних конструкцій розгалуження. Для цього було запропоновано критерії покриття умов. Використання цих критеріїв може дати добір тестів, що забезпечують перехід у програмі, яка пропускається при використанні критерію *C1*.

Наприклад, при тестуванні фрагмента

if(x&&y) function1();

else function2();

використовується критерій покриття переходів. Для переходу на гілку *else* достатньо, щоб виконувалася тільки умова *!x*, що не гарантує перевірки працездатності оператора умови в цілому, тобто того, що при *!y* теж буде здійснено перехід на гілку *else*.

Існують кілька критеріїв покриття умов і умов/рішень.

Простий критерій покриття умов: кожну з атомарних умов має бути протестовано на свої правильні й помилкові значення хоча б один раз. Цей

критерій забезпечує тільки перевірку на можливість прийняття атомарними умовами правильних і помилкових умов, але не забезпечує перевірку правильності подальшого логічного переходу. Критерій покриття умов може не гарантувати покриття всіх переходів.

Розглянемо ту саму конструкцію. Критерій покриття умов потребує два тести (наприклад, x = TRUE, y = FALSE і x = FALSE, y = TRUE), за яких може не виконуватися оператор, розташований в then-гілці оператора if.

Таким чином, цей критерій треба поєднувати з критеріями покриття рішень.

Критерій покриття умов/рішень: крім вимог, наведених для простого критерію покриття умов, додається умова, що кожну гілку алгоритму (кожний перехід) має бути пройдено (виконано) хоча б один раз.

На практиці найчастіше виникає ситуація, коли правильність функціонування атомарних умов не гарантує істинності загального виразу умови. З метою забезпечення цієї умови було введено модифікований критерій покриття умов/рішень.

Модифікований критерій покриття умов/рішень: кожну атомарну умову, що впливає на істинність загального виразу-умови, має бути протестовано, при цьому тести мають бути незалежними від інших часткових умов.

Повну перевірку правильності функціонування операторів умови можна здійснити лише шляхом перебору всіх варіантів комбінацій атомарних умов, що належать до загального предикатного виразу. Це знайшло відображення у комбінаторному критерії покриття умов/рішень.

Комбінаторний критерій покриття умов/рішень. Усі комбінації істинних значень кожного з атомарних предикатів, що належать до умови, має бути протестовано. Це найбільш ресурсомісткий критерій покриття умов.

Усі перелічені критерії було наведено у вигляді ієрархічної структури (рис. 3.1).

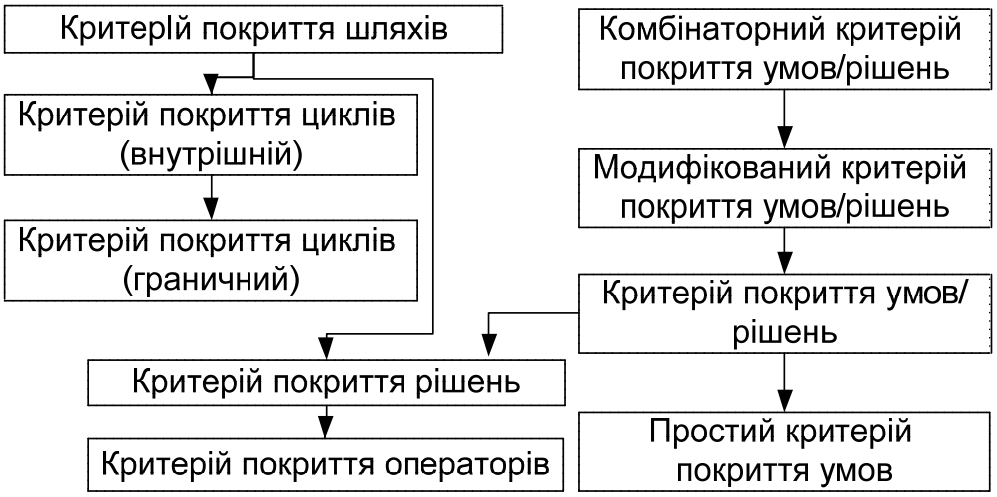


Рис. 3.1. Взаємозв'язок між критеріями покриття потоків керування програми

Проаналізуємо наведену структуру. Із критеріїв покриття умов/рішень найбільш сильним є комбінаторний, який забезпечує виконання модифікованого критерію. Останній, зі свого боку, забезпечує виконання критерію покриття умов/рішень. Простий критерій покриття умов, виходячи з його визначення, є складовою частиною критерію покриття умов/рішень.

Виконання критерію покриття шляхів забезпечує повне тестування циклічних структур, а отже, і виконання внутрішнього і граничного критеріїв тестування циклів. Зауважимо, що ці два критерії створено лише для циклів і навіть не забезпечують покриття всіх операторів, тобто їх необхідно використовувати тільки в сукупності з іншими.

Критерій покриття рішень забезпечує покриття всіх операторів. Зі свого боку, покриття всіх переходів (рішень) також можна досягнути завдяки виконанню більш повних критеріїв – покриття рішень/умов і шляхів.

Розглянемо практичне застосування критеріїв структурного тестування.

Приклад. Для тестування було вибрано частину програми, що завантажує файл .bmp на екран. Лістинг програми наведено у дод. 3. Перед початком аналізу програмного коду й складання тестів побудуємо блок-схему цієї програми (рис. 3.2), а на її основі – граф керування (рис. 3.3). Протестуємо програму, використовуючи критерії структурного тестування на основі потоку керування.

Першим розглянемо критерій покриття всіх вершин графа керування (покриття всіх операторів). Для покриття всіх операторів цієї програми необхідно провести тести.

Тест 1. Нехай змінна Path не вказує шлях реально існуючого файла, тоді тестована програма перевірить існування файла, виведе попередження й завершить роботу. При цьому значення інших змінних не впливають на покриття операторів. Таким чином, на графі керування буде пройдено шлях ([Початок] → [A] → [B] → [D] → [Кінець]), тобто буде покрито вершини A, B, D.

Тест 2. Для того щоб показати покриття операторів, які залишилися, достатньо, щоб виконувалися такі умови:

а) змінна Path має вказувати шлях реально існуючого файла;

б) має завантажуватися картинка, що має такий формат:

– Sagolovok.biWidth=2 (ширина зображення 2 пікселі);

– Sagolovok.biHeight=1 (висота зображення 1 піксель).

За таких початкових умов буде пройдено шлях ([Початок] → [A] → → [B] → [C] → [E] → [F] → [G] → [H] → [I] → [K] → [L] → [M] → [N] → [O] → → [P] → [Q] → [K] → [L] → [M] → [O] → [Q] → [K] → [L] → [M] → [O] → [Q] → → [K] → [L] → [M] → [O] → [Q] → [K] → [J] → [H] → [Кінець]), тобто буде покрито всі вершини, що залишилися.

При виконанні цих тестових наборів помилок не було виявлено. Критерій C0 для цієї програми виконується. Перейдемо до будування тестів на основі наступного критерію – покриття всіх рішень C1, тобто ребер графа керування.

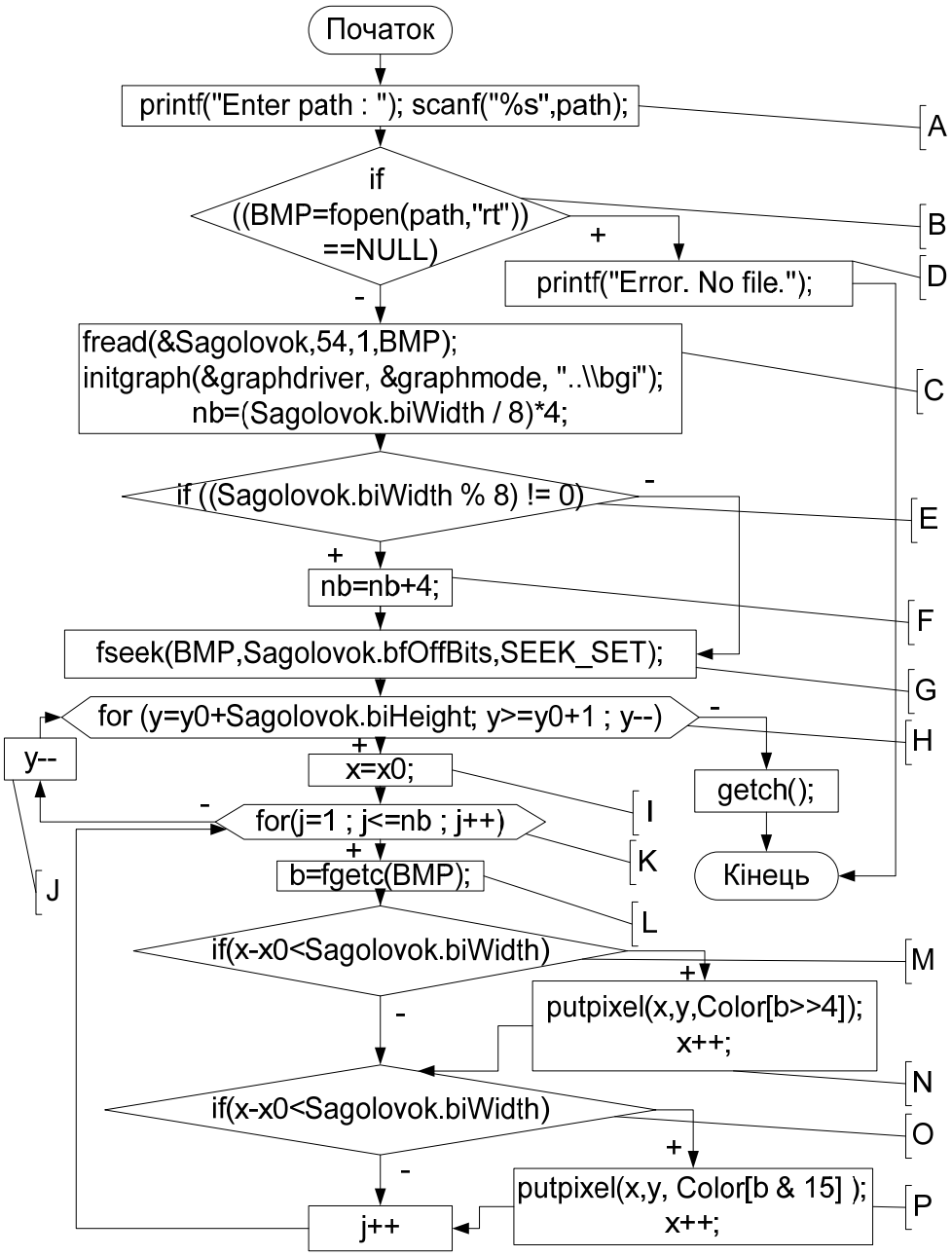


Рис. 3.2. Блок-схема програми

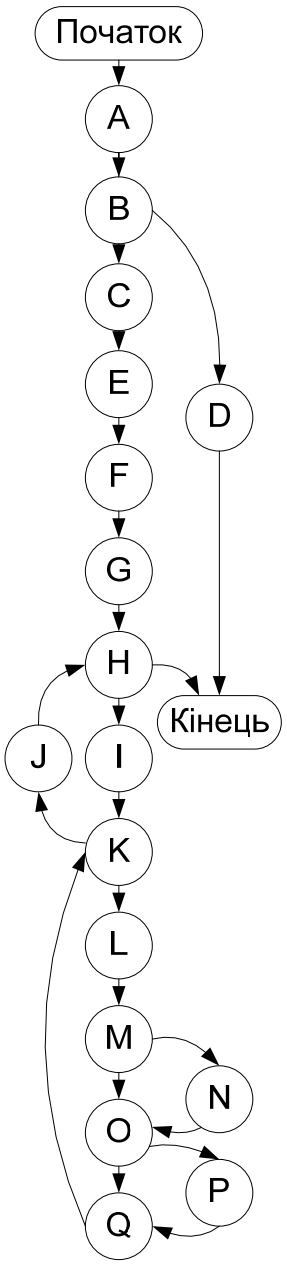


Рис. 3.3. Граф керування програми

Тест 3. Після проходження першого тесту залишилося непокритим лише одне ребро графа ([E] → [G]). Для проходження цього тесту необхідно додати до тестів 1 і 2 ще один тест:

– Path вказує шлях реально існуючого файла;

– Sagolovok.biWidth=8 (ширина зображення – 8 пікселів);

– Sagolovok.biHeight=1 (висота зображення – 1 піксель).

За таких початкових умов буде пройдено шлях ([Початок] → [A] → → [B] → [C] → [E] → [F] → [G] → [H] → [I] → [K] → [L] → [M] → [N] → [O] → → [P] → [Q] → [K] → [L] → [M] → [O] → [Q] → [K] → [L] → [M] → [O] → [Q] →

→ [K] → [L] → [M] → [O] → [Q] → [L] → [M] → [O] → [Q] → [K] → [L] → [M] → → [O] → [Q] → [K] → [L] → [M] → [O] → [Q] → [K] → [L] → [M] → [O] → [Q] → → [K] → [J] → [H] → [Кінець]).

Виходячи з наведеного, можна зробити висновок, що проходження тестів 1–3 забезпечує виконання критерію С1.

Розглянемо третій критерій – покриття всіх шляхів (С∞). Проаналізувавши граф потоку керування, можна зробити висновок, що неможливо подати такі вхідні дані, які забезпечували би прохід по шляху ([Початок] → → [A] → [B] → [C] → [E] → [F] → [G] → [H] → [I] → [K] → ([J] → [H] → [I] → → ([K]) → [L] → [M] → [O] → [P] → [Q]) → [K] → [J] → [H] → [Кінець]).

Більш того, оскільки в цьому шляху є цикли (які позначено дужками), виходить, що наведено не один недосяжний шлях, а кілька таких шляхів.

Проаналізуємо ділянку програмного коду, що унеможливлює проходження по знайденому шляху:

```
for(j=1 ; j<=nb ; j++)
{b=fgetc(BMP);
if( x-x0<Sagolovok.biWidth) {putpixel(x,y,Color[b>>4]); x++;}
if( x-x0<Sagolovok.biWidth) {putpixel(x,y, Color[b&15] ); x++;}
}
```

Якщо не виконується перший *if*, то другий також не буде виконуватися, тому що перевіряються однакові умови. Це підтверджує неможливість знаходження початкових умов, які б забезпечили проходження цього шляху.

Таким чином, виходячи з критерію С∞, програма, що тестується, містить помилку. Знайдений дефект не впливає на відповідність специфікації програми, тобто його не буде виявлено методами функціонального тестування, але буде вплив на швидкість виконання програми, тому що в деяких випадках можливе виконання до трьох зайвих циклів («холостий хід» програми).

Іншою можливістю подання внутрішньої логіки програми є граф потоку даних [3]. Розглянемо тестування і межі його застосування більш докладно.

Тестування потоків даних програми. Групу критеріїв цього методу побудовано на принципах досяжності й доступу до змінних.

Будемо вважати, що при використанні виразу $y = f(x_1, ..., x_n)$:

− *c-use* – це найменування функції, що означає використання зміних $x_1, ..., x_n$ для обчислювального процесу (Computational-use);

− *def* – це найменування функції, що означає наявність виразу, який визначає змінну *y*.

Будемо вважати, що у виразі $pr(x_1, …, x_n)$ використовуються змінні $x_1, ..., x_n$ як предикати (Predicate-use). Позначимо це як *p-use*.

Шлях $p = (n_{in}, ..., n_m)$ уважається таким, що не містить визначень змінної *x*, якщо в ньому немає *def(x)*. Для вершини n_i і змінної *x*, такої, що

$x \in def(n_i)$, вираз $dcu(x,n_i)$ означає набір всіх вершин n_j, таких, що $x \in c\text{-}use(n_j)$ і шлях від n_i до n_j не містить визначень x. Для вершини n_i та змінної x, такої що $x \in def(n_i)$, вираз $dpu(x,n_i)$ означає набір усіх ребер (n_j,n_k), таких що $x \in p\text{-}use(n_j ,n_k)$ і шлях від n_j до n_k не містить визначень x.

Позначимо як *du-шлях* (шлях визначення використання) один із двох шляхів:

– $p = (n_{i+1},... n_j, n_k)$, що містить глобальне визначення змінної x у вершині n_i, і такий, що p не містить визначень змінної x, але містить $c\text{-}use(x)$, і всі вершини $n_i, \ldots, n_k$ є попарно відмінними;

– $p(n_{i+1},... n_j, n_k)$, що не містить визначень змінної x, але містить предикатне використання x $(p\text{-}use(x))$, і всі вершини $n_i, \ldots, n_k$ є попарно відмінними.

Розглянемо критерії тестування потоків даних програми.

Критерій all-defs потребує створення набору тестів, які б містили для кожної вершини n_i і кожної змінної $x \in def(n_i)$ щонайменше один шлях, що не містить визначень x від n_i до елемента з $dcu(x, n_i)$ або $dpu(x, n_i)$. Критерій є забезпеченням перевірки правильної ініціалізації змінних, але не є гарантією їх правильного використання в обчислювальному процесі. Це завдання ставиться поетапно перед такими критеріями.

Критерій all p-uses. За цим критерієм потребується створення набору тестів, які б містили для кожної вершини n_i і кожної змінної $x \in def(n_i)$ щонайменше один шлях, який не містить визначень x від n_i до всіх елементів з $dpu(x, n_i)$.

Критерій all c-uses потребує створення набору тестів, які б містили для кожної вершини n_i і кожної змінної $x \in def(n_i)$ щонайменше один шлях, який не містить визначень x від n_i до елемента з $dcu(x, n_i)$.

Ці два критерії є гарантією перевірки того факту, що використовувані змінні в предикатних виразах і в обчисленнях було проініціалізовано.

Критерій all c-uses/some p-uses полягає у вирішенні завдання: чи використано всі ініціалізовані змінні в обчисленнях або хоча б у предикатних виразах. Критерій all c-uses/some p-uses потребує створення набору тестів, які б містили для кожної вершини n_i і кожної змінної $x \in def(n_i)$ щонайменше один шлях, що не містить визначень x від n_i до всіх елементів з $dcu(x, n_i)$, або якщо $dcu(x, n_i)$ – порожня множина, або шлях, що не містить визначень x від n_i до елемента з $dpu(x, n_i)$.

Критерій all p-uses/some c-uses є аналогічним попередньому, з тією різницею, що в ньому перевіряються входження шуканої змінної в предикатні вирази, а якщо таких не виявлено, то в обчислення. Цей критерій потребує створення набору тестів, які б містили для кожної вершини n_i і кожної змінної $x \in def(n_i)$ щонайменше один шлях, який не містить визначень x від n_i до всіх елементів з $dpu(x, n_i)$, або якщо $dpu(x, n_i)$ – порожня множина, або шлях, що не містить визначень x від n_i до елемента з $dcu(x, n_i)$.

Критерій all uses (усі використання) є узагальненим для останніх

двох критеріїв і потребує створення набору тестів, які б містили для кожної вершини n_i і кожної змінної $x \in def(n_i)$ щонайменш один шлях, який не містить визначень x від n_i до всіх елементів з $dpu(x, n_i)$ і до всіх елементів з $dcu(x, n_i)$.

Критерій all du-paths забезпечує найбільшу повноту покриття при цьому враховуються всі можливі використання певної змінної. Цей критерій потребує створення набору тестів, які б містили для кожної вершини n_i і кожної змінної $x \in def(n_i)$ усі *du-шляхи* цього визначення.

Усі критерії структурного тестування узагальнено й наведено у вигляді ієрархічної структури (рис. 3.4), яка дає змогу виявити наявні взаємозв'язки між критеріями.

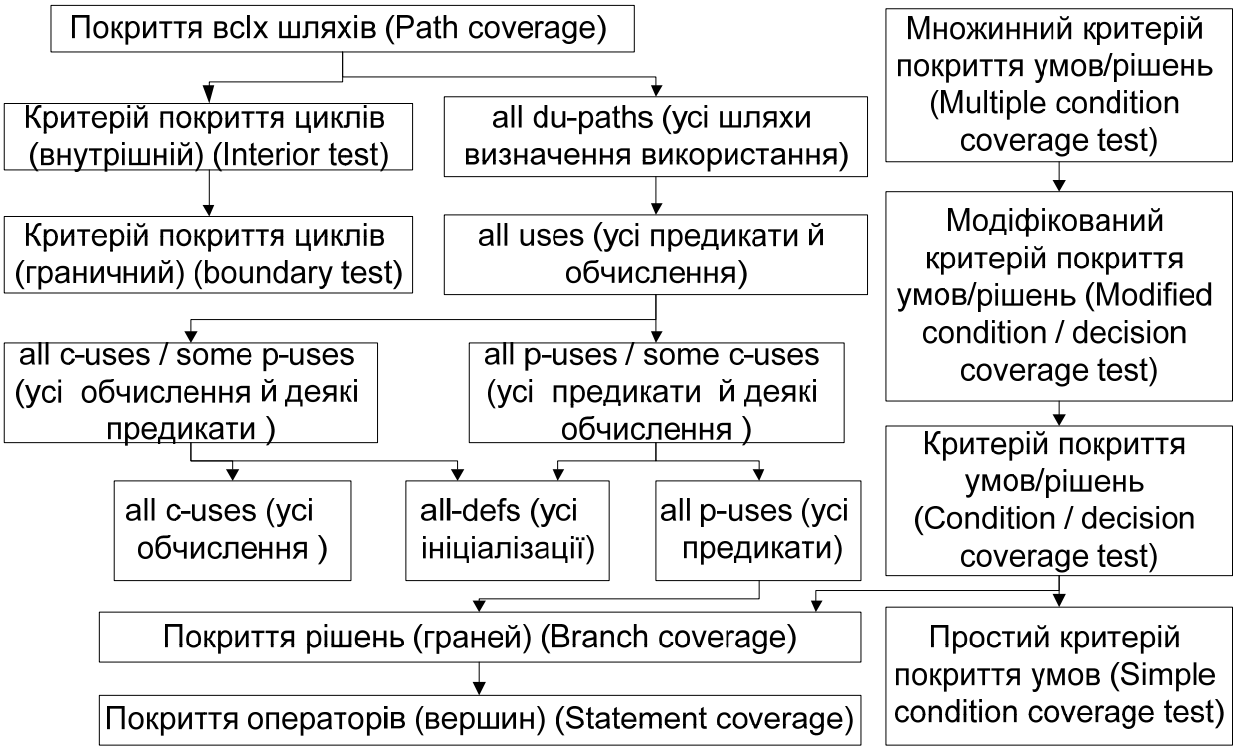


Рис. 3.4. Взаємозв'язок між критеріями структурного тестування

Взаємозв'язки між критеріями, у яких використовуються потоки керування програми, було вже проаналізовано вище, а критерії потоків даних позв'язані таким чином: найширшим є критерій all du-paths, виконання якого забезпечує виконання критерію all uses який, зі свого боку є об'єднанням критеріїв all p-uses / some c-uses і all c-uses / some p-uses Кожний з останніх двох забезпечує виконання критерію all-defs. При цьому, виходячи з означення, критерій all p-uses / some c-uses є гарантією виконання критерію all p-uses, а критерій all c-uses / some p-uses – виконання критерію all c-uses.

Очевидно, що покриття всіх шляхів забезпечить виконання критерію all du-paths, а критерій all p-uses перевірить проходження всіх переходів (усіх рішень).

Приклад 1. Проілюструємо будування графа потоку даних. Розглянемо функцію:

```
/* знаходження мінімального й максимального числа з двох наданих*/
    void MinMax (int& Min, int& Max)
    {int Hilf;
    if (Min > Max) {
        Hilf = Max; Max = Min; Min = Hilf;
        }
    }
```

Граф керування матиме вигляд, наведений на рис. 3.5. Для кожного вузла зазначено відповідність типовим структурам потоку даних (визначення, предикатні використання, використання в обчислювальному процесі).

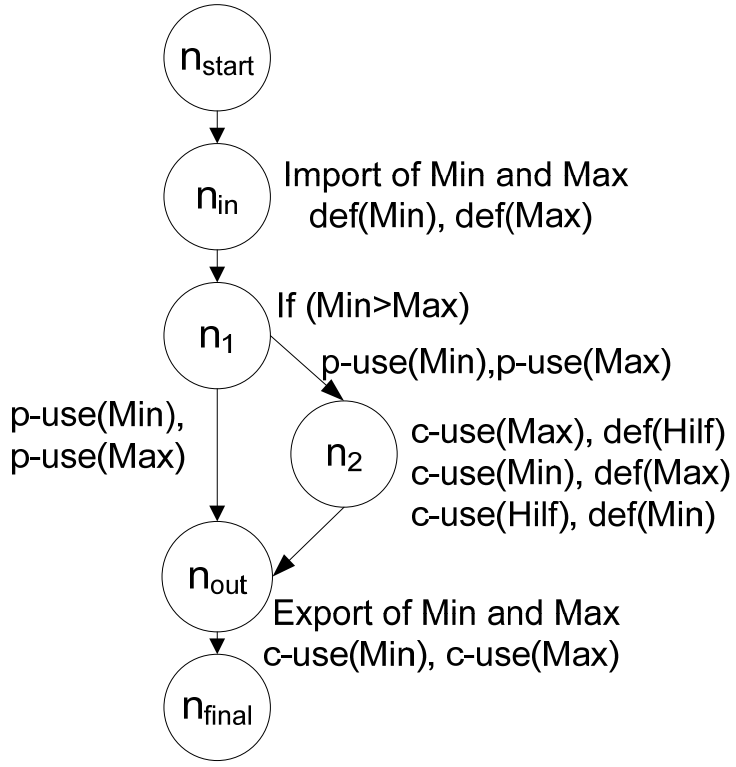


Рис. 3.5. Граф потоку даних

Розглянемо відповідності між вершинами потоку керування й потоку даних (табл. 3.2).

Таблиця 3.2

Вершина n_i	def(n_i)	c-use(n_i)
n_{in}	{Min, Max}	{ }
n_1	{}	{ }
n_2	{Min, Max}	{Min, Max}
n_{out}	{ }	{Min, Max}

Як було зазначено вище, предикатне використання відповідає ребрам графа (табл. 3.3).

Таблиця 3.3

Ребро n_i	p-use(n_i ,n_j)
(n_1, n_2)	{Min, Max}
(n_1, n_{out})	{Min, Max}

Побудуємо шляхи визначення використання (предикатне і в обчислювальному процесі (табл. 3.4).

Таблиця 3.4

Змінна x	Вершина (n_i)	dcu(x,n_i)	dpu(x,n_i)
Min	n_{in}	{n_2, n_{out}}	{(n_1, n_2), (n_1, n_{out})}
Min	n_2	{n_{out}}	{}
Max	n_{in}	{n_2, n_{out}}	{(n_1, n_2), (n_1, n_{out})}
Max	n_2	{n_{out}}	{}

Отже, було отримано всі необхідні дані для проведення структурного тестування на основі потоку даних.

Приклад 2. Проілюструємо різницю в роботі критеріїв тестування на основі потоку керування і потоку даних. Нехай дано такий фрагмент коду:

public int f(int a, int b, String c)

{…if (a > 0) c = null;… if (b < 0) b = c.length(); return b; }

Відповідний граф потоку керування і даних зображено на рис. 3.6.

Для забезпечення критерію покриття операторів необхідно пройти два шляхи: 1-2-3-4-5-6-8 і 1-2-3-5-6-7-8. Для критерію покриття рішень досить цих самих шляхів.

Умови виконано, проте можливою є така ситуація: у вершині 4 є нове визначення змінної *c*, яка використовується потім в вершині 7. У зазначеному переліку тестів, незважаючи на те, що вони задовольняли критерії, немає тесту, щоб містив одночасно проходження по вершинах 4 і 7. З таким завданням може справитися, наприклад, критерій all-uses (1-2-3-4-5-6-7-8). Отже, тестування на основі потоку керування і потоку даних є взаємодоповнювальними.

Мутаційне тестування є різновидом тестування білого ящика, для його здійснення необхідно мати доступ до вихідного коду програми.

Мутаційний критерій ґрунтується на штучному внесенні помилок до програми. У мутаційному критерії припускається, що програмісти пишуть майже коректні програми, що відрізняються від правильних незначними помилками в арифметичних операціях, перестановками індексів, некорек-

тними границями циклів, хибними значеннями констант та ін.

Для виправлення дефектів такого роду до програми вносяться дрібні помилки (мутації). Програми, що відрізняються від вихідних програм штучно внесеними помилками, називають мутантами. Зазвичай мутант відрізняється від вихідної програми невеликою кількістю мутацій. У вихідній програмі можуть піддаватися мутаціям ділянки коду, пов'язані з переліченними вище помилками (змінення значень змінних, модифікація індексів і границь циклів, внесення мутацій в умови).

Таким чином, з початкової програми шляхом внесення *n* мутацій одержують *k* мутантів, $n \geq k$ (щонайменше одна мутація на одного мутанта). Якщо сформована множина тестових наборів виявляє всі мутації у всіх мутантах, то вона відповідає мутаційному критерію. Якщо тестування вихідної програми і мутанта на заданій множині тестових наборів не виявило помилок, то програма оголошується еквівалентною мутанту. У випадку мутаційного тестування важливо створити таку кількість мутантів, щоб охопити всі можливі ділянки, де можуть виявитися помилки.

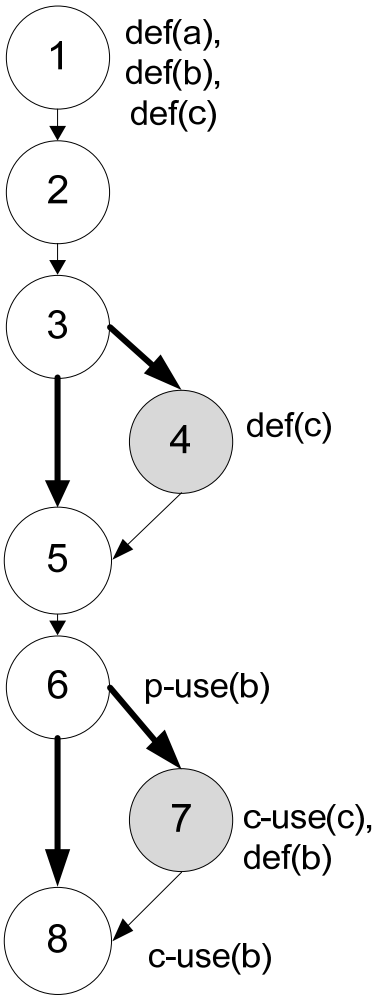


Рис. 3.6. Граф потоку керування з потоком даних

Уважається, що на основі дрібних помилок можна оцінити загальну кількість помилок, що залишилися в програмі.

Приклад застосування мутаційного критерію. Нехай необхідно протестувати програму P:

```
// З допомогою методу обчислюється невід‘ємний степінь n числа x
double PowerNonNeg(double x, int n)
    {double z=1;
    if (n>0) for (int i=1; n-1>=i;i++) {z = z*x;}
    else
    printf("Помилка! Степінь числа n має бути більше нуля.");
    return z;}
```

Для програми P створюються дві програми-мутанти P1 і P2. У програмі P1 змінено початкове значення змінної *z* з 1 на 2:

```
double PowerMutant1(double x, int n)
    {double z=2;
```

```
if (n>0) for (int i=1;n>=i;i++){z = z*x;}
else
printf("Помилка! Степінь числа n має бути більше нуля.");
return z;}
```

У програмі Р2 змінено початкове значення змінної з 1 на 0 і граничне значення індексу циклу з *n* на *n–1* (лістинг 3):

```
double PowerMutant2(double x, int n)
{double z=1;
if (n>0) for (int i=0; n-1>=i;i++){z = z*x;}
else
printf("Помилка! Степінь числа n має бути більше нуля.");
return z;}
```

При запуску тестів (X,Z)={(x=2,n=3,z=8),(x=999,n=1,z=999), (x=0,n=100,z=0)} виявляються всі помилки в програмах-мутантах і помилка в основній програмі, де в умові циклу замість *n* стоїть *n–1.*

3.3.3. Техніка тестування

Існує багато прийомів тестування ПЗ. Деякі з них перевершують інші, деякі можна використовувати разом з іншими для кращих результатів. Основні прийоми тестування:

– ручне тестування – тести виконуються людьми з наперед складеними або визначеними для кожного випробування тестовими даними;

– автоматизоване тестування – тести виконуються з допомогою спеціальних інструментів або можуть бути самостійними процесами і повторюватися багато разів; тестові дані попередньо вводяться або генеруються;

– регресивне тестування – тести (зазвичай автоматизовані) призначено для виявлення негативного впливу змін у програмі на функції, що пройшли попередні перевірки;

– димове тестування – тести, спрямовані на швидку перевірку базової функціональності з метою виявлення, чи вартий тестування новий білд (версія програми або її певного модуля);

– дослідне тестування – тести виконуються за відсутності специфікацій; тестувальник розроблює власну систему тестування, яка базується на накопиченому ним досвіді, та оцінює ризики, створюючи сценарії тестування;

– «мавпяче» тестування – тести не мають під собою певної системи, «швидка атака» програми тестувальником;

– стрес-тестування – тести призначено для перевірки стійкості програми до надмірного навантаження у разі нестачі ресурсів;

– тестування навантаження – тести виконуються при різних рівнях навантаження з метою перевірки поведінки програми й виявлення макси-

мально дозволеного рівня;

– тестування продуктивності – тести виконуються для порівняння поточної продуктивності з розрахунковою;

– тестування інсталяції – тести полягають у встановленні програми на різних платформах-комбінаціях і перевірки того, чи всі файли було переписано, чи коректно працює програма;

– тестування довгим використанням – тести виконуються довготривало, з метою виявлення такого роду помилок, які неможливо виявити при нетривалому використанні.

Існують кілька аксіом:

– неможливо повністю протестувати програму;

– тестування – це процес, що містить ризики; тестувальнику необхідно навчитися зменшувати область усіх можливих тестів до керованого набору та приймати, враховуючи ризик, розумні рішення: що важливо для тестування, а що ні; на рис. 3.7 проілюстровано залежність між обсягом тестування і якістю проекту стосовно кількості знайдених помилок;

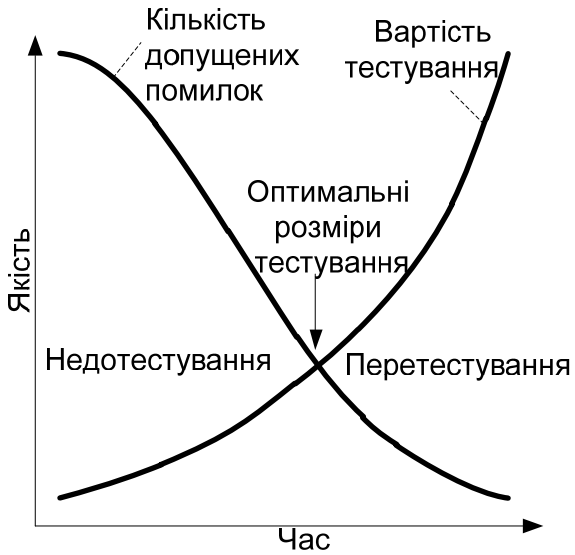


Рис. 3.7. Залежність між обсягом тестування і якістю проекту

– тестування не є показником того, що помилок немає;

– чим більше помилок знаходить тестувальник, тим більше їх існує.

Контрольні запитання

1. У чому полягає тестування за класами еквівалентності?
2. Що таке структурне тестування?
3. У чому полягає тестування потоку керування програм?
4. У чому полягає тестування потоку даних програм?
5. Як пов‘язані між собою критерії структурного тестування?
6. Що таке мутаційне тестування?
7. Яке основне припущення є основою методу мутаційного тестування?
8. Як з допомогою мутаційного тестування оцінити кількість помилок, що залишилися в програмі?

3.4. Випробування і супроводження програмних продуктів. Інтеграційне тестування компонентно-базованого ПЗ

Супроводження ПЗ – процес покращання, оптимізації та виправлення дефектів у ПЗ після його введення до експлуатації.

Супроводження відповідно до стандартів ISO/IEC 12207 і ISO/IEC 14764 проводиться з метою виконання і модифікаці ПЗ під час експлуатації за умови збереження його цілісності. Практика створення сучасного ПЗ полягає в тому, що акцент робиться на конструюванні складних компонентів з окремих, вже розроблених модулів. Це має назву компонентно-базованого підходу. Вирізняють чотири основних групи компонентів:

– комерційні компоненти інших постачальників;

– внутрішні компоненти, розроблені для інших проектів;

– внутрішні розроблені компоненти, які було оновлено для повторного використання;

– нові сконструйовані компоненти.

Характеристики компонентно-базованого ПЗ:

– різна природа програмних компонентів;

– повторне використання компонентів;

– використання підмножини наданої функціональності;

– недоступність вихідного коду.

Цей підхід дає змогу зменшити час і витрати на програмування, однак різна природа програмних компонентів, відсутність вихідного коду, використання підмножини наданої функціональності й складності інтеграції є основними проблемами для тестування компонентно-базованого програмного забезпечення, особливо для інтеграційного тестування. Відомо катастрофи літаків Therac-25, Ariane 5, у яких помилки ПЗ були пов'язані саме з повторним використанням попередньо «працюючих» компонентів, тобто було неякісно здійснено інтеграційне тестування.

Як було зазначено вище, для інтеграційного тестування потрібно не тільки згенерувати тестові приклади, а й розробити нові критерії і метрики [6], які дадуть змогу кількісно оцінювати якість цих тестових прикладів.

3.4.1. Критерії і метрики інтеграційного тестування

Основне припущення: усі компоненти попередньо вичерпно тестувалися, тобто модульне тестування вже було зроблено [6].

Критерій покриття операцій інтерфейсу. Синтаксична частина інтерфейсу містить оголошення сервісів, що надає або використовує компонент. Усі ці сервіси має бути протестовано. Тому для кожного інтерфейсу вводиться критерій його покриття: кожну операцію, оголошену в інтерфейсі, має бути протестовано, принаймні, один раз.

Щоб кількісно оцінити ступінь досягнення цього критерію, використо-

вують таку метрику:

$$\text{Покриття\_інтерфейсу}_i = \frac{\text{Кількість\_виконаних\_операцій\_в\_Інтерфейсі\_і}}{\text{Загальна\_кількість\_операцій\_в\_Інтерфейсі\_і}}.$$

Вважатимемо, що інтерфейс є покритим, якщо покрито всі операції, оголошені в ньому.

Тоді метрика для програмного продукту в цілому має такий вигляд:

$$\text{Покриття\_інтерфейсів\_ПЗ} = \frac{\sum_{i=1}^{N} \text{Покриття\_інтерфейсу}_i}{N},$$

де *N* – загальна кількість інтерфейсів.

Цей критерій дає змогу визначити можливість здійснення виклику операції, оголошеної в інтерфейсі компонента.

Критерій покриття інтерфейсів не є досить репрезентативним з таких причин:

– його вже має бути досягнуто на фазі модульного тестування;

– у ньому немає різниці між викликами, що виходять із різних компонентів.

З урахуванням цієї інформації визначимо **критерій покриття викликів операцій**. Для кількісного оцінювання ступеня досягнення цього критерію використовують таку метрику для кожної операції:

$$\text{Покриття\_викликів\_операцій} =$$

$$= \frac{\text{Кількість\_протестованих\_різних\_викликів}(s_{j,k})}{\text{Загальна\_кількість\_різних\_викликів}(s_{j,k})\text{\_які\_потрібно\_протестувати}}$$

де $s_{j,k}$ – сервіс, оголошений у *k*-му компоненті системи, *j* = 1...m_k, де m_k – кількість сервісів, оголошених у *k*-му компоненті.

Виклики операцій можна подати як повідомлення в UML-діаграмах взаємодії. Тоді цей критерій можна сформульовати таким чином: для кожної діаграми взаємодії у системі, що містить різні об'єкти *Ob1* і *Ob2*, необхідно протестувати повідомлення *m* хоча б один раз під час інтеграційного тестування, якщо це повідомлення безпосередньо з'єднує *Ob1* і *Ob2*, *Ob1≠Ob2*.

Критерій покриття викликів операцій дає змогу перевірити виклики операції з різних компонентів, що фактично існують у системі. Активізація інтерфейсів з різних компонентів з допомогою різних подій може мати різні наслідки.

З урахуванням цієї інформації визначимо **критерій покриття активізації інтерфейсу.** У кожній активізації інтерфейсу беруть участь два компоненти: викличний і відповідний. Нехай кожен компонент характеризується попереднім і наступним станом. Уважаємо, що дві активізації є однаковими, якщо збігаються викличні й відповідні компоненти, і всі чотири стани також є однаковими. Відповідно до цього означення однаковості активізації можна поділити на класи еквівалентності.

Щоб перевірити можливу поведінку кожного інтерфейсу під час роботи, необхідно протестувати кожний клас активізацій, принаймні, один раз.

Для того щоб оцінити кількісно ступінь досягнення цього критерію, використовують таку метрику:

$$\text{Покриття\_активізацій}_{i,j} = \\ = \frac{\text{Кількість\_протестованих\_різних\_активацій}(\text{Інтерфейс}_i, \text{операція}_j)}{\text{Загальна\_кількість\_класів\_еквівалентності}(\text{Інтерфейс}_i, \text{операція}_j)}.$$

Активізації зі своїми відповідними станами можна подати як переходи в діаграмах стану UML. Необхідно зауважити, що тригери вже було протестовано на фазі модульного тестування (виходячи з основного припущення інтеграційного тестування), а дії необхідно протестувати інтеграційно.

Критерій покритя активізацій інтерфейсу дає змогу здійснити перевірку подій, що відбуваються між будь-якими двома компонентами з урахуванням контексту даних.

Метрика відповідностей викликів і активізацій. Як було зазначено вище, *виклики* операцій можна подати як повідомлення *mk* в діаграмах взаємодії UML, при цьому активізації інтерфейсу характеризуються двома компонентами – викличним і відповідним, а кожен компонент характеризується попереднім і наступним станами. Активізації з відповідними станами можна подати як *переходи* в діаграмах стану UML. Метрика $\mu(mk)$ означає кількість можливих комбінацій між викликами й переходами, що відповідають викликам. Чим вище значення $\mu(mk)$, тим більше потрібно тестів. Метрику $\mu(mk)$ можна обчислити на найперших фазах проектування програмного забезпечення, при цьому рекомендується використовувати ті компоненти, які забезпечують мінімальне значення $\mu(mk)$.

Зауважимо, що цю метрику тестовий менеджер може використовувати для прийняття рішення про вибір того чи іншого компонента серед функціонально еквівалентних на найперших етапах проектування ПЗ.

Критерії покриття послідовностей. Наведені вище критерії є гарантією того, що кожний різновид взаємодії між компонентами (активізації, виклики й т. д.) буде виконано, принаймні, один раз. Однак під функціону-

ванням компонентно-базованого програмного забезпечення передбачається взаємодія сукупності елементів, тому порядок такої взаємодії може бути значущим. Розглянемо критерії для врахування порядку взаємодії.

Критерій покриття залежностей Для того щоб обробити інформацію про порядок взаємодій, введемо поняття відношення залежності: активізація *Inv2* пов'язана відношенням залежності з активізацією *Inv1*, якщо існує шлях (*execution path*), при якому активізація *Inv1* викличе активізацію *Inv2*. Це відношення є рефлексивним, асиметричним і транзитивним.

Будемо вважати, що *Inv2* пов'язана з *Inv1* послідовністю активізацій, що є реалізацією відношення залежності між цими двома активізаціями. Тому критерій покриття залежностей має такий вигляд: кожну послідовність активізацій, що реалізує кожне відношення залежності, має бути протестовано хоча б один раз.

Щоб оцінити кількісно ступінь досягнення ціого критерію, використовують таку метрику:

$$\text{Покриття\_залежностей} = \\ = \frac{\text{Кількість\_протестованих\_різних\_послідовностей\_активизацій}}{\text{Загальна\_кількість\_різних\_послідовностей\_активизацій}}.$$

Для систематизації процесу тестування відношення залежності рекомендується здійснювати його покроково:

– усі активізації має бути протестовано;

– усі функціонально можливі пари активізацій має бути протестовано;

– усі функціонально можливі трійки активізацій має бути протестовано і т. д.

Досягнення повного покриття цього критерію на практиці є дуже ускладненим через велику кількість необхідних тестів. Тому пропонується практичний спосіб вирішення цієї проблеми на основі використання UML-діаграм. Відповідно до цього підходу враховуються тільки фактичні UML-послідовності з діаграм взаємодії, а їх підпослідовності розглядаються у сукупності.

Критерій покриття послідовностей викликів операцій. Діаграми взаємодії UML (collaboration diagram − CD) містять інформацію про порядок взаємодії компонентів у вигляді впорядкованих повідомлень. Використовуючи позначення, уведені в метриці відповідності активізацій і викликів, критерій покриття послідовностей викликів операцій буде визначено таким чином: кожну послідовність повідомлень m_k у кожній діаграмі взаємодій UML має бути протестовано хоча б один раз.

Для того щоб оцінити кількісно ступінь досягнення цього критерію, використовують таку метрику:

$$\text{Покриття\_послідовностей\_повідомлень} =$$

$$= \frac{\text{Кількість\_протестованих\_різних\_послідовностей\_повідомлень}}{\text{Загальна\_кількість\_різних\_послідовностей\_повідомлень}}.$$

Цей критерій дає змогу протестувати послідовності повідомлень, що фактично існують у системі, його легко реалізувати на практиці, але, на жаль, у ньому не враховується контекст даних.

Критерій покриття послідовностей активізацій. З метою урахування контексту даних послідовності повідомлень із діаграм взаємодії слід доповнити інформацією про відповідні стани з діаграм станів.

Використовуючи позначення, уведені в матриці відповідності активізацій і викликів, критерій покриття послідовностей активізацій буде визначено таким чином: кожну послідовність активізацій m_k у кожній діаграмі взаємодій має бути протестовано хоча б один раз.

Для того щоб оцінити кількісно ступінь досягнення цього критерію, використовують таку метрику:

$$\text{Покриття\_послідовностей\_активізацій} =$$

$$= \frac{\text{Кількість\_протестованих\_різних\_послідовностей\_активізацій}}{\text{Загальна\_кількість\_різних\_послідовностей\_активізацій}}.$$

Цей критерій є компромісним між критеріями покриття залежностей і покриття послідовностей викликів операцій, у ньому враховується контекст даних, розглядаються фактичні послідовності активізацій, але не досліджується окремо кожна підпослідовність, що дає змогу полегшити реалізацію цього критерію на практиці.

Критерій покриття паралельних потоків. На практиці особливу увагу необхідно приділяти паралельному виконанню послідовностей повідомлень. Усі функціонально можливі комбінації виконання повідомлень у паралельних потоках має бути випробувано під час інтеграційного тестування. Повідомлення всередині кожного потоку мають виконуватися послідовно, відповідно до їх порядку в діаграмі взаємодії.

Отже, критерій покриття паралельних потоків буде визначено таким чином: для кожної діаграми взаємодії CD кожну функціонально можливу комбінацію виконання повідомлень у паралельному потоці має бути протестовано хоча б один раз.

Для того щоб оцінити кількісно ступінь досягнення цього критерію, використовують таку метрику:

– для кожного паралельного потоку в діаграмі взаємодії CD:

$$\text{Покриття\_паралельного\_потоку}_i =$$

$$= \frac{\text{Кількість\_протестованих\_різних\_комбінацій}}{\text{Загальна\_кількість\_функціонально\_можливих\_різних\_комбінацій}};$$

– для діаграми взаємодії CD і всіх паралельних потоків у CD:

$$\text{Покриття\_паралельних\_потоків} = \frac{\sum_{i=1}^{N} \text{Покриття\_паралельного\_потоку}_i}{N},$$

де N – загальна кількість паралельних потоків у діаграмі взаємодії.

Тестування відповідно до цього критерію дає змогу виявити помилки, характерні для паралельного виконання процесів.

Ієрархія й відповідність між критеріями інтеграційного тестування. Наведені вище критерії не є повністю незалежними і можуть бути зв'язані в ієрархічну структуру, показану на рис. 3.8 [6].

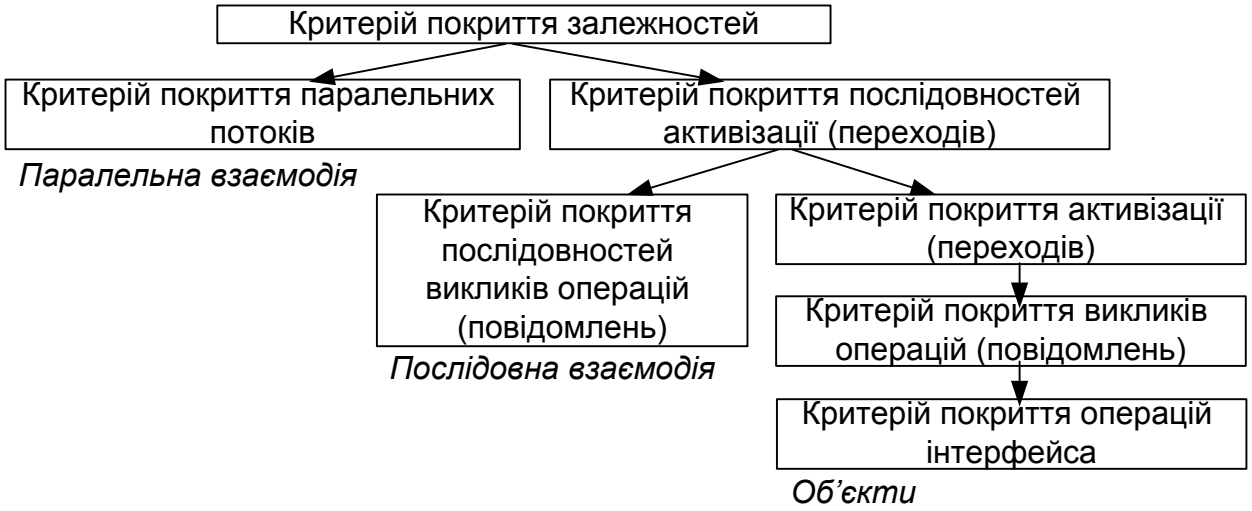


Рис. 3.8. Ієрархія зв'язків між запропонованими критеріями

Розглянемо докладніше, виконання яких критеріїв автоматично гарантує досягнення інших. Виходячи з нотації UML, основними досліджуваними об'єктами є операції, повідомлення й переходи, взаємодія між якими може бути послідовною і паралельною.

Критерії покриття для об’єктів:

– операцій інтерфейсу;

– викликів операцій;

– активізацій інтерфейсу.

Критерії покриття для послідовної взаємодії:

– послідовностей викликів операцій;

– послідовностей активізацій.

Критерії покриття для паралельної взаємодії:

– паралельних потоків і такий, що містить усі види взаємодії:

– залежностей.

Ці критерії можна ієрархічно впорядкувати: 100 % досягнення критерію покриття залежностей є гарантією виконання критеріїв послідовностей

активізацій і паралельних потоків.

Відповідно до означення критерію покриття залежностей усі функціонально можливі послідовності активізацій інтерфейсів, що реалізують відношення залежності, має бути протестовано хоча б один раз. Це означає, що UML-послідовності з відповідними переходами має бути протестовано (як послідовні, так і паралельні).

Стовідсоткове досягнення критерію покриття послідовностей активізацій є гарантією виконання критерію покриття послідовностей викликів операцій і покриття активізацій.

Відповідно до означення критерію покриття послідовностей активізацій усі UML-послідовності має бути протестовано з урахуванням інформації про відповідні переходи. Отже, критерій покриття послідовностей викликів операцій є частиною критерію покриття послідовностей активізацій. Аналогічно, виходячи з означення активізації, наведеного вище, при повному досягненні критерію покриття послідовностей активізацій буде враховано всі активізації, а отже, і відповідний критерій.

Стовідсоткове досягнення критерію покриття активізації інтерфейсу є гарантією виконання критерію покриття викликів операцій. Відповідно до означення, активізація являє собою виклик операції, у якому беруть участь два компоненти: викличний і відповідний, тому під час тестування активізації враховуються виклики операцій.

Стовідсоткове досягнення критерію покриття викликів операцій є гарантією виконання критерію покриття операцій інтерфейсу. У критерії покриття викликів операцій враховуються виклики операцій з різних компонентів, а в критерії покриття операцій інтерфейсу – не враховуються, отже, він є частиною критерію покриття викликів операцій.

3.4.2. Практичне дослідження застосування критеріїв інтеграційного тестування

Для аналізу практичного застосування наведених вище критеріїв тестування розглянемо компонентно-базовану систему віддаленої взаємодії користувача, авіакомпанії, туристичного агентства й банку.

Застосування критерію покриття операцій інтерфейсу «Досліджувана система “Base-IT.Com”» складається із чотирьох взаємодійних компонентів: «Банк» (*Bank*), «Авіакомпанія» (*Airline*), «Туристичне агентство» (*TravelAgency*) і «Користувач» (*CustomerInterface*), кожний з яких має свій інтерфейс. Розглянемо відповідну UML-схему компонентів (рис. 3.9).

Як видно з рис. 3.9, компонент «Банк» (*Bank*) має два інтерфейси «Рахунок» (*Account*) і «Банк» (*Bank*). Таке подання є зручним для розбиття наданих сервісів на функціонально зв‘язані групи.

На основі цієї схеми компонентів розробляються тести згідно з критерієм покриття операцій інтерфейсу. Для 100-відсоткового досягнення

необхідно забезпечити виклики всіх функцій, які оголошено в інтерфейсах відповідних компонентів.

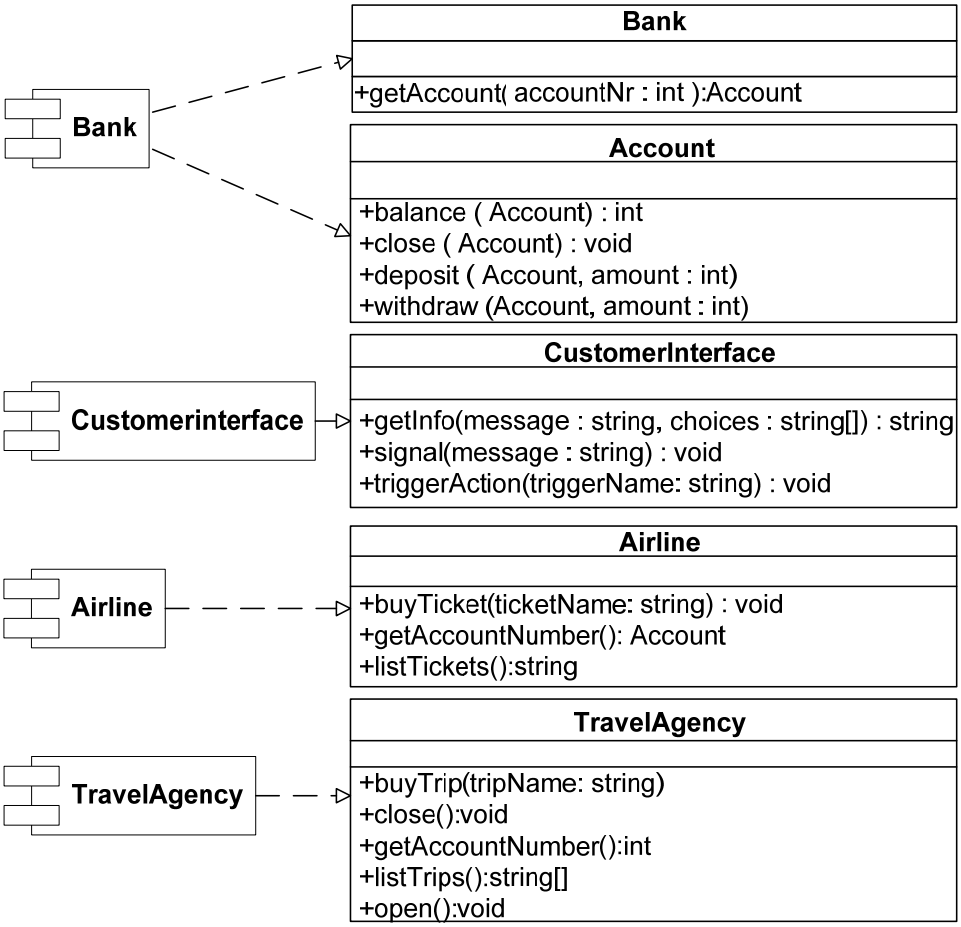


Рис. 3.9. Схема компонентів проектованої системи

Типи тестів. Для компонента «Банк» (*Bank*) має бути створено тести, що перевіряють можливість виклику таких функцій: *getAccount()*, *balance()*, *close()*, *deposit()*, *withdraw(),* для компонента «Авіакомпанія» (*Airline*): *buyTicket()*; *getAccountNumber()*; *listTickets()*, аналогічно – для двох інших компонентів.

Застосування критерію покриття викликів операцій. Розглянемо UML-діаграму взаємодії між Користувачем (*CustomerInterface*) і Авіакомпанією (*Airline*) (рис. 3.10). Користувач запитує в Авіакомпанії інформацію про наявність необхідного рейсу (*listTicket()*), у разі позитивної відповіді надсилає запит на купівлю квитка (*buyTicket()*). Авіакомпанія запитує в клієнта інформацію про номер його банківського рахунку (*getInfo()*), зв'язується з банком і надсилає запит на переказ вартості авіаквитка з рахунку клієнта на рахунок авіакомпанії (*transfer()*). Банк здійснює необхідну операцію з переказу грошей (*withdraw(), deposit()*).

Відповідну діаграму взаємодії показано на рис. 3.10.

Прямокутниками на цій діаграмі позначено об'єкти (екземпляри компонентів), лініями – зв'язки між об'єктами, а стрілочками – повідомлення (суцільною лінією із трикутною стрілкою позначено виклик процедури або іншого вкладеного потоку керування; суцільна лінія з напівстрілкою використовується для позначення асинхронного потоку керування; пунктирною лінією з V-подібною стрілкою позначено повернення з виклику процедури).

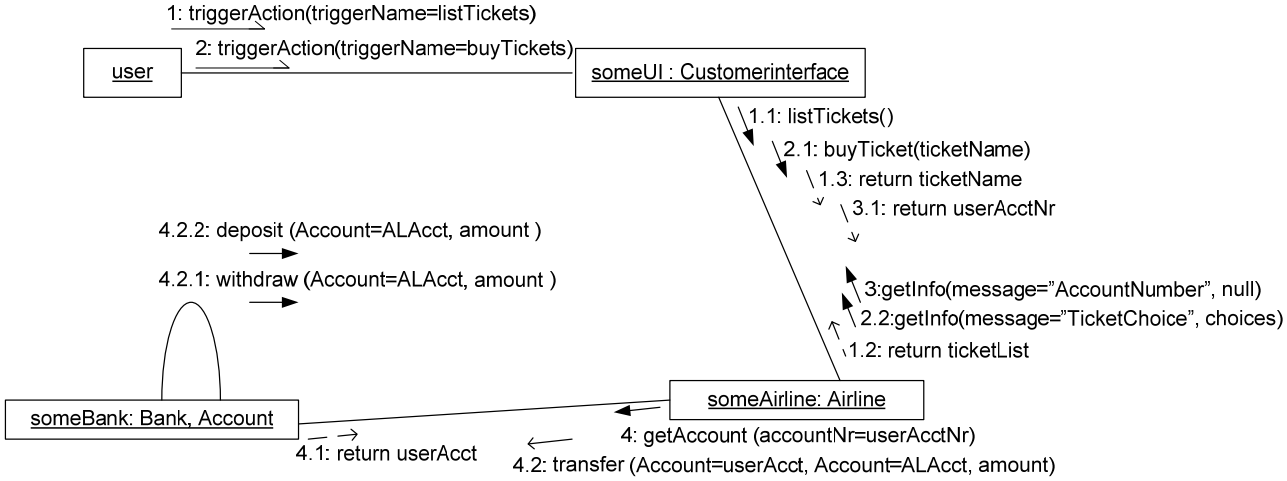


Рис. 3.10. Діаграма взаємодії між користувачем (CustomerInterface) і авіакомпанією (Airline).

На основі цієї діаграми виконується тестування відповідно до критерію покриття викликів операцій. Для досягнення цього критерію в загальному випадку необхідно в кожній діаграмі взаємодії протестувати всі повідомлення, які не є викликами компонента самого до себе.

Типи тестів. З рис. 3.10 видно, що більшість повідомлень у розглядуваному прикладі – це виклики сервісів, які оголошено в компонентах, що відрізняються від викличного. Виключення становлять повідомлення m1=4.2.1 і m2=4.2.2, які є викликами інтерфейсу «Рахунок» компонента «Банк» (*Account, Bank*) до себе самого.

Уважається, що такі виклики було вже перевірено під час модульного тестування й немає необхідності в їх повторному дослідженні. Усі інші повідомлення має бути протестовано для досягнення критерію покриття викликів операцій. Таким чином, спроектовані тести містять можливість виклику повідомлень: 1, 1.1, 1.2, 1.3, 2, 2.1, 2.2, 3, 3.1, 4.4.1, 4.2.

Застосування критерію покриття активізації інтерфейсу. Розглянемо UML-діаграму станів компонента «Авіакомпанія» (рис. 3.11).

Діаграма станів є графом спеціального виду, що являє собою кінцевий автомат. Вершинами цього графа є стани (на діаграмі їх зображено прямокутниками з округленими вершинами), а також псевдостани, які зображуються відповідними графічними символами. У розглядуванному прикладі таким псевдостаном є початковий стан, який являє собою окремий

випадок стану і не містить ніяких внутрішніх дій. У цьому стані перебуває об'єкт у початковий момент часу.

Графічно початковий стан в нотації UML зображується у вигляді зафарбованого кружка, з якого виходить стрілка, що відповідає переходу. Дуги графа позначають переходи зі стану в стан.

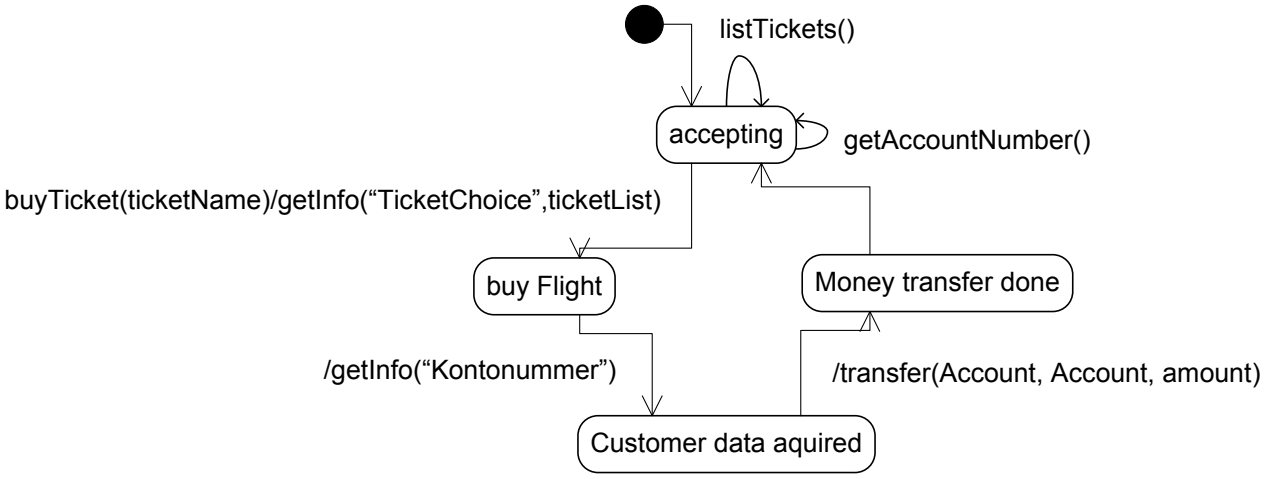


Рис. 3.11. Діаграма станів компонента «Авіакомпанія» (*Airline*)

Компонент «Авіакомпанія» перебуває в початковому стані, з якого виходить при надходженні запиту, наприклад *listTicket()* (запит списку авіаквитків, що є в наявності), при цьому компонент переходить у стан *Accepting* («приймання»). При наступному одержанні запиту на купівлю квитка (*buyTicket()*) компонент із цього стану переходить у стан *buy Flight* («купівля перельоту»), при цьому в користувача запитується інформація про вибраний квиток (*getInfo (TicketChoice, ticketList())*). Зі стану *buy Flight* компонент переходить у стан *Customer data aquired* (Одержання даних користувача) і посилає користувачеві запит про номер його банківського рахунку *getInfo* (*Kontonummer*). Одержавши необхідну інформацію, компонент викликає операцію переказу грошей з рахунку клієнта на свій *transfer* (*Account, Account, amount*) і переходить у стан *Money transfer done*.

Проведемо дослідження застосування критерію покриття активізації інтерфейсу на наведеному прикладі.

Для досягнення цього критерію в загальному випадку у всіх діаграмах стану має бути протестовано всі переходи, для яких дія (*effect*) – непорожня множина.

Розглянемо деякі з наявних переходів (рис. 3.11), беручи до уваги той факт, що здійснюється інтеграційне тестування:

– *Transition1=(Accepting, Accepting, listTickets(),-,-);*

– *Transition2=(Accepting, buy Flight, buyTicket(ticketName), getInfo(“TicketChoice”, ticketList()),-);*

– *Transition3=(Customer data acquired, Money transfer done, -, transfer(Account, Account, amount),-).*

У переході 1 (*Transition 1*) дії немає.

У переході 2 (*Transition 2*) дія *getInfo(“TicketChoice”, ticketList())*, це сервіс, який оголошено в інтерфейсі компонента «Користувач» (*CustomerInterface*).

У переході 3 (*Transition 3*) дія *transfer(Account, Account, amount)*, цей сервіс не оголошено в інтерфейсі жодного з компонентів, проблему має бути вирішено з допомогою спеціалізованих оболонок (*wrapper*) і підбору відповідності імен (*mapping*). У розглянутому прикладі функцію *transfer()* буде реалізовано як послідовність операцій *withdraw()* і *deposit()*, які оголошено в інтерфейсі компонента «Банк» (*Bank*) (див. рис. 3.9).

Типи тестів. Таким чином, відповідно до критерію покриття активізації інтерфейсу тести мають містити можливість запуску всіх переходів наведеної діаграми станів, крім Transition1 = (Accepting, Accepting, listTickets(),-,-) і Transition4 = (Accepting, Accepting, getAccountNumber(),-,-).

Помилки, що виявляються. При тестуванні згідно з цим критерієм виявляється клас помилок, пов'язаних з невідповідностями імен функцій у викличному й відповідно́му компонентах. У розглянутому прикладі компоненти «Авіакомпанія» і «Банк» надаються різними розробниками, отже, проблема є актуальною на сьогодні. Рішення полягає в написанні спеціалізованих оболонок («*wrapper*»), що забезпечують відповідність імен («*mapping*»).

Застосування метрики відповідності повідомлень і переходів. Дослідимо на практиці метрику відповідності повідомлень і переходів. Щоб одержати дані для цієї метрики, необхідно мати відповідні UML-діаграми станів компонентів і діаграми взаємодії. Скористаємося діаграмами, показаними на рис. 3.11 і 3.12.

Найпоширенішою ситуацією є однозначна відповідність між повідомленнями й переходами (викликами функцій і активізаціями інтерфейсу), однак існують виключення.

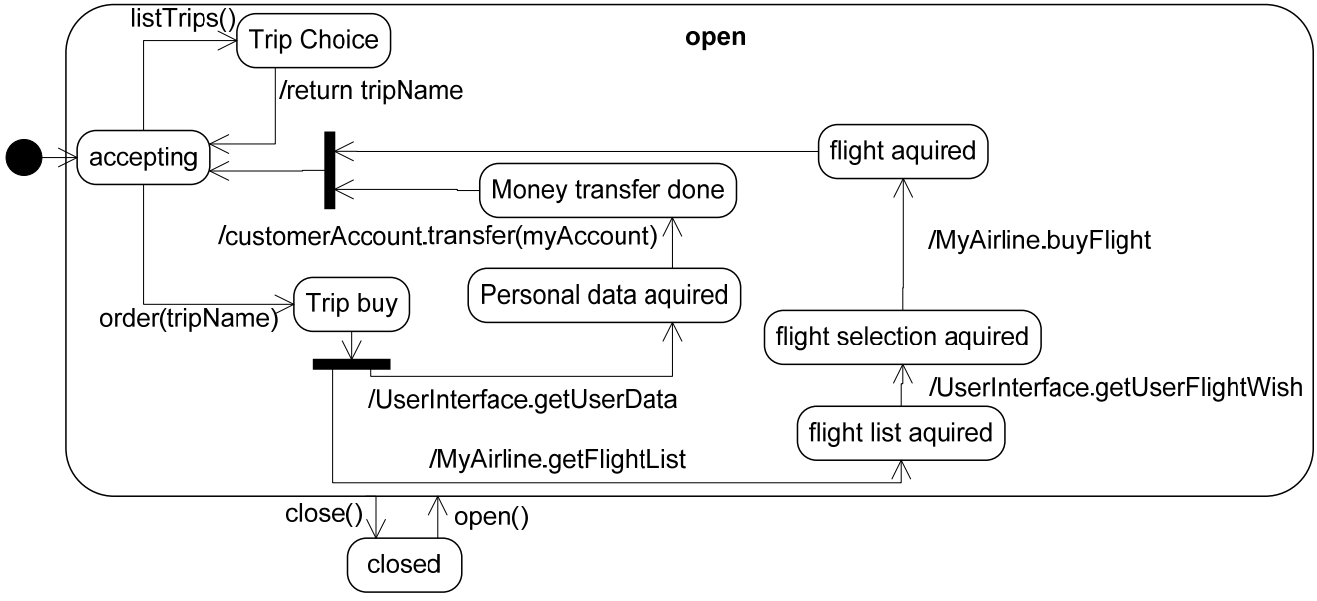


Рис. 3.12. Діаграма станів компонента «Туристичне агентство» (TravelAgency)

На діаграмі станів компонента «Авіакомпанія» (*Airline*) (див. рис. 3.10) є два переходи: якщо *getInfo()*, то *t1=(Accepting, buy Flight, buyTicket(),getInfo(),-); t2=(buy Flight, Customer data acquired, -,getInfo(),-); н*а цій діаграмі показано складений стан *Open*. Якщо компонент одержить повідомлення *close()*, то він може перебувати в кожному з восьми підстанів стану *Open*. У цьому випадку кількість необхідних тестів для повідомлення збільшується у вісім разів. Загальна кількість тестів для всієї діаграми теж зростає. Ця метрика є дуже важливою на етапі вибору компонентів. При однозначній відповідності повідомлень і переходів легше здійснювати процес тестування, а ймовірність виникнення помилок буде меншою. Тому за наявності функціонально еквівалентних компонентів рекомендується вибирати ті, для яких значення наведеної метрики будуть мінімальними. Особливу увагу слід приділяти наявності складених станів, тому що питання моделювання переходу з будь-якого підстану може виявитися нетривіальним завданням.

Застосування критеріїв покриття послідовностей. Вище було наведено критерії для тестування не тільки окремих об'єктів (операцій, викликів і активізацій), але і їх послідовностей. Розглянемо застосування цих критеріїв на досліджуваному проекті.

Застосування критерію покриття послідовностей викликів операцій. Для досягнення цього критерію необхідно протестувати всі послідовності повідомлень у всіх діаграмах взаємодії.

Типи тестів. На UML-діаграмі взаємодії між користувачем (*CustomerInterface*) і авіакомпанією (*Airline*) (див. рис. 3.10) є одна послідовність повідомлень: 1–4.2.2 (нумерацію повідомлень наведено відповідно до стандартів мови UML [11, 18]), яку необхідно протестувати. Для забезпечення виклику цієї послідовності слід ініціювати такі дії: *triggerAction (listTickets)*, *triggerAction (buyTicket)*.

Критерій покриття послідовностей викликів операцій легко реалізується на практиці, але через те, що в ньому не враховується контекст даних, тестування не забезпечує надійність перевірки функціонування. Наприклад, результати виконання послідовності 1–4.2.2 будуть різними, якщо вартість авіаквитка (*amount*) перевищить кількість грошей на рахунку користувача або ціна квитка виявиться меншою, ніж сума на рахунку клієнта. Відповідно до наведеного критерію цю відмінність не буде виявлено, однак вона є важливою для практики; урахування таких даних здійснюється в критерії покриття послідовностей активізацій.

Застосування критерію покриття послідовностей активізацій. При застосуванні критерію покриття послідовностей активізацій враховується інформація про відповідні стани компонентів. Для цього необхідно мати UML-діаграми станів компонентів і діаграми взаємодій.

З допомогою діаграм взаємодій отримують послідовності повідом-

лень, а з допомогою діаграм станів кожному повідомленню ставлять у відповідність переходи у викличному й відповідно̀му компоненті.

Типи тестів. Покажемо відповідності між деякими повідомленнями й переходами з послідовності 1–4.2.2 (див. рис. 3.10 і 3.11):

а) listTickets ():

– викличний компонент: CustomerInterface;

– викличний перехід: t = (Interface ready, Interface ready, triggerAction (listTickets), listTickets(),-);

– відповідний компонент: Airline;

– відповідний перехід: t = (Accepting, Accepting, listTickets(),-,-);

б) getInfo (message = AccountNumber, null):

– викличний компонент: Airline;

– викличний перехід: t = (buy Flight, Customer data acquired, -, getInfo (“AccountNumber”, -);

– відповідний компонент: CustomerInterface;

– відповідний перехід: t = (Interface ready, Waiting for user data, getInfo (“AccountNumber”),-,-);

в) transfer(Account=userAcct, Account=ALAcct, amount):

– викличний компонент: Airline;

– викличний перехід: t = (Customer data acquired, Money transfer done, -, transfer (Account, Account, amount),-);

– відповідний компонент: Bank;

– відповідний перехід: t = (Bank operating, Bank operating, transfer (Account, Account, amount), -,-.).

Для критерію покриття послідовностей активізацій таку відповідність має бути знайдено для кожного повідомлення в кожній послідовності, а тести мають забезпечувати виклик послідовності з усіма можливими активізаціями.

Застосування критерію покриття залежностей. Як було зазначено вище, найбільш повним є критерій покриття залежностей. Для його реалізації недостатньо протестувати всі послідовності повідомлень у всіх діаграмах взаємодії, необхідно також протестувати всі функціонально можливі підпослідовності (з урахуванням відповідних переходів).

Типи тестів. У дослідженій вище послідовності 1–4.2.2 (див. рис. 3.10), можна вирізнити кілька послідовностей із функціонально можливих: 1–1.1; 2–2.2; 4; 3–4.2; 1.1–3; 1.1–2.1. Кожну з них необхідно протестувати з урахуванням відповідних станів (тобто контексту даних).

Застосування критерію покриття паралельних потоків. Критерій покриття паралельних потоків було визначено таким чином: для кожної діаграми взаємодії кожну функціонально можливу комбінацію виконання повідомлень у паралельному потоці має бути протестовано хоча б один раз. Проаналізуємо застосування цього критерію на досліджуваному прикладі.

Типи тестів. На рис. 3.13 зображено UML-діаграму взаємодії між користувачем (*CustomerInterface*), авіакомпанією (*Airline*), банком (*Bank*) і туристичним агентством (*TravelAgency*).

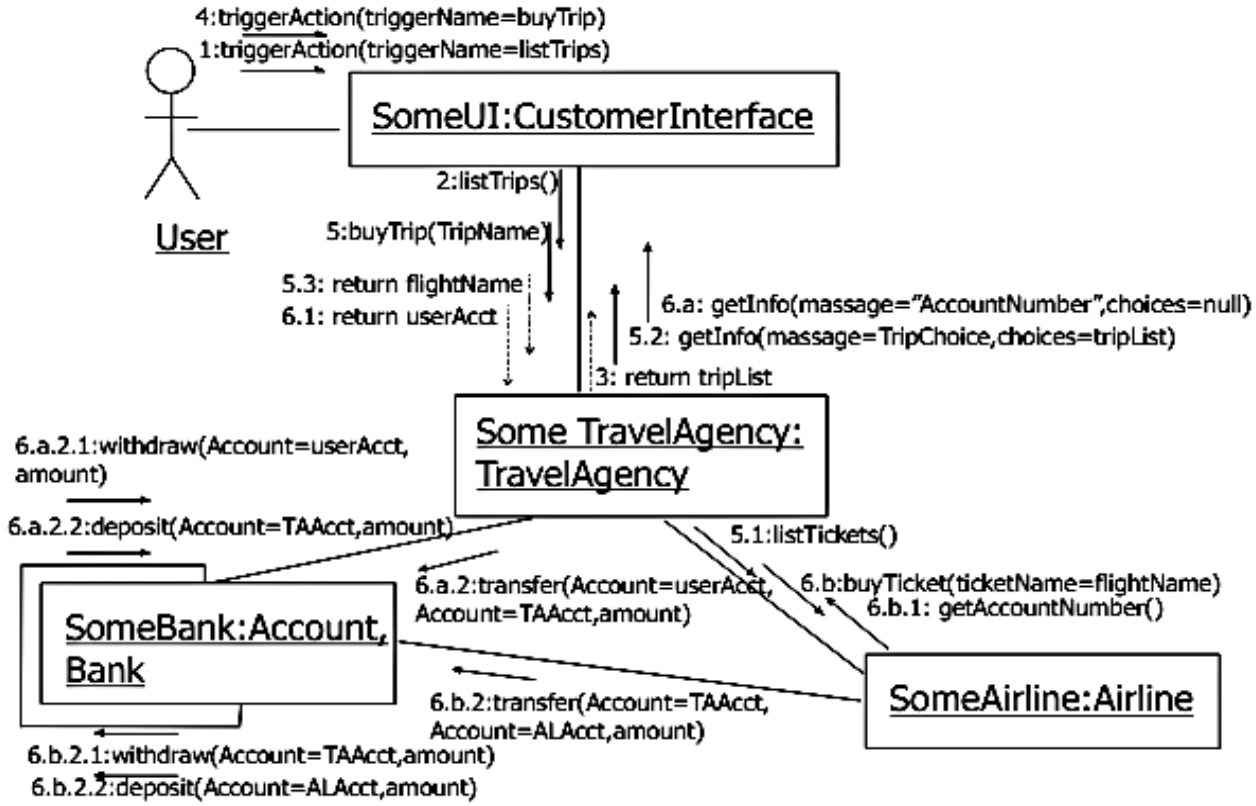


Рис. 3.13. Діаграма взаємодії між користувачем (CustomerInterface) і авіакомпанією

Туристичне агентство замовляє квиток в Авіакомпанії (повідомлення 6.b, 6.b.1) і оплачує його вартість (повідомлення 6.b.2–2–6.b.2.2). Якщо виявляється, що на рахунку клієнта немає необхідної суми, то за цією транзакцією відбувається відкіт, а паралельна транзакція виконується, унаслідок чого туристичне агенство зазнає збитків, пов'язаних з придбанням квитка в Авіакомпанії.

Таким чином, проведення тестування відповідно до цього критерію дає змогу виявити помилки й попередити фінансові втрати.

Оцінювання кількості тестів для інтеграційного тестування. Метою оцінювання є одержання верхніх оцінок кількості тестів, яку необхідно згенерувати для досягнення кожного із критеріїв. Вважаємо, що всі компоненти зв'язані між собою, а це є найбільш складною ситуацією для тестування. Уведемо такі позначення:

– n – кількість компонентів;

– M – максимальна кількість операцій в інтерфейсі компонента;

– t – максимальна кількість переходів у діаграмі станів;

– *L* – максимальна довжина послідовності (у найгіршому випадку може бути оцінена максимальною кількістю повідомлень у діаграмі взаємодій);

– *S* – кількість UML-послідовностей.

Критерій покриття операцій інтерфейсу. Для кожної оголошеної операції має бути згенеровано один тест. Таким чином, кількість тестів визначається як $Q_1 = n$.

Критерій покриття викликів операцій. Тест має бути згенеровано для кожного виклику операції. У найгіршому випадку кожна операція викликається з кожного компонента, крім того, який її містить (оскільки такі виклики вже протестовано на фазі модульного тестування, виходячи з основного припущення інтеграційного тестування). Тоді кількість тестів визначається як $Q_4 = n(n – 1) M$.

Критерій покриття активізацій інтерфейсу. Для кожної активізації інтерфейсу має бути розроблено тест. У кожній активізації беруть участь два компоненти – викличний та відповідний, і дві пари станів – попередній і наступний. Їх можна розглядати як стан-джерело і стан-приймач у відповідних переходах. У найгіршій ситуації всі попарно можливі комбінації переходів із викличного компонента має бути враховано. Переходи вже містять інформацію про операції. Загальна кількість тестів визначається як $Q_3 = n(n – 1) t^2$.

Критерій покриття послідовностей викликів операцій. Відповідно до означення цього критерію всі UML-послідовності має бути протестовано. Для кожної послідовності необхідно розробити свій тест. Таким чином, кількість тестів визначається як $Q_4 = S$.

Критерій покриття послідовностей активізацій. Для кожної UML-послідовності має бути розроблено тест із урахуванням інформації про відповідні переходи. У найгіршому випадку всі попарно можливі комбінації з викличного й відповідного компонентів має бути протестовано. Отже, найгірша оцінка кількості тестів визначається як $Q_{5w} = t^{2L}S$.

З метою одержання більш точних оцінок використовують метрику відповідності між переходами й повідомленнями (викликами й активізаціями). Кількість тестів визначається як $Q_5 = \sum_{j=1}^{J}(\sum_{\forall S \in CD_j} \prod_{\forall m \in S} \mu(m))$, де *S* – UML-послідовності; *m* – повідомлення; CD_j – діаграма взаємодії; *J* – кількість діаграм взаємодії в системі.

Критерій покриття залежностей. Відповідно до означення відношення залежності і її властивостей для кожної окремо взятої активізації і кожної послідовності активізацій необхідно знайти шлях обчислення (*execution path*), який їх реалізує. Для кожного такого шляху необхідно розробити тест. Кількість тестів, необхідних для досягнення цього критерію, є більшою, ніж кількість тестів, необхідних для досягнення критеріїв покрит-

тя активізацій і послідовностей активізацій, тому що враховуються ще й підпослідовності.

У найгіршому випадку всі компоненти є зв'язаними послідовностями й підпослідовностями активізацій. Допустимими є виклики компонентів самих до себе для розглядуваних послідовностей і підпослідовностей.

Тоді, використовуючи покрокове нарощення, можна оцінити кількість тестів, необхідних у такому випадку: усі активізації має бути протестовано: n^2t^2; усі пари активізацій має бути протестовано: n^3t^4; усі функціонально можливі трійки активізацій має бути протестовано: n^4t^6. Цю процедуру необхідно продовжувати до найдовшої послідовності активізацій *L*.

Загальна кількість тестів визначається як $Q_{6w} = \sum_{i=1}^{L} n^{i+1}t^{2i}$.

З метою більш точного оцінювання використовуємо метрику відповідності між переходами й повідомленнями (викликами й активізаціями): кількість тестів визначається як $Q_6 = \sum_{j=1}^{J}(\sum_{\forall S \in CD_j}(\sum_{\forall SS \in S}\prod_{\forall m \in S}\mu(m)))$, де S – UML-послідовності; *SS* – підпослідовності; *m* – повідомлення; CD_j – діаграма взаємодії; J – кількість діаграм взаємодії в системі.

Таким чином, з допомогою наведених оцінок кількості необхідних тестів можна на найперших фазах проектування ПЗ виходячи з розрахунку фінансових і часових витрат на тестування вибрати той критерій, досягнення якого реалізовано на практиці.

Контрольні запитання

1. Наведіть означення терміна «супроводження ПЗ».
2. Від яких параметрів залежить критерій покриття операцій інтерфейсу?
3. Для чого використовується обчислювальний шлях (execution path) у критерії покриття залежностей?
4. Яка кількість тестів необхідна для дослідження відповідності ПЗ критерію покриття послідовностей активізацій?
5. Які характеристики має компонентно-базоване ПЗ?
6. Чим відрізняється однокомпонентне супроводження ПЗ від багатокомпонентного?
7. Які інтерфейси реалізовано для компонентів «Банк» згідно з рис. 3.9? Для яких цілей їх застосовують?
8. Дайте означення терміна «компонент ПЗ».
9. Чому дорівнює критерій покриття викликів операцій для коду з дод. 3?
10. Сформуйте вимогу для технічного завдання, у якій буде передбачено інтеграційне тестування на етапі супроводу ПЗ. До якого розділу ця вимога буде належати?

4. ДОКУМЕНТУВАННЯ І МАРКЕТИНГ ПЗ

4.1. Експлуатаційна, операційна і рекламна документація на ПЗ

Часто вважають, що документація розробляється після створення певного програмного засобу. Однак для якості розроблення програмна документації має бути невід'ємною частиною процесу створення програмного засобу. Для цього потрібно провести доссть важку роботу з планування документації.

4.1.1. Види документації на ПЗ

На ПЗ має складатися експлуатаційна, операційна й рекламна документація англійською або рідною мовою користувача.

Експлуатаційна документація призначена для системних спеціалістів, які встановлюють ПЗ на ЕОМ, і прикладних спеціалістів і програмістів, що використовують ПЗ при розробленні алгоритмів і програмуванні прикладних програм.

Експлуатаційна документація, зазвичай має містити такі документи:

– загальний опис (концепції і можливості);

– інструкція для установки ПЗ в середовищі операційної системи (рідною мовою користувача);

– інструкція для установки використовуваного обладнання (ЕОМ);

– інструкція з експлуатації (використання) ПЗ;

– повідомлення, які видає ПЗ;

– інструкція для запуску контрольного приладу.

Кількість і обсяг документації залежить від складності й компонентного складу документації (опис мови, опис застосування ПЗ, інструкція для оператора та ін.).

Документація має бути написана простою, доступною мовою, особливо якщо її призначено для користувачів персональних комп'ютерів. Документацію розробника призначено насамперед для персоналу, що здійснює розвиток і підтримку цього ПЗ. Якщо передбачається розвиток і супровід такого ПЗ, то документація має містити:

– текстовий опис логіки ПЗ;

– докладні блок-схеми;

– тексти програм на вихідній мові програмування.

При здійсненні будь-якої господарської операції формується **операційний документ**, що підтверджує її здійснення. Сукупність операційних документів утворює документообіг фірми.

В операційних документах вирізняють документи-основи, тобто до-

кументи, які регламентують операції між юридичними особами. До них належать прості й багатоетапні договори, рахунки, рахунки-фактури, контракти, вимоги, гарантійні листи тощо, а також службові записки, протоколи, звіти, листи й т. ін.

Рекламна документація, яку призначено для маркетингу (потреб продавця), має містити основні дані про ПЗ (функціональні можливості, основні характеристики, вимоги до ЕОМ і пам'яті, підтримувані пристрої, умови експлуатації, кількість наявних користувачів, можливості порівняно з аналогічними продуктами). Рекламна документація має складатися з проспектів і загальних описів.

Документація має містити всі необхідні відомості.

4.1.2. Документація, що створюється під час розроблення ПЗ

При розробленні ПЗ створюється великий обсяг різноманітної документації, яка є засобом передання інформації між розроблювачами ПЗ, керування розробленням ПЗ і передання користувачам інформації, необхідної для застосування й супроводження ПЗ. На створення цієї документації припадає більша частина вартості ПЗ.

Цю документацію можна розбити на дві групи:

– документи, що стосуються керування розробленням ПЗ;

– документи, що належать до складу ПЗ.

У документах стосовно керування розробленням ПЗ (*process documentation*) запротокольовано процеси розроблення й супроводження ПЗ, відображено зв'язки всередині колективу розробників і між колективами розробників і менеджерів (*managers*) – особами, які керують цими процесами. Ці доку-менти поділяються на такі:

– плани, оцінювання, розклади, які створюють менеджери для прогнозування й керування процесами розроблення й супроводження;

– звіти про використання ресурсів під час розроблення, їх створюють менеджери;

– стандарти – принципи, правила, угоди, яких мають дотримуватися розробники; можуть бути міжнародними, національними, або спеціально створеними для організації;

– робочі документи забезпечують зв'язок між розробниками і містять фіксацію ідей і проблем, що виникають під час розроблення, опис використовуваних стратегій і підходів, а також робочі (тимчасові) версії документів;

– замітки й листування – у цих документах відображається взаємодія між менеджерами й розробниками.

У документах, що належать до складу ПЗ (*product documentation*), описуються програми щодо застосування ПЗ користувачами і з точки зору

їх розробників і супровідників (згідно з призначенням ПЗ). Тут слід зазначити, що ці документи будуть використовуватися не тільки на стадії експлуатації ПЗ (в її фазах застосування і супроводу), але й на стадії розроблення для керування цим процесом (разом з робочими документами); їх має бути перевірено (протестовано) на відповідність програмам ПЗ. Ці документи утворюють два комплекти з різним призначенням:

– користувацька документація ПЗ (П-документація);

– документація зі супроводження ПЗ (С-документація).

У користувацькій документації ПЗ (*user documentation*) пояснюються правила використання цього ПЗ. Цю документацію застосовують, якщо в ПЗ припускається якась взаємодія з користувачами. До неї належать документи, якими керується користувач при інсталяції ПЗ (при установленні ПЗ з відповідним налаштуванням на середовище застосування ПЗ), при застосуванні ПЗ для вирішення своїх завдань і при керуванні ПЗ (наприклад, коли ПЗ взаємодіє з іншими системами). Ці документи частково стосуються питання супроводу ПЗ, але не стосуються питань, пов'язаних з модифікацією програм.

У зв'язку з цим слід розрізняти ординарних користувачів ПЗ і адміністраторів ПЗ. Ординарний користувач (end-user) використовує ПЗ для вирішення своїх завдань у своїй предметній області. Це може бути інженер, який проектує технічний пристрій, або касир, що продає залізничні квитки з допомогою ПЗ і може й не знати багатьох деталей роботи комп'ютера або принципів програмування. Адміністратор ПЗ (*system administrator*) керує використанням ПЗ ординарними користувачами і супроводжує ПЗ, не пов'язане з модифікацією програм. Наприклад, він може регулювати права доступу до ПЗ між ординарними користувачами, підтримувати зв'язок з постачальниками ПЗ або виконувати певні дії, щоб підтримувати ПЗ у робочому стані, якщо його включено як частину в іншу систему. Склад користувацької документації залежить від аудиторії користувачів, на яку орієнтовано ПЗ, і від режиму використання документів. Під аудиторією тут розуміється контингент користувачів ПЗ, у яких є необхідність у певній користувацькій документації ПЗ. Правильно складений користувацький документ істотно залежить від точного визначення аудиторії, для якої його призначено. Користувацька документація має містити інформацію, необхідну для кожної аудиторії. Під режимом використання документа розуміється спосіб використання цього документа. Зазвичай користувачеві досить великих програмних систем потрібні документи для вивчення ПЗ (використання у вигляді інструкції) або для уточнення певної інформації (використання у вигляді довідника).

Для досить великих ПЗ можна вважати типовим такий склад користу-

вацької документації:

– загальний функціональний опис ПЗ, у якому наводиться коротка характеристика функціональних можливостей ПЗ; його призначено для користувачів, які мають вирішити, наскільки необхідним є для них це ПЗ;

– посібник з інсталяції ПЗ, який призначено для системних адміністраторів; у ньому має бути детально прописано, як встановлювати системи в конкретному середовищі, а також мають міститися опис машинно-зчитувального носія, на якому поставляється ПЗ, файли зі змістом ПЗ і вимоги до мінімальної конфігурації апаратури;

– інструкція щодо застосування ПЗ, яку призначено для ординарних користувачів; у ній має міститися необхідна інформація щодо застосування ПЗ, подана у формі, зручній для її вивчення;

– довідник із застосування ПЗ, який призначено для ординарних користувачів, у ньому має міститься необхідна інформація, подана у формі, зручній для виборчого пошуку окремих деталей;

– посібник з керування ПЗ, який призначено для системних адміністраторів; у ньому має бути описано повідомлення, що генеруються, коли ПЗ взаємодіє з іншими системами, і як правильно реагувати на ці повідомлення; крім того, якщо ПЗ використовує системну апаратуру, у цьому документі може пояснюватися, як супроводжувати цю апаратуру.

Розроблення користувацької документації починається відразу після створення зовнішнього опису. Від якості цієї документації може істотно залежати успіх ПЗ. Документація має бути досить простою і зручною для користувача. Тому, хоча чорнові варіанти (начерки) комерційних документів створюються основними розробниками ПЗ, до створення їх остаточних варіантів часто залучають професійних технічних письменників. Крім того, для забезпечення якості користувацької документації розроблено кілька стандартів, у яких передбачено порядок розроблення цієї документації, формулюються вимоги до кожного її виду і визначаються їхні структура і зміст.

Документація із супроводження ПЗ (*system documentation*) необхідна, якщо в ній описано влаштування (конструкція) ПЗ і модернізація його складових. Як вже зазначалося, супровід – це подальше розроблення, тому у випадку необхідності модернізації ПЗ до цієї роботи залучається спеціальна команда розроблювачів-супровідників. Цій команді доведеться працювати з тією самою документацією, у якій визначалася діяльність команди первинних (основних) розробників ПЗ, з тією лише різницею, що ця документація для команди розроблювачів-супровідників буде зазвичай «чужою» (її створювала інша команда). Команда розроблювачів-супровідників повинна буде вивчити цю документацію, щоб зрозуміти бу-

дову і процес розроблення модернізованого ПЗ і внести необхідні зміни, повторюючи майже всі технологічні процеси, з допомогою яких створювалося первинне ПЗ.

Супровідну документацію до ПЗ можна поділити на дві групи:

– така, що визначає будову програм і структур даних ПЗ і технологію їх розроблення;

– така, що допомагає вносити зміни в ПЗ.

Документація першої групи містить підсумкові документи кожного технологічного етапу розроблення ПЗ: зовнішній опис ПЗ (*Requirements document*) і опис архітектури ПЗ (*description of the system architecture*), у тому числі зовнішню специфікацію кожної її програми; для кожної програми ПЗ – опис її модульної структури, у тому числі зовнішню специфікацію кожного її модуля; для кожного модуля – його специфікацію і опис будови (*design description*); тексти модулів на вибраній мові програмування (*program source code listings*). У документах стосовно встановлення достовірності ПЗ (*validation documents*) описується весь процес для кожної програми ПЗ і зв'язок інформації про це з вимогами до ПЗ. Документи зі встановлення достовірності ПЗ містять насамперед документацію з тестування (схема тестування й опис тестів), а також результати інших видів перевірки ПЗ, наприклад доведення властивостей програм.

До другої групи належить така документація, як посібник зі супроводження ПЗ (*system maintenance guide*), у якому описуються відомі проблеми з ПЗ, апаратно- і програмно-залежні частини системи, будова (конструкція) ПЗ з урахуванням його розвитку.

Загальна проблема супроводу ПЗ – забезпечити, щоб всі його подальші версії залишалися узгодженими при змінені ПЗ. Для цього зв'язки й залежності між документами та їх частинами має бути зафіксовано в базі даних керування конфігурацією.

Існують кілька видів експлуатаційних документів:

− відомість (перелік експлуатаційних документів на програму);

− формуляр (основні характеристики програми, комплектність і відомості про експлуатацію програми);

− опис застосування (відомості про призначення програми, області й методи застосування, клас вирішуваних завдань, обмеження для застосування, мінімальна конфігурація технічних засобів);

− посібник системного програміста (відомості про перевірку, забезпечення функціонування й налаштування програми на умови конкретного застосування);

− посібник програміста (відомості про експлуатацію програми);

− посібник оператора (відомості про забезпечення процедури спілку-

вання оператора з обчислювальною системою під час виконання програми);

− опис мови (опис синтаксису й семантики мови);

− посібник з технічного обслуговування (відомості про застосування тестових і діагностичних програм при обслуговуванні технічних засобів).

Види програмних документів на різних стадіях розроблення ЖЦ наведено в табл. 4.1, де «!» – документ потрібно розробити для компонентів, які можуть використовуватися самостійно; «+» – обов'язковий документ; «–» – документи не розробляються; «?» – необхідність розроблення документа вирішується на етапі затвердження технічного завдання.

Таблиця 4.1

Код виду документа	Вид документа	Стадія розроблення			
		Ескізний проект	Технічний проект	Робочий проект	
				Компонент	Комплекс
–	Специфікація	−	−	!	+
05	Відомість власників оригіналів	−	−	−	?
12	Текст програми	−	−	+	?
13	Опис програми	−	−	?	?
20	Відомість експлуатаційних документів	−	−	?	?
30	Формуляр	−	−	?	?
31	Опис застосування	−	−	?	?
32	Посібник системного програміста	−	−	?	?
33	Посібник програміста	−	−	?	?
34	Посібник оператора	−	−	?	?
35	Опис мови	−	−	?	?
46	Посібник з технічного обслуговування	−	−	?	?
51	Програма і методика випробувань	−	−	?	?
81	Пояснювальна записка	?	?	−	−
91–99	Інші документи	?	?	?	?

Специфікація має містити перелік і короткий опис призначення всіх файлів ПЗ, у тому числі й файлів документації на нього, та є обов'язковою для програмних систем і їх компонентів, що застосовуються самостійно.

Відомість власників оригіналів має містити список підприємств, на яких зберігаються оригінали програмних документів. Необхідність цього документа визначається на етапі розроблення й затвердження технічного завдання тільки для ПЗ зі складною архітектурою.

Текст програми повинен містити текст програми з коментарями. Необхідність цього документа визначається на етапі розроблення й затвер-

дження технічного завдання.

Опис програми має містити відомості про логічну структуру і функціонування програми. Необхідність цього документа також визначається на етапі розроблення й затвердження технічного завдання.

Відомість експлуатаційних документів має містити перелік експлуатаційних документів на програму. Необхідність цього документа також визначається на етапі розроблення й затвердження технічного завдання.

Формуляр має містити основні характеристики ПЗ, комплектність і відомості про експлуатацію програми.

Опис має містити відомості про призначення ПЗ, галузі застосування, застосовувані методи, клас розв'язуваних задач, обмеження для застосування, мінімальну конфігурацію технічних засобів.

Програма і методика випробувань має містити вимоги, які необхідно перевіряти під час випробування ПЗ, а також порядок і методи їх контролю.

Пояснювальна записка має містити інформацію про структуру й конкретні компоненти ПЗ, у тому числі схеми алгоритмів, їх загальний опис, а також обґрунтування прийнятих технічних і техніко-економічних рішень. Її складають на стадії ескізного й технічного проекту.

Інші документи складаються на будь-яких стадіях розроблення (ескізного, технічного і робочого проектів).

Допускається об'єднувати окремі види експлуатаційних документів, крім формуляра й відомості. Необхідність об'єднання зазначається в технічному завданні, а назву беруть з одного з об'єднаних документів. Наприклад, сьогодні часто використовують експлуатаційний документ, який частково входить у посібник системного програміста, програміста й оператора, що має назву «**Посібник користувача**».

Контрольні запитання та завдання для самостійної роботи

1. Які документи обов'язково створюються при розробленні ПЗ?
2. Для наведеного фраґмента програми створіть блок-схему (дод. 4).
3. Що містить документ «Експлуатаційна документація»?
4. Для кого створюється «Пояснювальна записка»?
5. Створіть документ «Посібник користувача» для наведеного програмного коду (дод. 4).
6. Чи є обов'язковим документ «Посібник системного програміста»?
7. Що таке Product documentation? Що таке П-документація?
8. Створіть документ «Специфікація» для наведеного модуля (дод. 4).
9. Які існують відмінності між end-user і адміністратором?

4.2. Маркетинг програмних продуктів

Інформаційні технології (ІТ) змінили уявлення про межі підприємств, технології виробництва й керування, ведення бізнесу в цілому.

Успіх комерційного розповсюдження інформації як товару обумовлюється тим, якою мірою запропоновані дані будуть для споживачів інформативними, тобто наскільки повними й ефективними стосовно витрат, часу та ін. Інформативність залежить від даних і їх подання.

4.2.1. Інформація як предмет комерційного розповсюдження

Інформація – це дані, які усувають невизначеність стосовно якого-небудь питання. Унаслідок вирішення завдання знання пройшли перевірку відомостей, їх узагальнено у вигляді теорій тощо.

Інформаційна модель – це сукупність уявлень про конкретну предметну область і про те, які дані й у якій формі найбільш адекватно відображають її.

На споживчу цінність і ринкову ціну інформації впливають як зміст, так і форма їх подання.

Форма структурування даних реалізує жорстку інформаційну модель виробника даних.

Перетворення даних на інформацію здійснюється споживачами на основі їх інформаційної моделі. Інформаційні моделі виробника і користувачів ніколи не можуть цілком збігатися. Їх розбіжність виявляється в тому, що користувачу потрібні дані в іншому обсязі й в іншій структурі, ніж це зроблено в інформаційному продукті.

Нові інформаційні технології, у яких передбачено не тільки надання інформаційного продукту, але й засобів доступу до нього, сприяють зближенню інформаційних моделей виробників і користувачів і здешевлюють інформацію, тому не треба показувати весь продукт цілком. Таким чином, основним товаром, створюваним з допомогою інформаційних технологій, є інформаційні продукти й послуги (ІПП).

4.2.2. Індустрія комерційного розповсюдження інформації

Основними інформаційними продуктами є БД (бази даних) і метадані (дані про дані).

Основною організаційною формою, у якій розвиваються сучасні технології, є автоматизовані банки даних (АБД) – системи спеціально організованих БД, а також програмних, технічних, мовних організаційно-методичних засобів, призначених для колективного використання цих БД.

На ринку діють:

а) самі виробники ІПП, серед яких:

– виробники БД, які здійснюють збір інформації та її переведення в машинопрочитувану форму;

– інтерактивні служби, які розробляють та експлуатують АБД;

– інтегровані виробники (виробники БД та інтерактивні служби);

б) телекомунікаційні служби, які здійснюють передання інформації по мережах ЕОМ.

в) користувачі, які поділяються на проміжні, тобто посередники (бібліотеки, інформаційні центри загального користування і брокери-фахівці) і кінцеві.

4.2.3. Організація інформаційного маркетингу

Специфіка інформаційного маркетингу обумовлена таким:

– інформаційні потреби не завжди усвідомлюються самими користувачами і їх не локалізовано фіксованими моментами часу, отже, потрібна значна робота з просування пропонованих послуг до потреб користувачів і підтримки стійкого попиту;

– інформаційні технології постійно змінюються кожні 2–3 роки, отже, змінюються інструменти маркетингу.

В інформаційному маркетингу керуються принципами основного маркетингу і його напрямків. Основні напрямки інформаційного маркетингу:

– аналіз ринків;

– формалізація цін на ІПП;

– установлення відносин між учасниками інформаційного ринку;

– просування ІПП.

Аналіз ринку дає змогу визначати, яку інформацію необхідно додати до АБД і яким вимогам він має задовольняти, щоб користувачі погодилися працювати з інформацією в АБД. Для вирішення цих питань використовуються аналіз аналогів АБД, насамперед зарубіжних, і аналіз сегментації ринку.

Існують аналоги за змістом і за призначенням. Вивчення аналогів першого типу дає уявлення про частоту додавання певних даних в АБД, таким чином можна отримати характеристику структури попиту. Вивчення аналогів другого типу дає змогу уточнити ринкові стандарти щодо функціональних параметрів АБД і їх ПЗ (змістовні й формалізовані характеристики).

Критерії сегментації на інформаційному ринку:

– фахова й галузева належність, за якою визначають тематику даних;

– наявність досвіду роботи в інтерактивному режимі (для інтерактивних послуг);

– характер інформаційної діяльності (кінцеві користувачі або посередники).

На ціни ІПП впливають:

– витрати на розроблення й експлуатацію АБД;

– якість послуг;

– очікуваний попит.

Використовувана система цін складається з базових тарифів, пільг і знижок. Основні види базових цін:

– ціна години приєднання до АБД;

– ціна отриманих даних;

– ціна передплати на АБД і окремі БД.

Для розрахунку конкретних цін рекомендується застосування імітаційного моделювання. Фіксуючи розмір виручки й задаючи певні значення попиту, можна добрати структуру й значення цін з допомогою імітаційного моделювання.

Відносини між виробниками АБД та інтерактивними службами будуються на основі контракту: інтерактивна служба завантажує БД в АБД і здійснює їх комерційну експлуатацію, а виробник відповідає за достовірність, ліцензійну частоту й регулярність оновлення.

Існують дві схеми розрахунків між виробниками й інтерактивними службами:

– орендна, коли інтерактивна служба сплачує виробнику БД фіксовану суму, а частину виручки, що залишилася, забирає собі; для інтерактивної служби ця схема буде вигідною, якщо БД є комерційно перспективною, а орендна плата невеликою, для виробника ця схема буде вигідною тому, що він не має ризиків, пов'язаних з невизначеністю попиту;

– розподільна, коли попередніх платежів немає, а розподіляється вже отримана виручка, виробнику виплачується авторський гонорар (роялті), розмір якого фіксується в договорі і встановлюється у відсотках або гривнях.

Основними методами інформаційного маркетингу є реклама, розповсюдження довідкових матеріалів, консультування користувачів.

Особливості реклами: інформування користувача про характеристики і конкретни переваги АБД. Зміст реклами має бути лаконічним, доступним, точно відображати переваги й бути чітко оформленим.

Для реклами використовуються друковані й електронні засоби, розробники ПЗ беруть участь у виставках, ярмарках, організації різноманітних семінарів, демонстраційних сеансів і надають користувачеві право роботи з метою ознайомлення зі своєю продукцією.

Комплект довідкової документації, яка періодично оновлюється, надсилається користувачеві відразу після укладення контракту. Довідкова документація повинна містити відомості про БД, у посібнику користувача має бути описано, як скласти запит і наведено повну метаінформацію, а також інформацію про ціни й можливі знижки.

Користувач може ознайомитися з можливостями АБД і отримати відповіді на конкретні запитання на семінарах, електронною поштою тощо.

4.2.4. Інтернет-маркетинг

Сьогодні основні маркетингові ходи пов'язані з інтернет-рекламою. Більшою мірою це пов'язано з тим, що продавець відразу може продемонструвати якусь частину функціональних можливостей рекламованого товару. Величезне поширення мережі Інтернет дає змогу охоплювати великі сегменти ринку.

Особливості рекламування продуктів в Інтернеті пов'язані з тим, що для досягнення ефекту необхідно обов'язково мати сайт, на якому буде розміщено всю інформацію про товари. Просто реклама в Інтернеті продукту, яку неможливо тут же переглянути (характеристики, властивості, ціна, контакти для зв'язку тощо), є малоефективною.

Для реклами в Інтернеті спочатку необхідно створити сайт, на якому буде продемонстровано всі можливості й переваги, описано способи купівлі, виставлено Посібники користувача тощо, потім додати сайт до пошукових систем і каталогів. Це необхідно для того, щоб його можна було знайти, використовуючи пошукові системи. Хоча пошукові ресурси автоматично додають нові сайти до своїх каталогів, цю процедуру необхідно зробити самостійно, не покладаючись на автоматику, оскільки продавець зацікавлений у якнайшвидшому збільшенні продажів. Основною причиною недовіри до автоматичного функціоналу є повільність цієї процедури.

Слід зазначити, що не треба поспішати з платними рекламними послугами, оскільки в Інтернеті існує безліч ресурсів, які розміщують банери й оголошення абсолютно безкоштовно. Укладати гроші в рекламування продукту слід акуратно й осмислено, оскільки реклама на дуже популярному сайті може не принести очікуваної вигоди.

Отже, для досягнення потрібного ефекту основним напрямком рекламної діяльності стане популяризація сайта, на якому буде розміщено інформацію про ПЗ, а також доказ того, що рекламована продукція краща за існуючі.

Якщо один програмний продукт вже є популярним, то фірма використовує його для реклами наступних своїх розробок. Як приклад можна навести компанію Google (рис 4.1), що діє дуже ефективно на ринку.

Наприклад, якщо виникла необхідність внести підприємство (організацію) на Google Map, то для цього буде необхідно зареєструвати пошту на gMail, потім створити акаунт на Google+, після чого стане доступним функціонал з додатковими корисними програмними продуктами від Google.

Ще один приклад відмінного маркетингового ходу – використання Google Doc, робота з яким є доступною тільки при використанні gMail-пошти, при цьому документи зберігаються на Google Disk. Про це постійно нагадується і пропонується зарезервувати більше місця або скористатися цим функціоналом, наприклад, щоб відправити великий обсяг даних елек-

тронною поштою. Якщо робота з Google Doc здійснюється не в браузері Google Chrome, то постійно надходять нагадування про те, що не всі функціональні можливості є доступними, про можливі збої в роботі й т. д. Слід також зазначити, що Google, створивши Google Doc, продублював функції Microsoft Office 365, надавши безкоштовну можливість використання функціонала (яку Microsoft надає за гроші) і відібравши у них таким чином частину ринку.

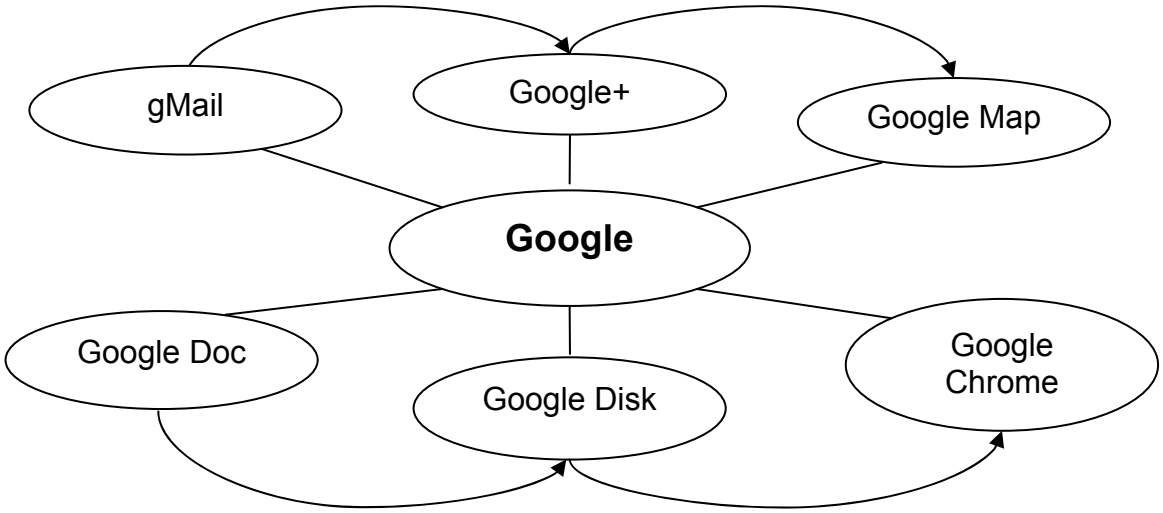


Рис. 4.1. Взаємодія продуктів Google

4.2.5. Пошукова оптимізація SEO

Для знаходження потрібної інформації в Інтернеті застосовуються різні пошукові механізми, тому необхідно, щоб пошуковий ранґ був високим. Якщо враховувати, що користувачі нечасто переглядають більше трьохчотирьох сторінок під час пошуку, то при низькому ранзі інформація залишиться незатребуваною.

Пошукову оптимізацію спрямовано на збільшення кількості відвідувачів Web-сайта з неоплачуваних списків пошукових сайтів завдяки підвищенню рангу сайта.

Пошуковий механізм являє собою синтез багатьох складних взаємозв'язаних алгоритмів, на який впливають як негативні, так і позитивні фактори.

Посилальна цінність

Посилання забезпечують структуру і надають як явну, так і неявную інформацію. Без посилань World Wide Web був би просто набором незв'язаних документів. Наприклад, якщо посилання на Web-сторінку є на багатьох інших Web-сайтах, то зазвичай це означає, що ця сторінка є більш важливою, ніж сторінка з меншою кількістю посилань. Якщо текст в дескрипторах цих посилань містить слово «пряник», то для пошукових механізмів це означає, що сторінка містить інформацію саме про пряники.

Посилання додають цінність Web-сторінкам і тому мають велике значення в пошуковій оптимізації. Посилальна цінність визначається як цінність, передана іншому URL цим посиланням. Для ясності термін «посилальна цінність» (*link equity*) буде вживатися для призначення або пересилання цінності, а термін «цінність» *(URL equity*) – для самої цінності, що міститься в цьому URL.

Серед усіх чинників, які пошукові механізми враховують при обчисленні ранґу Web-сайтів, посилальна цінність є основною.

Посилальна цінність поділяється на кілька форм.

Цінність для пошукового ранжування. Сучасні пошукові механізми використовують кількість і якість посилань на URL для оцінювання його якості, релевантності й корисності. Web-сайт з хорошою оцінкою буде мати більший ранґ. URL-адреса нарівні з контентом має й економічне значення, а контент, зі свого боку, становить цінність URL. При переміщенні вмісту в нове місце попередній URL буде видалено з пошукового індексу. Однак лише перенесення інформації не призведе до перенесення відповідної цінності, якщо при цьому всі вхідні посилання не буде переорієнтовано на нове місце. Можна проінформувати пошукові механізми про змінення адреси з допомогою перенаправлення – це перенесе цінність. Без правильно виконаного перенаправлення пошукового механізму неможливо повідомити, що всі посилання мають указувати на новий URL, і, таким чином, вся цінність URL може бути загубленою.

Зàкладна цінність. Користувачі часто залишають зàкладки на потрібні їм URL-адреси в браузері і на Web-сайтах соціальних зàкладок. Переміщення вмісту в нове місце призведе до припинення трафіку з цих зàкладок (якщо не буде перенаправлення), завдяки чому браузер буде знати про переміщення вмісту. Без перенаправлення користувач, швидше за все, отримає повідомлення про те, що контент більше не є доступним.

Цінність прямого цитування. Інші сайти можуть цитувати й посилатися на URL-адреси певного Web-сайта, що може бути причиною інтенсивного трафіка на нього. Переміщення контенту на нову адресу призведе до припинення трафіка з цих посилань (якщо не буде перенаправлення), завдяки чому браузер буде знати про переміщення вмісту.

Google PageRank

PageRank – це запатентований компанією Google алгоритм, який визначає важливість конкретної сторінки стосовно інших сторінок, доданих в індекс пошукового механізму. Його розробили наприкінці 1990-х років Ларрі Пейдж (Larry Page) і Сергій Брін (Sergey Brin). Як фактор ранжування в PageRank застосовується концепція посилальної цінності.

PageRank визначає важливість сторінки, вважаючи кожне посилання на неї одним голосом.

PageRank відповідає ймовірності, що користувач, випадковим чином

натискаючи на посилання в Інтернеті, потрапить на певну сторінку. Сторінка, на яку користувачі потрапляють частіше, мабуть, є більш важливою і тому має більш високу оцінку PageRank. Кожна сторінка з посиланням на іншу сторінку підвищує її оцінку PageRank. Зрозуміло, що сторінки з більш високою оцінкою PageRank зазвичай підвищують оцінки PageRank тих сторінок, на які вони посилаються.

PageRank – лише один із факторів в об'єднаному алгоритмі, який Google застосовує для будування сторінок з результатами пошуку (Search Results Pages – SERP). Цілком можливо, що для конкретного запиту сторінка з меншою оцінкою PageRank буде знаходитися вище за сторінку з більшою оцінкою PageRank. Крім того, у PageRank не враховується релевантність, оскільки підсумкова популярність розраховується в ньому тільки на основі посилань, а не теми викладу.

Google PageRank не єдиний алгоритм ранжирування на основі обліку посилань, але є найбільш популярним.

Зручність (usability) Web-сайта визначається як легкість його використання.

Під *доступністю (accessibility*) мається на увазі те ж саме, що й зручність, але стосовно користувачів з такими вадами, як обмежений зір або слух.

Для фахівця з пошукового маркетингу зручність і доступність належать до властивості «оптимізація для користувачів».

Фактори, що впливають на пошуковий ранґ

Алгоритми, що застосовуються в Google, Yahoo! і MSN Live Search для обчислення результатів пошуку, можуть змінитися в будь-який момент.

На пошуковий ранґ впливають такі фактори: наявні видимі, наявні невидимі, тимчасові, зовнішні.

Наявні фактори *− це критерії, які обумовлені вмістом самої Web-сторінки.* Ці фактори є важливими для компаній пошукового маркетингу, але вже не так, як раніше, тому ними дуже легко маніпулювати. Оскільки пошукові механізми є очевидними інструментами для спамерів, вони почали звертати увагу й на інші фактори, але це не означає, що наявні фактори вже не є важливими.

Зручно розбити ці фактори ще на дві категорії: видимі й невидимі. Видимі фактори є важливішими за невидимі. Багато фахівців з пошукового маркетингу вважають, що наявні невидимі фактори знецінилися до такого стану, що їх майже не варто брати до уваги. Причиною цього є легкість маніпуляції ними без будь-якого впливу на зовнішній вигляд сторінок. Ці фактори дають змогу ретельно замаскувати спам на Web-сторінці, тому неможливо бути впевненим у їх коректності.

Наявними видимими факторами на сторінці є назва сторінки, заголо-

вки, текст, вихідні посилання, ключові слова в URL і доменному імені, структура внутрішніх посилань і анкери, загальна актуальність сайта.

Наявні невидимі фактори — це частини Web-сторінки, які є невидимими для людей, що її переглядають. Однак вони є видимими для пошукових механізмів, що аналізують Web-сайт. До таких невидимих факторів належать опис в дескрипторі meta, ключові слова у дескрипторі meta, атрибути alt і title, структура сторінки.

Тимчасові фактори, які використовуються для ранжирування, *− це вік сайта і сторінки, а також давність посилань на них.*

Крім того, на ранґ може вплинути і тривалість реєстрації доменного імені.

Зовнішні фактори: кількість, якість і релевантність вхідних посилань; мерехтіння посилань; швидкість набуття посилань; анкерний текст посилань і їхній навколишній текст; зворотні посилання; кількість посилань на сторінці; семантичний взаємозв'язок посилань на сторінці; IP-адреси сайтів з перехресними посиланнями; зони доменних імен для посилань; розташування посилань; відповідність Web-стандартам; заборонені шкідливі фактори та ін.

Заборонені шкідливі фактори. Оштрафований Web-сайт має мало, а то й зовсім ніяких шансів пробитися в SERP. Основними штрафами є «ефект відстійника» (*sandbox effect*) Google; штраф за домен із закінченим терміном придатності; штраф за дублювання контенту; додатковий індекс Google.

Штраф за домен із закінченим терміном придатності. Запуск нового веб-сайта в домені із раніше закінченим терміном придатності якраз і виконувався для усунення «ефекту відстійника». У цьому випадку Google не міг визначити, що сайт тільки що виник, але потім Google увів штраф за такі домени і зараз їх застосування часом дає абсолютно зворотний ефект.

Тимчасовий штраф потрібен для створення затримки, перш ніж сайт знайде хороший ранґ. У деяких випадках Google навіть відмовляється індексувати сторінки протягом усього цього періоду, і Web-сайт стає доступним для крадіжки контенту.

Штраф за дублювання контенту. Пошукові механізми намагаються не виконувати індексування декількох копій одного й того самого дубльованого контенту.

Додатковий індекс Google, сам по собі не є штрафом, але може бути його наслідком. Google зберігає свої результати індексування в двох індексах: основному й додатковому. У додатковому індексі зберігаються відомості про сторінки, які Google з якихось причин уважає менш важливими. Результати додаткового індексу зазвичай виводяться в кінці списку результатів пошукового запиту Google (за винятком дуже специфічних запитів), і ці результати помічено як додаткові.

Фактори, які призводять до включення посилання в додатковий, а не основний індекс, є недоліком суттєво унікального контенту або вхідних посилань на цей контент. Крім того, це може бути й результатом явного штрафування.

Web-аналітика. Інструменти Web-аналітики заміряють трафік і відстежують поведінку користувачів на конкретному Web-сайті. Зазвичай Web-аналітика використовується для підвищення ефективності виробництва на основі різних метрик, наприклад коефіцієнта активного відвідування і рентабельності вкладень. Особливо зручно відстежувати рентабельність вкладень для PPC-реклами, де є конкретна вартість одного клацання, але можна також відстежувати ключові слова з органічного трафіка, на які реагують користувачі. Такі результати допомагають фахівцю з пошукового маркетингу визначити ключові фрази, які потрібно використовувати при виконанні пошукової оптимізації як для поліпшення, так і для підтримки своєї позиції.

Важливо не тільки мати у своєму розпорядженні дані про свій Web-сайт, але й знати ринок і своїх конкурентів. Перше, що необхідно засвоїти: як використовувати вбудовані можливості пошукових механізмів. Наприклад, щоб знайти всі сторінки сайта http://www.dialektika.com, індексовані Google, Yahoo!, MSN, Yandex тощо, потрібно ввести запит site:www.dialektika.com.

4.2.6. Аудит

Стандарт SWEBOK

Згідно зі стандартом SWEBOK, призначення аудиту – це незалежне оцінювання продуктів і процесів на відповідність, регулювання і регламентування документів (планів, стандартів та ін.), а також формулювання звіту про випадки невідповідності й пропозицій для їх коригування.

Аудит ПЗ *– це діяльність, що виконується для незалежного оцінювання програмних продуктів і процесів, «формальна відповідність» (conformance) чинним інструкціям, стандартам, керівним документам, планам і процедурам (див. IEEE 1028-97 «Standard for Software Reviews»).*

Аудити проводять відповідно до певних процесів, що містять і визначають ролі й обов'язки аудиторів. Кожний процес аудиту має бути чітко сплановано і може потребувати залучення багатьох фахівців для виконання різних завдань (визначених процедурою аудиту) за досить короткий інтервал часу. Автоматизовані засоби, що забезпечують підтримку планування й проведення аудиту, можуть суттєво полегшити й прискорити цей процес. У стандарті SWEBOK зазначається, що рекомендації щодо проведення аудиту можна знайти в багатьох джерелах, у тому числі в стандарті IEEE 1028-97 «Standard for Software Reviews».

Аудит конфігурації *– це процедура, що виконується для оцінювання відповідності продукту і процесів стандартів, інструкцій, планів і процедур.*

З допомогою аудиту програмних конфігурацій визначається ступінь відповідності конфігурації функціональним і фізичним (апаратним) характеристикам системи, а також кожного елемента конфігурації (SCI) − заданим (наприклад, на рівні вимог і (або) запитів на змінення) функціональним і фізичним характеристикам. Неформальний аудит такого типу може бути пов'язаний з ключовими точками життєвого циклу (віхами проекту, у термінах керування проектами – *milestones*). Існують два досить поширених види формального аудиту конфігурацій (необхідного для певних категорій контрактів, наприклад, на створення критично-важливого ПЗ): функціональний (Functional Configuration Audit, FCA) і фізичний (Physical Configuration Audit, PCA). Успішне (стосовно відповідності результатів заданим умовам) завершення аудитів може бути обов'язковою вимогою для фіксування базової лінії продукту. Водночас, щоб порівняти контексти FCA і PCA для програмного й апаратного забезпечення, перед їх виконанням необхідно чітко оцінити реальні потреби в таких видах аудиту (оскільки для них потребуються суттєві, іноді просто «непідйомні» витрати ресурсів при оцінюванні їх з огляду на задані обмеження проекту).

Функціональний аудит програмних конфігурацій (Software Functional Configuration Audit) полягає в тому, щоб переконатися, що контрольований програмний елемент повністю відповідає заданим специфікаціям. «Вихід», тобто результат перевірки й атестації (V&V, verification and validation) ПЗ є ключовим «входом» (вихідними даними) для проведення аудиту.

Фізичний аудит програмних конфігурацій (Software Physical Configuration Audit) полягає в тому, щоб переконатися, що дизайн і документація точно узгоджуються з програмним продуктом.

Внутрішні аудити базових ліній (In-process Audits of Software Baseline). Як вже було зазначено вище, аудити можуть виконуватися протягом усього процесу розроблення для отримання поточного статусу заданих елементів конфігурацій. У цьому випадку аудит може проводитися відносно вибіркових елементів базових ліній для переконання, що задані специфікаціями характеристики процесу, швидкості і якості розвитку продукту дотримуються, а документація відповідає і підтримується в узгодженому стані з документовними елементами продукту під час їх еволюції протягом життєвого циклу. У посібнику «Цілі контролю для інформаційних та суміжних технологій (COBIT)» містяться кращі практики на рівні доменів (груп та окремих IT-процесів) у вигляді керованої і логічної структури. Кращі практики в COBIT базуються на консенсусі експертів.

Принципи аудиту COBIT – книга стандартів, яку більшою мірою зорієн-

нтовано на аудит IT-процесів, ніж на аудит конкретних функцій або додатків. COBIT складається з високорівневих цілей контролю (визначених для IT-процесів організації), які охоплюють усі параметри інформаційних систем і застосовуваних інформаційних технологій, у яких ураховано цикл життя і специфічні завдання, що вирішуються в IT-галузі.

Для забезпечення високої якості надання послуг, професіоналізму аудиторів і вирішення складних етичних ситуацій, що виникають під час аудиту, асоціація ISACA (the Information Systems Audit and Control Association) визначила основні вимоги до аудитора, які описано в «Етичному кодексі аудитора».

Етичний кодекс аудитора IT-галузі:

– сприяти тому, щоб інформаційні системи відповідали прийнятим стандартам та інструкціям;

– здійснювати свою діяльність відповідно до стандартів в області аудиту інформаційних систем, прийнятих ISACA;

– діяти старанно, лояльно і чесно в інтересах роботодавців, акціонерів, клієнтів і суспільства;

– свідомо не брати участі в незаконній або недобросовісній діяльності;

– зберігати конфіденційність інформації, отриманої під час виконання своїх посадових обов'язків;

– не використовувати конфіденційну інформацію для отримання особистої вигоди і не передавати її третім особам без дозволу власника;

– виконувати свої посадові обов'язки, залишаючись незалежним і об'єктивним;

– уникати діяльності, яка загроджує незалежності аудитора;

– підтримувати на належному рівні свою компетентність у галузях знань, пов'язаних з проведенням аудиту інформаційних систем, брати участь у професійних заходах;

– виявляти сумлінність при отриманні й документуванні фактографічних матеріалів, на яких базуються висновки і рекомендації аудитора;

– інформувати всі зацікавлені сторони про результати проведення аудиту;

– сприяти підвищенню обізнаності керівництва організацій, клієнтів і суспільства щодо питань, пов'язаних з проведенням аудиту інформаційних систем;

– відповідати найвищим етичним стандартам професійної та особистої діяльності;

– вдосконалювати свої особисті якості.

Структура принципів аудиту COBIT. Для кожного IT-процесу, визначеного за COBIT, наведено таку інформацію:

– у секції високого рівня принципів аудиту COBIT відображаються назва бізнес-процесу, вимоги бізнесу (об'єкти контролю високого рівня), пра-

вила здійснення контролю, а також те, що саме треба враховувати;

– для переходу на рівень детального аудиту IT-процесу відображаються детальні об'єкти контролю, описується як зрозуміти IT-процес (кому задавати запитання), оцінити контроль IT-процесу і відповідність цього контролю керування, як довести ризик невиконання цілей керування.

Стандарт COBIT має три рівня застосування:

− базовий рівень аудиту IT (використовуються вимоги процесу аудиту, контроль, загальні принципи аудиту);

− рівень процесів (використовується основна частина стандарту);

− рівень аудиту додаткових цілей контролю (використовуються: спеціалізовані галузеві критерії, промислові стандарти, вимоги виробників елементів інфраструктури, застосування детальних методів контролю).

На практиці при проведенні аудиту для кожного IT-процесу аудитору необхідно виконати таку роботу: визначити високорівневий об'єкт контролю, визначити IT-процес, проаналізувати межі аудиту, визначити детальні об'єкти контролю, провести інтерв'ю із співробітниками (орієнтовні назви посад для кожного об'єкта контролю наведено у принципах керування), призначити завдання на оцінювання засобів контролю, оцінити відповідність, перевірити докази.

Ліцензування

Аудит легальності ПЗ *– це аналіз стану використовуваного ПЗ щодо відповідності умов застосування вимогам угоди правовласника програмного продукту.*

Іншими словами, аудит легальності ПЗ – це початковий етап з його легалізації, що дає змогу визначити, які програми необхідно легалізувати, а які можна вільно використовувати.

Аудит легальності ПЗ складається з кількох основних етапів: сканування ПЗ, встановленого на робочих станціях і серверах; оброблення отриманих даних; подання загального звіту про проведений аудит з докладним описом ПЗ, встановленого на кожному комп'ютері, ризиків і рекомендацій; складання загальної програми з легалізації організації.

Результатом проведення аудиту легальності ПЗ є повна картина легальності встановленого в організації ПЗ, спільна програма з легалізації неліцензійного ПЗ, відомості про відповідальність при використанні неліцензійного ПЗ, можливість уникнути технічних проблем, що виникають при використанні неліцензійного ПЗ.

В українському законодавстві за використання й розповсюдження неліцензійного ПЗ і порушення авторських прав передбачається покарання у вигляді штрафів і навіть позбавлення волі. У зв'язку з цим питання переходу організацій і підприємств на ліцензійне ПЗ стає дуже актуальним.

Програмне забезпечення *– це об'єкт авторського права, для ви-*

користання якого потрібна ліцензія.

Ліцензія надає право на встановлення, використання, доступ, відображення, запуск або будь-яку іншу взаємодію з ПЗ.

***Купити ліцензійне ПЗ** означає придбати ліцензію (права) на його використання у правовласника.* Ліцензія необхідна для кожної використовуваної програми.

Обсяг прав і умови використання описано у відповідних ліцензійних угодах. При порушенні умов ліцензійної угоди використання ПЗ вважається нелегальним.

Перевести офіс на ліцензійне ПЗ досить складно, оскільки необхідно добрати саме таке ПЗ, яке дасть змогу зменшити тимчасові витрати на період адаптації користувачів (співробітників організації), а також не спричинить серйозних фінансових втрат організації.

***Ліцензія на ПЗ** – це правовий інструмент, у якому визначено правила використання й розповсюдження ПЗ, захищеного авторським правом.*

Зазвичай ліцензія на ПЗ дає змогу одержувачу використовувати одну або кілька копій програми, причому без ліцензії таке використання розглядалося б у межах закону як порушення авторських прав видавця. По суті ліцензія є гарантією того, що видавець, якому належать виключні права на програму, подасть в суд на того, хто нею користується незаконно.

Ліцензійні угоди за обсягом прав, що передаються користувачу програми, поділяються на дві великі групи:

До першої групи належить ПЗ із закритим кодом, яке розповсюджується на основі ліцензійних угод: платні, умовно безкоштовні (*shareware*) і безкоштовні.

До другої групи належить ПЗ (Open Source), що вільно розповсюджується, – програми з відкритим програмним кодом. Саме друга група ліцензійних угод виникла як альтернатива першій групі. Одним із перших таких ліцензійних договорів був General Public License (GNU GPL), розроблений 1988 року Річардом Столлманом.

Програмне забезпечення, яке розповсюджується на основі безкоштовної ліцензійної угоди, містить такі положення:

– є повністю безкоштовним, але не містить вихідні коди певної програми;

– може розповсюджуватися без обмежень;

– майнові і немайнові права зберігаються за правовласником.

Прикладом ліцензії безкоштовного ПЗ є угода Adware, за якою програма, що містить рекламу, розповсюджується і використовується безкоштовно, а автор ПЗ для ЕОМ отримує винагороду від рекламодавців за розміщення реклами і від користувачів, які зацікавлені в тому, щоб реклами не було.

Програмне забезпечення, що розповсюджується на основі умовно-

безкоштовної ліцензійної угоди (*Shareware*), належить до змішаного виду безоплатних договорів, у яких використовуються умови безкоштовних програм з дотриманням певних обмежень.

До обмежень, що застосовуються в умовно безкоштовному ПЗ, належить можливість застосування користувачем функціонально або тимчасово обмеженого екземпляра ПЗ з особистою метою. Водночас користувач за наданий період безкоштовного використання повинен визначити, чи потрібна йому програма, після чого програма перестає функціонувати до сплати розробникам певної суми. Можливим є також інший варіант: програма є безкоштовною, але функціонально обмеженою, і користувач може оцінити, чи потрібен йому більш професійний варіант, за який необхідно заплатити. Після закінчення терміну роботи такого програмного продукту його необхідно видалити або оплатити й придбати права на нього цілком, тобто можна говорити про сукупність двох залежних ліцензійних угод, причому закінчення другої угоди безпосередньо залежить від умов першої.

Для декомпіляції не потрібні згода правовласника й виплата додаткової винагороди, але необхідно дотримуватися таких умов:

– інформація, необхідна для досягнення здатності до взаємодії, раніше не була доступною цій особі з інших джерел;

– зазначені дії здійснюються стосовно тільки тих частин декомпільовної програми для ЕОМ, які є необхідними для досягнення здатності до взаємодії;

– інформація, яку отримано внаслідок декомпілювання, може використовуватися лише для досягнення здатності до взаємодії незалежно розробленої програми з іншими програмами, і не може передаватися іншим особам (за винятком випадків, коли це необхідно для досягнення здатності до взаємодії незалежно розробленої програми для ЕОМ з іншими програмами), а також не може використовуватися для розроблення програми для ЕОМ, за своїм виглядом істотно схожою з декомпільованою програмою для ЕОМ, або для здійснення іншої дії, що порушує виключне право на програму для ЕОМ.

Дії з декомпіляції можливі тільки стосовно ПЗ і БД. Проте такі дії не повинні перешкоджати нормальному використанню ПЗ для ЕОМ або БД і обмежувати законні інтереси автора або іншого правовласника.

Усе це ускладнює поширення і використання програм для ЕОМ. Для порівняння, стосовно програм для ЕОМ правовласник має можливість самому визначати правила використання програмного продукту, а стосовно літературного твору він такої можливості не має. Крім того, у більшості випадків юридичні особи, наприклад корпорація Microsoft, утворювали монополії на використання розроблених ними програм для ЕОМ, що з огляду на безпеку для програми із закритим кодом не є оптимальним варіантом. Через їх закритість перевірити ці програми на наявність помилок і вбудованих шпигунських програм законно неможливо.

Корпорація Microsoft 24 серпня 2007 р. здійснила масовий несанкціонований доступ в десятки мільйонів комп'ютерів по всьому світу, підключених до мережі Інтернет. Несанкціонований доступ полягав в установленні на комп'ютери з операційною системою Windows XP і Windows Vista дев'яти файлів без згоди користувачів, при цьому файли було змінено навіть на комп'ютерах, де функцію автоматичного оновлення ОС було відключено. Після публікації 14 вересня в ЗМІ такої інформації Microsoft визнала цей факт.

Усе це сприяло поширенню іншого класу програм для ЕОМ – програм з відкритим кодом.

Вільне ПЗ. Його виникнення пов'язане з ідеєю створення ПЗ, вільного від обмежень у використанні, причому концепцію, що містить такі ідеї, називають «копілефт» (на відміну від концепції «копірайт»). У концепції «копілефт» припускається використання законів авторського права, щоб будь-яка людина мала змогу без обмежень використовувати, змінювати й поширювати як вихідний твір, так і твори, що є похідними від нього.

За пропозицією американських розробників Брюса Перенса і Еріка Реймонда 1998 року було створено організацію «Ініціатива відкритих кодів» (Open Source Initiative, OSI), яка розробила й опублікувала «Визначення ПЗ з відкритим програмним кодом», що складається з десяти пунктів.

Авторство й ім'я автора охороняються безтерміново. Відмова від цих прав є незначною. У зв'язку з тим, що положеннями ліцензійної угоди щодо програм для ЕОМ передбачено укладення угоди кожним користувачем з відповідним правовласником договору приєднання, умови якого викладено на придбаному примірнику таких програм (або баз даних) або на упаковці цього примірника. Це і дає змогу розповсюджувати програму для ЕОМ з відкритим кодом, розроблену Е. Реймондом.

Звичайно, не можна виключити ситуації, коли автор програми для ЕОМ з відкритим кодом може звернутися до суду за захистом немайнових прав, якщо було виявлено факт використання цієї програми третьою особою.

З метою створення законної схеми передання усім, хто звернувся, виключних прав на ПЗ по всьому світу було розроблено ліцензію на вільне ПЗ з відкритим кодом (Open Source).

Таке ПЗ схоже на безкоштовне, але із суттєвою різницею – виключні права на вихідний код безкоштовного ПЗ залишаються за його правовласником, а такі права на ПЗ з відкритим кодом у правовласника програми не зберігаються. На основі цієї ліцензії розповсюджується програма з відкритим вихідним кодом, який не потрібно відкривати і можна модифікувати.

ПЗ з відкритим кодом можна допрацьовувати для подальшого вдосконалення, поширення й використання без додаткових умов і без отримання на те згоди автора. Вихідний код програми буде доступним усім, і

будь-який користувач зможе змінювати, розповсюджувати, використовувати програми, створені на основі цієї програми. ПЗ, користувачам якого не надається права на модифікацію відкритого коду, є невільним незалежно від будь-яких інших умов.

Якість такого ПЗ можна порівняти з якістю комерційних програм. Пов'язано це з тим, що ПЗ тестується, змінюється, покращується усіма бажаючими. Таким чином, кількість розробників, що працюють над поліпшенням ПЗ, дорівнює, а в деяких випадках і є набагато більшою за кількість розробників комерційного забезпечення із закритим кодом.

Таке ПЗ виявилося не тільки відкритим, але й надійним, причому свобода творчості для його авторів є важливішою за винагороду. Звичайно, автори розповсюджуваних ліцензій на ПЗ з відкритим кодом зобов’язані дотримуватися немайнових прав, таких, як проставлення знака охорони, року першого публікування та імені автора чи іншого правовласника.

Таким чином, специфіка ліцензійної угоди на використання ПЗ з відкритим програмним кодом визначається обсягом переданих виключних прав ліцензіату. Важливо відразу зазначити, що такі ліцензійні договори ніяк не впливають на особисті немайнові права авторів програм, а більшою мірою захищають їх. Програми з відкритим кодом є більш надійними й безпечними, оскільки код програми можна перевірити і в разі необхідності перепрограмувати.

Контрольні запитання

1. Які інформаційні продукти користуються попитом на ринку?
2. Які існують напрямки в інформаційному маркетингу?
3. З чого складається інтернет-маркетинг?
4. Що таке посилальна цінність?
5. Що таке цінність URL?
6. Які існують форми посилальної цінності?
7. Що таке SERP?
8. Що таке зручність і доступність Web-сайта?
9. На які категорії можна поділити фактори, що впливають на пошуковий ранґ?
10. За що штрафують Web-сайти?
11. Для чого необхідна Web-аналітика?
12. Що таке аудит? Які існують стандарти з аудиту?
13. Для чого необхідне ліцензування? Які види ліцензування існують?

БІБЛІОГРАФІЧНИЙ СПИСОК

1. Галузевий стандарт вищої освіти України з напряму підготовки 6.050101 «Комп'ютерні науки» : зб. нормат. док. вищ. освіти / О. А. Павлов, Т. В. Ковалюк, А. І. Петренко та ін. − К. : Вид. група BHV. − 2011. − 85 с.
2. Бандура, И. Н. Технология программирования и создания программных продуктов : учеб. пособие / И. Н. Бандура. – Х. : Нац. аэрокосм. ун-т им. Н. Е. Жуковского «Харьк. авиац. ин-т», 2005. – 20 с.
3. Иванова, Г. С. Технология программирования : учебник / Г. С. Иванова. − М. : Изд-во МГТУ им. Н. Э. Баумана, 2003. – 320 с.
4. Ушакова, І. О. Основи системного аналізу об'єктів і процесів комп'ютеризації : навч. посіб. / І. О. Ушакова. − Х. : Вид-во ХНЕУ, 2008. − Ч. 2. − 308 с.
5. Новиков, Ф. А. Анализ и проектирование на UML [Электронный ресурс] : учеб.-метод. пособие / Ф. А. Новиков. − Электрон. дан. − СПб. : СПбГУ ИТМО, 2008. − 286 с. − Режим доступа : http://books.ifmo.ru/file/pdf/424.pdf, свободный. − Загл. с экрана.
6. Дідковська, М. В. Тестування: основні визначення, аксіоми та принципи [Електронний ресурс] : курс лекцій / М. В. Дідковська, Ю. О. Тимошенко. − Електрон. дані. − К. : НТУ КПІ, 2010. − 61 с. − Режим доступу : http://mmsa.kpi.ua/, вільний. − Назва з екрана.
7. Буч, Г. Объектно-ориентированный анализ и проектирование с примерами приложений : пер. с англ. / Г. Буч. − 3-е изд. − М. : Вильямс, 2010. − 720 с.
8. Wirth, N. Program Development by Stepwise Refinement / N. Wirth // Communications of the ACM. − Vol. 14 (4). − 1971. − P. 221−227.
9. Dahl, O. Structured Programming / O. Dahl, E. Dijkstra, C.A.R. Hoare. − London : Academic Press, 1972. − P. 175−220.
10. Алексенко, О. В. Технології програмування та створення програмних продуктів : конспект лекцій / О. В. Алексенко. – Суми : СумДУ, 2013. – 133 с.
11. Буч, Г. Язык UML. Руководство пользователя : пер. с англ. / Г. Буч, Дж. Рамбо, А. Джекобсон. − СПб. : Питер, 2004. − 432 с.
12. Боггс, У. Язык UML и Rational Rose : пер. с англ. / У. Боггс, М. Боггс. − М. : Лори, 2004. − 608 с.
13. Дуг, Р. Применение объектного моделирования с использованием UML и анализ прецедентов : пер. с англ. / Р. Дуг, К. Скотт. − М. : Пресс, 2002. − 160 с.
14. Орлов, С. А. Технология разработки программного обеспечения : учеб. пособие / С. А. Орлов. − 2-е изд. − СПб. : Питер, 2003. – 528 с.
15. Леоненков, А. В. Самоучитель UML / А. В. Леоненков. – 2-е изд., перераб. и доп. − СПб. : БХВ-Петербург, 2004. − 432 с.

16. Лингер, Р. Теория и практика структурного программирования : пер. с англ. / Р. Лингер, Х. Миллс, Б. Уитт. − М. : Мир, 1982. − 406 с.
17. Макконнелл, С. Совершенный код : пер. с англ. / С. Макконнелл. − СПб. : Питер, 2005. − 896 с.
18. Фаулер, М. UML. Основы : пер. с англ. / М. Фаулер. − 3-е изд. − СПб. : Символ, 2005. − 188 с.
19. Ларман, К. Применение UML и шаблонов проектирования : пер. с англ. / К. Ларман. − 2-е изд. − М. : Изд. дом «Вильямс», 2002. – 624 с.
20. Калянов, Г. Н. CASE: структурный системный анализ (автоматизация и применение) / Г. Н. Калянов. − М. : Лори, 1996. − 242 с.
21. Леффингуэлл, Д. Принципы работы с требованиями к программному обеспечению : пер. с англ. / Д. Леффингуэлл, Д. Уидриг. − М. : Изд. дом «Вильямс», 2002. − 448 с.
22. Тернер, М. Основы Microsoft Solution Framework : пер. с англ. / М. Тернер. − М. : Питер, 2008. − 336 с.

Додаток 1

ДІЮЧИ СТАНДАРТИ ТА ОРГАНІЗАЦІЇ-РОЗРОБНИКИ СТАНДАРТІВ

Таблиця Д.1.1

Стандарти, що використовуються при розробленні ПЗ і документації

Індекс і номер (індекс і номер ідентичного стандарту)	Обєкт стандартизації	Найменування підрозділу
ДСТУ 2844–94	Програмні засоби ЕОМ	Забезпечення якості. Терміни та визначення
ДСТУ 2850–94		Показники і методи оцінювання якості
ДСТУ 2851–94		Документування результатів випробувань
ДСТУ 2853–94		Підготовлення і проведення випробувань
ДСТУ 2873–94	Системи оброблення інформації	Програмування. Терміни та визначення
ДСТУ 2941–94		Розроблення систем. Терміни та визначення
ДСТУ 3918–1999 (ISO/IEC 12207:1995)	Інформаційні технології	Процеси життєвого циклу програмного забезпечення
ДСТУ 3919–1999 (ISO/IEC 14102:1995)		Основні напрямки оцінювання та відбору CASE-інструментів
ДСТУ 4071–2001 (ISO/IEC 13244:1998, Adm. 1:1999, MOD)		Архітектура відкритого розподіленого керування та підтримка загальної архітектури брокера об'єктних запитів (CORBA)
ДСТУ 4302: 2004 (ISO/IEC 6592:2000, MOD)		Настанови щодо документування комп'ютерних програм
ДСТУ ISO/IEC 11411–2002 (ISO/IEC 11411:1995, IDT)		Зображення переходу стану програмного засобу для спілкування людей
ДСТУ ISO/IEC 12119–2003 (ISO/IEC 12119:1994, IDT)		Пакети програм. Тестування і вимоги до якості
ДСТУ ISO/IEC TR 12182:2004 (ISO/IEC TR 12182:1998, IDT)		Класифікація програмних засобів
ДСТУ ISO/IEC 14598–1:2004 (ISO/IEC 14598–1:1999, IDT)		Оцінювання програмного продукту. Частина 1. Загальний огляд
ДСТУ ISO/IEC 14598–2:2005 (ISO/IEC 14598–2:2000, IDT)		Оцінювання програмного продукту. Частина 2. Планування та керування

Продовження табл. Д.1.1

Індекс і номер (індекс і номер ідентичного стандарту)	Обєкт стандартизації	Найменування підрозділу
ДСТУ ISO/IEC 14598–3:2005 (ISO/IEC 14598-3:2000, IDT)		Оцінювання програмного продукту. Частина 3. Процес для розробників
ДСТУ ISO/IEC 14598–4:2005 (ISO/IEC 14598–4:1999, IDT)		Оцінювання програмного продукту. Частина 4. Процес для замовників
ДСТУ ISO/IEC 14598–5:2005 (ISO/IEC 14598–5:1998, IDT)		Оцінювання програмного продукту. Частина 5. Процес для оцінювачів
ДСТУ ISO/IEC 14598–6:2005 (ISO/IEC 14598–6:2001, IDT)		Оцінювання програмного продукту. Частина 6. Документація модулів оцінювання
ДСТУ ISO/IEC 14764–2002 (ISO/IEC 14764:1999, IDT)		Супроводження програмного забезпечення
ДСТУ ISO/IEC 15288:2005 (ISO/IEC 15288:2002, IDT)		Процеси життєвого циклу системи
ДСТУ ISO/IEC TR 15504–1–2002 (ISO/IEC TR 15504–1:1998, IDT)		Оцінювання процесів життєвого циклу програмних засобів. Частина 1. Концепції та вступна настанова
ДСТУ ISO/IEC TR 15504–2–2002 (ISO/IEC TR 15504–2:1998, IDT)		Оцінювання процесів життєвого циклу програмних засобів. Частина 2. Еталонна модель процесів та потужності процесу
ДСТУ ISO/IEC TR 15504–3–2002 (ISO/IEC TR 15504–3:1998, IDT)		Оцінювання процесів життєвого циклу програмних засобів. Частина 3. Виконання оцінювання
ДСТУ ISO/IEC TR 15504–4–2002 (ISO/IEC TR 15504–4:1998, IDT)		Оцінювання процесів життєвого циклу програмних засобів. Частина 4. Настанови з виконання о
ДСТУ ISO/IEC TR 15504–5–2002 (ISO/IEC TR 15504–5:1999, IDT)		Оцінювання процесів життєвого циклу програмних засобів. Частина 5. Модель оцінювання та настанови щодо показників
ДСТУ ISO/IEC TR 15504–6:2003 (ISO/IEC TR 15504–6:1998, IDT)		Оцінювання процесів життєвого циклу програмних засобів. Частина 6. Настанови з визначання компетентності оцінювачів

Продовження табл. Д.1.1

Продовження табл. Д.1.1

Індекс і номер (індекс і номер ідентичного стандарту)	Обєкт стандартизації	Найменування підрозділу
ДСТУ ISO/IEC TR 15504–7:2003 (ISO/IEC TR 15504–7:1998, IDT)		Оцінювання процесів життєвого циклу програмних засобів. Частина 7. Настанови з удосконалювання процесу
ДСТУ ISO/IEC TR 15504–8:2003 (ISO/IEC TR 15504–8:1998, IDT)		Оцінювання процесів життєвого циклу програмних засобів. Частина 8. Настанови з визначання потужності процесу постачальника
ДСТУ ISO/IEC TR 15504–9:2003 (ISO/IEC TR 15504–9:1998, IDT)		Оцінювання процесів життєвого циклу програмних засобів. Частина 9. Словник термінів
ДСТУ ISO/IEC 15939:2008 (ISO/IEC 15939:2007, IDT)	Інженерія систем і програмних засобів	Процес вимірювання
ДСТУ ГОСТ 31078:2004 (ГОСТ 31078–2002, IDT)	Захист інформації	Випробовування програмних засобів на наявність комп'ютерних вірусів. Типова настанова
ДСТУ ISO/IEC 90003:2006 (ISO/IEC 90003:2004, IDT)	Програмна інженерія	Настанови щодо застосування ISO 9001:2000 до програмного забезпечення
ДСТУ ISO/IEC TR 9126–2:2008 (ISO/IEC TR 9126–2:2003, IDT)		Якість продукту. Ч. 2. Зовнішні метрики
ГСТУ 3–71–24–94	Специфікація програмних виробів	Вимоги до змісту та оформлення.
ГОСТ 19.001–77	Единая система программной документации	Общие положения
ГОСТ 19.002–80		Схемы алгоритмов и программ. Правила выполнения
ГОСТ 19.004–80		Термины и определения
ГОСТ 19.005–85		Р-схемы алгоритмов и программ. Обозначения условные графические и правила выполнения
ГОСТ 19.101–77		Виды программ и программных документов
ГОСТ 19.102–77		Стадии разработки
ГОСТ 19.103–77		Обозначение програм и программных документов
ГОСТ 19.104–78		Основные надписи
ГОСТ 19.105–78		Общие требования к программным документам

Продовження табл. Д.1.1

Індекс і номер (індекс і номер ідентичного стандарту)	Обєкт стандартизації	Найменування підрозділу
ГОСТ 19.106–78		Требования к программным документам, выполненым печатным способом
ГОСТ 19.201–78		Техническое задание. Требования к содержанию и оформлению
ГОСТ 19.202–78		Спецификация. Требования к содержанию и оформлению
ГОСТ 19.301–79		Программа и методика испытаний. Требования к содержанию и оформлению
ГОСТ 19.401–78		Текст программы. Требования к содержанию и оформлению
ГОСТ 19.402–78		Описание программы
ГОСТ 19.403–79		Ведомость держателей подлинников
ГОСТ 19.404–79		Пояснительная записка. Требования к содержанию и оформлению
ГОСТ 19.501–78		Формуляр. Требования к содержанию и оформлению
ГОСТ 19.502–78		Описание применения. Требования к содержанию и оформлению
ГОСТ 19.503–79		Руководство системного программиста. Требования к содержанию и оформлению
ГОСТ 19.504–79		Руководство программиста. Требования к содержанию и оформлению
ГОСТ 19.505–79		Руководство оператора. Требования к содержанию и оформлению
ГОСТ 19.506–79		Описание язика. Требования к содержанию и оформлению
ГОСТ 19.507–79		Ведомость эксплуатационных документов
ГОСТ 19.508–79		Руководство по техническому обслуживанию. Требования к содержанию и оформлению
ГОСТ 19.601–78		Общие правила дублирования, учета и хранения

Закінчення табл. Д.1.1

Індекс і номер (індекс і номер ідентичного стандарту)	Обєкт стандартизації	Найменування підрозділу
ГОСТ 19.602–78		Правила дублирования, учета и хранения программных документов, выполненных печатным способом
ГОСТ 19.603–78		Общие правила внесения изменений
ГОСТ 19.604–78		Правила внесения изменений в программные документы, выполненые печатным способом

Таблиця Д.1.2

Організації-розробники міжнародних стандартів у сфері програмної інженерії

Скорочена назва	Назва англійською мовою	Назва українською мовою	Види стандартів
ISO	International Organiza tion for Standardization	Міжнародна організація зі стандартизації	Міжнародні стандарти: – якості ISO 9000; – процесів життєвого циклу ПЗ ISO/IEC 12207
SEI	Software Engineering Institute	Інститут програмної інженерії	Програмна інженерія: зрілості організацій, що розробляють ПЗ (CMM™)
PMI®	Project Management Institute	Інститут керування проектами	Загальний менеджмент проектів: основи знань у галузі керування проектами
IEEE	Institute of Electrical and Electronics Engineers	Інститут інженерів електротехніки й електроніки	Зведення стандартів з програмної інженерії

Додаток 2

ЗРАЗОК ОФОРМЛЕННЯ ТЕХНІЧНОГО ЗАВДАННЯ

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
Факультет літакобудування
Кафедра інформаційних технологій проектування

«ЗАТВЕРДЖЕНО»
Завідувач кафедри
_____ Є. А. Дружинін
«___»_____ 2015 р.

ПРОГРАММА РОЗРАХУНКУ НАЙМЕНШОЇ ВІДСТАНІ

Технічне завдання

«УЗГОДЖЕНО»
Керівник проекту:
_____ О. К. Погудіна
Виконавець:
_____ І. І. Іванов

2015

1. НАЙМЕНУВАННЯ І ГАЛУЗЬ ЗАСТОСУВАННЯ

Назва розробки: програма розрахунку найменшої відстані.

Галузь застосування: інформаційні технології.

2. ПІДСТАВИ ДЛЯ РОЗРОБЛЕННЯ

Підставою для розроблення є домашнє завдання, затверджене кафедрою інформаційних технологій проектування Національного аерокосмічного університету ім. М.Є.Жуковського «ХАІ», варіант №1.

3. ПРИЗНАЧЕННЯ РОЗРОБКИ

Розробку призначено для використання як інформаційного забезпечення роботи кафедри з метою демонстрації принципу роботи алгоритму Дейкстри.

4. ВИМОГИ ДО ПРОГРАМНОГО ПРОДУКТУ

4.1. Вимоги до функціональних характеристик:

1) введення вхідних даних викладачем в консольному режимі або з використанням текстового файла; граф має бути у вигляді матриці, де строка і стовпець відображають номер вершини (номер пункту), а в клітинку внесено вагу ребра (відстань між двома пунктами);

2) перевірка коректності даних;

3) уведення ідентифікаційних даних студента (прізвище, ім’я та по батькові) для реєстрації в системі;

4) випадковий вибір ПЗ (коректний або випадковий результат вихідних даних для демонстрації студенту);

5) уведення вершини, від якої необхідно знайти найменші маршрути до всіх інших;

6) розрахунок маршрута з використанням алгоритму Дейкстри або випадкових маршрутів (відповідно до п. 4) і виведення маршрутів від вибраної студентом вершини до всіх інших;

7) запитання до студента: чи є коректним маршрут;

8) оцінювання знань студента стосовно принципу роботи алгоритму Дейкстри (за 5-бальною шкалою).

Таблиця Д.2.1

Вихідні дані

Вихідні дані	Тип даних	Діапазон	Приклад
Граф: – кількість вершин – вага ребра	 int int	 0…10 0…99	0 5 3 5 0 14 3 14 0 Кількість вершин 3 Вага 1–2 : 5 Вага 1–3 : 3 Вага 2–3 : 14
Прізвище, ім'я по батькові	char [100]	0…100 символів	Іванов В. П.
Номер вершини	int	Залежить від кількості вершин	3
Оцінювання знань	bool	0…5	0 – «треба повторити» 5 – «відмінно»

4.2. Вимоги до надійності:

– контроль вхідних даних;

– блокування некоректних дій користувача при роботі з системою.

4.3. Умови експлуатації: дві категорії користувачів – викладач і студент, які мають необхідні навички для роботи з ПК.

4.4. Вимоги до складу і параметрів технічних засобів: система має працювати на IBM сумісних ПК. Мінімальна конфігурація:

– тип процесора – Pentium і вище;

– місткість пам'яті, що займає ПЗ, – 2 Мб.

4.4. Вимоги до інформаційної і програмної сумісності: розроблення виконати на платформі Microsoft Visual Studio з використанням мови програмування C++.

4.5. Вимоги до маркування й упаковки – не передбачено.

4.6. Вимоги до транспортування й зберігання: усі файли проекту мають знаходитися в каталозі 231180/ДЗТСПП.

5. ТЕХНІКО-ЕКОНОМІЧНІ ПОКАЗНИКИ

Розроблення забирає 60 годин, одна година роботи виконавця з урахуванням розміру стипендії: 720,70 / 176 = 4,10 грн, тому вартість розробки становить 246,00 грн.

6. ВИМОГИ ДО ПРОЕКТНОЇ ДОКУМЕНТАЦІЇ

Під час виконання проекту необхідно розробити таку документацію:

1) пояснювальна записка;

2) діаграми UML: варіантів використання, класів, діяльності, послідовності, компонентів;

3) лістинг ПЗ і посібник користувача;
4) програма й методика тестування.

7. ЕТАПИ ПРОЕКТУВАННЯ

Розроблення й узгодження технічного завдання ……..... 30.09.2014
Розроблення діаграм ……..... 10.10.2014
Програмна реалізація ……..... 30.10.2014
Тестування ПЗ ……..... 15.11.2014

8. ПОРЯДОК ТЕСТУВАННЯ РОЗРОБКИ

Контроль ПЗ відбувається поетапно. Робота оцінюється від 0 до 5 балів, невиконання в строк – від 0 до 2 балів. Максимальна кількість балів за документацію і ПЗ – 20.

ЛІСТИНГ ПРОГРАМИ, ЩО ЗДІЙСНЮЄ ЗАВАНТАЖЕННЯ .BMP ФАЙЛА НА ЕКРАН

```
#include <stdlib.h>
#include <graphics.h>
#include <conio.h>
#include <stdio.h>
#include <math.h>
#include <string.h>
#include <alloc.h>
struct BmpHeader
{
unsigned int bfType;
long bfSize;
unsigned int bfReserved1;
unsigned int bfReserved2;
long bfOffBits;
long biSize;
long biWidth;
long biHeight;
unsigned int biPlanes;
unsigned int biBitCount;
long biCompression;
long biSizeImage;
long biXPelsPerMeter;
long biYPelsPerMeter;
long biClrUsed;
long biClrImportant;
};
const int Color[16] =
{0,4,2,6,1,5,3,7,8,12,10,14,9,13,11,1
5};
int main()
{
int x0=0, y0=0;
char * path;
struct BmpHeader Sagolovok;
int x,y,j,nb;
short int b;
FILE *BMP;
printf("Enter path : ");
scanf("%s",path);
if ((BMP=fopen(path,"rt"))==NULL)
{printf("Error. No file."); return 1;}
fread(&Sagolovok,54,1,BMP);
int graphdriver = DETECT,
graphmode;
initgraph(&graphdriver,
&graphmode, "..\\bgi");
nb=(Sagolovok.biWidth / 8)*4;
if ((Sagolovok.biWidth % 8) != 0)
{
nb=nb+4;
}
fseek(BMP,Sagolovok.bfOffBits,SEE
K_SET);
for (y=y0+Sagolovok.biHeight;
y>=y0+1 ; y--)
{
x=x0;
for(j=1 ; j<=nb ; j++)
{
b=fgetc(BMP);
if(x-x0<Sagolovok.biWidth)
{
putpixel(x,y,Color[b>>4]);
x++;
}
if(x-x0<Sagolovok.biWidth)
{
putpixel(x,y, Color[b & 15] );
x++;
}
}
}
fclose(BMP);
getch();
closegraph();
return 0;}
```

Додаток 4

ПРОГРАМНИЙ КОД ДЛЯ ВИКОНАННЯ ЗАВДАНЬ ДЛЯ САМОСТІЙНОГО ВИРІШЕННЯ

/* Пошук найкоротших шляхів між вершинами ґрафа.
Як вихідні дани використовується матриця довжин ребер ґрафа (МДРҐ), що читається з текстового файла

Приклад файла, що містить опис ґрафа:

1000	10	30	1000	1000	100
10	1000	80	1000	50	1000
1000	1000	1000	40	1000	10
30	1000	1000	1000	1000	60
1000	1000	70	1000	1000	1000
1000	1000	1000	1000	20	1000

У кінці файла не повинно бути порожніх записів, а також пробілів на початку і в кінці запису. Значення 1000 означає відсутність ребра.*/

```
#include "stdafx.h"
#include "Dijkstra.h"
#include <ConIO.h>
#include <fstream>
#ifdef _DEBUG
#define new DEBUG_NEW
#endif
CWinApp theApp;
using namespace std;
void FreeMemory(int ** Graph,int VertexCount)
{
        for(int i=0;i<VertexCount;i++) delete [] Graph[i];
        delete Graph;
}
int MagicNumber=1000;
/* Магічне число, що означає у файлі ґрафа довжину неіснуючого ребра */
/*ReadGraphData − читання даних про ґраф з файла */
void ReadGraphData(CString FName, // Ім'я вхідного файла
                   int **&Graph,  // МДРҐ
                   int &VertexCount)//Кількість вершин ґрафа. Vertex==0 - по-
милка у файлі
{
    VertexCount=0;
    ifstream File; // Файл для зчитування
```

```
        File.open(FName);
        if(!File)return;
        /* Підраховуємо кількість записів файла, щоб виділити пам'ять для
        матриці МДРҐ і прочитати її */
        int NumRec=0;
        char Buffer[10240]; //Максимальна кількість символів у записі файла

        while(!File.eof())
                {
                File.getline(Buffer,sizeof(Buffer),'\n');
                if(!strlen(Buffer)) return; // У файлі не повинно бути порожніх за-
писів
                NumRec++;
                }
                File.seekg(0,ios_base::beg); //Переміщаємося на початок файла
!!!                     /* Виділяємо пам'ять для матриці МДРҐ */
                Graph=new int* [NumRec];
                for(int i=0;i<NumRec;i++) Graph[i]=new int [NumRec];
                /* Читаємо файл і заповнюємо матрицю МДРҐ */
                char Next;
                bool Error=false;
                for(int i=0;i<NumRec;i++)
                {
                        for(int j=0;j<NumRec;j++)
                        {
                                File>>Graph[i][j];
                        }
                        Next=File.peek();
                        Error=(Next!='\n')&&(Next!=EOF);
                        if(Error) break;
                }
                if(Error)
                {
                        FreeMemory(Graph,VertexCount);
                        VertexCount=0;
                        return;
                }
                VertexCount=NumRec;
}

/*FindMinDistance − пошук найкоротших шляхів з початкової вершини */
void FindMinDistance(int ** Graph, int StartVertex, int VertexCount)
{
```

```
        int *Distance,   // мінімальний маршрут
        count,           // кількість обходів
        index, u;    //u, index − наступна вершина
        bool *Visited;       // відвідані вершини
        int * VertexOrder; // номери вершин у порядку обходу
        Distance = new int [VertexCount];
        Visited        = new bool [VertexCount];
        VertexOrder = new int [VertexCount];
        for (int i = 0; i < VertexCount; i++) // ініціалізуємо масиви
        {
                Distance[i] = INT_MAX;
                Visited[i] = false;
                VertexOrder[i]=0;
        }
        Distance[StartVertex-1] = 0; //початкова вершина
        cout<< endl<<"Обхід вершин за алгоритмом Дейкстри:" << StartVertex;
        for (count = 0; count < VertexCount; count++) // цикл для всієї кількості
обходів
        {
        int min = INT_MAX;
        for (int i = 0; i < VertexCount; i++) // пошук вершини, відстань до якої є
мінімальною
        if (!Visited[i] && (Distance[i] < min) )
                {
                        min = Distance[i];
                        index = i;
                }
        if (index != StartVertex-1) VertexOrder[count]=index + 1;
        u = index;   // запам'ятовуємо вершину
        Visited[u] = true; // робимо вершину відвіданою
        for (int i = 0; i < VertexCount; i++)
         // якщо вершину не відвідано й сусідні вершини є суміжними
                if ((!Visited[i]) && (Distance[u] != INT_MAX) &&
                   (Distance[u] + Graph[u][i] < Distance[i]))
                Distance[i] = Distance[u] + Graph[u][i];
        }
        // виведення результатів
        cout<<" Номери вершин у порядку обходу: "<<StartVertex<<' ';
        for (int i = 1; i < VertexCount; i++) cout<<VertexOrder[i]<<' ';
        cout<<endl;
        cout<<endl<<" Довжина шляху з початкової вершини до інших: \ t \ n ";
        for (int i = 0; i < VertexCount; i++)
        {
```

```
        if(i==StartVertex-1) continue;
        if ((Distance[i] != INT_MAX))
                cout << StartVertex << " -> " << i + 1 << " = " << Distance[i] << endl;

        else  cout<<StartVertex<<" -> "<<i+1<<"=" << "маршрут є недоступним
"       << endl;
        }
delete []     Distance;
delete [] Visited;
delete [] VertexOrder;
}
/* GetFileName − уводить ім'я файла з допомогою стандартного діалогу
Windows.
   Повертає false, якщо користувач відмовився від уведення імені файла */
bool GetFileName(CString &FName)
{
        CFileDialog dlg(TRUE,_T("txt"),_T("*.txt")); //FALSE - збереження, TRUE
- відкриття
        if(dlg.DoModal()==IDOK) /* виклик діалогового вікна */
                {
                        FName=dlg.GetPathName();
                        /* Функція dlg.GetPathName () повертає повне ім'я файла */
                        return true;
                }
        else  return false; /* Користувач завершив діалог натисканням кнопки
Cancel*/
}

void OutputGraphData(int ** Graph, int VertexCount)
{
        cout << endl << "Кількість вершин ґрафа == "<<VertexCount<<endl;
        for (int i = 0; i < VertexCount; i++)  cout << "\t" << i + 1;
        cout << endl << endl;
        for (int i = 0; i < VertexCount; i++)
        {
                cout << i + 1;
                for (int j = 0; j < VertexCount; j++)
                if (Graph[i][j] == MagicNumber) cout << "\t-" << " ";
                        else cout << "\t" << Graph[i][j] << " ";
                cout << endl;
        }
}
```

```
void main()
{
	// initialize MFC and print and error on failure
	if (!AfxWinInit(::GetModuleHandle(NULL), NULL, ::GetCommandLine(),
0))
	{
		_tprintf(_T("Fatal Error: MFC initialization failed\n"));
		_getch(); return;
	}
		setlocale(LC_ALL, "ukr");
	// Запитуємо ім'я файла, що містить матрицю довжин ребер ґрафа
(МДРҐ)			CString FName;
		if(!GetFileName(FName)) return;
		int VertexCount;  // кількість вершин графа
		int **Graph; // МСРГ
		ReadGraphData(FName,Graph,VertexCount);	// Читаємо
МДРҐ з файла
		if(!VertexCount) // Помилки при читанні МДРҐ з файла
		{
			cout<<" Помилка при читанні файла"<<endl;
			_getch();
			return;
		}
		// Висновок МДРҐ на екран
		OutputGraphData(Graph, VertexCount);
		int StartVertex;
		do
		{
		cout << endl << "Уведіть номер початкової вершини
		[1.."<<VertexCount<<"]>";
			cin >> StartVertex;
		}while((StartVertex<1)||(StartVertex>VertexCount));
		// Пошук найкоротших шляхів з початкової вершини
		FindMinDistance(Graph, StartVertex, VertexCount);
		FreeMemory(Graph,VertexCount);
	 _getch();
}
```

Навчальне видання

Погудіна Ольга Костянтинівна
Крицький Дмитро Миколайович
Бабак Ірина Миколаївна та ін.

ТЕХНОЛОГІЯ СТВОРЕННЯ ПРОГРАМНИХ ПРОДУКТІВ

Редактор О. Ф. Серьожкіна

Зв. план, 2015
Підписано до друку 25.12.2015
Формат 60×84 1/16. Папір офс. № 2. Офс. друк
Ум. друк. арк. 8,9. Обл.-вид. арк. 10. Наклад 100 пр.
Замовлення 246. Ціна вільна

Видавець і виготовлювач
Національний аерокосмічний університет ім. М. Є. Жуковського
«Харківський авіаційний інститут»
61070, Харків-70, вул. Чкалова, 17
http://www.khai.edu
Видавничий центр «ХАІ»
61070, Харків-70, вул. Чкалова, 17
izdat@khai.edu

Свідоцтво про внесення суб'єкта видавничої справи
до Державного реєстру видавців, виготовлювачів і розповсюджувачів
видавничої продукції сер. ДК № 391 від 30.03.2001