МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ
імені ІГОРЯ СІКОРСЬКОГО»

С. О. Цибульник,
К. С. Барандич

ТЕХНОЛОГІЇ РОЗРОБЛЕННЯ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

ЧАСТИНА 1. ЖИТТЄВИЙ ЦИКЛ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

Підручник

Затверджено Вченою радою КПІ ім. Ігоря Сікорського
як підручник для здобувачів ступеня бакалавра
за спеціальністю 151 «Автоматизація та комп’ютерно-інтегровані технології»

Електронне мережне навчальне видання

Київ
КПІ ім. Ігоря Сікорського
2022

Рецензенти: *Сергій Савастюк, к.ф.-м.н., CEO, Tickeron Inc., USA*

Тертичний Віктор, провідний програміст ТОВ «Гіротех», Україна

Матвієнко Сергій, к.т.н., начальник лабораторії комп'ютерних мереж ТОВ «ВІРКОМ», Україна

Відповідальний редактор: *Павловський О.М.,* к. т. н., доцент кафедри комп'ютерно-інтегрованих оптичних та навігаційних систем КПІ ім. Ігоря Сікорського

Гриф надано Вченою радою КПІ ім. Ігоря Сікорського
(протокол № 6 від 03.10.2022 р.)

Підручник складено для студентів, які навчаються за спеціальністю 151 «Автоматизація та комп'ютерно-інтегровані технології». Даний підручник також може використовуватися студентами, які навчаються в інших закладах вищої освіти або на інших спеціальностях.

Метою даного навчального видання є допомога студентам в отриманні необхідних знань та умінь з об'єктно-орієнтованого програмування, основ проектування архітектури програмного забезпечення, використання патернів та принципів програмування при вирішенні задач розроблення програмного забезпечення.

Першу частину підручника присвячено основам технології розроблення програмного забезпечення, зокрема, на основі світового досвіду та результатів опитування персоналу провідних компаній-розробників розглядаються основні причини невиконання проектів та етапи життєвого циклу, які проходить програмне забезпечення від моменту появи ідеї до моменту виведення його з експлуатації.

Окремою глобальною темою розглядаються питання організації процесу розроблення програмного забезпечення. У першу чергу це: формування вимог, планування проекту та керування персоналом. Систематизовані та наведені в останній частині матеріали дозволять узагальнити та конкретизувати основні теоретичні положення, на які необхідно звернути увагу при управлінні проектом з розроблення програмного забезпечення.

Остання частина підручника охоплює перелік питань, які описують процес проектування архітектури програмного забезпечення. За даним напрямком наводиться опис основних класів архітектури, які використовуються на сучасному етапі розвитку ІТ-сфери, розглядаються архітектурні стилі, моделі та шаблони, які використовуються в об'єктно-орієнтованому програмуванні для розроблення якісного програмного забезпечення.

Реєстр. № 22/23-007 Обсяг 12,25 авт. арк.

Національний технічний університет України
«Київський політехнічний інститут імені Ігоря Сікорського»
проспект Перемоги, 37, м. Київ, 03056
https://kpi.ua
Свідоцтво про внесення до Державного реєстру видавців, виготовлювачів
і розповсюджувачів видавничої продукції ДК № 5354 від 25.05.2017 р.

© С. О. Цибульник, К. С. Барандич
© КПІ ім. Ігоря Сікорського, 2022

ЗМІСТ

СКОРОЧЕННЯ ТА УМОВНІ ПОЗНАКИ………………………………………… 6
ВСТУП…………………………………………………………………………… 7
РОЗДІЛ 1. ОСНОВИ ТЕХНОЛОГІЇ РОЗРОБЛЕННЯ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ…………………………………………………………………… 9
1.1 Сучасний стан сфери розроблення програмних засобів………………….. 9
1.2 Перспективи розвитку інформаційних технологій……………………….. 13
1.3 Структурний та об’єктно-орієнтований принципи розроблення програмного забезпечення…………………………………………………………. 15
1.3.1 Непроцедурні парадигми…………………………………………………… 15
1.3.2 Структурне програмування………………………………………………… 18
1.3.3 Об’єктно-орієнтоване програмування…………………………………... 22
1.4 Життєвий цикл……………………………………………………………… 24
1.4.1 Стратегії розроблення програмного забезпечення……………………... 31
1.4.2 Моделі життєвого циклу…………………………………………………… 37
1.5 Гнучке розроблення програмного забезпечення………………………….. 47
Контрольні запитання…………………………………………………………… 51
РОЗДІЛ 2. ОРГАНІЗАЦІЯ ПРОЦЕСУ РОЗРОБЛЕННЯ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ ……………………………………………………………... 52
2.1 Процеси управління………………………………………………………… 54
2.2 Управління проектом……………………………………………………….. 56
2.2.1 План проекту………………………………………………………………… 61
2.2.2 Контрольні позначки етапів робіт……………………………………….. 62
2.2.3 Графік робіт………………………………………………………………….. 63
2.3 Вимоги до програмного забезпечення……………………………………... 75
2.3.1 Функціональні вимоги……………………………………………………… 78
2.3.2 Нефункціональні вимоги…………………………………………………… 80
2.3.3 Вимоги предметної області………………………………………………… 86
2.4 Методи визначення вимог………………………………………………….. 88

2.4.1 Інтерв'ювання…………………………………………………………... 88
2.4.2 Мозковий штурм і відбір ідей…………………………………………. 91
2.4.3 Спільне розроблення додатків………………………………………… 94
2.4.4 Розкадрування…………………………………………………………. 96
2.4.5 Обігравання ролей……………………………………………………... 98
2.4.6 Швидке прототипування………………………………………………. 100
2.5 Формалізація вимог……………………………………………………. 101
2.6 Створення прототипу програмного забезпечення…………………… 106
2.6.1 Прототипування у процесі розроблення програмного забезпечення…… 109
2.6.2 Технології швидкого прототипування……………………………... 117
2.6.3 Прототипування інтерфейсів користувача…………………………… 126
2.7 Управління персоналом при реалізації проектів…………………….. 127
2.7.1 Організація людської пам'яті…………………………………………. 128
2.7.2 Рішення задач…………………………………………………………... 133
2.7.3 Мотивація………………………………………………………………. 135
2.7.4 Групова робота…………………………………………………………. 138
2.7.5 Підбір та збереження персоналу……………………………………… 147
2.7.6 Робоче середовище……………………………………………………... 150
2.7.7 Модель оцінки рівня розвитку персоналу……………………………. 152
Контрольні запитання…………………………………………………………. 155
РОЗДІЛ 3. АРХІТЕКТУРНЕ ПРОЕКТУВАННЯ……………………………. 158
3.1 Основи архітектурного проектування……………………………….. 158
3.2 Бажані характеристики проекту………………………………………. 163
3.3 Системна архітектура………………………………………………….. 165
3.3.1 Архітектурні моделі……………………………………………………. 166
3.3.2 Архітектурні парадигми……………………………………………….. 173
3.3.3 Сценарії використання………………………………………………… 181
3.3.4 Приклади архітектурних стилів та моделей…………………………. 186
3.3.5 Документування архітектури…………………………………………. 214
Контрольні запитання…………………………………………………………. 222

РОЗДІЛ 4. ПРИНЦИПИ SOLID ТА ШАБЛОНИ ОБ'ЄКТНО-ОРІЄНТОВАНОГО ПРОГРАМУВАННЯ…………………………………………… 224
4.1 Принцип єдиної відповідальності……………………………….. 225
4.2 Принцип відкритості/закритості………………………………… 236
4.3 Принцип заміни (підстановки) Барбари Лісков………………… 243
4.4 Принцип поділу інтерфейсів…………………………………….. 245
4.5 Принцип інверсії залежностей…………………………………... 247
4.6 Шаблони проектування GRASP та інші………………………… 250
4.6.1 Шаблони GRASP……………………………………………… 251
4.6.2 Шаблони «Банди Чотирьох»…………………………………….. 260
4.6.3 Інші принципи програмування…………………………………... 263
Контрольні запитання……………………………………………………. 267
ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ……………………………………….. 268

СКОРОЧЕННЯ ТА УМОВНІ ПОЗНАКИ

БД – база даних

ЖЦ – життєвий цикл

ІТ – інформаційні технології

НВ – нефункціональні вимоги

ООП – об’єктно-орієнтоване програмування

ООП – об’єктно-орієнтований підхід

ПЗ – програмне забезпечення

ПП – програмний продукт

ПС – програмна система

ТЗ – технічне завдання

ФВ – функціональні вимоги

ФС – формальна специфікація

MVP – Minimum Viable Product, мінімально життєздатний продукт

ВСТУП

Підручник складено відповідно до силабусу навчальної дисципліни «Технології розроблення програмного забезпечення» для студентів приладобудівного факультету, які навчаються за спеціальністю 151 «Автоматизація та комп’ютерно-інтегровані технології» за освітньо-професійною програмою «Комп’ютерно-інтегровані системи та технології в приладобудуванні». Даний підручник також може використовуватися студентами інших спеціальностей або освітніх програм.

Нині бракує літератури, яка б максимально широко охоплювала усі етапи процесу розроблення програмного забезпечення. Найбільш відомими у вітчизняній практиці є: «Інженерія програмного забезпечення» [1], «Розроблення вимог до програмного забезпечення» [2], «Чистий код: створення, аналіз та рефакторинг» [3], книги, підручники та навчальні посібники для різних етапів розроблення [4-13]. Проте, наведені видання не є узагальнюючими та не завжди в повному обсязі містять відповіді на питання, які стосуються усіх етапів розроблення програмного забезпечення.

Саме тому, метою даного навчального видання є допомога студентам в отриманні необхідних знань та умінь з об’єктно-орієнтованого програмування, основ проектування архітектури програмного забезпечення, використання шаблонів та принципів програмування при вирішенні задач розроблення програмного забезпечення.

Перший розділ підручника присвячено основам технології розроблення програмного забезпечення, зокрема, на основі світового досвіду та результатів опитування персоналу провідних компаній-розробників розглядаються основні причини невиконання проектів, сучасний стан IT-сфери та прогнози щодо подальшого її розвитку в майбутньому. Також розглядаються етапи життєвого циклу, які проходить програмне забезпечення від моменту появи ідеї до моменту виведення його з експлуатації. Детально описуються основні стратегії розроблення програмного забезпечення та моделі, які реалізуються в рамках даних

стратегій.

Окремою глобальною темою в межах другого розділу підручника розглядаються питання організації процесу розроблення програмного забезпечення. У першу чергу це: формування вимог, планування проекту та керування персоналом. У багатьох випадках програмне забезпечення не доводиться до етапу реалізації саме через помилки керівників та менеджерів у прийнятті рішень щодо внесення змін до ключових етапів розроблення або розподілу ресурсів (часу, коштів, людей, тощо). Саме тому систематизовані та наведені у даному розділі матеріали дозволять узагальнити та конкретизувати основні теоретичні положення, на які необхідно звернути увагу при управлінні проектом з розроблення програмного забезпечення.

Третій розділ підручника «Технології розроблення програмного забезпечення» охоплює перелік питань, пов'язаних з проектуванням архітектури програмного забезпечення. За даним напрямком наводиться опис основних класів архітектури, які використовуються на сучасному етапі розвитку ІТ-сфери, розглядаються архітектурні стилі, моделі та шаблони, починаючи з класичного файл-север і закінчуючи найбільш використовуваним сьогодні MVVM. Також у даному розділі підручника розглядається декілька варіантів документування архітектурних рішень для надання програмістам повної та всеосяжної інформації про структуру розроблюваного програмного забезпечення.

У четвертому розділі детально описуються (з наведенням деяких прикладів) сучасні принципи та шаблони, які використовуються в об'єктно-орієнтованому програмуванні для розроблення якісного програмного забезпечення.

Опрацювання контрольних запитань, пов'язаних з викладеним у даному підручнику теоретичним матеріалом, сприятиме закріпленню, поглибленню та узагальненню теоретичних основ дисципліни, а також розвитку навичок самостійної творчої роботи студентів у процесі їх навчання, зокрема при виконанні практичних, курсових та інших видів робіт з дисципліни «Технології розроблення програмного забезпечення».

РОЗДІЛ 1

ОСНОВИ ТЕХНОЛОГІЇ РОЗРОБЛЕННЯ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

1.1 Сучасний стан сфери розроблення програмних засобів

У даний час в усі сфери діяльності людини широко впроваджуються інформаційні технології. Процес зростання інтересу до ІТ-сфери призводить до розроблення величезної кількості програмних засобів різного функціонального призначення, обсяг і складність яких постійно зростають.

У зв’язку з цим з кожним роком деякі методології розроблення програмного забезпечення (ПЗ), які застосовувалися на початкових етапах розвитку комп’ютерних та інформаційних технологій, втрачають свої позиції. Це відбувається через те, що вони не дозволяють отримати ПЗ необхідного рівня якості за заданий проміжок часу при обмежених ресурсах (фінансових, людських, технічних, тощо) і може бути пов’язано з багатьма причинами, наприклад:

- по-перше, інтуїтивний підхід до розроблення ПЗ, заснований на знаннях, уміннях і талантах окремих програмістів-одинаків, не дозволяє розробляти складні ПЗ і суперечить принципам їх колективного розроблення;
- по-друге, використання колективних методів розроблення вимагає структурованого підходу до понять життєвого циклу (ЖЦ) і моделі життєвого циклу програмного забезпечення. В іншому випадку виникають істотні ризики не довести проект до завершення або не одержати продукт задовільної якості;
- по-третє, використовувані методології розроблення ПЗ із зростанням їх складності і критичності перестають задовольняти цілям і завданням, які стоять перед їх розробниками;
- по-четверте, зростання складності та обсягу розроблюваних ПЗ автоматично призводить до появи досить складних у застосуванні методологій аналізу, проектування і подальших етапів розроблення.

Використання таких методологій стає неможливим без застосування інструментальних засобів підтримки.

Вищеназвані причини найчастіше призводять до незадовільних результатів виконання проектів.

Pulse of the Profession® [14] – це глобальне опитування практиків з управління проектами, яке проводиться з 2006 року. Дане опитування показує основні тенденції управління проектами зараз і прогноз на майбутнє, а також містить оригінальне дослідження ринку, яке відображає відгуки та уявлення керівників проектів чи програм, власників портфелів (портфель, *фінанси* – сукупність активів юридичної або фізичної особи), а також аналіз даних третіх сторін.

Видання Pulse за 2017 рік [14] містить відгуки та висновки від 3234 фахівців з управління проектами, 200 вищих керівників та 510 директорів офісів з управління проектами з різних галузей промисловості, а також інтерв'ю з 10 корпоративними лідерами. Респонденти охоплюють Північну Америку; Азіатсько-Тихоокеанський регіон; Європу, Близький Схід, Африку, регіони Латинської Америки та Карибського басейну.

Отже, у загальній практиці розроблення ПЗ (рис. 1.1-рис. 1.2) близько 10-20% усіх проектів не доходили до завершення, 40-60% проектів не виконуються вчасно (середній проект завершується із запізненням на 150-200%), 25-40% проектів реалізують поставлені завдання не в повному обсязі, 40-55% проектів потребували додаткового фінансування, у 20% проектів не всі зміни, що вносилися замовником (замовники беруть участь у ЖЦ ПЗ), бралися до уваги, а також однією з причин незавершення проектів є відсутність або неадекватне спілкування в команді розроблення.

Усі ці причини провалу розроблення ПЗ призвели до того, що в останні десятиліття у всьому світі провідними фахівцями в області теорії і практики розроблення програмного забезпечення активно виконуються роботи з удосконалення підходів до розроблення ПЗ. Ці роботи ведуться в різних напрямках. Основними з них є наступні:

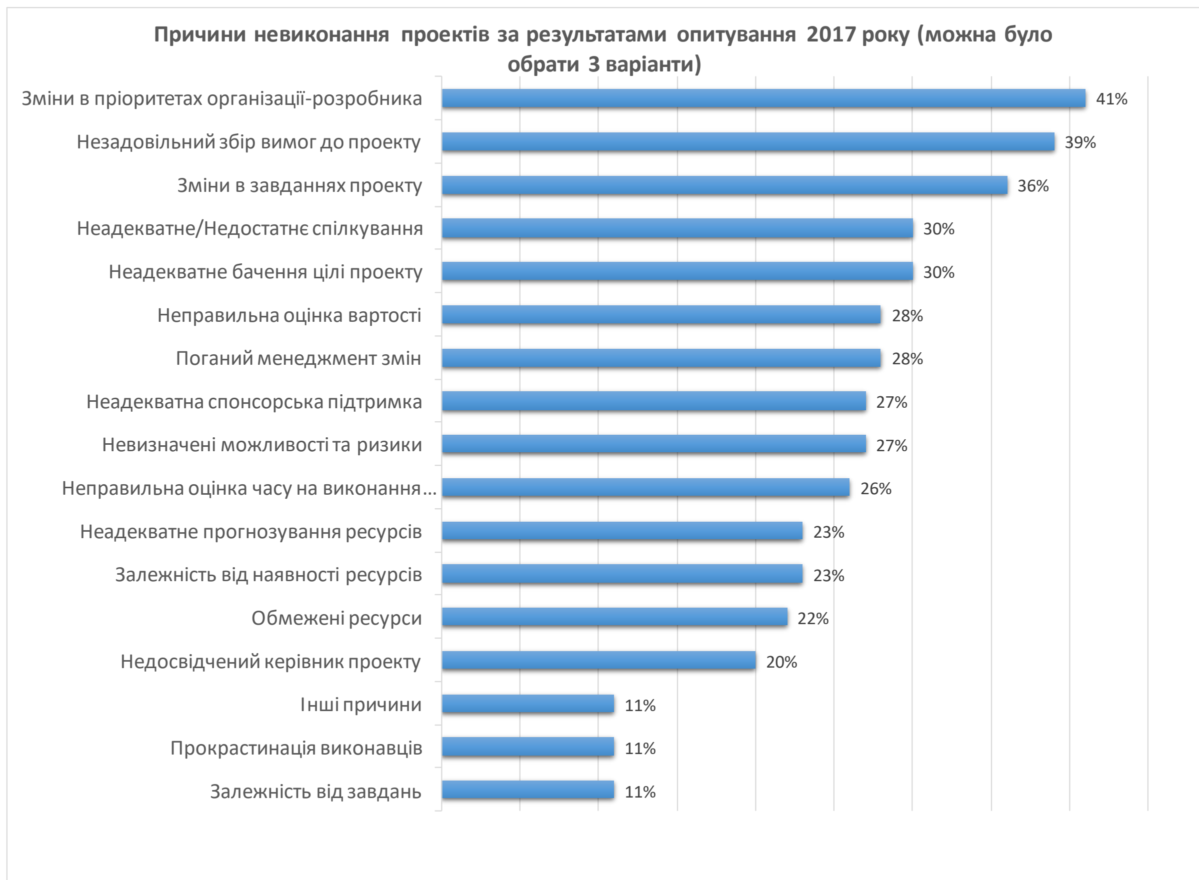

Рис. 1.1. Результати опитування за 2017 рік

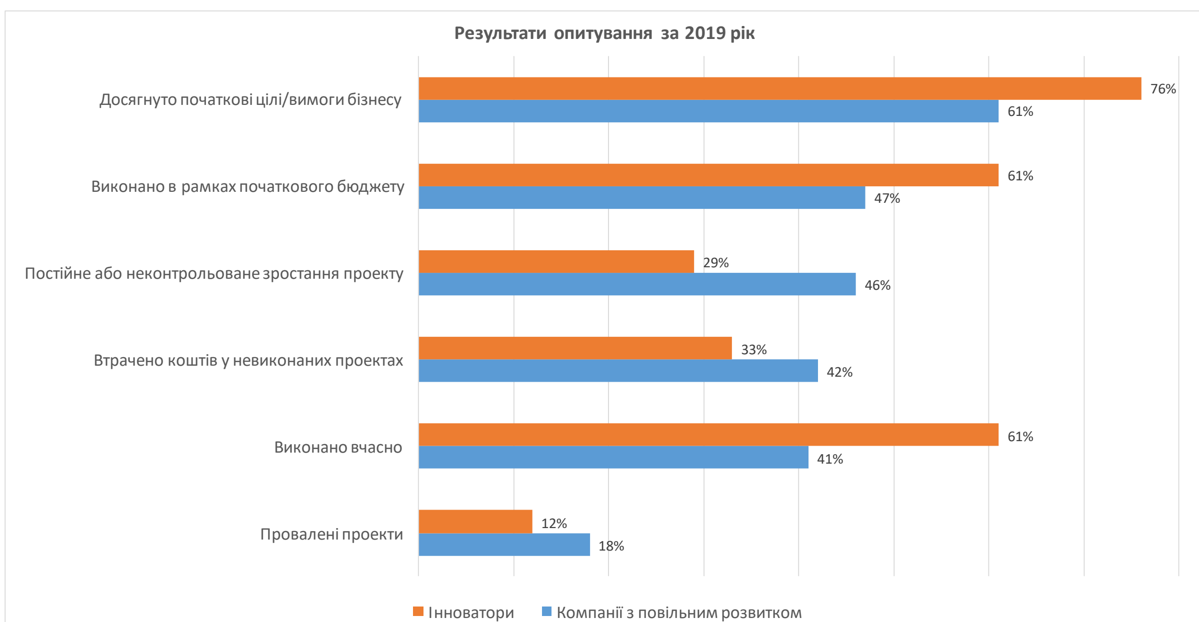

Рис. 1.2. Результати опитування за 2019 рік

1. **Стандартизація життєвого циклу програмних засобів**. У даний час розробляється і постійно оновлюється велика кількість міжнародних і національних стандартів, присвячених різним аспектам ЖЦ ПЗ.

2. **Структуризація моделей життєвого циклу програмних засобів**. З 80-х р XX століття ведуться роботи з удосконалення стратегій розроблення ПЗ і створення моделей ЖЦ, що реалізують дані стратегії. У даний час широко використовуються три базові стратегії розроблення ПЗ (хоча існує їх значно більше): каскадна, інкрементна, еволюційна. Розроблена велика кількість моделей ЖЦ, що реалізують дані стратегії.

3. **Розроблення методів вибору моделей життєвого циклу**. До теперішнього моменту розроблений ряд методик і процедур вибору моделей ЖЦ, виходячи з умов і характеристик конкретного проекту.

4. **Створення методологій аналізу та проектування програмних засобів**. У даний час створена велика кількість методологій, спрямованих у першу чергу на початкові етапи процесу розроблення ПЗ, а саме: аналіз предметної області, розроблення вимог, проектування системи та ПЗ.

5. **Удосконалення інструментальних засобів підтримки сучасних методологій розроблення програмних засобів і систем**. З 80-х років XX ст. бурхливо розвиваються CASE-засоби (*англ. Computer-Aided Software Engineering, засоби автоматизації процесу розроблення ПЗ*), призначені для автоматизації процесів ЖЦ ПЗ і систем. До теперішнього часу багатьма компаніями розроблені лінійки CASE-засобів, що підтримують практично весь ЖЦ ПЗ і систем.

6. **Управління якістю розроблюваних програмних засобів**. Основу управління якістю становить оцінка якості ПЗ. У даний час у нашій країні ведуться активні роботи в області стандартизації оцінки якості ПЗ і їх сертифікації.

Ряд провідних світових компаній, які займаються інформаційними технологіями, щороку роблять прогнози щодо перспективних напрямків розвитку програмних засобів і систем.

1.2 Перспективи розвитку інформаційних технологій

Швидка зміна параметрів сучасного зовнішнього середовища призводить до збільшення обсягів і швидкості поширення інформації, тому для успішного ведення бізнесу необхідно скорочувати час прийняття рішень, що потребує збільшення швидкості передачі і обробки інформації за рахунок використання нових інформаційних технологій. Аналіз тенденцій і закономірностей розвитку інформаційних процесів у сфері бізнесу підтверджує висновок про високі темпи інформатизації, як процесів управління, так і процесів виробництва товарів і надання послуг.

У 2020 році International Data Corporation (IDC, міжнародна дослідницька і консалтингова компанія) [15] та Gartner (міжнародна дослідницька і консалтингова компанія) [16] у своїх звітах зазначили наступні перспективні напрямки розвитку інформаційних технологій [15, 16]:

1. **Прискорений перехід до хмарних технологій**. IT-директори (Information Technology, інформаційні технології) повинні просувати перехід до хмарних IT-моделей, щоб залишатися конкурентоспроможними. У звіті IDC вказується, що «до кінця 2021 року 80% підприємств стануть удвічі швидше переходити на хмарну інфраструктуру і додатки, ніж в умовах до пандемії».

2. **Периферійні обчислення стають пріоритетними**. Edge Computing – технологічна парадигма, що означає перенесення обчислювальних ресурсів фізично ближче до кінцевих пристроїв. Периферійні обчислення і хмара пов’язані симбіотичними відносинами. Вони вирішують різні набори проблем, але прекрасно доповнюють один одного. Це, в свою чергу, відкриває безліч можливостей для компаній (організацій). Технологічні експерти та IT-керівники очікують, що периферійні обчислення поліпшать існуючі процеси, а також стимулюватимуть нові операційні моделі та види діяльності.

3. **Перехід до автономних IT-операцій**. До 2023 року всі нові проекти в області IT та автоматизації використовуватимуть хмарну екосистему в якості базової структури, яка розширює можливості управління ресурсами і аналітики в реальному часі. Для цього компанії повинні інтегрувати аналітику на основі

штучного інтелекту і машинного навчання, впроваджувати автоматизацію і автономну інфраструктуру.

4. **Інтернет поведінки**. Аналітики Gartner вперше ввели термін «інтернет поведінки» у прогнозі на 2020 рік – це технології для моніторингу поведінкових явищ і управління даними, які на них впливають. Сюди входять розпізнавання осіб, відстеження місця розташування і Big Data. Gartner прогнозує, що до кінця 2025 року понад половини населення світу буде задіяно хоча б в одній програмі Інтернету поведінки.

5. **Технологія комірчастої мережі в кібербезпеці**. Комірчаста топологія мережі – це побудована на принципі осередків топологія, у якій робочі станції мережі поєднуються одна з одною та здатні приймати на себе роль комутаторів для інших учасників мережі. Подібна організація мережі є доволі складною в налаштуванні, але натомість реалізовує високу відмовостійкість. Як правило, вузли мережі з такою топологією з'єднуються за принципом «кожен вузол з кожним сусіднім вузлом». Таким чином, велика кількість вузлових зв'язків забезпечує широкі можливості з вибору маршруту всередині мережі, а отже, обрив одного з'єднання не призведе до порушення функціонування мережі в цілому. Gartner прогнозує, що до 2025 року комірчаста мережа кібербезпеки буде підтримувати більше половини запитів на управління цифровими правами.

6. **Гіперавтоматізація**. Аналітики відзначають величезний попит на автоматизацію повторюваних ручних процесів і завдань. При цьому компанії будуть переходити від автоматизації окремих завдань до автоматизації процесів з декількома завданнями, а також до функціональної автоматизації декількох процесів і навіть до автоматизації на рівні бізнес-екосистеми.

7. **Проектування систем штучного інтелекту**. Згідно з дослідженням Gartner, тільки 53% проектів штучного інтелекту проходять шлях від прототипів до виробництва. Тому ще одним трендом стане забезпечення надійної структури, яка створить основу для проектування, масштабування і переходу нових систем штучного інтелекту у виробництво.

1.3 Структурний та об’єктно-орієнтований принципи розроблення програмного забезпечення

Основним предметом технологій розроблення ПЗ було і залишається питання підвищення низької продуктивності програмування. Технології розроблення ПЗ знаходяться в центрі уваги провідних програмістів починаючи з кінця 60-х років, коли відбувалася так звана «перша революція в програмуванні». На даний момент у технології розроблення ПЗ інвестовані величезні ресурси, є значні досягнення, але проблема низької продуктивності праці все ще далека від остаточного вирішення. Підтвердженням цьому служить кількість та різноманітність парадигм програмування, які постійно виникають, але не витісняють одна одну з часом.

Парадигма програмування [17] – це набір основоположних принципів, які служать методичною основою для конкретних технологій та інструментальних засобів програмування.

У даний час широку популярність набули два базові принципи (парадигми) розроблення ПЗ: модульний і об’єктно-орієнтований.

Розроблення модульного ПЗ ґрунтується на використанні структурних методів проектування, метою яких є декомпозиція (розбиття, поділ) за деякими правилами проектованого ПЗ на структурні компоненти, кожен з яких виконує окреме завдання.

Об’єктно-орієнтоване розроблення базується на застосуванні об’єктних методів, до яких відносяться методології об’єктно-орієнтованого аналізу, проектування і програмування. Під об’єктом в програмуванні розуміється віртуальна сутність, яку можна описати за допомогою її стану (властивостей) та поведінки.

1.3.1 Непроцедурні парадигми

Парадигми програмування у деякій мірі створені завдяки моделям обчислення.

Модель обчислення – це формальний опис алгоритму, що використовується при обчисленні.

Однією з класичних моделей обчислення є машина Тьюринга, яка складається з пам'яті і процесора [17]. Для даної моделі характерно, що потенційно нескінченна пам'ять розподілена на окремі комірки, в яких можуть бути записані символи кінцевого алфавіту. Машина працює дискретно і завжди знаходиться в одному стані з певного обмеженого їх набору. За один крок машина виконує одну команду, яка визначається поточним станом і символом в поточній комірці. Виконання команди полягає в записі нового символу в поточну комірку, його переміщенні до іншої комірки і зміні стану.

Ця проста модель обчислення відображена в найбільш поширеній архітектурі комп'ютерів, яка називається архітектурою фон Неймана [17]. За такої архітектури комп'ютер має адресовану пам'ять, що складається з окремих комірок, а також процесор, який дискретно виконує команди для взаємодії з комірками пам'яті. Як правило, одна команда працює з однією коміркою, тобто процесор в кожен момент часу може або зчитувати інструкцію/дані з пам'яті, або записувати одиницю даних у пам'ять. Одночасно ці обидві дії виконуватися не можуть, оскільки інструкції і дані використовують один і той же потік.

У всіх поширених системах програмування є два фундаментальних поняття, які диктуються архітектурою машини [17] – це поняття змінної, яка має ім'я і може змінювати значення (абстракція комірок адресованої пам'яті) і поняття управління (абстракція дискретного послідовного процесора), тобто послідовності виконання команд програми, яка визначається програмістом. Програмування, в якому використовуються обидва цих ключових поняття, називається процедурним програмуванням. Програмування, в якому одне або обидва поняття не використовується, називається непроцедурним.

Використання тільки процедурного програмування на сьогодні не є обов'язковим, тому що існують (та іноді з'являються нові) різноманітні парадигми непроцедурного програмування, які засновані на інших моделях обчислення, ніж машина Тьюринга, а також орієнтовані на іншу архітектуру комп'ютера, ніж класична архітектура фон Неймана.

Наприклад, логічне програмування – це один з підходів, в якому в якості мови високого рівня використовується логіка предикатів першого порядку в формі фраз Хорна. Мови логічного програмування є мовами більш високого рівня, ніж алгоритмічні (наприклад, Pascal, C) і функціональні (Лісп, CLOS). Найбільшого поширення отримала реалізація логічного програмування, яка відома під назвою Prolog. У логічної програми немає явного управління, а також присутнє неявне присвоювання, яке зводиться до запам'ятовування уніфікованих підстановок.

Програмування, в якому використовується явне поняття управління, називається імперативним. На противагу йому логічне програмування є декларативним, тобто описує загальну мету, яку необхідно досягти, але не методи її досягнення.

Іншим варіантом непроцедурного підходу є функціональне програмування. Воно тісно пов'язане з λ-обчисленням Черча. Функціональна програма є композицією функціоналів (тобто функцій, аргументами і результатами яких можуть бути функціонали). У функціональній програмі немає ні явного управління, ні явних змінних, отже, немає і явного присвоювання.

Дуже часто практична реалізація не процедурного погрworld програмування є простішою, зрозумілішою, більш зручною у використанні, ніж процедурні системи, та не завжди вимагає від програміста володіння специфічним математичним апаратом. Оскільки непроцедурні програми менше орієнтовані на класичну архітектуру комп'ютера, їх реалізація виявляється менш ефективною на звичайних комп'ютерах. Проте в промислових масштабах існують та застосовуються комп'ютери з нетрадиційною архітектурою, для яких непроцедурне програмування виявляється більш природним, продуктивним і ефективним [17].

Перераховані парадигми непроцедурного програмування порівняно не часто застосовуються при вирішенні завдань розроблення вертикальних додатків та інформаційних систем, тому в даному підручнику розглядатиметься виключно імперативний підхід до програмування.

1.3.2 Структурне програмування

Історично однією з перших парадигм прийнято вважати структурне програмування [17]. Едсгером Дейкстрою була виявлена зворотна залежність між часткою операторів безумовного переходу *go to* і якістю ПЗ. Це призвело до бурхливого розвитку концепції структурного програмування. Дану концепцію можна розуміти як «програмування без go to».

Відмова від використання оператору безумовного переходу go to призвела до формування нового бачення структури програми: «будь-яка структура управління може бути функціонально еквівалентно вираженою суперпозицією послідовного виконання, розгалуження за умовою і циклу з передумовою» [17] (рис. 1.3).

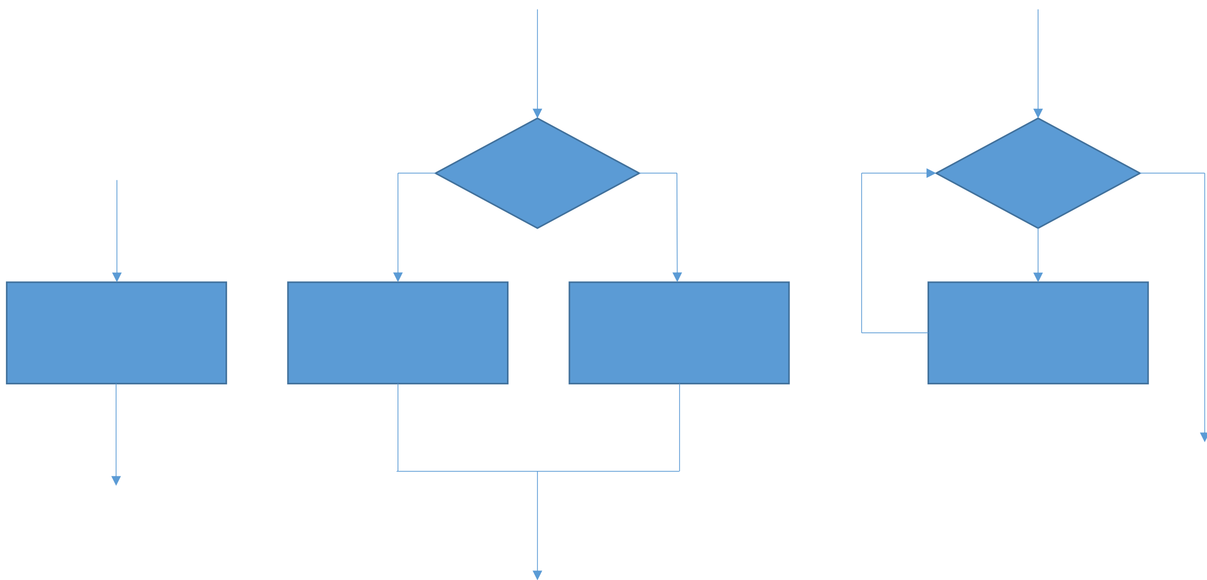

Рис. 1.3. Базові структури керування без «go to»

Кожна з перелічених вище базових структур має властивість «один вхід – один вихід». Дана властивість зберігається при будь-якій лінійній комбінації двох базових структур. Наявність цієї властивості корисна тим, що вона дозволяє встановити просту відповідність між динамікою виконання програми та її статичним текстом.

Таким чином, програмування без оператора *go to* дозволяє частково вирішити протиріччя між статикою і динамікою в ПЗ. При цьому синтаксичні конс-

трукції різного роду (наприклад, switch, repeat, for next, for each та інші) не змінюють суті справи, тому що принцип один вхід – один вихід зберігається [17]. Це означає, що відсутність або наявність структур управління в мові майже не впливає на програмування, бо необхідно та достатньо мати відповідні процедури з процедурними параметрами.

Відсутність або наявність обмежених операторів переходу (наприклад, break, exit, тощо) також не змінює суті справи з тієї ж причини. Обмежені переходи є корисними, тому що дозволяють писати більш короткі та ефективні програми такого ж ступеню структурованості.

Наявність ієрархічної структури вкладеності (наприклад, вкладені цикли *for*) також дозволяє частково вирішити інше протиріччя, а саме: послідовне розгортання структури вкладеності забезпечує декомпозицію загального завдання на окремі підзадачі і, таким чином, на кожному кроці обмежує комбінаторну складність. Це називається програмуванням методом покрокового уточнення або ітераційним методом.

Практично усі системи програмування підтримують поняття модуля – завершеного блоку кода, який має чітке функціональне призначення і може знаходитися в окремому програмному файлі. У різних системах програмування поняття «модуль» може визначатися по-іншому: функція, клас, підпрограма, бібліотека, набір бібліотек, тощо. Проте суть цього поняття залишається однією й тією ж: модульність є засобом подолання кількісних обмежень.

Як правило, модульна структура допускає зв’язок між окремими модулями або їх вкладення. Також допускається їх одночасне використання. Така декомпозиція призводить до того, що «в добре структурованій програмі текст кожного модуля займає одну сторінку» (Е. Дейкстра) [17]. Таким чином, модульне програмування є безпосередньою реалізацією принципу «розділяй і володарюй» у програмуванні.

Треба також зазначити, що в системах програмування крім безпосередньої структуризації тексту програми на модулі реалізується ще безліч додаткових функцій: інкрементальна компіляція, скорочення обсягу коду, інкапсуляція даних, обмеження областей дії, тощо.

Структурне програмування поклало основу для декількох підходів до розроблення ПЗ [17]. Розглянемо їх більш детально.

Програмування «зверху-вниз» (рис. 1.4) – це узагальнююча назва для модульного програмування без використання оператора *go to* методом покрокового уточнення. Англійська назва – Top-down approach – більш точно пояснює даний підхід: проектування і реалізація (кодування) програми є двома фазами одного процесу, які нерозривно пов'язані. При цьому проектування є первинним процесом, а реалізація – вторинним.

При програмуванні «зверху-вниз» процес програмування полягає в наступному: проводиться декомпозиція вихідної задачі на підзадачі до тих пір, поки підзадача не стане настільки простою, що її реалізація стає очевидною.

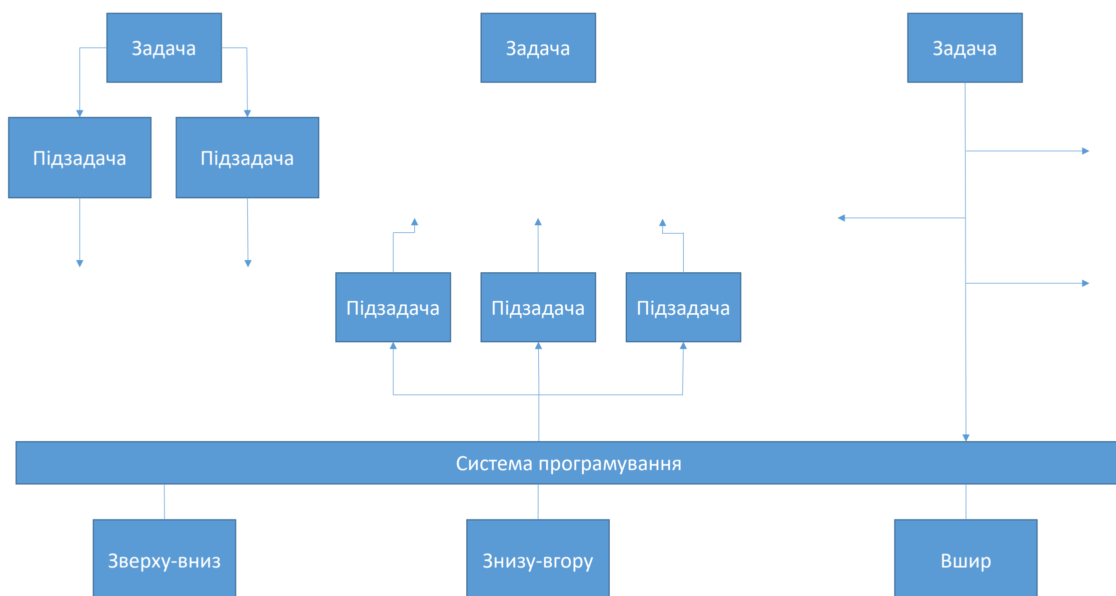

Рис. 1.4. Схеми різних підходів до процесу розроблення

Аналогом програмування «зверху-вниз» є програмування «знизу-вгору» (рис. 1.4). При такому підході рівень структурних елементів мови програмування (функцій, методів, класів, компонентів, тощо) підвищується до тих пір, поки він не стане настільки високим і близьким до вихідної задачі, що її реалізація стане очевидною.

Обидва методи мають очевидні переваги, але мають і недоліки [17]: при програмуванні «зверху-вниз» налагодження можливе лише після закінчення

всього проектування, при програмуванні «знизу-вгору» є великий ризик створення непотрібних модулів, які не будуть використані для формування готового ПЗ. Саме тому на практиці завжди застосовується комбінація цих підходів, адже вони не суперечать один одному (табл. 1.1).

Таблиця 1.1. Порівняння підходів

Основа для порівняння	«зверху-вниз»	«знизу-вгору»
Принцип	Розбиває велику загальну проблему на маленькі підзадачі	Вирішує фундаментальні проблеми низького рівня та об'єднує їх у більшу
Процес	Підмодулі аналізуються одноосібно	Визначається чи потрібна інкапсуляція і для яких даних
Зв'язок	Не потрібен	Потрібне спілкування між розробниками
Надлишковість	Може міститись надлишковість	Надлишковості можна позбутися
Мови програмування	Процедурні та структурні мови	Об'єктно-орієнтовні мови
Використовується в	Документація модуля, створення тестового набору, реалізація коду та налагодження.	Тестування

Існує й інший підхід до процесу розроблення в рамках структурної парадигми. Термін «програмування вшир» винайшов Святослав Сергійович Лавров [17]. Даний підхід полягає в тому, що починаючи з самого першого кроку, створюється працездатна версія майбутньої програми, яка підтримується на всіх наступних етапах розроблення. Звичайно, перша версія програми не може виконувати ніяких корисних функцій у межах предметної області (бізнес-логіки), але вона також не містить непотрібних функцій і готова для демонстрації у будь-який момент. Такий стиль вимагає проведення тестових випробу-

вань усіх готових модулів при завершенні або зміні будь-якого модуля і не відтерміновує налагодження ПЗ на момент закінчення розроблення. Це вимагає додаткових трудовитрат, але сприятливо впливає на замовника. Через чверть століття після Лаврова подібні ідеї запропонував Кент Бек в екстремальному програмуванні.

Важливо підкреслити, що структурність програми реалізується програмістом, а не зумовлюється обраною системою (парадигмою) програмування. У будь-якій системі програмування будь-яку програму можна написати структурно або не структурно, як з використанням сумнівних операторів (наприклад, go to), а також інструментальних засобів (наприклад, ітераційного уточнення), так і повністю уникаючи їх.

1.3.3 Об’єктно-орієнтоване програмування

Парадигма об’єктно-орієнтованого програмування (ООП) – це найбільш популярна [17] на даний момент парадигма, яка є розширенням згаданих вище підходів.

Програмування без використання оператора безумовного переходу *go to* задовільним чином вирішує проблему структурування управління. Однак, як відомо, програми визначаються зв’язками не тільки за управлінням, а й за даними. Необмежене присвоювання викликає стільки ж помилок, як і використання оператора *go to*, тому що створює ті ж проблеми: труднощі динамічного відстеження історії змінної і наявність занадто великої кількості змінних, які часто ототожнюються.

Для вирішення даної проблеми необхідно дотримуватися наступного принципу: для кожної змінної існує єдиний модуль, який має до неї доступ. Локалізація змінної в модулі не є рішенням, оскільки процедурне програмування не може обходитися без змінних, час життя яких виходить за межі часу активізації процедур їх обробки. Це означає, що потрібні змінні, які будуть доступними для процедури, та які існують не тільки під час виклику цієї процедури. Глобальні ж змінні допускають необмежену кількість присвоювань.

Саме тому центральною ідеєю парадигми ООП є інкапсуляція [17], тобто структурування програми на модулі особливого виду, які об'єднують дані і процедури їх обробки. При цьому внутрішні дані модуля не можуть бути оброблені інакше, крім як передбаченими для цього модуля процедурами. Крім основної ідеї інкапсуляції, в ООП прийнято вважати основними також поняття спадковості та поліморфізму.

Важливим елементом об'єктно-орієнтовної парадигми програмування є наступна ідея: клас, який теж є об'єктом, може мати методи (конструктори та деструктори), що дозволяють під час виконання програми динамічно створювати і знищувати екземпляри (об'єкти) даного класу. Об'єкти одного класу схожі між собою (наприклад, успадковують методи класу – функції, які описують поведінку об'єктів), але можуть мати відмінності (наприклад, різні значення властивостей).

Механізм динамічного створення екземплярів виявляється дуже корисним, а в деяких випадках незамінним, однак не завжди. Об'єктна орієнтованість є атрибутом стилю програмування, властивого конкретному програмісту, а не зумовлюється мовою або системою програмування. У будь-якій системі програмування, навіть в тій, де немає ніяких спеціальних засобів підтримки об'єктів і класів, можна використовувати принципи ООП, так само як і в сучасній об'єктно-орієнтованій системі ніхто не застрахований від недоцільного використання класичної процедурної парадигми.

З парадигмою ООП пов'язаний розвиток і практичне втілення важливої думки про нерозривний зв'язок проектування і кодування. В уявленні людини реальний світ складається з певних об'єктів. Кодуванню завжди передує проектування, найважливішою частиною якого є моделювання, тобто визначення в предметній області завдання об'єктів і зв'язків, які істотні для вирішення цього завдання. ООП дозволяє встановити пряму відповідність між програмними конструкціями і об'єктами реального світу через об'єкти моделі. Під час написання об'єктно-орієнтованої програми програміст витрачає основні зусилля на визначення складу класів, їх методів і властивостей. Ця інформація є в чистому

вигляді накладеною структурою, оскільки в виконуваному коді програми явно ніяк не відображається.

Об'єкти визначаються суб'єктивно, по волі програміста. Це визначення не є однозначним і може бути зроблено як більш-менш вдало, так і навпаки. Програмісти, які вміють вдало виділяти об'єкти та зв'язки, називаються системними аналітиками. У багатьох предметних областях, з метою зниження негативних наслідків невдалого аналізу, є готові модельні рішення, в яких акумульовано вдалий досвід попередніх розробок.

1.4 Життєвий цикл

У програмного забезпечення, як і у будь-якого живого організму, є свій життєвий цикл (ЖЦ). Під **життєвим циклом** програмного засобу або системи мається на увазі [10] сукупність процесів, робіт і завдань, що включає в себе розроблення, експлуатацію та супровід програмного засобу або системи і охоплює їхнє життя від формулювання концепції до виходу з обігу. ЖЦ ПЗ складається з ряду процесів. Кожен процес ЖЦ розділений на визначений набір робіт, а кожна окрема робота розділена на набір завдань. Процеси ЖЦ ПЗ діляться на три групи (рис. 1.5, [10]):

- основні;
- допоміжні;
- організаційні.

Основні процеси життєвого циклу – це процеси, якими управляють основні сторони, які приймають участь у ЖЦ ПЗ. Основними сторонами є: замовник, розробник, постачальник, оператор і персонал супроводу програмних продуктів (ПП). До основних можна віднести наступних п'ять процесів [10]:

- замовлення;
- розроблення;
- експлуатація;
- поставка;
- супровід.

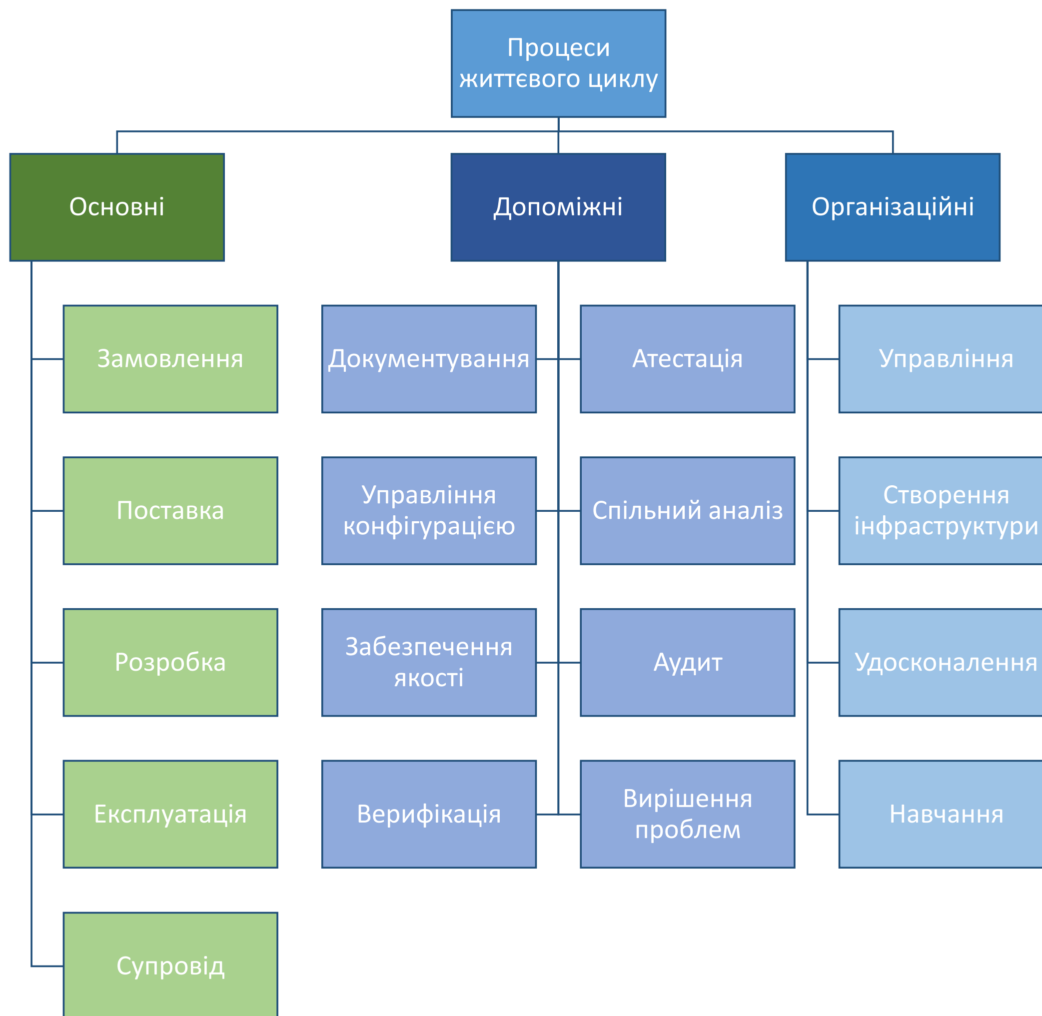

Рис. 1.5. Процеси життєвого циклу

Процес **замовлення** визначає роботи і завдання замовника. Він складається з визначення вимог замовника до системи або ПП (визначення елементів бізнес-логіки), підготовки і випуску заявки на підряд, вибору постачальника і управління процесами замовлення та приймання системи або ПП.

Процес **постачання** визначає роботи і завдання постачальника. Даний процес починається з рішення про підготовку пропозиції у відповідь на заявку на підряд від замовника, або з підписання договору з замовником на поставку системи чи ПП. Далі визначаються необхідні для управління і забезпечення проекту процедури та ресурси, включаючи розроблення проектних планів і їх виконання.

Процес **розроблення** складається з робіт і завдань, які виконуються розробником. Даний процес в загальному випадку може містити близько тринадцяти робіт [10]:

1) підготовка процесу розроблення;

2) аналіз вимог до системи;

3) проектування системної архітектури;

4) аналіз вимог до програмних засобів;

5) проектування програмної архітектури;

6) технічне проектування програмних засобів;

7) програмування і тестування програмних засобів;

8) складання програмних засобів;

9) кваліфікаційні випробування програмних засобів;

10) складання системи;

11) кваліфікаційні випробування системи;

12) уведення в дію програмних засобів;

13) забезпечення приймання програмних засобів.

При виконанні роботи 1 «Підготовка процесу розроблення» обирається модель ЖЦ ПЗ або системи, в яку структуруються процеси, роботи і завдання. Також обираються і адаптуються стандарти, методи, інструментальні засоби розроблення, мови програмування і формується план проведення робіт процесу розроблення.

При виконанні роботи 2 «Аналіз вимог до системи» аналізується область застосування системи. На підставі результатів аналізу предметної області визначаються вимоги до неї, а також виконується оцінка розроблених вимог.

При виконанні роботи 3 «Проектування системної архітектури» визначається загальна архітектура системи. Здійснюється розподіл вимог до системи між об’єктами технічних і програмних засобів архітектури і ручними операціями. Проводиться подальше уточнення вимог, здійснюється оцінка архітектури системи та вимог до об’єктів архітектури.

При виконанні роботи 4 «Аналіз вимог до програмних засобів» аналізується призначення ПЗ. Виходячи з результатів аналізу визначаються і уточнюються вимоги та проводиться їх оцінка.

При виконанні роботи 5 «Проектування програмної архітектури» розробляється ескізний проект ПЗ. Вимоги перетворюються в архітектуру ПЗ, розподіляються між його компонентами і уточнюються далі. Проводиться оцінка результатів ескізного проектування.

При виконанні роботи 6 «Технічне проектування програмних засобів» здійснюється детальне проектування ПЗ. Компоненти проектуються до рівня їх подання у вигляді набору програмних модулів, складність яких повинна забезпечити можливість їх безпосереднього кодування в наступній роботі (роботі 7). Проводиться розподіл технічних вимог до компонентів між програмними модулями, подальше уточнення вимог та оцінювання технічного проекту.

При виконанні роботи 7 «Програмування та тестування програмних засобів» проводиться кодування і тестування програмних модулів, а також оцінювання результатів.

При виконанні роботи 8 «Складання програмних засобів» здійснюється складання програмних модулів і компонентів в кінцеве ПЗ, тестування проміжних і кінцевих результатів складання, а також виконується оцінка результатів побудови та тестування.

При виконанні роботи 9 «Кваліфікаційні випробування програмних засобів» проводяться кваліфікаційні випробування ПЗ у змодельованому середовищі зі змодельованими вихідними даними. Оцінюються результати випробувань, а також, при необхідності, проводиться доопрацювання розроблюваного ПЗ.

При виконанні роботи 10 «Складання системи» здійснюється складання об'єктів програмної та технічної конфігурації, ручних операцій, підсистем в єдину систему. Проводяться випробування складеної системи та її оцінка.

При виконанні роботи 11 «Кваліфікаційні випробування системи» проводяться кваліфікаційні випробування зібраної системи в змодельованому середовищі зі змодельованими вихідними даними. За результатами випробувань виконується оцінка системи і при необхідності її доопрацювання.

У наведеній послідовності можна виділити два види робіт [10]: системні та програмні. Системні роботи починають і завершують процес розроблення. До них відносяться роботи з номерами 2, 3, 10, 11. Роботи процесу розроблення з 4-ї (аналіз вимог до програмних засобів) по 9-у (кваліфікаційні випробування програмних засобів) представляють собою програмні роботи. Вони виконуються над програмними засобами, виділеними з системи.

Таким чином, системні роботи є розширенням сукупності програмних робіт.

При виконанні роботи 12 «Уведення в дію програмних засобів» ПЗ вводиться в дію в середовищі експлуатації.

При виконанні роботи 13 «Забезпечення приймання програмних засобів» забезпечується проведення замовником приймальних випробувань, після чого готовий ПП поставляється замовнику.

Процес **експлуатації** визначає роботи і завдання оператора. Даний процес включає експлуатацію ПП і підтримку користувачів у процесі експлуатації.

Процес **супроводу** визначає роботи і завдання персоналу супроводу і реалізується при розширенні функціональних можливостей ПП. Призначенням процесу є зміна існуючого ПП при збереженні його цілісності. Даний процес також охоплює питання зняття ПП з експлуатації.

Допоміжні процеси життєвого циклу – це процеси, які є цілеспрямованими складовими частинами інших процесів і призначені для забезпечення успішної реалізації і якості виконання програмного проекту. До допоміжних процесів можна віднести вісім наступних процесів [10]:

- документування;
- забезпечення якості;
- управління конфігурацією;
- верифікація;
- спільний аналіз;
- атестація;
- аудит;

- вирішення проблем.

Допоміжні процеси викликаються іншими процесами ЖЦ ПЗ.

Процес **документування** призначений для формалізованого опису створеної в процесі ЖЦ інформації. Він включає планування, проектування, розроблення, випуск, редагування, поширення та супровід документів по ПП.

Процес **управління конфігурацією** призначений для визначення стану (базової лінії) програмних об’єктів в системі, управління їх змінами і випуском.

Процес **забезпечення якості** призначений для забезпечення гарантій того, що ПП і процеси в ЖЦ проекту відповідають розробленим вимогам і планам.

Процес **верифікації** призначений для визначення відповідності функціонування ПП реалізованим в попередніх роботах вимогам і умовам. У процесі розроблення верифікація пов’язана з експертизою результатів конкретної роботи з метою визначення їх відповідності встановленим на вході даної роботи вимогам.

Процес **атестації** призначений для визначення повноти відповідності встановлених вимог створеної системи або ПП їх прямому функціональному призначенню. У процесі розроблення ПЗ атестація пов’язана з експертизою проміжного або кінцевого продукту з метою визначення його відповідності потребам користувача (тобто вихідним вимогам до проекту).

Процес **спільного аналізу** призначений для оцінки стану і результатів робіт за проектом у цілому. Даний процес може виконуватися двома сторонами, які беруть участь у договорі, коли одна сторона перевіряє (аналізує) іншу.

Процес **аудиту** призначений для визначення відповідності вимогам, планам та умовам договору. Аналогічно до попереднього випадку, даний процес може виконуватися двома сторонами, які беруть участь у договорі.

Процес **рішення проблем** призначений для аналізу і вирішення проблем, включаючи знайдені невідповідності, які виявлені в ході розроблення, експлуатації, супроводу або інших процесів.

Організаційні процеси ЖЦ – це процеси, призначені для створення (та їх подальшого вдосконалення) організаційних структур, що охоплюють процеси

ЖЦ і відповідний персонал. До організаційних можна віднести чотири наступних процеси [10]:

- управління;
- удосконалення;
- створення інфраструктури;
- навчання.

Процес **управління** складається з загальних робіт і завдань, які можуть бути використані стороною, яка управляє відповідним процесом. У даному процесі розробляються плани виконання процесів ЖЦ ПЗ, здійснюється управління і поточний нагляд за ходом процесів ЖЦ, забезпечується управління оцінкою планів, ПП, робіт і завдань.

Процес **створення** інфраструктури призначений для створення і супроводу інфраструктури, необхідної для будь-якого іншого процесу. Інфраструктура містить технічні та програмні засоби, інструментальні засоби, стандарти, методики і умови для розроблення, експлуатації або супроводу.

Процес **удосконалення** призначений для створення, оцінки, поліпшення, вимірювання та контролю будь-якого процесу ЖЦ ПЗ.

Процес **навчання** є процесом забезпечення навчання персоналу розробці, супроводу або експлуатації ПП.

Процес ЖЦ будь-якої системи або ПП може бути описаний за допомогою моделі ЖЦ. **Модель життєвого циклу** [8, 10] – це сукупність процесів, робіт і завдань ЖЦ, що відображає їх взаємозв’язки та послідовність виконання. Моделі можуть використовуватися як глобально (для всього проекту), так і локально (для окремого етапу проекту, який виконується в межах іншої моделі ЖЦ). Кожна модель ЖЦ залежить від певних умов конкретного ПП і включає в себе поділ на стадії (етапи) з визначенням відповідних результатів та ключові події (точки завершення робіт та прийняття рішень).

Під стадіями будемо розуміти обмежену в часі частину процесу створення ПЗ, яка закінчується відповідним результатом (документація, моделі, тощо), який визначається висунутими вимогами. Різні компанії-розробники можуть використовувати різноманітні стадії, але кожна стадія реалізується з належним

аналізом інформації (наприклад, з попередніх етапів) та ретельним прийняттям рішень щодо подальшого курсу процесу розроблення ПП для забезпечення його задовільної якості.

Враховуючи вищесказане, можна зробити висновок, що існують тісні взаємозв'язки між моделлю ЖЦ, використовуваними стратегіями розроблення ПЗ, технологіями розроблення і рівнем якості розробленого ПП.

1.4.1 Стратегії розроблення програмного забезпечення

На початковому етапі розвитку обчислювальної техніки ПЗ розроблялися за принципом «кодування-усунення помилок». Модель такого процесу розроблення ПЗ ілюструє рис. 1.6.

Вибір тієї чи іншої стратегії розроблення визначається характеристиками:

- проекту;
- команди розробників;
- вимог до продукту;
- команди користувачів.

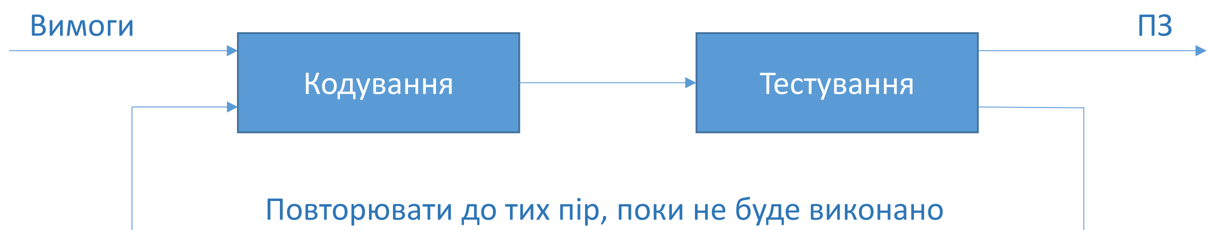

Рис. 1.6. Застаріла модель розроблення ПЗ

Кожна зі стратегій має як переваги, так і недоліки, які визначаються правильністю вибору даної стратегії для реалізації конкретного проекту. Слід підкреслити, що одні й ті ж властивості стратегії можуть проявляти себе як переваги при правильному виборі стратегії, так і як її недоліки, якщо стратегія обрана неправильно.

Каскадна стратегія [10] являє собою одноразовий прохід етапів розроблення та базується на повному визначенні всіх вимог щодо розроблюваного ПЗ

або системи на початку процесу розроблення. Кожен наступний етап розроблення починається після завершення попереднього етапу і повернутися до вже виконаних етапів найчастіше неможливо. Проміжні версії розроблюваного ПП не поставляються замовнику чи кінцевим користувачам.

Основними представниками моделей, які реалізують каскадну стратегію, є waterfall і V-образна моделі. Дані моделі будуть розглянуті пізніше.

Основними перевагами каскадної стратегії, які проявляються при розробці відповідного їй проекту, є [10]:

1) стабільність визначених вимог протягом ЖЦ ПЗ;

2) простота планування та управління проектом;

3) простота застосування стратегії, яка проявляється у необхідності тільки одного проходу етапів розроблення;

4) доступність для розуміння замовниками.

До основних недоліків каскадної стратегії, що проявляється при її використанні в проекті, який їй не відповідає, слід віднести [10]:

1) лінійність структури процесу розроблення; в умовах сьогодення найчастіше розробляються дуже великі та складні проекти, для яких одноразове виконання кожної стадії процесу розроблення є недоцільним; повернення до попередніх кроків для вирішення поточних проблем призводить до порушення графіка робіт і збільшення фінансових витрат;

2) складність повного формулювання вимог на початку процесу розроблення і неможливість їх динамічної зміни протягом ЖЦ;

3) недостатня участь користувача в процесі розроблення ПЗ – тільки на самому початку (при визначенні вимог) і в кінці (під час приймальних випробувань); це призводить до неможливості попередньої оцінки користувачем якості ПП;

4) непридатність проміжних продуктів для використання.

Області застосування каскадної стратегії визначаються її перевагами і обмежені її недоліками. Використання даної стратегії найбільш ефективно в наступних випадках [10]:

1) при виконанні великих проектів в якості складової частини моделей ЖЦ, що реалізують інші стратегії розроблення;

2) при розробці проектів невисокої складності, наприклад:

- створення ПЗ або системи такого ж типу, які вже розроблялися в минулому даним виконавцем;
- створення нової версії вже існуючого ПЗ або системи;
- перенесення вже існуючого ПП на нову платформу.

3) при розробці проектів з чіткими, незмінними протягом ЖЦ вимогами і зрозумілою реалізацією.

Інкрементна стратегія являє собою багаторазовий прохід етапів розроблення із запланованим поліпшенням результату.

Дана стратегія вимагає визначення більшості вимог щодо розроблюваного ПЗ або системи на початку процесу розроблення, а уточнення початкових вимог реалізується поступово (у відповідності до плану) у послідовних циклах розроблення.

Результат кожного циклу називається **інкрементом**.

Перший інкремент реалізує лише базові функції ПЗ. У наступних інкрементах функціональні можливості ПЗ поступово розширюються, поки не буде реалізований весь набір правил бізнес-логіки. Відмінності між інкрементами сусідніх циклів у ході розроблення поступово зменшуються, а результат кожного циклу розроблення може поставлятися на ринок як чергова версія ПЗ або системи.

Особливістю інкрементної стратегії є велика кількість циклів розроблення з незначною тривалістю кожного окремого циклу і невеликих відмінностях між інкрементами сусідніх циклів. Наприклад, дана стратегія розроблення ПЗ і систем використовувалася в компанії Microsoft [10], де на кожну версію ПЗ розроблялося близько тисячі інкрементів, а час розроблення інкременту становив одну добу (наприклад, вдень інкремент розробляється, а вночі тестується). У ряді компаній використовується тижневий період розроблення інкременту (найчастіше п’ять днів триває процес розроблення і ще два дні – тестування).

Інкрементна стратегія зазвичай заснована на об'єднанні елементів каскадної моделі і прототипування, яке дозволяє істотно скоротити тривалість розроблення кожного окремого інкременту і всього проекту в цілому.

Під **прототипом** [10] розуміється «робоча модель розроблюваного ПЗ або системи, яка легко піддається модифікації і розширенню, а також дозволяє користувачеві отримати уявлення про її ключові властивості до повної реалізації».

Основними перевагами інкрементної стратегії при розробці відповідного їй проекту є [10]:

1) розроблювані специфікації вимог не є громіздкими; вимоги залишаються стабільними під час створення кожного інкременту, а також є можливість обліку змінених вимог;

2) коротка тривалість створення інкременту, що призводить до скорочення термінів початкової поставки; дозволяє знизити витрати на первинну і наступні поставки ПП;

3) можливість отримання функціонального ПП після реалізації кожного інкременту;

4) включення в процес користувачів дозволяє оцінити функціональні можливості ПП на більш ранніх етапах розроблення і в кінцевому підсумку призводить до підвищення його якості, зниження витрат і часу на його розроблення;

5) зниження ризиків у порівнянні з каскадною стратегією.

До основних недоліків інкрементної стратегії, що проявляються в результаті її невідповідного застосування, слід віднести [10]:

1) складність планування і розподілу робіт;

2) можливість поточної зміни вимог, які вже реалізовані в попередніх інкрементах;

3) необхідність якомога більш повного функціонального визначення системи або ПЗ на початку ЖЦ для забезпечення адекватного планування інкрементів і управління проектом;

4) прояв людського фактора, пов'язаного з тенденцією до відтягування вирішення важких проблем на пізні інкременти може порушити графік робіт або знизити якість ПП.

Області застосування інкрементної стратегії визначаються її перевагами і обмежені її недоліками. Використання даної стратегії є найбільш ефективним у наступних випадках [10]:

1) при необхідності швидко поставити на ринок продукт, що має базові функціональні можливості;

2) при виконанні складних проектів (для яких важко реалізувати весь функціонал за один цикл розроблення) із заздалегідь сформульованими вимогами;

3) при виконанні проектів, в яких більшість вимог можна сформулювати заздалегідь, але частина з них можуть бути уточнені через певний період часу;

4) при виконанні проектів з низьким або середнім ступенем ризиків;

5) при виконанні проектів з застосуванням нових технологій.

Еволюційна стратегія являє собою багаторазовий прохід етапів розроблення. Дана стратегія базується на частковому визначенні вимог щодо розроблюваного ПЗ або системи на початку процесу розроблення. Вимоги поступово уточнюються в послідовних циклах розроблення. Результат кожного циклу розроблення зазвичай являє собою чергову версію ПЗ або системи, яка може поставлятися.

Слід зазначити, що в загальному випадку для еволюційної стратегії характерною є істотно менша кількість циклів розроблення при більшій тривалості циклу в порівнянні з інкрементною стратегією [10]. При цьому результат кожного циклу розроблення набагато сильніше відрізняється від результату попереднього циклу.

Як і при використанні інкрементної стратегії, при реалізації еволюційної стратегії часто використовується прототипування.

У даному випадку основною метою прототипування є забезпечення повного розуміння вимог. Воно дозволяє ітеративно уточнювати вимоги до ПП при досягненні гранично високої продуктивності розроблення проекту і одночасному зниженні витрат. Використання прототипування є найбільш ефективним у тих випадках, коли в проекті застосовуються нові концепції або нові технології. Це пов’язано з тим, що в цих випадках досить складно повністю і коректно роз-

робити детальні технічні вимоги до системи або ПЗ на ранніх стадіях циклу розроблення.

Варто зазначити, що для ітеративного уточнення вимог при застосуванні прототипування в циклі розроблення обов'язково повинен брати участь замовник або його представник.

Основними перевагами еволюційної стратегії при розробленні відповідного їй проекту є [10]:

1) можливість управління ризиками;

2) придатність проміжного продукту для використання;

3) можливість уточнення і внесення нових вимог у процесі розроблення;

4) реалізація переваг каскадної та інкрементної стратегій;

5) забезпечення повноцінної участі кінцевого користувача в проекті, починаючи з ранніх етапів, що мінімізує можливість розбіжностей між замовниками і розробниками і забезпечує створення продукту високої якості.

До недоліків еволюційної стратегії, що проявляється при її невідповідному виборі, слід віднести [10]:

1) невідомість точної кількості необхідних ітерацій і складність визначення критеріїв для продовження процесу розроблення на наступній ітерації; це може викликати затримку реалізації кінцевої версії системи або ПП;

2) необхідність використання потужних інструментальних засобів і методів прототипування;

3) необхідність активної участі користувачів в проекті, що реально не завжди можливо;

4) складність планування і управління проектом;

5) можливість відкладення вирішення важких проблем на наступні цикли, що може привести до невідповідності отриманих продуктів вимогам замовників.

Використання даної стратегії найбільш ефективно в наступних випадках [10]:

1) при розробці проектів, що використовують нові технології;

2) при розробці складних проектів, у тому числі:

- великих довгострокових проектів;
- проектів зі створення нових ПП, які не мають аналогів (взагалі або в межах компанії-розробника);
- проектів з середнім і високим ступенем ризиків;
- проектів, для яких потрібна перевірка концепції, демонстрація технічної здійсненності або проміжних продуктів;

3) при розробці проектів, для яких вимоги занадто складні, невідомі заздалегідь, непостійні або вимагають уточнення.

1.4.2 Моделі життєвого циклу

Waterfall – класична каскадна модель (рис. 1.7). У межах даної моделі кожен наступний етап залежить від попереднього, тобто початок наступного етапу можливий лише у випадку завершення поточного. Таким чином утворюється каскадний (поступальний) рух вперед, який і лежить в основі даної моделі.

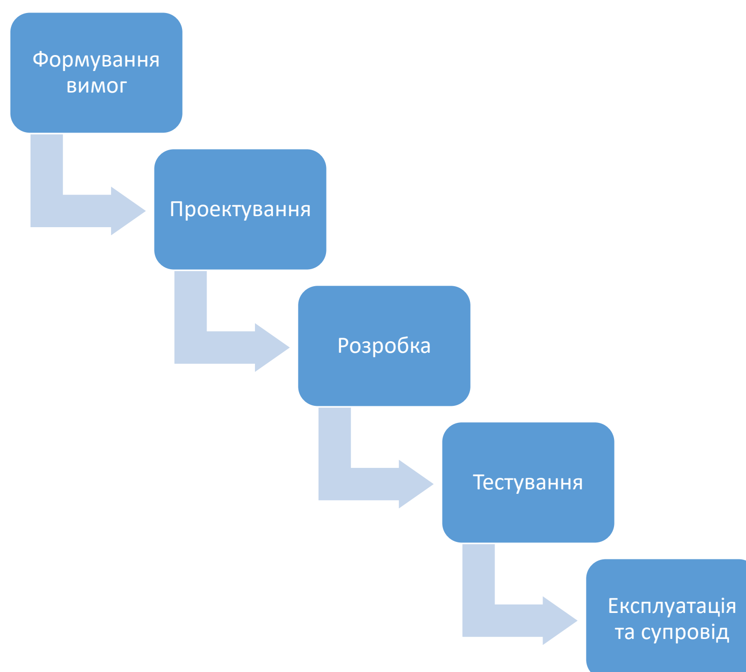

Рис 1.7. Класичне уявлення каскадної моделі

Ключовим фактором моделі *Waterfall* є сувора послідовність реалізації етапів процесу розроблення ПЗ. Спочатку формуються вимоги щодо розроблю-

ваного ПП, потім починається етап проектування, на якому створюється докладна документація з описом плану розроблення на основі раніше розроблених вимог до даного ПП.

Після остаточної підготовки всієї необхідної документації та завершення етапу проектування ПП переходить в стадію розроблення. На даному етапі відбувається безпосередня реалізація раніше спроектованого функціоналу або його частин, які на цьому ж етапі інтегруються в загальну систему.

Підготовка до тестування включає в себе створення тестової документації. Оскільки вимоги, які є основою цієї документації, вже розроблені, процес підготовки може починатися ще до завершення стадії розроблення, але ПП переходить до етапу безпосереднього тестування лише після закінчення етапу розроблення.

На етапі тестування перевіряється весь розроблений функціонал ПП. Основною метою є виявлення дефектів (критичних багів) та зведення до мінімуму можливості виникнення помилок після введення ПП в експлуатацію. ПП може перейти до етапів експлуатації та супроводу тільки у тому випадку, якщо команда тестування приймає рішення про недоцільність подальшого тестування у зв'язку з повною відсутністю (дуже малою кількістю) дефектів.

В уяві більшості замовників каскадна модель *Waterfall* виглядає лінійною (рис. 1.8) і простою. Виконання етапів відбувається послідовно крок за кроком до самого завершення розроблення, доки не буде досягнута кінцева мета. Саме тому реалізація даної моделі в межах технічного завдання є повністю зрозумілою: повне визначення всіх необхідних вимог, етап проектування, етап програмування, тестування, експлуатація та супровід.

У межах каскадної стратегії паралельність етапів процесу розроблення хоч і обмежена, але може бути реалізованою (рис. 1.9) для деяких незалежних між собою робіт. За такої умови інтеграція елементів, які виконуються паралельно, все одно відбувається, але не в межах одного і того ж етапу, тобто результат виконання одного з паралельних етапів буде імплементовано пізніше за інший. При цьому команди виконавців різних етапів між собою не спілкуються – кожна команда відповідає виключно за свій етап.

V-подібна модель ЖЦ є модернізацією моделі *Waterfall*. Її мета [8] полягає у визначенні та встановленні відповідного етапу тестування для кожного етапу проектування. Тестування (у першу чергу створення тестової документації) у *V-подібній* моделі починається ще на етапі формування вимог. Для кожного етапу тестування створюється окремий план, який визначає необхідні критерії переходу до наступної стадії проектування. *V-подібна* модель зображає усі стадії проектування в низхідній послідовності. У центрі (в основі) знаходиться стадія розроблення, а усі етапи тестування йдуть у висхідній послідовності.

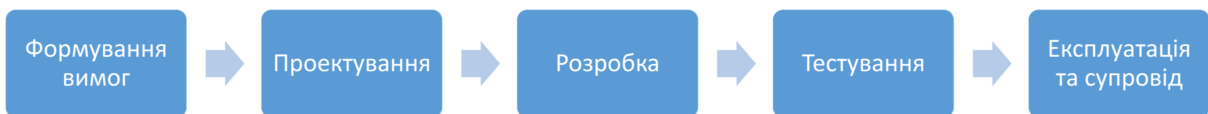

Рис. 1.8. Як бачить каскадну модель замовник

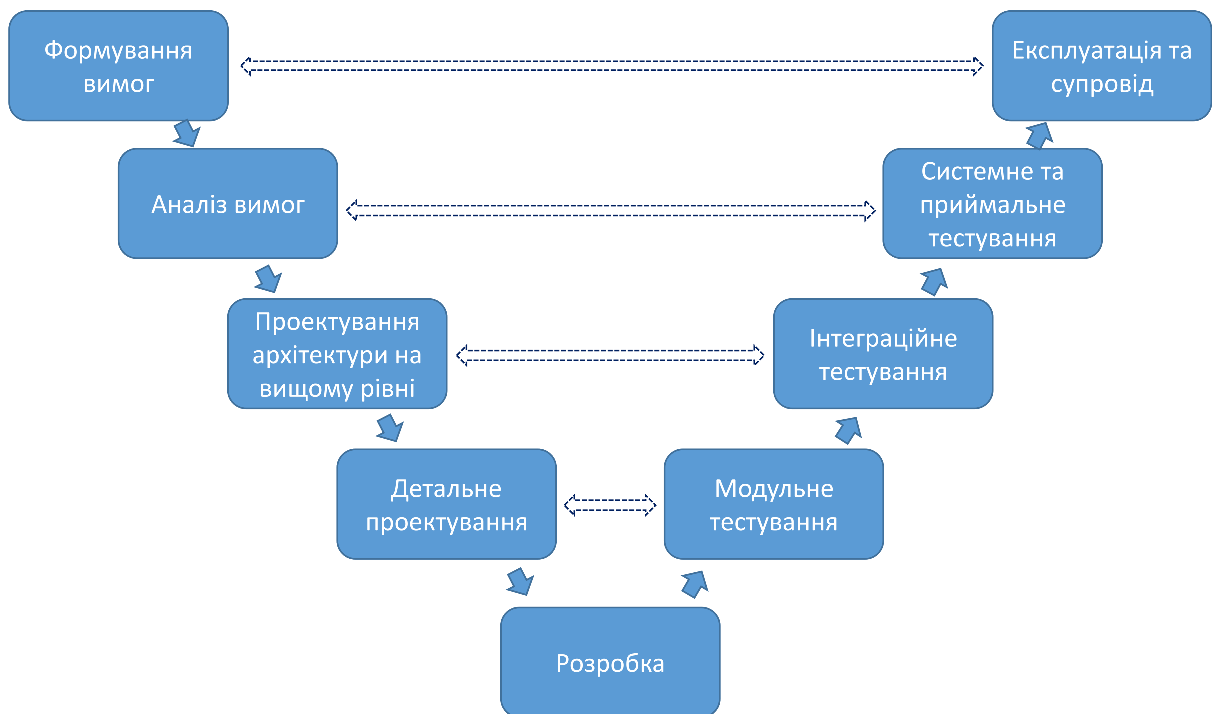

Рис. 1.9. V-подібна модель

В якості першого етапу тестування проводиться модульне тестування – тестування кожного з розроблених модулів (структур, методів, класів, пакетів, які відповідають за реалізацію лише однієї функції бізнес-логіки) окремо. Після

завершення тестування всіх модулів виконується інтеграційне тестування – перевірка правильності взаємодії модулів між собою та їх інтеграції один з одним. Вимоги до інтеграції та, відповідно, тестова документація для даного етапу розробляються на етапі проектування архітектури. Останніми етапами тестування є системне та приймальне. Вони дозволяють переконатися в правильному функціонуванні ПП в цілому та його відповідності поставленим вимогам.

Обидві каскадні моделі зрозуміло і легко викладаються в документах [8], наприклад, у договорі, технічному завданні чи дорожній карті, особливо якщо контрольні точки чітко визначені. За такої умови у будь-який момент часу можна легко визначити була пройдена відповідна контрольна точка чи ні, а також чи витримано терміни виконання. Саме тому великі довготривалі проекти, які розраховані на роки, інколи використовують каскадну стратегію розроблення, якщо впевнені, що проект не буде стрімко змінюватись.

Однак така простота каскадної моделі є оманливою. Клієнт, у переважній більшості випадків, не є спеціалістом в області розроблення ПЗ, тому його бачення цього процесу є дуже обмеженим. Через те, що каскадні моделі не передбачають залучення замовника в деталі процесів розроблення на кожному окремому їх етапі, наглядний результат роботи можна оцінити лише під час проходження контрольних точок і в самому кінці виконання проекту.

Ще одна причина складності використання каскадних моделей полягає у складності управління ними в реальних умовах. При бажанні замовника внести певні зміни в процесі розроблення або при появі у проекті серйозних не передбачених заздалегідь ризиків каскадну модель доводиться перебудовувати, графіки перепланувати. Якщо уважно проаналізувати ці наслідки, то можна зрозуміти, що такий випадок є рівнозначним початку нового проекту з нуля.

Зважаючи на все вищесказане, можна виділити основні недоліки моделей, які реалізують каскадну стратегію розроблення, а саме [8]:

- результат буде отримано тільки після проходження всіх етапів ЖЦ;
- дуже складно виявити помилки;
- повернутися навіть до попереднього етапу процесу розроблення дуже важко й інколи неможливо;

- якщо стався збій на одному з етапів, його наслідки стане видно тільки в кінці процесу розроблення;
- велика вартість внесення змін (особливо при використанні V-подібної моделі ЖЦ);
- велика тривалість процесу розроблення і, як результат, треба довго очікувати на працездатну версію ПП.

Серед переваг наведених моделей можна зазначити простоту планування термінів виконання та оцінки вартості розроблення, а також мінімум помилок при проектуванні архітектури ПП.

Інкрементна модель – це класична модель інкрементної стратегії розроблення ПЗ. Найчастіше вона являє собою цикл лінійних послідовностей каскадної моделі (рис. 1.10).

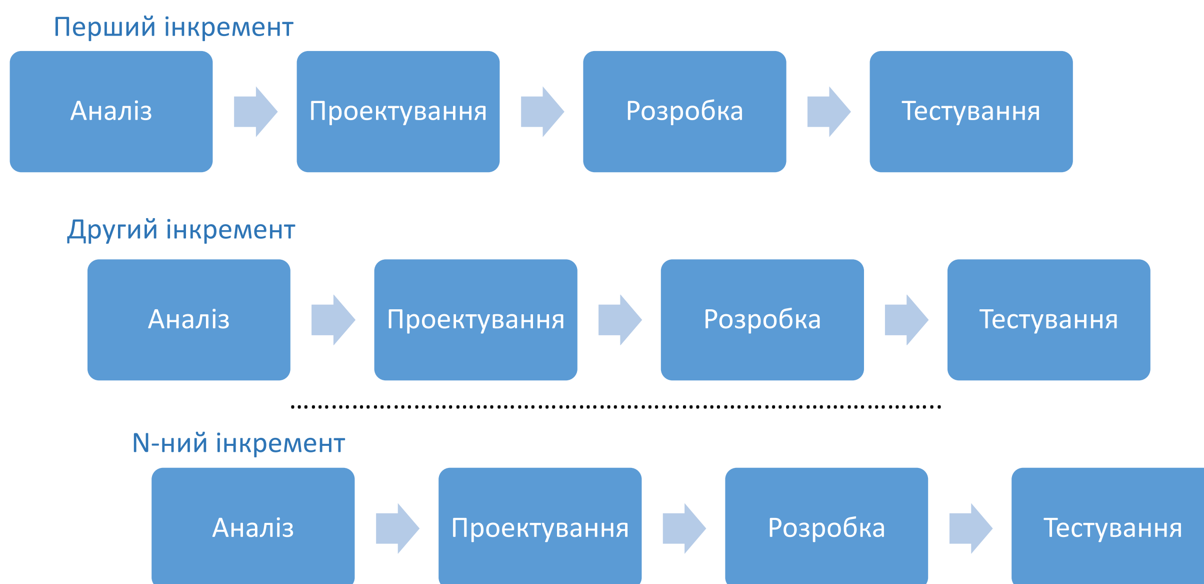

Рис. 1.10. Класичне уявлення інкрементної моделі

Кожна окрема послідовність називається інкрементом. Особливість інкрементів полягає в тому, що результатом кожного з них є функціональний ПП, який може бути поставлений замовнику. Перший інкремент завершується створенням ПП з базовими функціональними можливостями, наприклад, графічним інтерфейсом користувача, сервісом аутентифікації, тощо. Оскільки функціона-

льні вимоги бізнес-логіки на даному етапі все ще залишаються нереалізованими, план наступного інкремента передбачає модифікацію попереднього шляхом імплементації ряду (або однієї) додаткових функцій, після чого процес повторюється.

Отже, після завершення кожного окремого інкремента забезпечується випуск готового ПП з певним функціоналом.

Дуже часто інкрементну модель поєднують з ітераційним підходом. Така модель називається **ітераційно-інкрементною** (IID, Iterative and incremental development, *англ.*). Основна ідея цієї моделі полягає (рис. 1.11) в розробці ПП за допомогою повторюваних циклів (ітерацій) і невеликими частинами за раз (інкрементів). Такий підхід має значну перевагу [8]: можна користуватися знаннями, досвідом та результатами, які були отримані на більш ранніх етапах розроблення.

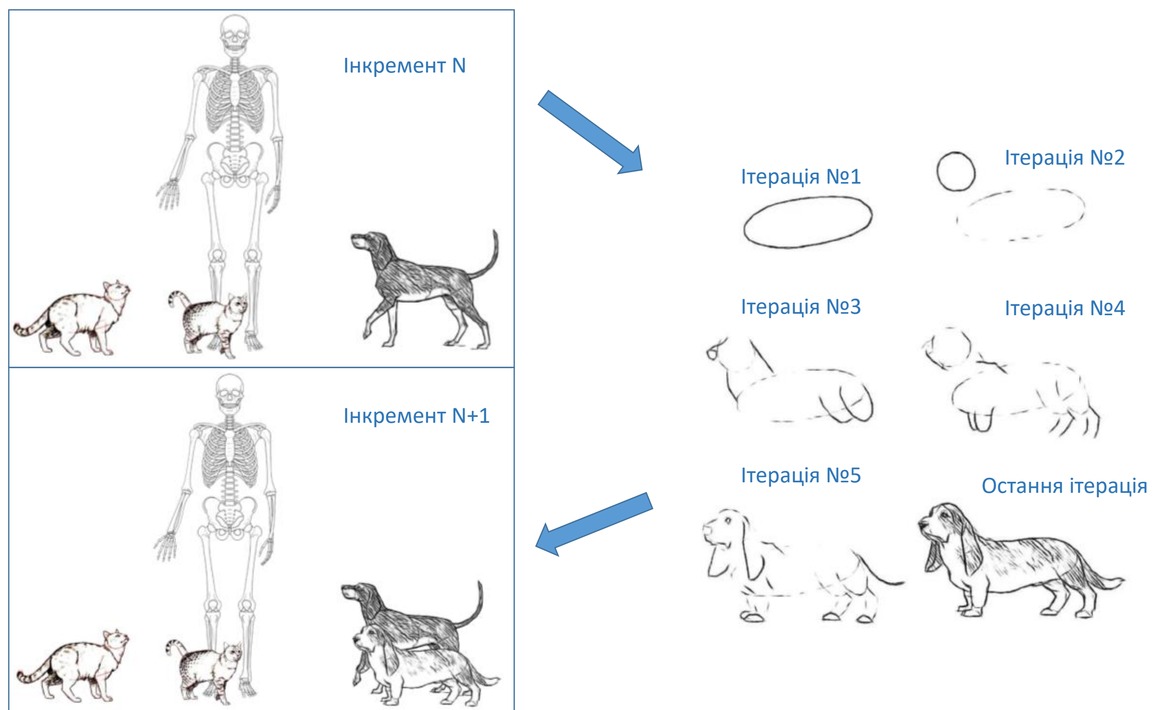

Рис. 1.11. Графічне уявлення ітераційно-інкрементної моделі

Ітераційний підхід передбачає розбиття ПП (або підсистеми чи модулю) на окремі частини (ітерації) і проходження етапів ЖЦ в межах кожної з них.

Кожна ітерація є закінченою сама по собі, а їх сукупність формує кінцевий результат (інкремент).

Оскільки з кожною ітерацією відбувається постійне уточнення вимог та розроблення наближається до кінцевого результату, у будь-який момент поточна ітерація може стати останньою для даного інкременту, якщо її результат задовольнятиме вимоги поточного етапу. Це призводить до зменшення невизначеності та дозволяє перевіряти правильність прийнятих рішень, адже в будь-який момент можна зупинитися, переосмислити підхід й почати ітерацію знову.

Поєднання інкрементної стратегії розроблення з ітераційним підходом дозволяє значно знизити глобальні ризики провалу проекту. Завдяки постійному уточненню вимог як замовник, так і кожний окремий учасник команди розроблення більш повно розуміють кінцеву мету кожного етапу, що дає змогу більш ретельно планувати бюджет та уникнути його розтрати.

Узагальнивши все вищесказане, можна сформулювати основні переваги ітераційно-інкрементної моделі розроблення [8]:

- зниження ризиків за рахунок раннього виявлення конфліктів між вимогами, використовуваними моделями і реалізацією проекту;
- динамічне формування вимог;
- гнучке управління вимогами;
- організація ефективного зворотного зв'язку команди розроблення з замовником (створений ПП реально відповідає потребам споживача);
- швидкий випуск мінімально життєздатного продукту (MVP, Minimum Viable Product, *англ.*).

Основні недоліки такої моделі полягають в наступному [8]:

- проблеми з архітектурою і накладні витрати (при роботі з хаотичними вимогами і без чіткого глобального плану може постраждати архітектура ПП, для доопрацювання якої можуть знадобитися додаткові ресурси);
- немає фіксованого бюджету і термінів виконання;

- необхідна сильна залученість замовника (кінцевих користувачів) у процес розроблення, що не завжди можливо.

Спіральна модель є однією з моделей еволюційної стратегії ЖЦ ПП. Усі етапи ЖЦ при використанні спіральної моделі зображаються у вигляді витків (рис. 1.12), на кожному з яких відбуваються наступні процеси: аналіз вимог, формалізація вимог, проектування системи, детальне проектування, тощо. При цьому, набір процесів на кожному новому витку може змінюватися, але результати проходження кожного окремого витка обов'язково ведуть до досягнення головної мети.

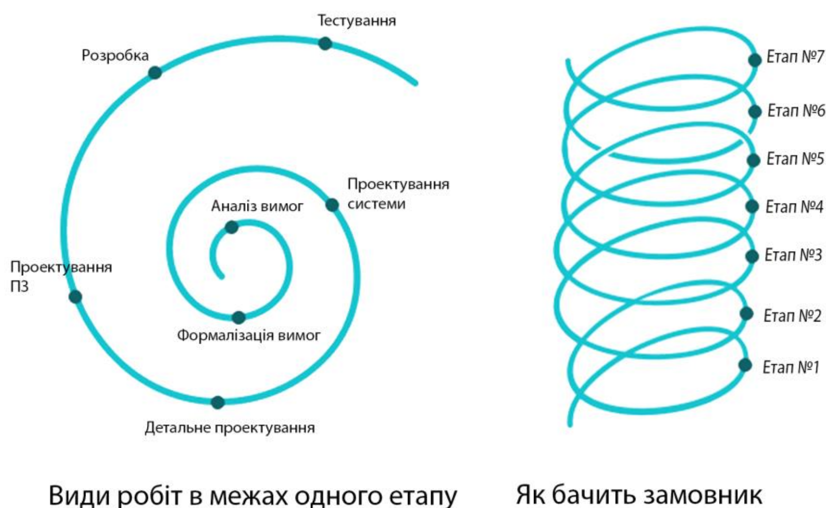

Рис. 1.12. Уявлення спіральної моделі

Спіральна модель ЖЦ характерна для розробок з використанням інноваційних або нетипових для компанії-розробника технологій. Це означає, що на початку роботи над проектом замовник та розробник можуть не мати чіткого уявлення кінцевого ПП. Тобто у даному випадку вимоги не можуть бути чітко визначені або у замовника немає стовідсоткової впевненості в успішній реалізації проекту (дуже великі ризики). У зв'язку з цим у рамках еволюційної стратегії приймається рішення проведення розроблення по частинах з можливістю доповнення вимог або відмови від подальшого розвитку. Подальший розвиток розроблюваного ПП, у свою чергу, може бути завершено не тільки після стадії

впровадження в експлуатацію готового продукту, але навіть після стадії аналізу ризиків на початкових етапах.

Серед переваг спіральної моделі, яка реалізує еволюційну стратегію розроблення, можна відзначити наступні:

- швидкий випуск мінімально життєздатного продукту (MVP, Minimum Viable Product, *англ.*);
- гнучке управління вимогами;
- більша гнучкість в управлінні проектом;
- у результаті розроблення отримуються більш надійні та стійкі ПП за рахунок того, що на кожній ітерації легко виявляються і виправляються слабкі місця і помилки;
- дозволяє удосконалювати процес розроблення – аналіз, проведений в кожній ітерації, дозволяє проводити оцінку того, що необхідно змінити в організації розроблення, щоб поліпшити її на наступній ітерації;
- зменшуються ризики замовника за рахунок того, що він може завершити розвиток безперспективного проекту з мінімальними (для себе) фінансовими втратами.

Недоліками спіральної моделі розроблення є:

- збільшується невизначеність у перспективах розвитку проекту;
- розроблення триває довго і коштує дорого;
- існує ризик застрягнути на першому етапі й нескінченно вдосконалювати початкову версію ПП;
- через динамічну зміну вимог складно планувати час та ресурси для виконання проекту.

Два останніх недоліки можна більш-менш легко вирішити шляхом уведення часових обмежень на кожен етап ЖЦ. Перехід від етапу до етапу у такому випадку здійснюється відповідно до плану незалежно від того, чи виконано всю заплановану на поточний етап роботу. План складається на основі отриманих у попередніх проектах статистичних даних і особистого досвіду керівника процесу розроблення.

Знання різних моделей ЖЦ і вміння їх застосовувати на практиці необхідні будь-якому керівнику. Правильний вибір моделі ЖЦ дозволяє грамотно планувати терміни, ресурси та обсяги фінансування, необхідні для виконання робіт, зменшити ризики як розробника, так і замовника. Це сприяє підвищенню авторитету розробників в очах замовника та впливає на перспективу подальшої співпраці.

Саме тому у таблиці 1.2 зібрано основну інформацію для порівняльного аналізу ефективності різних моделей (стратегій) ЖЦ ПП. Аналіз даної інформації дозволить більш ефективно обирати модель ЖЦ для зменшення ризиків та забезпечення відповідного рівня якості розробленого ПП чи системи.

З таблиці 1.2 видно, що спіральна модель має багато переваг у порівнянні з іншими, але вважати, що вона краща за інші, – не варто. Це пояснюється тим, що на кожен проект заключається окремий договір на певну вартість. Укладати договір на велику суму з невизначеним підсумковим результатом замовник ніколи не буде. У цьому випадку він запропонує виконати спочатку невеликий проект за невелику суму, а за результатами першої версії (ікременту) буде вирішувати питання про укладання додаткового договору на розвиток системи.

Таблиця 1.2. Порівняння різних моделей ЖЦ

<table>
<tr><td rowspan="2">Характеристика проекту</td><td colspan="3">Модель або стратегія</td></tr>
<tr><td>Каскадна</td><td>Інкрементна</td><td>Спіральна</td></tr>
<tr><td rowspan="2">Новизна розробки та забезпеченість ресурсами</td><td colspan="2">Типовий прцес розроблення. Якісно пророблені методи та відома технологія</td><td rowspan="2">Інноваційний підхід, нова технологія</td></tr>
<tr><td>Ресурсів замовника та розробника вистачає для реалізації ПП у короткий термін</td><td>Ресурсів замовника та розробника не вистачає для реалізації ПП у короткий термін</td></tr>
<tr><td>Строки виконання проекту</td><td>До року</td><td colspan="2">До декількох років. Розроблення однієї версії (інкременту) може тривати від декількох днів до року</td></tr>
</table>

Продовження таблиці 1.2

Зміна вимог з часом	Ні	Незначна зміна	Так
Розповсюдження проміжних версій ПП	Ні	Може бути	Так
Масштаб проекту	Малі та середні проекти. Великі проекти майже ніколи	Середні та великі проекти	Будь-які проекти
Укладання окремих договорів на окремі версії ПП	Укладається один договір на одну версію ПП	Окремий договір заключається на кожну окрему версію ПП або декілька послідовних версій	
Розроблення інкрементами (версіями)	Ні	Так	Так
Визначення основних вимог на початку проекту	Так	Так (у більшості випадків)	Не обов’язково

1.5 Гнучке розроблення програмного забезпечення

Agile – це гнучкий підхід до організації роботи та система установок. Даний підхід зосереджений на особистій відповідальності, тісному контакті та командній роботі.

Гнучке розроблення за принципом Agile найчастіше використовується, щоб покращувати клієнтський досвід, швидше постачати продукт на ринок, скорочувати кількість нераціональних і неефективних дій та підвищити якість ПП в цілому. Особливо ефективним є застосування Agile при роботі в умовах високої невизначеності.

Основні ідеї Agile [18]:

- люди і їх взаємодія є важливішими за процеси та інструменти;
- працюючий ПП є важливішим за вичерпну документацію;
- співпраця з замовником є важливішою за узгодження умов контракту;

- готовність до змін є важливішою за дотримання початкового плану.

Принцип взаємодії має на увазі, що команда взаємодіє з замовником і, навпаки, замовник з командою. Такий підхід дозволяє обмінюватися досвідом між замовником та учасниками команди розроблення, а також кожному з них брати участь у процесі прийняття рішень. Як наслідок, зниження ризиків втрати коштів і часу, а також підвищення здатності команди до вирішення нестандартних складних завдань з високим рівнем невизначеності.

Недоліки Agile випливають з його переваг. Так взаємодія всіх з усіма може призвести до хаотичного процесу розроблення, що впливає на всі етапи. Саме тому, використовуючи підходи Agile, потрібно зважати на певні обмеження: команди розроблення повинні бути невеликими, учасники повинні бути мотивованими та (що більш важливо) компетентними, має бути встановлено чіткі обмеження за часом, кожна ітерація має бути короткою з максимально зрозумілими цілями, а кінцевий результат повинен бути очевидним.

Agile прогнозує результат на більш короткий період у порівнянні зі стандартними стратегіями розроблення, тому дуже добре вирішує проблему невизначеності. Для підвищення ефективності використовується наступне правило: чим вище невизначеність, тим коротшою має бути ітерація. Варто зазначити, що дуже часто її тривалість може бути меншою (або рівною) 24 годин. У такому випадку на початку кожної ітерації обов’язково необхідно виконати аналіз готової роботи та запланувати наступну ітерацію.

Ідея іншого підходу – **Lean** – полягає в бережливому ставленні до ресурсів (час також є одним з основних ресурсів), а усі завдання вирішуються найпростішим способом. Наприклад, відразу весь ПП не розробляється. Це необхідно, щоб провести рекламну кампанію та збір інформації для перевірки того факту, що цей продукт є цікавим для кінцевого користувача. Після цього приймається усвідомлене рішення про необхідність подальшого розроблення.

У випадку розроблення ПП або прийняття виробничого чи інженерного поліпшення спочатку розробляється MVP. Термін «MVP» походить саме з підходу Lean. Звичайно, MVP-версію ПП можна покращувати і покращувати, але в цілому продукт на стадії MVP повинен бути зрозумілим та корисним кінцевому

користувачу. Аналіз роботи з MVP має дати змогу прийняти однозначне рішення щодо його подальшого вдосконалення. В іншому випадку ПП визнається невдалим, а також проводиться тестування нової гіпотези з мінімальною втратою ресурсів.

У цілому, підхід Lean орієнтується на тестування цінностей і потреб, щоб потрапити в тренди ринку самими мінімальними затратами часу та ресурсів. Lean – це підхід, який регулює не рішення технічних проблем інженерними засобами, а підприємницький підхід до вирішення завдань. Даний підхід бере за основу необхідність пошуку найпростішого рішення для досягнення результату: технічного, організаційного чи будь-якого іншого, спрощуючи при цьому все, що не є дійсно важливим.

Проте спрощення є також і однією з головних проблем підходу Lean: прагнення все спростити іноді призводить до ситуацій, в яких ПП спрощується настільки, що губляться дійсно важливі для нього функції і він виявляється непотрібним для кінцевого користувача.

Далі буде наведено короткий огляд основних методологій гнучкого розроблення з описом їх суті. Даний огляд не є вичерпним і дає лише загальне уявлення про розглянуті методології.

В основі методології **Scrum** лежить поняття спринту (sprint), у межах якого виконується робота над ПП. Перед початком кожного спринту проводиться етап планування. У межах кожного планування відбувається аналіз та оцінка вмісту списку завдань з розвитку ПП (Product Backlog) і формування завдань, які необхідно виконати (Sprint Backlog) у межах спринту. Дані списки завдань визначають напрямок, у рамках якого діє команда розроблення. Для спринту завжди існують певні обмеження у часі (найчастіше від одного тижня до місяця), а ЖЦ ПП розбивається на рівні по тривалості спринти.

Методологія **Kanban** полягає в тому, що проект ділиться на етапи (наприклад, планування, розроблення, тестування, тощо), які візуалізуються у вигляді канбан-дошки. Завдання на дошці подаються у вигляді «карток» і переміщаються з етапу на етап. Нові завдання на дошку можна додавати в будь-який час.

Закриття завдання відбувається при зміні його статусу на «завершено» незалежно від часу. Kanban це одна з методологій концепції Lean.

RUP (Rational Unified Process) – метод розроблення ПП, який у загальному випадку складається з чотирьох етапів: початкова стадія, стадія уточнення, побудова та впровадження. Кожен з етапів може включати в себе одну або декілька ітерацій. RUP – величезна методологія, яку важко розкрити в одному абзаці тексту (навіть у декількох сторінках тексту), але її методи є ефективними, бо засновані на статистиці комерційно успішних проектів.

RAD (Rapid Application Development) – методологія швидкого розроблення ПП, яка передбачає застосування інструментальних засобів прототипування і розроблення. У межах даної методології зазвичай формуються невеликі команди, замовник залучається до проекту з самих ранніх етапів розроблення, а терміни виконання становлять до 4 місяців. В основі даної методології лежить якомога повне визначення вимог до ПП, але існує також можливість їх уточнення та зміни безпосередньо в процесі розроблення ПП. Використання цієї методології дозволяє скоротити витрати коштів і часу до мінімуму.

DSDM (Dynamic Systems Development Model) – методологія, яка заснована на концепції швидкого розроблення додатків RAD. Особливістю DSDM є ітераційно-інкрементний підхід до процесу розроблення. Ключовим фактором цього підходу є активна та постійна участь кінцевих користувачів у процесі розроблення, а основною метою – здача готового проекту вчасно та в межах початкового бюджету, враховуючи постійне управління зміною вимог під час етапів розроблення.

На відміну від традиційних каскадних моделей (Waterfall, V-подібної, тощо) має дуже розвинену модель управління вимогами, у разі використання якої залишається не дуже багато об’єктів для управління.

XP (Extreme Programming) – є методологією гнучкого розроблення, яка реалізує еволюційну стратегію, має на меті значне скорочення термінів процесу розроблення при збереженні високої якості ПЗ, реагуючи на вимоги кінцевих користувачів. Це досягається завдяки постійній взаємодії з користувачами та наданні реальних доказів успішного розвитку ПП. Мінімізація помилок на ран-

ніх стадіях розроблення в межах XP дозволяє отримати готовий ПП високої якості максимально швидко. Це досягається за рахунок використання простої архітектури ПЗ, виконанням одного і того ж завдання одночасно декількома командами, а також великої кількості тестів.

Контрольні запитання

1. Які основні причини не виконання проектів з розроблення програмного забезпечення?

2. Які існують тренди та перспективи розвитку ІТ-сфери?

3. Що таке парадигма програмування? Які парадигми існують сьогодні?

4. Що таке модель обчислення? Що таке машина Тьюринга?

5. Чим відрізняються процедурні та непроцедурні парадигми?

6. Яка різниця між імперативним та декларативним програмуванням?

7. У чому полягає основна ідея структурного програмування?

8. Чим об'єктно-орієнтоване програмування відрізняється від структурного?

9. Що таке об'єкт, екземпляр класу, поле, метод?

10. Що таке життєвий цикл? Які процеси проходять у межах життєвого циклу програмного забезпечення?

11. Які є основні стратегії розроблення програмного забезпечення?

12. У чому різниця між моделями каскадної стратегії Waterfall та V-подібною?

13. У чому різниця між інкрементною (ітераційно-інкрементною) та спіральною моделями?

14. Які підходи гнучкого розроблення сьогодні широко використовуються?

15. У чому полягає різниця між основними методологіями гнучкого розроблення?

РОЗДІЛ 2

ОРГАНІЗАЦІЯ ПРОЦЕСУ РОЗРОБЛЕННЯ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

На початку 70-х років минулого століття багато великих проектів з розроблення ПП завершилися провалом. Це вперше було пов'язано не з кваліфікацією керівників та програмістів, яка була вище середнього, а з затримками в створенні ПЗ, низькою надійністю, перевищенням у декілька разів (у порівнянні з початковими оцінками) витрат на розроблення, низькою продуктивністю створених ПС [1]. Основною причиною цих невдач було неправильне використання підходів до процесу управління проектами, тому що застосовувалися методики, які були засновані на досвіді управління технічними проектами. При розробленні ПЗ вони довели свою неефективність.

Між професійним та аматорським розробленням ПЗ існує істотна різниця. Процес професійного створення ПЗ завжди пов'язаний з плануванням бюджету компанії-розробника, має чітко визначені вимоги та часові рамки, що призводить до необхідності використання певних методологій управління програмними проектами. Основним завданням керівника такого програмного проекту є необхідність гарантувати ефективну організацію робочого процесу, виконання як часових, так і бюджетних обмежень з урахуванням бізнес-цілей компанії (організації) щодо розроблюваного ПЗ.

Основні завдання, які мають виконувати менеджери програмних проектів: планування етапів розроблення ПП, контроль ходу виконання робіт, контроль дотримання всіх необхідних стандартів та відповідності результатів ТЗ. Постійний контроль за ходом виконання усього переліку робіт необхідний для того, щоб процес розроблення не виходив за обмеження по часу та бюджету. Якісне управління процесом розроблення не гарантує успішного та вчасного завершення проекту, але в іншому випадку проект гарантовано завершиться провалом. Наслідками провального проекту є значна затримка термінів здачі готово-

го ПЗ, перевищення кошторисної вартості проекту, невідповідність готового ПЗ специфікації вимог, тощо.

Як було сказано вище, процес розроблення ПЗ істотно відрізняється від процесів реалізації технічних проектів. Наведемо невеликий список цих відмінностей [1]:

1) **ПП є нематеріальним**. Менеджер проекту з побудови літака може бачити результати роботи на власні очі у будь-який момент часу. Якщо реалізація проекту відстає від графіка, – це відразу видно, так як частина конструкції не завершена. На противагу цьому ПП є нематеріальним і його не можна побачити чи доторкнутися до нього. Саме тому менеджер програмного проекту може покладатися лише на документацію, яка фіксує усі результати процесу розроблення ПП.

2) **Не існує повністю однакових процесів розроблення ПЗ**. На сьогоднішній день процеси розроблення технічних виробів ретельно випробувані й перевірені завдяки тому, що більшість технічних дисциплін мають тривалу історію розвитку. Розвиток технологій розроблення ПЗ відбувається лише протягом декількох останніх десятиліть. Саме тому в наш час не можна заздалегідь досить точно передбачити, на якому саме етапі процесу розроблення ПЗ можуть виникнути проблеми, що загрожують всьому проекту.

3) **Унікальність великих проектів**. Як правило, великі програмні проекти дуже сильно відрізняються від раніше реалізованих, тому для зменшення невизначеності у плануванні проекту менеджери та керівник повинні мати дуже великий практичний досвід. Але з плином часу він знецінюється через постійний розвиток технологічної бази, комп’ютерної техніки і комунікаційного устаткування. Накопичені знання та навички можуть не знадобитися також через необхідність використання в великому проекті нових технологій, яка може виникати як на початку, так і на будь-якому етапі процесу розроблення.

Будь-яка з перерахованих ситуацій може призвести до того, що реалізація проекту вийде за часові рамки або перевищить фінансування. Наприклад, ПС часто є ідейними або технічними новинками і, як інноваційні проекти, дуже часто порушуються терміни їх виконання [14]. Тому при плануванні процесу роз-

роблення варто завжди оцінювати ризики та заздалегідь збільшувати час виконання проекту, що дозволить вчасно реагувати на виникнення проблемних ситуацій без значної втрати ресурсів.

Основними процесами, на які треба звернути увагу менеджерам та керівникам у рамках управління проектами, є [1]: планування проекту, складання графіку робіт, управління ризиками, управління персоналом, тощо.

2.1 Процеси управління

Перелік робіт, які необхідно виконати менеджерам проекту, багато в чому залежать від компанії, у межах якої проводиться розроблення ПЗ, та типу створюваного ПП. Саме тому складно всеосяжно описати та стандартизувати весь перелік цих робіт. Як правило, більшість менеджерів несуть відповідальність за виконання деяких з наведених нижче процесів управління [1]:

- обговорення та формування договорів зі створення ПЗ;
- підбір персоналу;
- розроблення графіка робіт зі створення ПЗ;
- контроль за ходом виконання робіт;
- оцінювання вартості проекту;
- оцінювання ризиків;
- написання звітів.

Перша стадія будь-якого програмного проекту включає в себе процеси обговорення ТЗ та формування договору на розроблення відповідного ПЗ. Робота менеджера у даному випадку починається з написання пропозицій щодо реалізації проекту. Пропозиції повинні містити мету проекту, опис першочергових цілей і способів їх досягнення, оцінки витрат коштів та часу на виконання проекту. Також за необхідності керівник або менеджер проекту має навести обґрунтування необхідності передачі частини проекту на виконання сторонньою компанією або командою розробників.

Основним менеджером є керівник проекту. Саме тому в його обов’язки дуже часто входить підбір виконавців. Команда розробників, яка буде залучена

до виконання проекту, є особливим видом ресурсів. При формуванні команди також не варто забувати про інший вид ресурсів – кошти на можливі відрядження персоналу для обміну досвідом, вивчення нових технологій, підвищення кваліфікації, тощо. В ідеальному випадку професійний рівень та досвід виконавців ще на початку процесу розроблення повинен відповідати тій роботі, яку вони будуть виконувати в ході реалізації програмного проекту. Однак у більшості випадків менеджери повинні покладатися на команду розробників, яка далека від ідеальної. Така ситуація може бути викликана наступними причинами [1]:

1) Бюджет проекту не дозволяє залучити висококваліфікований персонал.

2) Неможливо знайти фахівців необхідної кваліфікації. Наприклад, кваліфіковані кадри як в компанії-розробника, так і поза нею можуть бути вже задіяними в інших проектах.

3) Підвищення кваліфікації. Якщо організація-розробник хоче підвищити професійний рівень своїх працівників, вона може залучити до участі в будь-якому проекті недосвідчених або недостатньо кваліфікованих працівників.

Отже, підбір фахівців для виконання проекту має певні обмеження і не завжди є вільним. Разом з тим необхідно зважати на те, щоб хоча б декілька розробників мали достатні для роботи над проектом кваліфікацію і досвід. В іншому випадку неможливо буде уникнути помилок при розробці ПЗ.

На етапі планування проекту визначаються етапи, процеси та очікувані результати, які повинні відповідати ТЗ. Реалізація цього плану має призводити до досягнення мети та цілей проекту. Також на даному етапі оцінюються необхідні для виконання плану ресурси та загальна вартість проекту.

Контроль за ходом виконання робіт (моніторинг проекту, [1]) – це безперервний процес, що триває протягом усього терміну реалізації проекту. Менеджер повинен постійно відслідковувати хід реалізації проекту та порівнювати фактичні і планові показники виконання робіт з їх вартістю. Хоча багато компаній мають механізми формального моніторингу робіт, досвідчений менеджер може скласти ясну картину про стадії розвитку проекту просто шляхом неформального спілкування з розробниками.

Неформальний моніторинг часто допомагає виявити потенційні проблеми, які в явному вигляді можуть проявитися пізніше. Наприклад, щоденне обговорення ходу виконання робіт може виявити окремі недопрацювання в створюваному ПП. Замість очікування звітів, в яких буде відображено відставання від графіка робіт, менеджер може обговорити з фахівцями виявлені проблеми.

У межах ЖЦ будь-якого проекту також відбувається декілька формальних контрольних перевірок ходу виконання робіт зі створення ПЗ. Такі перевірки повинні показати керівнику загальну картину ходу реалізації проекту.

ЖЦ великих проектів може складати від одного року до декількох десятиліть. Протягом цього часу вимоги до ПП та кінцева мета організації-розробника можуть істотно змінитися. Це означає, що може виникнути необхідність внесення значних змін у розроблюваний ПП. У деяких ситуаціях така необхідність модифікації функціональних можливостей є рівносильною розробленню нового ПП з нуля, тому іноді керівниками проекту та компанії-розробника приймається рішення про зупинення подальшого розвитку.

Також варто зазначити, що менеджери проекту зобов'язані постійно формувати звіти про поточний стан проекту та надсилати їх своїм керівникам, представникам замовника або організаціям-підрядчикам. Кожен такий звіт оформлюється у вигляді короткого документу, в якому зібрано основну інформацію для кожної групи зацікавлених осіб відповідно. Представлена інформація береться з об'ємних детальних звітів та має давати чітке розуміння поточного стану створюваного ПП.

2.2 Управління проектом

Як було сказано вище, основною метою процесу розроблення є досягнення кінцевого результату з чітко визначеними фінансовими та часовими рамками [19]. Розглянемо поняття проектного (залізного) трикутника, яке включає в себе три основні елементи, а саме: час, бюджет та обсяг робіт (рис. 2.1). При внесенні змін хоча б в один з даних елементів змінюються й усі інші. Це означає, що для будь-якого проекту вони всі є важливими, але на практиці, як правило, тільки одному з них (у залежності від пріоритетів) приділяється більше уваги.

Рис. 2.1. Проектний трикутник

Характер впливу змін одного з елементів на інші залежить в основному від конкретних обставин та специфіки проекту. Наприклад, зменшення терміну виконання проекту може призводити в одному випадку до збільшення його вартості через необхідність залучення більшої кількості розробників або, в іншому випадку, до зменшення вартості завдяки відмови від реалізації певних функціональних можливостей.

Якість – четвертий елемент проектного трикутника, який знаходиться в його центрі. Це символізує те, що навіть незначна зміна будь-якого з інших елементів обов'язково впливає на неї.

Наприклад, якщо команда розробників випереджає графік виконання проекту, то можна збільшити обсяг робіт з тестування функціональних елементів ПП. Цей додатковий час на тестування дозволить виявити неочевидні помилки в кодовій базі та домогтися більш високого рівня якості кінцевого продукту. Якщо ж виникне ситуація зменшення бюджету в ході виконання робіт, необхідно буде знизити витрати, наприклад, відмовившись від виконання деяких завдань або зменшивши тривалість їх виконання. І в одному, і в іншому випадку буде значна втрата якості кінцевого ПП.

Досягти балансу між усіма елементами проектного трикутника можна завдяки правильному плануванню робіт, які є необхідними для досягнення результату з відповідною якістю. Побудова чіткого глобального (проектного) плану робіт допомагає менеджеру передбачити ряд відомих проблем, які мо-

жуть виникнути на деяких етапах ЖЦ ПЗ, а також розробити превентивні заходи для їх попередження або вирішення. Оскільки план проекту є глобальним, він має досить детально описувати завдання всіх етапів процесу розроблення та бажані результати, а його виконання має призводити до успішного завершення проекту. Саме тому він (або його проект на початкових стадіях) розглядається як керівний документ для всіх учасників ЖЦ.

Крім розроблення глобального плану, менеджери повинні додатково розробити інші типи планів, які будуть його доповнювати (табл. 2.1, [19]).

Таблиця 2.1. Основні типи планів

План якості	Описує стандарти і заходи з підтримки якості розроблюваного ПЗ
План атестації	Описує ресурси, способи і перелік необхідних для атестації ПС робіт
План управління конфігурацією	Описує структуру і процеси управління конфігурацією
План супроводу ПЗ	Пропонує план заходів для супроводу ПЗ в процесі його експлуатації; розрахунок вартості супроводу і необхідних для цього ресурсів
План з управління персоналом	Описує заходи, спрямовані на підвищення кваліфікації членів команди розробників

У наступному лістингу показано процес планування ЖЦ ПЗ у вигляді псевдокоду [20]:

Визначення проектних обмежень.
Початкова оцінка параметрів програмного проекту.
Визначення етапів виконання проекту і контрольних відміток.
***while** поки програмний проект не завершиться чи не буде зупинений*
***loop** Складання графіка робіт.*
Початок виконання робіт.
Очікування закінчення чергового етапу робіт.
Відстеження ходу виконання робіт.
Перегляд оцінок параметрів програмного проекту.
Зміна графіка робіт.

Перегляд проектних обмежень.
if *(виникла проблема)* ***then***
перегляд технічних або організаційних параметрів проекту
end if
end loop

У даному псевдокоді зроблено акцент на тому, що планування є ітераційним процесом.

До важливих чинників, які мають обов'язково враховуватися в процесі розроблення можна віднести наступні [1, 19]: фінансові та ділові зобов'язання організації-розробника, бізнес-правила, функціональні та не функціональні вимоги, зворотній зв'язок від замовника або кінцевих користувачів, тощо. Якщо вони змінюються, то повинен також змінитися і план робіт. На практиці доведено, що кожен розроблений план має постійно переглядатися й оновлюватися, оскільки в процесі виконання програмного проекту досить часто один (чи декілька) з наведених чинників розширюється або уточнюється.

Процес планування починається з визначення часових, бюджетних, ресурсних та інших обмежень, які повинні визначатися паралельно з оцінюванням основних параметрів проекту (наприклад, його структури та розміру), а також з розподілом обов'язків між виконавцями. На наступному кроці зазвичай визначаються етапи розроблення та результати (прототипи, підсистеми, версії ПП, документація, тощо), які повинні бути отримані після завершення цих етапів. Після створення детального графіка виконання робіт починається циклічна частина планування, яка полягає в їх контролі (зазвичай через 2-3 тижні) та визначенні відмінностей між початковим планом та поточним станом справ і оновленні планів.

Первинні оцінки параметрів проекту можуть переглядатися у міру надходження нових звітів про хід виконання програмного проекту. Це, у свою чергу, може привести до зміни розробленого графіка робіт. Якщо в результаті цих змін порушуються терміни завершення проекту, повинні бути переглянуті та обов'язково погоджені з замовником проектні обмеження.

Світова практика розроблення ПЗ показує, що при виконанні більшості проектів виникає ряд проблем [14], навіть якщо їх менеджери (керівники) на початку повністю впевнені в ідеальності розроблених планів. Бажано зважати на можливі проблеми ще до моменту їх виникнення, але, звичайно, неможливо побудувати план, який зміг би врахувати всі (у тому числі випадкові) проблеми та затримки виконання проекту. Саме тому і виникає необхідність періодичного оновлення проектних обмежень і етапів розроблення ПП.

На рис. 2.2 показано інше уявлення процесу планування у вигляді багатошарового ітераційного процесу [20].

Рис. 2.2. Послідовність дій при плануванні програмного проекту

2.2.1 План проекту

Як було сказано вище, план проекту є глобальним і повинен чітко показувати необхідні для реалізації проекту ресурси, етапи та часовий графік їх виконання. План проекту іноді складається у вигляді єдиного документу, який містить усі інші види планів. Проте найчастіше він описує тільки технологічний процес розроблення ПЗ з посиланнями на інші типи планів, які створюються окремо.

У залежності від компанії-розробника та від виду ПП деталізація плану проекту може дуже сильно відрізнятися, але переважна їх більшість містять наступні розділи [1]:

1) Вступ, в якому коротко описуються актуальність, мета, цілі та обмеження (ресурси, час, тощо) проекту, які важливі для управління ним.

2) Аналіз ризиків. Містить розрахунок ймовірностей виникнення відомих для подібних проектів ризиків, пов'язаних з персоналом, використовуваними технологіями та стратегіями ЖЦ, а також перелік спрямованих на їх зменшення заходів.

3) Управління кадрами. У даному розділі наводяться наступні елементи: перелік необхідних для успішного виконання проекту компетентностей, визначення основних обов'язків, методи підбору команди розробників, методики прийняття рішень, тощо.

4) Програмні та апаратні ресурси. Містить перелік ліцензійного ПЗ, яке необхідно використати в процесі розроблення, його вартість (у разі необхідності закупівлі або аренди за тимчасовою ліцензією), а також перелік необхідних апаратних засобів спільно з графіком закупівлі і поставки.

5) Етапи проведення робіт. ЖЦ розроблюваного ПЗ у відповідності до обраної стратегії розбивається на окремі етапи, визначаються бажані результати (артефакти) завершення кожного етапу та контрольні позначки (точки).

6) Графік робіт, в якому відображаються основні залежності між кожним етапом, часом на їх завершення та виконавцями.

7) Контроль ходу виконання проекту. Даний розділ містить основні типи звітів, які мають формувати керівник та менеджери іншим учасникам процесу розроблення, терміни їх надання, а також механізми глобального моніторингу ходу виконання проекту.

Оскільки план проекту регулярно оновлюється для відображення змін, наприклад, у графіку виконання робіт або інших елементах процесу розроблення, необхідно якісно організувати документообіг, який дозволить відстежувати ці зміни.

2.2.2 Контрольні позначки етапів робіт

Для якісної організації розроблення ПЗ та управління цим процесом менеджерам необхідна певна інформація, яка може бути отримана тільки у вигляді документів, в яких описано поточний стан виконання чергового етапу. Без наявності цієї інформації керівник та замовник не можуть зробити висновки про ступінь готовності ПП, а також неможливо оновити графік робіт чи скоригувати бюджет проекту.

Саме тому при плануванні будь-якого проекту обов'язково визначаються **контрольні позначки (точки)** – моменти часу, які характеризують закінчення відповідного етапу робіт з отриманням певного результату та супроводжуються формуванням звіту для зацікавлених осіб (рис. 2.3). Контрольні точки можуть бути основними (глобальними, зовнішніми) або другорядними (локальними, внутрішніми). Наприклад, після досягнення глобальних контрольних точок (завершення етапів формування вимог, проектування системної та програмної архітектур, кодування, тестування, тощо) результати можуть бути оформлені відповідним чином для перевірки представниками замовника. Ці результати зазвичай називаються **контрольними проектними елементами** та надаються замовнику у вигляді документації, готової підсистеми або прототипу ПП, тощо.

Хоч контрольні проектні елементи можуть бути подібними до результатів виконання будь-якого етапу, проте вони не повністю відповідають контрольним точкам (ні локальним, ні глобальним), які є елементами внутрішнього контролю результатів виконання проекту. Отже, контрольні проектні елементи фор-

муються на основі проходження однієї чи декількох контрольних позначок. Наприклад, на рис 2.3 зображено декілька початкових етапів ЖЦ ПЗ, яке розробляється на основі прототипу. Результати кожного етапу отримуються у визначених контрольних точках, а контрольним проектним елементом є об'єднана специфікація вимог – системних та користувача.

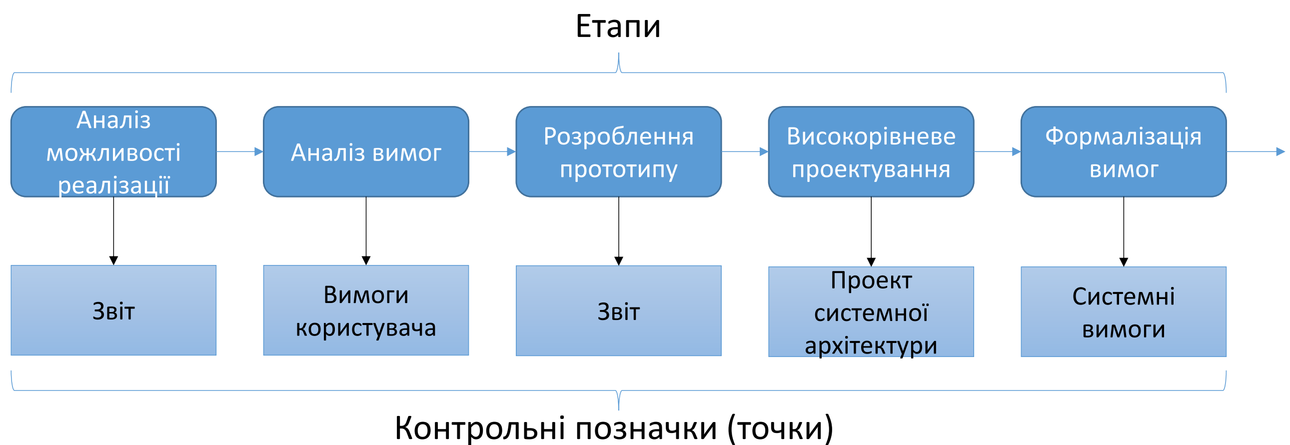

Рис. 2.3. Етапи процесу розроблення специфікації

2.2.3 Графік робіт

Складання графіка – один з найвідповідальніших видів робіт, які виконує менеджер проекту. Тут оцінюється тривалість проекту, визначаються ресурси, необхідні для реалізації окремих етапів робіт, і формується список етапів у вигляді узгодженої послідовності. Якщо компанія-розробник отримує замовлення на проект, подібний до якого вона виконувала в минулому, то попередній графік робіт (у залежності від ступеня подібності проектів) можна взяти за основу для поточного. Проте не варто забувати, що кожен проект з розроблення ПЗ є по-своєму унікальним і графік робіт у будь-якому випадку доведеться доопрацьовувати.

Для інноваційних проектів, які не мають аналогів (взагалі або в межах компанії-розробника) початкові оцінки необхідних часу та ресурсів зазвичай є занадто оптимістичними. Це актуально і для випадків, коли дуже досвідчений керівник намагається передбачити всі можливі проблеми. Завжди існує достатньо велика ймовірність виникнення непередбачуваних ситуацій. Навіть очевид-

на необхідність уточнення вимог у процесі розроблення ПП може призводити до необхідності внесення значних змін у графік робіт. Цим проекти з розроблення ПЗ не відрізняються від великих технічних (інженерних) проектів, наприклад, зведення аеропорту, моста, тощо. Саме тому при надходженні нової інформації про хід виконання проекту графік робіт обов'язково необхідно оновити.

Процес складання графіка робіт зображено на рис. 2.4. Основними цілями цього процесу є визначення усіх етапів ЖЦ розроблюваного ПЗ, оцінка часу виконання кожного етапу та контрольних позначок. Оскільки при виборі деяких стратегій ЖЦ частина етапів можуть виконуватися паралельно, необхідно також враховувати цю можливість і передбачити перерозподіл ресурсів найбільш оптимально. Однією з основних причин затримки виконання проекту є саме брак ресурсів (людських, матеріальних, тощо) для виконання певного етапу.

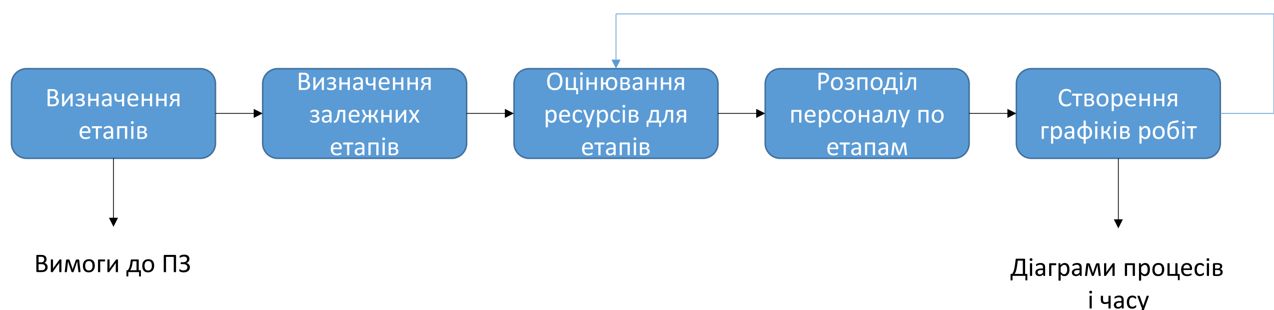

Рис. 2.4. Процес складання графіка робіт

Мінімальна тривалість етапу зазвичай знаходиться в межах від двох днів до тижня (наприклад, при тривалості інкременту в двадцять чотири години). Для великих проектів цей період може збільшуватись до місяця. Зменшення тривалості етапу може призводити до необхідності оновлювати графік робіт занадто часто, тому кожного разу необхідно ретельно аналізувати ризики для вибору оптимального інтервалу. Команда розробників може допускати помилки під час виконання, можуть вийти з ладу програмні або апаратні засоби підтримки процесу розроблення, технічно складні або інноваційні проекти за замов-

чуванням мають більшу ймовірність прийняття неправильних рішень – це лише невеликий перелік ситуацій, які обов’язково виникнуть при дуже малій тривалості етапів розроблення.

Не тільки нижня межа тривалості етапу впливає на хід виконання проекту, але й верхня. Як правило, максимальна тривалість етапу навіть для великих проектів не перевищує декількох місяців. При цьому занадто великі етапи можуть розбиватися на декілька менших з відповідними контрольними позначками. Усе це обов’язково має враховувати менеджер під час складання графіка робіт.

Враховуючи світовий досвід розроблення програмного забезпечення [14], емпірично отримано правило для оцінювання витрат часу: «необхідно визначити тривалість ідеального проекту (сума інтервалів усіх етапів), збільшити цей час на 30% від отриманого числа для вирішення можливих відомих проблем, збільшити ще на 20%, щоб бути готовими до вирішення непередбачуваних ситуацій». Можливі проблеми, які важко спрогнозувати, залежать від типу проекту, його параметрів, кваліфікації та досвіду виконавців, тощо. Також варто пам’ятати, що у відповідності до проектного трикутника (рис. 2.1) збільшення часу виконання проекту призводить до підвищення витрат коштів та ресурсів.

Як правило, графік робіт представляється у вигляді часових (показує час початку, закінчення та тривалість кожного етапу), мережевих (відображає залежності між різними етапами програмного проекту) та інших діаграм. Деякі з них можна створювати в автоматизованому режимі на основі інформації з бази даних (БД) за допомогою програмних засобів підтримки управління.

Розглянемо етапи довільного проекту, які представлені в табл. 2.2. У даному прикладі видно, що етап Т3 знаходиться в прямій залежності від етапу Т1. Отже, етап Т3 не може початися до завершення етапу Т1. Наприклад, високорівневе проектування ПС (Т3) не може початися до визначення вимог (Т1).

Мережева діаграма (мережевий графік) будується на основі наведених в табл. 2.2 значень тривалості етапів, враховуючи зв’язки та залежності між ними. На рис. 2.5 зображено приклад мережевої діаграми, з якого добре видно, що деякі види робіт (позначені прямокутниками етапи) виконуються паралельно, а

інші – ні. Літера «М» з номером відповідає контрольним точкам та контрольним проектним елементам (зображені овалами). Діаграма читається зверху-вниз і зліва-направо, а зображені на ній дати відповідають початку виконання відповідних етапів.

Таблиця 2.2. Приклад етапів проекту

Етап	Тривалість, днів	Зв'язок (контрольна точка)
Т1	7	
Т2	14	
Т3	17	Т1 (М1)
Т4	10	
Т5	10	Т4, Т2 (М2)
Т6	6	Т2, Т1 (М3)
Т7	20	Т1 (М1)
Т8	25	Т4 (М5)
Т9	14	Т6, Т3 (М4)
Т10	14	Т7, Т5 (М7)
Т11	7	Т9 (М6)
Т12	10	Т11 (М8)

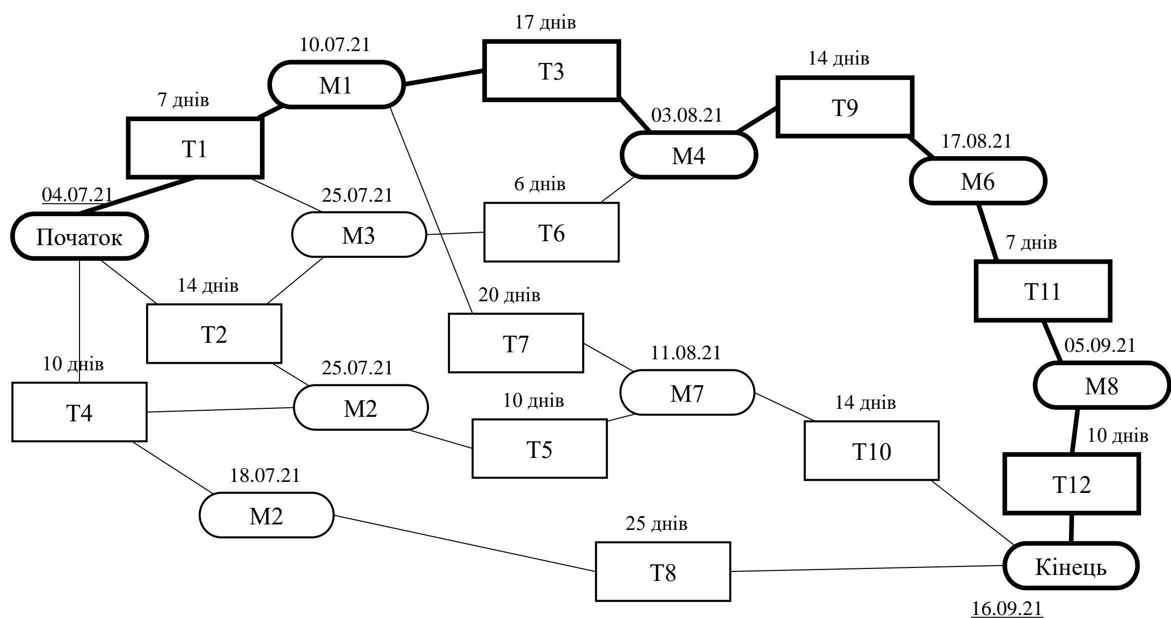

Рис. 2.5. Приклад мережевої діаграми

При створенні мережевої діаграми засоби підтримки управління проектом автоматично ставлять контрольні точки після закінчення кожного етапу. Контрольні проектні елементи (як і контрольні позначки) можуть залежати від виконання декількох попередніх етапів. Саме тому в третьому стовпчику табл. 2.2 наведено контрольні точки, після яких може починатися наступний етап виконання проекту.

З рис. 2.5 також видно, що певний етап не може розпочатися до тих пір, поки не будуть виконані усі етапи, які ведуть до нього від самого початку процесу розроблення ПЗ. Наприклад, поки не пройдена контрольна позначка M7, яка свідчить про завершення етапів T5 і T7, етап T10 не зможе початися.

Критичний шлях – найдовша за сумарною тривалістю послідовність етапів на мережевій діаграмі від початку процесу розроблення до його завершення. Він характеризує мінімально необхідний час на виконання всього проекту. Наприклад, на рис.2.5 критичний шлях зображено жирними лініями і тривалість проекту становить 55 робочих днів або 11 тижнів.

Будь-яка затримка виконання етапів критичного шляху автоматично призводить до затримки виконання проекту в цілому. Затримка виконання етапів, які не входять до критичного шляху, у загальному випадку не впливає на загальну тривалість проекту. Останнє твердження залишається актуальним до того моменту, коли сумарна тривалість етапів на інших шляхах через виникнення непередбачуваних проблем не перевищить тривалість критичного шляху.

Отже, щодо мережевої діаграми можна зробити наступні висновки:

- вона дозволяє виявити рівень значущості кожного етапу в межах проекту;
- детальний аналіз критичного шляху дозволяє знайти способи оптимізації процесу розроблення, що може призвести до скорочення тривалості процесу створення ПЗ;
- менеджери можуть її використовувати для розподілу робіт за етапами між окремими командами розроблення.

На рис.2.6 зображено часову діаграму (діаграму Ґанта, стрічкову діаграму), яка є іншим представленням графіка робіт. Вона показує тривалість вико-

нання, час початку і закінчення кожного етапу проекту, а також прогнозований час затримки (затінені частини стрічок), не вдаючись в її причини. Стрічки без затінених частин відповідають етапам критичного шляху. Таким чином дана діаграма показує, що затримка цих етапів призведе до збільшення часу виконання всього проекту.

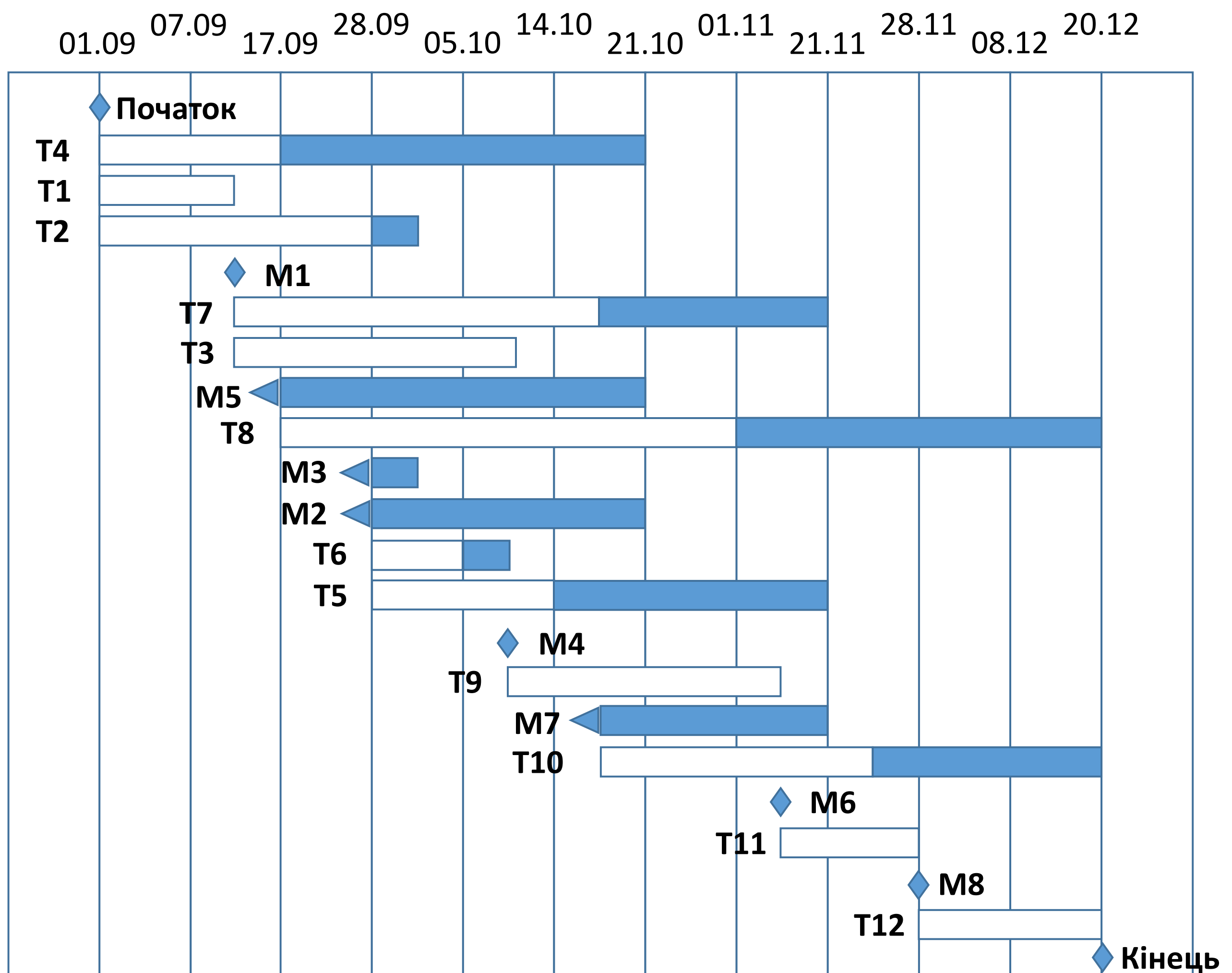

Рис. 2.6. Приклад часової діаграми тривалості етапів

Як і мережева, діаграма Ґанта також може бути побудована з використанням спеціалізованих програмних засобів підтримки процесу управління.

Окрім підбору виконавців в обов'язки менеджера проекту також входить розподіл ресурсів та персоналу за етапами. У табл. 2.3 наведено розподіл розробників за представленими на рис. 2.6 етапами.

Таблиця 2.3. Приклад розподілу виконавців за етапами

Етап	Тривалість, днів
Т1	Ольга
Т2	Ганна
Т3	Ольга
Т4	Олег
Т5	Світлана
Т6	Ганна
Т7	Петро
Т8	Олег
Т9	Ольга
Т10	Ганна
Т11	Олег
Т12	Олег

За даними з таблиці можна додатково побудувати іншу стрічкову діаграму: діаграму зайнятості (рис. 2.7). Як і всі попередні, вона може бути побудована в автоматизованому режимі за допомогою спеціалізованих засобів. Усі кадри (для даного прикладу) не задіяні в роботі над проектом протягом всього періоду розроблення. Це означає, що у вільний від даного проекту час вони можуть бути у відпустці, на курсах підвищення кваліфікації, працювати над іншим проектом, тощо.

У великих компаніях зазвичай працює велика кількість персоналу, який задіється для виконання проекту за необхідності. Такий підхід до організації робочого процесу створює для менеджерів проекту додаткові проблеми, наприклад, якщо за графіком певний розробник вже має підключитися до виконання, але він все ще працює над проектом, який вийшов за свої часові рамки (затримується).

Саме тому початковий графік робіт є скоріше проектним і неминуче буде містити відхилення в одну чи іншу сторону від реальної ситуації. У ході виконання проекту розраховані менеджером оцінки тривалості етапів порівнюються

з поточним станом проекту й графіки оновлюються. Це стосується не лише термінів виконання поточного етапу. За результатами порівняння відбувається коригування також графіку етапів, які ще не розпочалися, щоб спробувати зменшити тривалість критичного шляху.

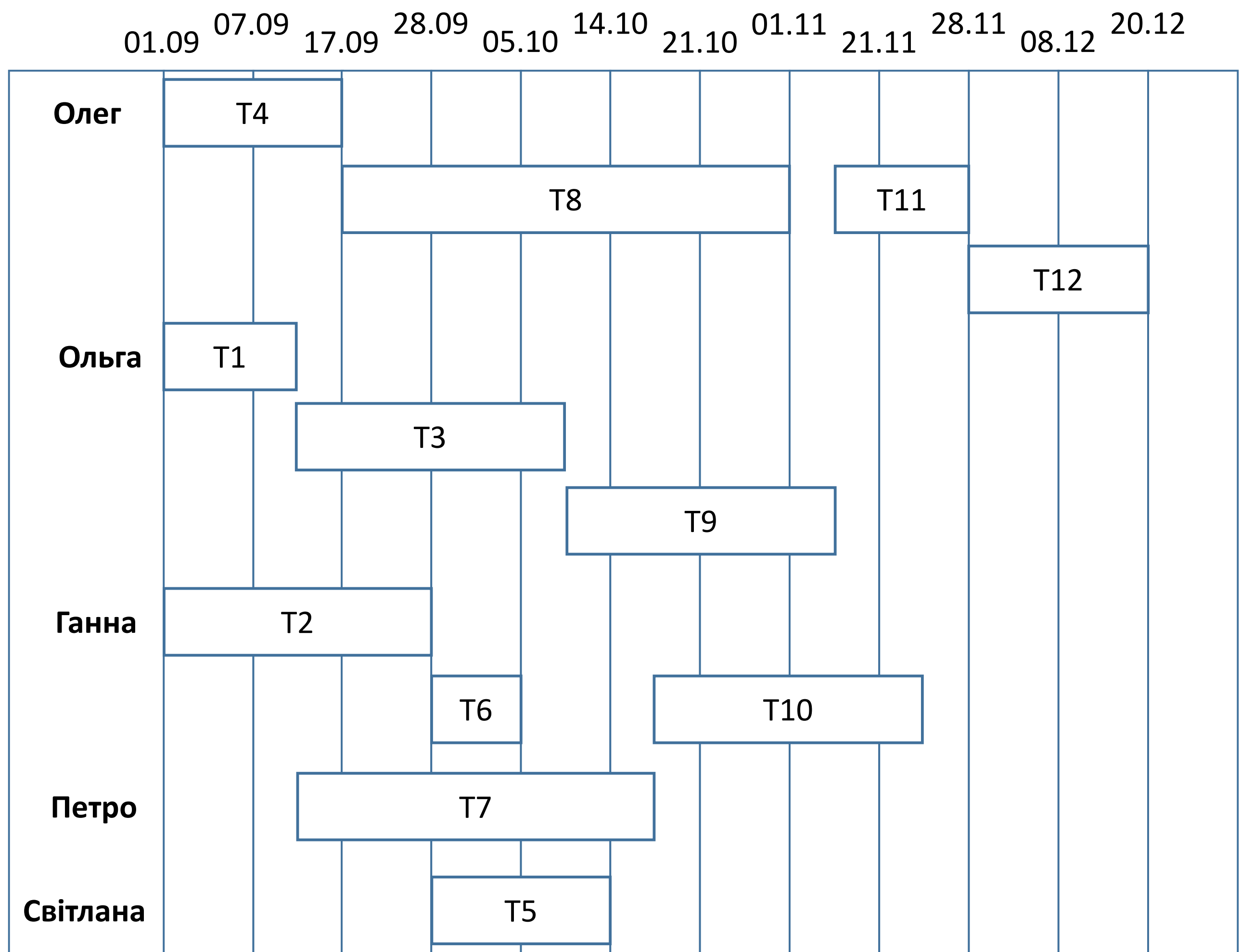

Рис. 2.7. Приклад часової діаграми зайнятості

Розглянемо ще один приклад використання мережевого графіку та діаграми Ґанта. Це класична задача планування: компанія працює у нормальному режимі та отримала замовлення на розроблення ПЗ. Для планування робіт необхідно побудувати мережевий графік та діаграму Ґанта.

У даному прикладі початковий мережевий графік зображено у вигляді орієнтованого графа (рис. 2.8, [19]) з двома виділеними вершинами, які відповідають за початок і кінець роботи. У вершинах графа розміщено події, які відпо-

відають пунктам наведеного нижче плану, а стрілками позначено відповідні типи виконуваних робіт. Опис подій має містити дієслова доконаного виду, наприклад, «тести виконано», «якість оцінено як достатню», «результати передано в БД», тощо. Стрілки (ребра) графа також мають містити поруч інформацію щодо тривалості поточного етапу робіт в днях або тижнях (у місяцях – майже ніколи).

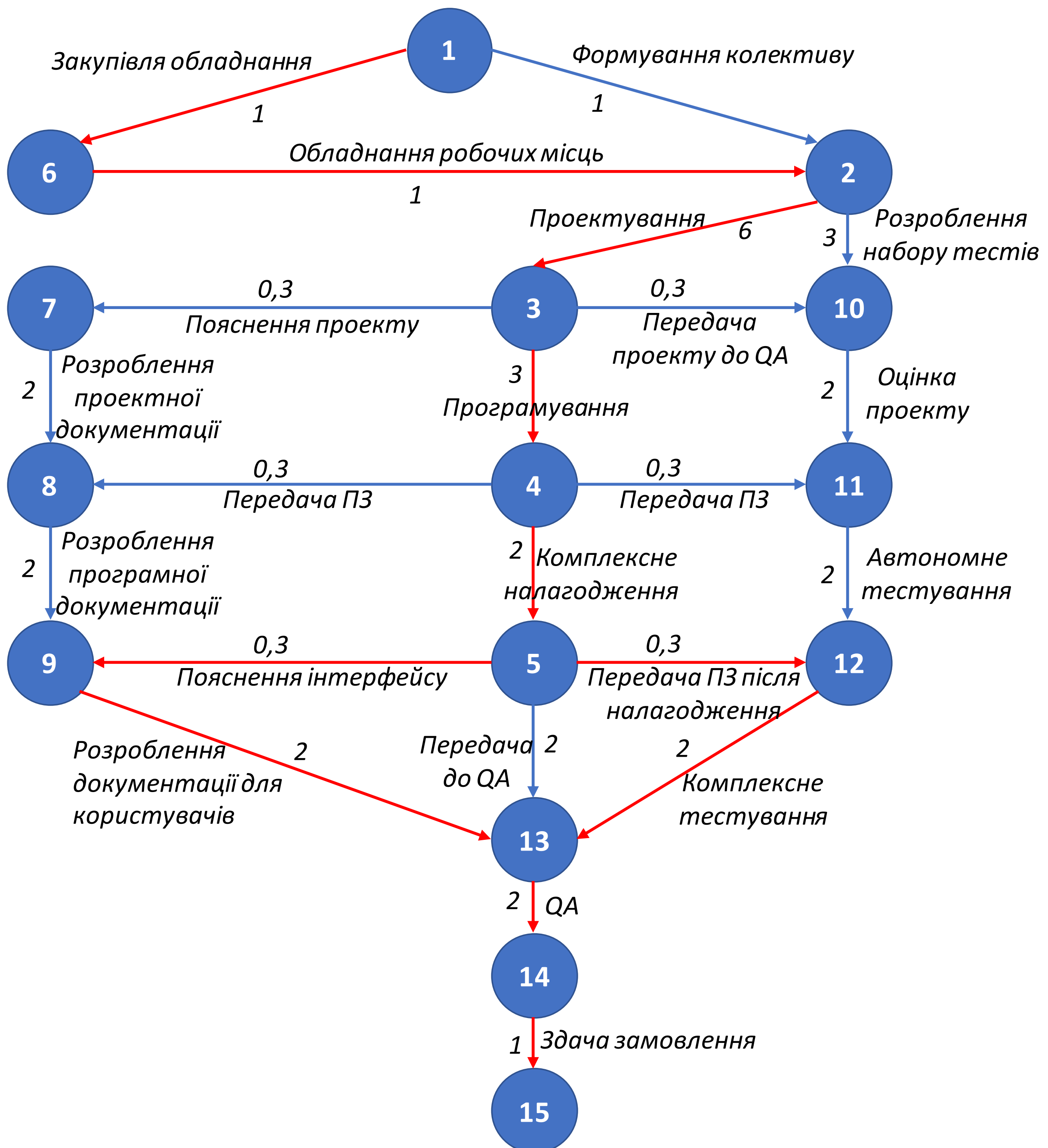

Рис. 2.8. Приклад мережевого графіка

Розглянемо події, які присутні в даному мережевому графіку [19]:

1. Роботу розпочато.

2. Робочі місця підготовлено, колектив виконавців сформовано.

3. Високорівневе проектування ПЗ завершено.

4. Детальне проектування та програмування завершено.

5. Комплексне налагодження ПЗ завершено.

6. Необхідне обладнання закуплено.

7. Групою з документування отримано опис проекту та всі необхідні пояснення від проектувальників.

8. Групою з документування завершено створення проектної документації.

9. Групою з документування отримано всю необхідну інформацію про користувацькі інтерфейси, розроблення програмної документації завершено.

10. Групою оцінки якості розроблено тести.

11. Групою оцінки якості оцінено проект позитивно.

12. Групою оцінки якості завершено автономне тестування.

13. Групою оцінки якості завершено комплексне тестування, отримано всю документацію та діючий варіант ПС.

14. Перевірку якості завершено.

15. Закінчення роботи.

Під кожною стрілкою зображеного на рис. 2.8 графа записано початкову тривалість роботи в тижнях. Критичними шляхами є наступні: 1-6-2-3-4-5-12-13-14-15 та 1-6-2-3-4-5-9-13-14-15. Це означає, що проект не можна закінчити швидше за 18,3 тижні.

З одного боку, можна спробувати розробити мережеву діаграму таким чином, щоб усі можливі шляхи від початку до кінця мали приблизно однакову тривалість. Наприклад, можна примусити групу оцінки якості проводити навіть початкове тестування. Це призведе до зменшення навантаження на програмістів. З іншого боку, команда розроблення у відповідності до проектного трикутника (рис. 2.1) для дотримання термінів може видавати результат низької якості. Повернення роботи після тестування для доопрацювання займе ще більше

часу. На практиці (у проектах з великою кількістю робіт) можна покращити мережевий графік за рахунок перерозподілу робіт між етапами.

Розглянемо останню мережеву діаграму (графік) більш детально. Події 1, 2 та 6 сплановані не зовсім вдало: команда розробників формується за тиждень, але робочі місця ще не будуть готові; фахівці з документування починають працювати лише через шість тижнів після проектувальників; група оцінки якості має майже тиждень перерви до завершення проектування. Вирішення цих проблем є складним процесом, який вирішується кожною компанією по-своєму.

Діаграму Ґанта для розглянутого прикладу [19] зображено для порівняння на рис. 2.9.

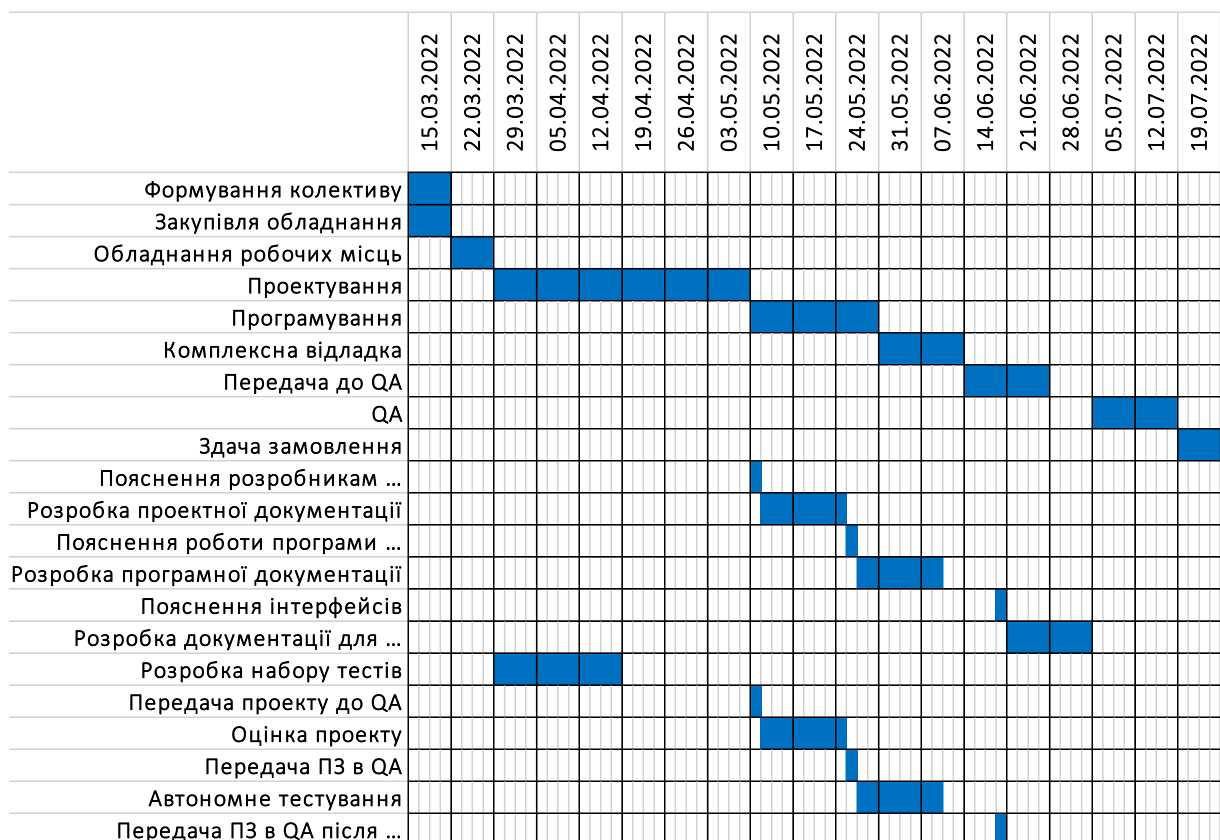

Рис. 2.9. Приклад діаграми Ґанта

Мережева діаграма наочно показує залежність етапів між собою, а з діаграми Ґанта можна зрозуміти, що саме відбуватиметься в кожен конкретний момент часу. Наприклад, можна побачити, що група оцінки якості між роботами 11-12 та 12-13 має перерву в роботі майже 2 тижні. Тому керівники проектів

дуже часто віддають перевагу саме діаграмі Ґанта. На ній також можна над кожним відрізком роботи зазначати кількість задіяних у даній роботі виконавців, а на мережевій діаграмі це робити не зручно, тому що треба відмітити ще й тривалість роботи. Для керівників проектів дана властивість стрічкової діаграми дуже часто є визначальним фактором при виборі діаграм для організації управління процесом розроблення.

Технічні менеджери частіше віддають перевагу мережевим графікам, оскільки для них більш важливою є інформація про взаємну залежність робіт та критичні шляхи, оскільки їх доволі часто доводиться перераховувати та оновлювати.

Отже, щодо основ управління проектом можна зробити наступні висновки:

1) Ознакою хорошого управління програмним проектом є виконання проекту відповідно до графіка робіт і в рамках запланованого бюджету.

2) Управління програмними проектами відрізняється від управління іншими технічними проектами, оскільки ПЗ є нематеріальним. Накопиченого при реалізації попередніх проектів досвіду може виявитися недостатньо для побудови стратегії управління інноваційними або складними програмними проектами.

3) Менеджери програмних проектів виконують безліч різнотипних робіт. Серед них основними є: планування проекту, оцінювання необхідних для реалізації проекту ресурсів, а також складання графіка робіт. Планування проекту та оцінювання необхідних ресурсів – це ітераційні види діяльності, які тривають протягом усього терміну виконання проекту. У міру надходження додаткової інформації про хід виконання проекту плани і графіки можуть переглядатися.

4) Контрольні позначки (точки) – це прогнозовані результати етапів реалізації проекту, які спільно з коротким звітом про виконання етапу передаються керівництву проектом. Контрольні проектні елементи – це глобальні контрольні позначки, які надаються замовнику ПС.

5) Складання графіка робіт полягає у створенні різних графічних представлень окремих частин плану проекту. Сюди відносяться мережеві діаграми ета-

пів, що показують взаємозалежність етапів робіт, а також часові діаграми тривалості етапів.

2.3 Вимоги до програмного забезпечення

Процес проектування ПП починається із визначення вимог до розроблюваного ПЗ і його початкових даних. У результаті аналізу вимог отримують специфікації ПЗ у вигляді текстових описів, структурних схем і діаграм. У процесі визначення специфікацій будують загальну модель предметної області і конкретизують основні функції ПП та його поведінки при взаємодії з навколишнім середовищем [21].

IEEE Standard Glossary of Software Engineering Terminology (1990) визначає вимоги як:

1. Умови або можливості, необхідні користувачу для вирішення проблем або досягнення цілей;

2. Умови або можливості, якими повинна володіти система або системні компоненти, щоб виконати контракт або задовольняти вимоги стандартів, специфікацій або інших формальних документів;

3. Документоване представлення умов або можливостей для пунктів 1 і 2.

Ці визначення охоплюють вимоги як користувачів (зовнішня поведінка системи), так і розробників (деякі приховані параметри).

Вимоги – це специфікація того, що повинно бути реалізовано в ПЗ. У них описано поведінку системи, її властивості або атрибути. Вимоги можуть бути обмежені процесом розроблення системи [8]. Інше визначення вимог [20] говорить, що «вимогами називається опис функціональних можливостей і обмежень, які накладаються на створюваний ПП».

Зазвичай вимоги виражають що ПЗ повинно виконувати, але не уточнюють як саме цього досягти. Розглянемо приклад вимоги до банківської системи: «ПС повинна надати клієнту можливість виконання наступних операції з його рахунком: перегляд, зняття грошей, поповнення». Проте наступний запис не є правильною вимогою до ПЗ: «інформація банківського рахунку повинна зберігатися у вигляді таблиці системи управління базами даних MySQL».

У другому прикладі вказуються не просто вимоги до функціональних можливостей ПС, а й уточнюється як мають бути реалізовані її внутрішня будова та взаємозв'язки між елементами. Втім, можуть бути і виключення із цього правила. Наприклад, у замовника можуть бути особливі причини для такої реалізації.

Розрізняють дві категорії представлення вимог: вимоги замовника (первинні вимоги) і вимоги розробника (детальні вимоги). Відрізняються вони один від одного ступенем опрацювання описів. Первинні вимоги документують побажання і потреби замовника, пишуться мовою, яка є зрозумілою саме для нього. Детальні вимоги документують у спеціальній, структурованій формі. Вони є більш деталізованими по відношенню до первинних вимог.

Робота зі створення первинних вимог проводиться на етапі підготовки ЖЦ ПЗ і називається збором або формуванням вимог. Робота зі створення детальних вимог проводиться на етапі моделювання ЖЦ ПЗ і називається аналізом вимог.

Деякі проблеми можуть бути породжені відсутністю чіткого розуміння відмінностей між цими категоріями вимог, отже, розглянемо різницю між ними. Наприклад, вимога замовника звучить наступним чином: «ПЗ повинно забезпечити засоби для введення і збереження різноманітних даних абонента-користувача». При цьому відповідна вимога розробника може записуватися в структурованій формі:

1.1) Користувач повинен мати можливість визначати тип інформації, яка вводиться.

1.2) Для кожного типу даних має бути відповідний засіб, що забезпечує введення і збереження елемента даних цього типу.

1.3) Кожен тип даних повинен представлятися відповідною піктограмою на дисплеї користувача.

1.4) Користувачеві має пропонуватися піктограма для кожного типу даних. Крім того, повинна надаватися можливість самостійного вибору піктограми для кожного типу даних.

1.5) При виборі користувачем піктограми типу даних до елементу даних має бути застосовано асоційований зі вказаним типом засіб.

Вимоги замовника містять деталізований опис функцій і обмежень ПС і переносяться в системну специфікацію, яка слугує основою для укладення контракту між покупцем і розробниками. Додатково до них можуть прикріплятися діаграми з поясненнями та макети.

Часто стандарти програмної інженерії інтегрують обидві категорії вимог в єдиний документ (рис. 2.10).

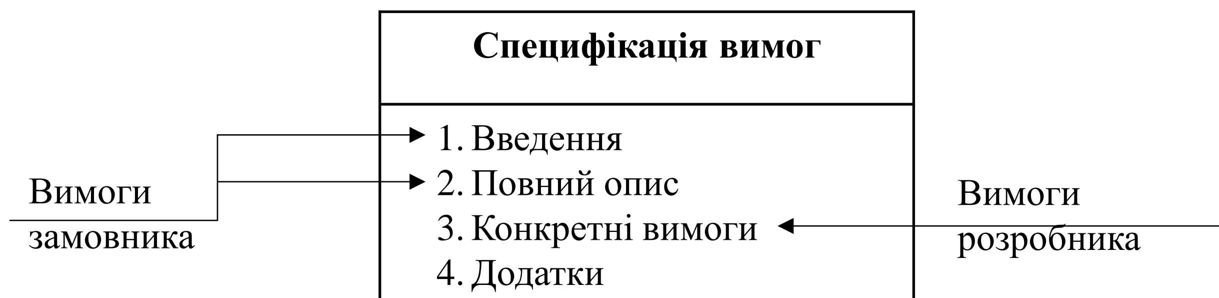

Рис. 2.10. Інтеграція вимог замовника і розробника в єдину специфікацію

Специфікація вимог (*англ. software requirements specification, SRS*) – документ, в якому зафіксовано усі вимоги, які висуваються до розроблюваного ПЗ. Дане поняття зазвичай ототожнюється з ТЗ на розроблення.

Зазвичай розрізняють два типи вимог:

- *Функціональні вимоги* (ФВ) описують поведінку програмної системи, а також сервіси (функції), які вона повинна надавати. Для їх визначення необхідно провести всебічний аналіз проблемної (предметної) області. У ході визначення функціональних вимог розглядаються різноманітні варіанти поведінки програмної системи, які визначаються різними вхідними даними і станами зовнішнього середовища.
- *Нефункціональні (експлуатаційні) вимоги* (НВ) відносяться до характеристик якості ПС та її зовнішнього оточення. Додатково можуть перераховуватися обмеження, які накладаються на дії та функції

системи, а також на умови розроблення (обмеження за часом, обмеження на організацію проекту, стандарти тощо).

У реальності ці два типи вимог пов'язані між собою. Це пов'язано з тим, що не можна повноцінно реалізувати один з цих типів без реалізації іншого. Наприклад, вимоги безпеки ПС, які пов'язані з користувачем, можна віднести до НВ. Однак, при більш детальному аналізі дані вимоги можна віднести також і до ФВ, оскільки вони створюють необхідність включення в систему засобів авторизації користувача.

4.3.1 Функціональні вимоги

Кожен ПП призначений для виконання певних функцій. Для того, щоб визначити чи підходить та чи інша програма для вирішення завдань, необхідно мати чіткий набір критеріїв, на підставі якого можна зробити правильний вибір.

ФВ описують сервіси, що надаються ПЗ [21], його стандартну поведінку, реакцію на різноманітні вхідні дані, а також перелік дій, які програма дозволить виконувати кінцевим користувачам. Іноді сюди додаються відомості про те, чого ПС робити не повинна.

При написанні ФВ необхідно враховувати, що чим більш докладними вони будуть, тим точнішу оцінку робіт за вартістю та термінами можна отримати перед розробленням технічного завдання (ТЗ) на створення ПЗ. Якщо на подальших етапах розроблення ПЗ не виникне доповнень до сформованих на початку ФВ, то ця оцінка залишиться досить точною. У той же час, при описі вимог не потрібно заглиблюватися в дуже дрібні деталі, наприклад, «для отримання результату користувач має натиснути з лівого краю другу зверху круглу жовту кнопку з написом «ОК». Такі деталі повинні бути опрацьовані вже в процесі розроблення ТЗ для команди програмістів.

ФВ документуються в специфікації вимог до ПЗ, де якомога повніше описується очікувана поведінка ПС.

Необхідно, щоб функціональна специфікація ПЗ була розроблена з математичною точністю. Вона повинна базуватися на чітких поняттях і твердженнях, бути однозначно зрозумілою розробникам і замовникам ПП. Щоб виріши-

ти дане завдання необхідно застосовувати математичні методи і формалізовані мови.

У загальному випадку функціональна специфікація має містити три основні частини:

1. Опис зовнішнього інформаційного середовища, з яким буде взаємодіяти розроблюване ПЗ. Повинні бути визначені всі використовувані канали введення і виведення даних і всі інформаційні об'єкти, до яких буде застосовуватися розроблюване ПЗ, а також зв'язки між цими інформаційними об'єктами.

2. Визначення основних функцій розроблюваного ПЗ, визначених на множині станів цього інформаційного середовища. Для всіх функцій визначається зміст, вводяться позначення, специфікуються вхідні дані і результати виконання, із зазначенням типів даних і завдань усіх обмежень, яким повинні задовольняти ці дані і результати.

3. Визначення та опис виняткових ситуацій та реакцій з боку ПЗ на них. Перераховуються всі випадки, коли ПЗ з тих чи інших причин не зможе коректно виконувати будь-яку зі своїх функцій. Реакція визначається для кожного такого випадку без виключень.

Розглянемо для прикладу ФВ до бібліотечної системи університету, яка призначена для замовлення документів і книг з інших бібліотек [1]:

1. Користувач повинен мати можливість проводити пошук книг і документів по всій множині доступних каталогів (їх БД) або за визначеними їх підмножинами.

2. Система повинна надавати користувачеві зручний засіб перегляду бібліотечних документів.

3. Кожне замовлення має бути забезпечене унікальним ідентифікатором, який копіюється в картку користувача для постійного зберігання.

Дані ФВ користувача визначають властивості, якими повинна володіти ПС. Зазвичай вони виділяються з документа, який містить основні вимоги користувача, і показують, що ФВ можуть бути описані з використанням різних рівнів деталізації (наприклад, перша і третя вимоги).

Розглянемо більш детально другу вимогу з наведеного вище прикладу. Дана вимога є суб'єктивною (розмитою), тому що має декілька можливих варіантів трактування фрази «зручний засіб перегляду документів». У наведеній вимозі мається на увазі, що ПС повинна надавати засоби для перегляду будь-яких документів, які представлені в широкому спектрі доступних форматів. Але оскільки ця умова описана не чітко, розробники можуть створити найпростіший засіб для перегляду текстових документів і наполягати на тому, що саме таке рішення є найбільш «зручним».

Багато з проблем, які виникають при розробці ПЗ, прямо пов'язані з неточністю специфікації вимог та з незрозумілими вимогами, які в ній наведені. Розробники завжди інтерпретують вимоги з двояким тлумаченням таким чином, щоб їм було легше реалізувати систему. Проте даний варіант може не збігатися з очікуваннями замовника. Виникнення подібних ситуацій призводить до розроблення нових вимог і внесення змін в ПС, що автоматично веде до затримки виконання та подорожчання системи.

Специфікація ФВ має бути комплексною (визначення та опис усіх системних сервісів) і несуперечливою (відсутність несумісних і взаємовиключних визначень сервісів). На практиці вкрай важко розробити таку специфікацію ФВ для великих і складних систем. Причина криється частково в складності самої ПС, а частково – у неузгоджених точках зору на те, що вона повинна робити. Ця неузгодженість може не виявитися на етапі первинного формулювання вимог – для її виявлення необхідний більш глибокий аналіз специфікацій. Коли неузгодженість системних функцій проявиться на будь-якому етапі ЖЦ ПС, у системну специфікацію доведеться внести відповідні зміни.

2.3.2 Нефункціональні вимоги

НВ визначають характеристики розроблюваного ПЗ, які проявляються в процесі його використання [21]. До таких характеристик належать:

- **правильність** – це функціонування у відповідності до ТЗ. Ця вимога є обов'язковою для будь-якого ПП, але ніяке тестування не дає абсолютної гарантії відсутності помилок у коді. У даному випадку мова йде

про певну ймовірність наявності помилок. Наприклад, імовірність збою системи управління ядерним реактором повинна бути близькою до нуля;

- **універсальність** – забезпечення належних функціонування (при будь-яких допустимих даних) і захисту від неправильних даних. Так само як в попередньому випадку, довести універсальність ПЗ неможливо, тому є сенс говорити про ступінь її універсальності;
- **надійність (завадостійкість)** – забезпечення правильності результатів (тобто їх повна повторюваність) при наявності різного роду збоїв. Джерелами завад можуть бути технічні та програмні засоби, а також люди, які працюють з цими засобами. Сьогодні існує достатня кількість способів уникнути втрат інформації при програмних збоях. Наприклад, можна зберігати проміжні результати, тобто використовувати так звані «контрольні точки». Це дозволить продовжити роботу з даними, записаними в останній контрольній точці, після збою ПЗ. Іншими способами зменшення кількості помилок є, наприклад, дублювання підсистем або введення надлишкової інформації;
- **можливість перевірки отриманих результатів**. Для цього необхідно документально фіксувати вихідні дані, встановлені режими і іншу інформацію, яка впливає на одержувані результати (особливо при надходженні сигналів безпосередньо від первинних перетворювачів);
- **точність результатів** – це ступінь наближення результатів роботи ПЗ до прийнятого в ТЗ значення. Величина отриманої похибки залежить від точності вихідних даних, ступеня адекватності використовуваної моделі, точності обраного методу і похибки виконання операцій комп'ютером. Жорсткі вимоги до точності пред'являють системи управління технологічними процесами і системи навігації (наприклад, система стикування космічних літальних апаратів);
- **захищеність** – ступінь конфіденційності інформації. Наприклад, до систем, які зберігають інформацію рівня держави, висуваються найбільш жорсткі вимоги по її захисту. Також високий рівень захищеності

забезпечується для зберігання комерційних таємниць. Для захисту інформації можуть використовуватися програмні, правові, криптографічні та інші методи;

- **програмна сумісність** – можливість спільного функціонування з іншим ПЗ. Найчастіше мова йде про функціонування програми під управлінням заданої операційної системи. Однак, може знадобитися обмін даними з деякою іншою програмою. У цьому випадку визначається конкретний формат переданих даних;
- **апаратна сумісність** – можливість спільного функціонування з деяким обладнанням. Ця вимога пов'язана з мінімально можливою конфігурацією обладнання, на якому буде працювати дане ПЗ. Інколи також виникає необхідність використання нестандартного (або невідомого на момент розроблення ПЗ) обладнання. У такому випадку чітко описуються інтерфейси, які відрізняються від стандартизованих;
- **ефективність** – це необхідність використання мінімальної кількості ресурсів технічних засобів (наприклад, обсягів оперативної та зовнішньої пам'яті, часу мікропроцесора, тощо). Для кожного використаного ресурсу також оцінюється ефективність. Це призводить до того, що дуже часто вимоги ефективності суперечать одна одній. Наприклад, наслідком зменшення обсягу використовуваної оперативної пам'яті буде збільшення часу виконання програми і навпаки;
- **здатність до адаптації** – можливість швидкого розширення функціональних можливостей ПЗ при зміні специфікації вимог до нього. Майже неможливо оцінити цю характеристику кількісно. Можна тільки констатувати, що при розробці даного ПЗ використовувалися прийоми, що полегшують його розширення;
- **рентабельність** – можливість одночасного використання декількома користувачами. Для задоволення цієї вимоги для кожного користувача може, наприклад, створюватись окрема копія даних.

На рис.2.11 показано три великі групи НВ [1]:

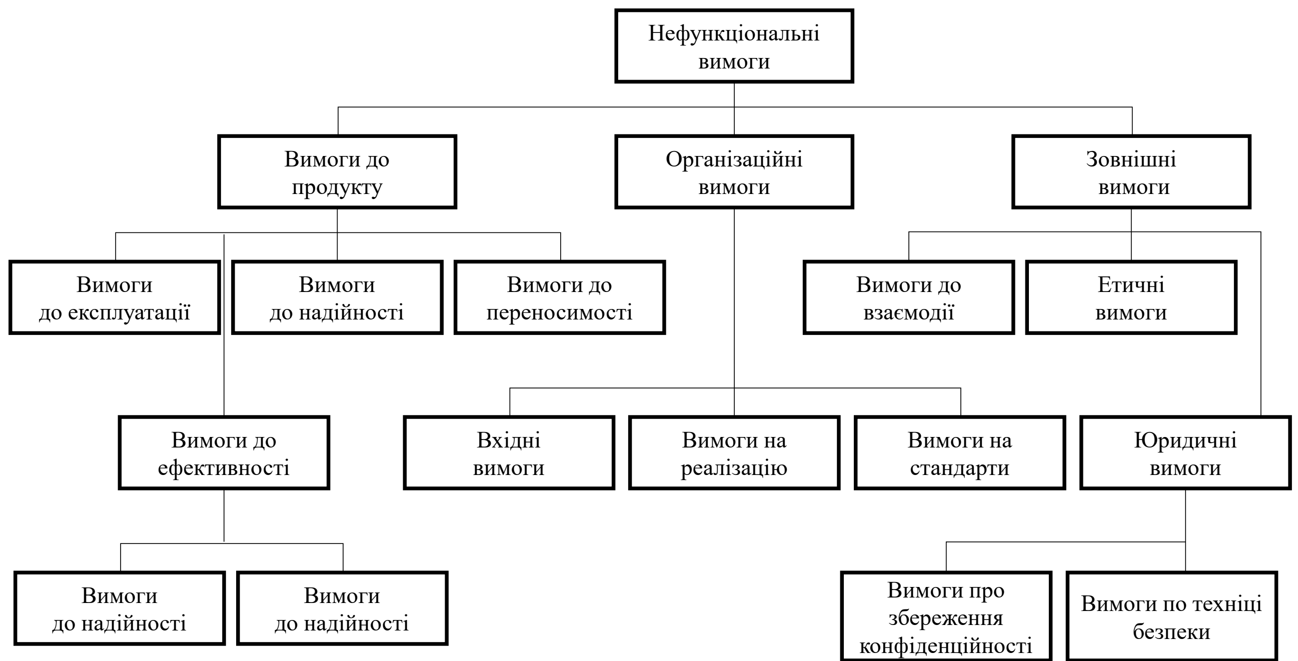

Рис. 2.11. Основні групи нефункціональних вимог

1. **Вимоги до продукту**. Характеризують експлуатаційні властивості ПП. До них можна віднести, наприклад, вимоги до необхідного обсягу пам'яті, продуктивності ПС, можливості її використання на інших платформах, надійності, зручності експлуатації, тощо. Наприклад, «Усі взаємодії між інтерфейсом APSE і кінцевим користувачем здійснюються за замовчуванням на основі стандартної множини символів мови програмування Ada».

2. **Організаційні вимоги**. Відображають організаційні процедури замовника та розробника ПЗ. Вони включають стандарти розроблення ПП, вимоги до його реалізації (мова програмування, методи проектування, тощо), вихідні вимоги (терміни створення та необхідна документація). Наприклад, «Основні процеси, які мають бути реалізованими в ПС, а також види додаткової супровідної документації описано в стандарті АБВ.Ко-Пр.ISO-2022».

3. **Зовнішні вимоги**. Враховують фактори, які є зовнішніми по відношенню до ПС і процесу її розроблення. Вони включають юридичні вимоги (система має функціонувати в рамках чинного законодавства країни, в якій використовується), вимоги щодо взаємодії з іншими системами, а також етичні вимоги (система має бути прийнятною для кінцевих користувачів та замовника). Наприклад, «ПС не повинна розкривати конфіденційну інформацію про її користува-

чів та замовника (крім їх імен), а також (за необхідності) телефонного номера операторів служби підтримки».

Проведемо аналіз наведених прикладів. Прив'язка вимог продукту до середовища програмування APSE та мови Ada обмежує свободу проектувальника ПС, тому що можна використовувати тільки визначений набір символів. Це означає, що дана вимога не просто вказує що саме треба зробити, але і як – явна ознака неправильно побудованої вимоги. Організаційні вимоги вказують на необхідність розроблення ПС у відповідності до внутрішнього стандарту компанії, який має код АБВ.Ко-Пр.ISO-2022. Внаслідок виконання зовнішніх вимог щодо необхідності дотримання законодавства в області збереження конфіденційності інформації оператори та кінцеві користувачі ПС не будуть мати прямого доступу до даних.

Основна проблема НВ полягає в тому, що вони часто є суб'єктивними і їх виконання важко перевірити. Через нечітке формулювання, наприклад, «зручність супроводу», «простота експлуатації», «можливість відновлення після збоїв (надійність)» реалізація подібних вимог може виявитися для системних розробників складною.

Розглянемо ще один приклад:

- *НВ, яку важко перевірити*: «ПС повинна бути простою в експлуатації для досвідченого оператора і зводити кількість його помилок до мінімуму».
- *НВ, яку можна перевірити*: «після двох годин навчання роботі з даною ПС досвідченому оператору мають бути доступними всі системні функції. Середнє число помилок оператора після навчання не повинно перевищувати двох за робочий день».

«Простота експлуатації» є суб'єктивним поняттям (залежить від точки зору кожної окремої людини), тому виконання такої вимоги не можна об'єктивно перевірити. У другому випадку НВ сформульовано так, що її можна було перевірити. Саме тому НВ мають виражатися через кількісні показники, досягнення яких можна однозначно визначити.

У табл. 2.4 наведені показники, за допомогою яких можна уточняти нефункціональні системні властивості.

Таблиця 2.4. Кількісні показники для нефункціональних вимог

Показник	Одиниці вимірювання
Швидкість	Час реакції на дії користувача; кількість виконаних операцій в секунду; час оновлення екрану.
Розмір	Кількість модулів пам’яті; одиниці вимірювання; кількість зовнішніх накопичувачів.
Простота експлуатації	Кількість статей в довідковій системі; час навчання персоналу;
Надійність	Ймовірність виходу системи з ладу; середнє значення часу між двома послідовними збоями у ПС; коефіцієнт готовності системи.
Стійкість до збоїв	Відсоток подій, які призводять до збоїв; час відновлення ПС після збою; ймовірність псування даних при збоях.

На практиці досить важко виразити НВ за допомогою кількісних показників. Замовник не завжди може оформити своє бачення майбутньої ПС у вигляді виражених кількісними показниками вимог, а деякі вимоги (наприклад, зручність супроводу) взагалі не можна виразити за допомогою кількісних показників. Витрати на об’єктивне вимірювання кількісних НВ можуть виявитися вкрай високими. Саме тому документ, який визначає вимоги до ПС, часто містить окрім чітко сформульованих вимог додатково опис системних цілей. Останні є корисними, оскільки відображають пріоритети замовника щодо майбутньої ПС. Проте варто не забувати (особливо замовнику), що системні цілі неможливо об’єктивно проконтролювати і вони можуть (і будуть) трактуватися різними способами.

НВ часто вступають в прямий конфлікт з іншими вимогами, які висуваються до ПС. Наприклад, ПС повинна повністю поміститися на постійний запам'ятовуючий пристрій ємністю 128 Мб, але інша вимога зобов'язує для написання системи використовувати мову програмування Java. При цьому, припустимо, що скомпільована програма займає більше 128 Мб, отже, виконати ці вимоги одночасно неможливо. У даній ситуації найкращим рішенням буде відмова від однієї з вимог: використати іншу мову програмування або збільшити обсяг пам'яті, яка виділяється для ПС.

Існує загальна рекомендація щодо створення специфікації вимог: ФВ та НВ мають бути відображені в різних розділах. На практиці цю умову виконати досить складно, оскільки у такому випадку буде важко простежити взаємозв'язки між ними. З іншого боку, якщо всі вимоги зведено до одного загального списку, складно провести окремо аналіз ФВ, НВ та визначити ті вимоги, які відносяться до ПС в цілому. Незважаючи на це, вимоги, які описують інтеграційні властивості ПС, мають бути обов'язково відокремлені, наприклад, у інший розділ (підрозділ, пункт або підпункт).

2.3.3 Вимоги предметної області

Даний тип вимог зазвичай не виділяють в окрему велику категорію. Це пов'язано з тим, що вони сильно інтегровані у ФВ та, хоч і значно менше, у НВ. Дана категорія вимог відображає умови експлуатації ПС, які висуваються замовником у вигляді нових ФВ, обмежень на вже сформульовані вимоги або у вигляді вказівок того, як саме ПС повинна виконувати обчислення. Через те, що ці вимоги є дуже важливими для предметної області (бізнесу), у межах якої буде використовуватися ПС, їх ще називають бізнес-правилами або бізнес-логікою. Наприклад, розглянута вище організаційна вимога, яка пов'язана з використанням внутрішнього стандарту (АБВ.Ко-Пр.ISO-2022) компанії, є бізнес-правилом, а реалізовані в ПС на її основі функціональні вимоги – бізнес-логікою. Невиконання даних вимог може призвести до виходу ПС з ладу або втрати її актуальності в межах поставлених задач.

Доповнимо розглянуті вище вимоги до бібліотечної системи декількома пунктами:

1. Стандартний інтерфейс користувача, який надає доступ до всіх бібліотечних БД, повинен ґрунтуватися на стандарті О17-60.

2. За бажанням користувача документи можуть бути роздрукованими на мережевому принтері. Деякі з цих документів мають відразу бути вилученими з системи для забезпечення виконання юридичних вимог щодо авторських прав.

Перша вимога є обмеженням системної ФВ, яка вказує на необхідність дотримання відповідного бібліотечного стандарту. Друга вимога є зовнішньою і направлена на виконання закону про авторське право, який застосовується до матеріалів бібліотеки. Ця вимога є коректною, тому що говорить, що деякі файли не можна зберігати, дозволено лише їх друк, але прямо не вказує як саме це повинен реалізувати програміст.

Розглянемо інший приклад бізнес-правил, пов'язаний з проведенням розрахунків. Наприклад гальмування потягу може обчислюватися за наступною формулою:

$$D_{потягу} = D_{градієнт} + D_{керування}.$$

Даний приклад бізнес-правила хоч і конкретизує як саме повинні виконуватися обчислення, але є обов'язковим для даної предметної області. Формулу взято зі специфікації системи автоматичного гальмування потягу, градієнт відомий для різних типів потягів, а ПС повинна автоматично зупиняти потяг на червоний сигнал семафора у спеціально відведеному для нього місці.

У даному прикладі використано специфічну термінологію, яка застосовується у відповідній предметній області. Для того, щоб в ній розібратися, менеджеру (або аналітику вимог) необхідні знання про характеристики систем управління поїздами та особливості їх функціонування.

Наведені приклади ілюструють основну проблему, яка пов'язана з бізнес-правилами (вимогами предметної області). Вимоги цього типу використовують термінологію та позначення, які властиві конкретній предметній області, що

ускладнює їх сприйняття командою розробників. Як правило, це призводить до незадовільного (з точки зору замовника) їх виконання.

Отже, формулювання вимог та створення їх детальної специфікації – досить відповідальне і складне завдання, яке вимагає від менеджера проведення перед проектних досліджень.

2.4 Методи визначення вимог

2.4.1 Інтерв’ювання

Одним з найбільш важливих і зрозумілих методів визначення вимог є інтерв’ю з клієнтом – метод, який можна використовувати практично в будь-якій ситуації [7].

Одне з основних завдань інтерв’ювання – зробити все можливе, щоб упередження і вподобання осіб, у яких беруть інтерв’ю, не вплинули на вільний обмін інформацією. Це складна задача через те, що неможливо сприймати навколишній світ, не фільтруючи його у відповідності зі своїм походженням і накопиченим досвідом.

Процес проведення інтерв’ю складається з наступних етапів [7]:

1) Розроблення питань (зразок питань для інтерв’ю представлений в табл. 2.5).

2) Вибір опитуваних користувачів. Існує декілька груп користувачів: персонал початкового рівня, організатори проекту середнього рівня, менеджери та інші замовники особливого роду – генеральні директори і віце-президенти, наукові співробітники, звичайні користувачі системи.

3) Планування контактів.

4) Проведення інтерв’ю. Інтерв’ю можна провести по телефону, персонально або за допомогою мережі Інтернет (відео конференція, чат, електронна пошта), проте кращим способом інтерв’ювання є безпосереднє спілкування.

5) Завершення зустрічі та визначення подальших дій.

Таблиця 2.5. Приклади питань для проведення інтерв'ю

Частина I. Визначення профілю замовника або користувача
Ваше ім'я? Ваша компанія? Галузь, у якій Ви працюєте? Ваша посада? (*Зауваження*. Вищенаведена інформація, як правило, може бути внесена заздалегідь) Які Ваші основні обов'язки? Що Ви в основному виготовляєте? Для кого? Як вимірюється успіх Вашої діяльності? Які проблеми впливають на успішність Вашої діяльності? Які тенденції, якщо такі існують, роблять Вашу роботу простішою або складнішою?
Частина II. Оцінювання проблеми
Для яких проблем (прикладного типу) Ви відчуваєте брак хороших рішень? Назвіть їх. (*Зауваження*. Не забувайте питати: «А ще?») Для кожної проблеми з'ясовуйте наступне: Чому існує ця проблема? Як вона вирішується в даний час? Як замовник (користувач) хотів би її вирішувати?
Частина III. Розуміння середовища користувача
Яке коло користувачів? Яка у них освіта? Які їх навички в комп'ютерній області? Чи мають користувачі досвід роботи з подібним типом додатків? Яка платформа використовується? Які у Вас плани щодо майбутніх платформ? Чи використовуються спеціальні програми, які мають відношення до даного додатку? Якщо так, то про них необхідна додаткова інформація. Які очікування замовника щодо практичності продукту? Скільки часу необхідно для навчання роботі з продуктом?

Продовження таблиці 2.5

В якому вигляді повинна бути представлена довідкова інформація для користувача (наприклад, в інтерактивному або у вигляді друкованої копії)?
Частина IV. Резюме (перераховуються основні пункти, щоб перевірити, чи все правильно зрозуміли обидві сторони) Отже, Ви сказали мені… (перерахуйте описані замовником проблеми своїми словами) Чи адекватно цей список описує коло проблем, які є в існуючих рішеннях? Які ще проблеми (якщо такі існують) Ви можете назвати/згадати?
Частина V. Припущення, аналітика щодо проблеми замовника (перевірені або неперевірені припущення) Які проблеми, якщо вони є, пов’язані з ..? (*аналітик перераховує всі потреби або додаткові проблеми, які, на його думку, може відчувати замовник або користувач*) Для кожної із зазначених проблем з’ясовується наступне: Чи є вона реальною? Які її причини? Як вона вирішується в даний час? Як би замовник (користувач) хотів її вирішити? Наскільки важливим для замовника (користувача) є рішення цієї проблеми в порівнянні з іншими, згаданими?
Частина VI. Оцінювання запропонованого аналітиком рішення (якщо це доречно) Аналітик характеризує основні можливості запропонованого ним рішення та задає користувачеві наступні питання: Чи доречним буде запропоноване рішення проблеми? Як Ви розцінюєте важливість цього?
Частина VII. Оцінювання можливостей Хто в компанії має потребу у цьому додатку? Скільки користувачів зазначених типів буде використовувати його? Наскільки значимо для Вас успішне вирішення поставлених завдань?
Частина VIII. Оцінювання необхідного рівня надійності і продуктивності, а також потреби в супроводі

Продовження таблиці 2.5

Які Ваші очікування щодо надійності ПЗ? Якою, на Вашу думку, повинна бути продуктивність? Чи буде Ваша компанія займатися підтримкою продукту або цим буде займатися хтось інший? Чи відчуваєте Ви потребу в підтримці ПЗ? Що Ви думаєте про доступ для супроводу і обслуговування? Які вимоги щодо безпеки? Які вимоги щодо установки і конфігурації? Чи існують спеціальні вимоги по ліцензуванню?
Частина IX. Інші вимоги Чи існують законодавчі вимоги, вимоги інформаційного середовища, інструкції або інші стандарти, яких необхідно дотримуватися? Чи немає інших вимог, про які необхідно знати?
Частина X. Закінчення Чи існують інші питання, які мені треба було б Вам задати? Якщо мені ще знадобиться задати Вам декілька питань, чи можу я Вам зателефонувати? Чи будете Ви брати участь у обговоренні вимог?
Частина XI. Висновок аналітика Після інтерв'ю, поки його дані ще свіжі в пам'яті аналітика, він має письмово зафіксувати основні виявлені під час бесіди з даним замовником (користувачем) потреби або проблеми з найвищими пріоритетами.

4.4.2 Мозковий штурм і відбір ідей

Мозковий штурм – це метод проведення зборів, при якому група людей намагається знайти вирішення специфічної проблеми за допомогою накопичення всіх спонтанних ідей.

Даний процес має ряд очевидних переваг [7]:

- підтримує участь всіх присутніх;
- дозволяє учасникам розвивати ідеї один одного;
- ведучий або секретар веде запис всього ходу обговорення;

- його можна застосовувати при різних обставинах;
- як правило, у результаті отримується множина можливих рішень для будь-якої поставленої проблеми;
- метод сприяє вільному мисленню, не обмеженому звичайними рамками.

Мозковий штурм складається з двох фаз: генерація ідей і їх відбір. Основна мета на етапі генерації полягає в тому, щоб описати як можна більше ідей, не обов'язково глибоких. На етапі відбору головним завданням є аналіз усіх ідей, які виникли. Відбір ідей включає в себе відсікання, організацію, упорядкування, розвиток, угруповання, уточнення тощо.

Генерація ідей під час мозкового штурму може проходити різними способами. Наприклад, усі основні учасники збираються в одній кімнаті, їм роздають матеріали для нотаток. Це може бути просто стопка паперу і чорний маркер. Аркуші паперу повинні бути не менше 9×9 см і не більше 12×17 см [7]. Кожному учаснику потрібно видати не менше 25 листів на кожен сеанс мозкового штурму.

Кожен учасник сеансу «мозкового штурму» виконує одну з трьох ролей: лідер, секретар або член команди. Лідер відповідає за спрямовування процесу в правильне русло, за порядок і допомагає секретарю робити записи. Секретар записує всі ідеї таким чином, щоб кожна людина, що знаходиться в кімнаті, могла бачити ці записи. Учасники генерують ідеї.

Ведучий пояснює правила проведення мозкового штурму, після чого визначається мета засідання.

Основні правила проведення мозкового штурму наступні [7]:

1) Не допускається критика або дебати.

2) Свобода фантазії.

3) Генеруйте якомога більше ідей.

4) Переробляйте і комбінуйте ідеї.

У міру створення ідей секретар просто збирає їх і прикріплює на стіну кімнати засідань. Більшість засідань по генерації ідей триває близько години (іноді 2-3 години).

Після завершення фази генерації ідей починається процес відбору, який складається з декількох етапів [7]:

1) Відсікання. Полягає у відсіканні лідером тих ідей, які не заслуговують на увагу. Якщо буде виявлено дві однакові ідеї, вони об'єднуються.

2) Групування ідей. Групи отримують назви в залежності від того, за яким принципом здійснюється групування. Наприклад, групи можуть мати такі назви: нові функції, питання продуктивності, пропозиції щодо вдосконалення існуючих функцій, інтерфейс користувача і питання простоти звернення, тощо. Для будь-якої з цих груп можна відновити генерацію ідей, якщо виявиться, що процес групування стимулював виникнення нових ідей або деяка область важливих функціональних можливостей залишилася неохопленою.

3) Визначення функцій. Ведучий перераховує всі ідеї, які залишилися, і просить авторів дати їх опис, що складається з одного речення (табл. 2.6).

4) Розстановка пріоритетів.

Таблиця 2.6. Приклад визначення функцій

Область застосування додатку	Аналізована функція	Опис функції
Автоматизація домашнього освітлення	Автоматичне завдання освітлення	Домовласник може попередньо задавати послідовності виникнення певних освітлювальних подій в залежності від часу дня та ночі
Система введення замовлень на покупку	Швидкість	Швидкість відповіді має бути достатньою, щоб не заважати проведенню інших операцій
Система виявлення неполадок	Автоматичне повідомлення	Усіх зареєстрованих користувачів відповідних груп буде повідомлено за допомогою електронної пошти, коли що-небудь зміниться

2.4.3 Спільне розроблення додатків

Метод спільного розроблення додатків (*англ. Joint Application Design, JAD*) – це зареєстрований товарний знак компанії IBM. Він являє собою груповий підхід до визначення вимог, при реалізації якого особлива увага приділяється удосконаленню групового процесу і правильному підбору людей, яких залучено до роботи над проектом.

Сеанси JAD аналогічні до сеансів мозкового штурму, однак не у всьому. Сеанси мозкового штурму тривають близько двох годин, а сеанси JAD – до трьох днів. На сеансах мозкового штурму відбувається швидке генерування ідей, а на сеансах JAD – високорівневих специфічних програмних моделей функцій, даних і ліній поведінки.

Сеанс JAD має певну структуру, на ньому дотримуються певної дисципліни, і він проходить під керівництвом арбітра. У його основі лежить обмін інформацією з використанням документації, фіксованих вимог і правил роботи. З моменту появи методики JAD на сеансах використовуються CASE-інструменти та інші програмні засоби, призначені для побудови діаграм потоку даних (*англ. Data flow diagram, DFD*), діаграм взаємозв'язків між сутностями (*англ. Entity relationship diagrams, ERD*), діаграм зміни станів та інші об'єктно-орієнтовані діаграми.

У сеансах JAD є наступні ролі [7]:

1) Розробники. У задачі розробника входить надання допомоги організаторам у формулюванні їх потреб, які зазвичай є рішенням існуючих проблем. Таким чином, визначення вимог до ПЗ відбувається спільно з організаторами проекту.

2) Учасники. Для успішного проведення сеансу ключовою вимогою є високий рівень кваліфікації запрошених учасників. Правильний підбір людей дозволить швидко приймати рішення і розробляти правильні моделі, навіть якщо вони не будуть завершені.

3) Арбітр/консультант. Арбітр повинен зводити до мінімуму прояв непродуктивних людських емоцій, на зразок агресивності або самозахисту. Арбітр не

володіє ні процесом розроблення, ні ПП. Він присутній тільки для того, щоб допомагати організаторам проекту розробляти ПП.

4) Секретар. Секретар сеансу JAD документує ідеї і допомагає стежити за часом.

Сеанс JAD – це своєрідна майстерня, що працює в максимально напруженому режимі, де рішення приймаються спільно всіма учасниками [7]. При цьому учасники є великими фахівцями в розглянутому питанні.

Процес дослідження проекту розбивається на наступні етапи:

- пошук фактів і збір інформації;
- підготовка сеансу JAD;
- проведення сеансу;
- перевірка зібраної інформації.

Результатами проведення сеансу JAD можуть бути [7]:

- діаграма контексту даних;
- діаграма потоку даних першого рівня;
- глобальна модель даних – діаграма взаємозв’язків між сутностями;
- перелік первинних об’єктів;
- об’єктна модель високого рівня;
- обов’язки кандидатів і співробітників для кожного об’єкта;
- перелік первинних процесів/сценарії вибору;
- інші діаграми потоку даних, діаграми стану, дерева альтернатив, таблиці рішень, тощо;
- вимоги, що пред’являються до даних для кожного процесу;
- перелік припущень;
- документація з аналізу або з призначенням відкритих питань.

Результати сеансу JAD використовуються в процесі визначення вимог для організації наступного етапу – створення специфікації вимог (SRS).

Проте у методі JAD є свої недоліки [7]:

1) Учасники передають свої ідеї арбітру та/або секретарю. Для уникнення невірної інтерпретації зібраних даних, необхідно прийняти деякі запобіжні за-

ходи. Використання під час сеансу автоматизованих інструментальних засобів і перевірка результатів усіма учасниками зменшить ризик.

2) На сеансах JAD розглядаються переважно інформаційні системи, в яких особлива увага приділяється елементам даних і проекту інтерфейсу. Є мало інформації про використання методу JAD для визначення вимог, що пред'являються до систем реального часу.

3) На проведення триденного сеансу JAD з представниками всіх груп організаторів проекту, кожна з яких складається з кваліфікованих фахівців, витрачається чимало коштів. Три дні – це середня тривалість. Для аналізу складних систем реального часу і систем, від яких залежить людське життя, часто потрібно більше. Якщо сеанс триває «до тих пір, поки не будемо втомлюватися», то втома може настати вже в той момент, коли будуть визначені тільки сценарії вибору.

2.4.4 Розкадрування

Метою процесу розкадрування є отримання ранньої реакції користувачів на запропоновані концепції ПЗ, ще до того, як концепції будуть перетворені в код, а в багатьох випадках навіть до розроблення докладної специфікації.

Розкадрування має наступні переваги:

- максимально недороге;
- дружнє користувачу, неформальне та інтерактивне;
- забезпечує ранній аналіз призначених для користувача інтерфейсів ПС;
- легко створити і модифікувати.

Розкадрування можна використовувати для прискорення концептуальної розроблення різних граней додатків. Їх можна застосовувати для розуміння візуалізації даних, визначення та розуміння бізнес-правил, які будуть реалізовані в новому бізнес-додатку, для визначення алгоритмів та інших математичних конструкцій, які будуть виконуватися всередині систем, для демонстрації звітів та інших результатів на ранніх етапах. Розкадрування можна використовувати

практично для всіх додатків, в яких раннє отримання реакції користувачів є ключовим фактором успіху.

Результати процесу розкадрування називаються розкадровками, які діляться на три типи залежно від режиму взаємодії з користувачем [7]:

- пасивні;
- активні;
- інтерактивні.

Пасивні – це історії, які розповідають користувачу. Вони можуть складатися зі схем, картинок, скріншотів екрану, презентацій PowerPoint або зразків вихідної інформації системи. При пасивному розкадруванні аналітик грає роль системи і просто проводить користувачів по розкадровках, пояснюючи наступним чином: «Коли ви робите це, відбувається ось це…».

Активні розкадровки забезпечують автоматизований опис поведінки системи використанні її в типовому або операційному сценарії. Вони створюються за допомогою анімації або автоматизації, наприклад, за допомогою автоматичного послідовного показу слайдів, анімаційних засобів або навіть фільму.

Інтерактивні розкадровки дають користувачу досвід роботи з ПС майже настільки ж реальний, як на практиці. При такому варіанті розкадровки вимагають взаємодії з користувачем. Інтерактивні розкадровки можуть бути імітаційними, у вигляді макетів або коду, який в подальшому можуть відкинути. Складні інтерактивні розкадровки, засновані на такому коді, можуть бути дуже схожими на прототип ПП.

З рис. 2.12 видно, що, ці три типи розкадрувань пропонують широкий спектр можливостей – від зразків вихідної інформації до «живих» демонстраційних версій. Різниця між складними розкадровками та ранніми прототипами ПП досить умовна.

Вибір типу розкадровок залежить від складності ПС і того, наскільки великим є ризик того, що команда неправильно розуміє її призначення. Для безпрецедентної нової системи, що має розпливчасте визначення, може знадобитися декілька розкадровок (від пасивних до інтерактивних) у міру того, як команда удосконалює своє розуміння ПС.

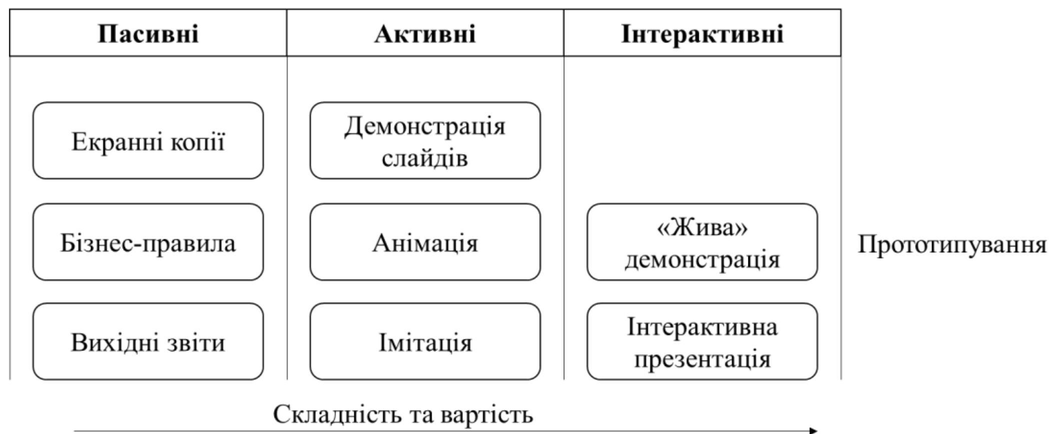

Рис. 2.12. Типи розкадровок

2.4.5 Обігравання ролей

Метод обігравання ролей дозволяє команді розробників відчути світ користувача, побувавши в його ролі. Концепція, що лежить в основі даного методу, досить проста [7]: хоча, спостерігаючи і задаючи питання, підвищується рівень розуміння, наївно вважати, що за допомогою одного спостереження розробник/аналітик може отримати істинно глибоке розуміння розв'язуваної проблеми або вимог до тієї системи, яка покликана дану проблему вирішити.

Суть методу обігравання ролей полягає в тому, що аналітик або будь-який інший член команди займає місце користувача і виконує його звичайні дії. Розглянемо, наприклад, проблему введення замовлень на покупку. Розробник/аналітик може «відчути» проблеми і неточності, притаманні існуючій системі введення замовлень на покупку, просто зайнявши місце оператора і спробувавши ввести декілька замовлень. Отриманий протягом певного часу досвід назавжди змінить розуміння командою суті проблеми.

Існує два різновиди методу обігравання ролей [7]:

- сценарний перегляд;
- CRC-картки (*англ. Class - Responsibility - Collaboration, укр. Клас - Обов'язок - Взаємодія*).

Сценарний перегляд – це виконання ролі на папері. У даному випадку кожен учасник слідує за сценарієм, який задає конкретну роль в «п'єсі». Перегляд демонструватиме будь-які неточності в розумінні ролей, брак інформації,

яка доступна актору чи підсистемі, або брак конкретного опису поведінки, який необхідний акторам для успішного виконання їх ролей.

Однією з переваг сценарного перегляду є те, що сценарій можна модифікувати і програвати знову стільки разів, скільки необхідно, поки актори не вважатимуть його правильним. Сценарій можна також повторно використовувати для навчання нових членів команди. Його можна модифікувати і програвати знову, коли необхідно змінити поведінку системи. У певному сенсі цей сценарій стає «живою» розкадровкою для проекту.

CRC-картки часто застосовуються в об'єктно-орієнтованому аналізі. У даному випадку кожному учаснику видається набір індексних карток, які описують клас (об'єкти певного типу), обов'язки (поведінку), а також взаємодії (з якими сутностями взаємодіє об'єкт). Ці взаємодії можуть просто представляти сутності проблемної області (наприклад, користувачі, натиснуті кнопки, лампи і підйомники) або об'єкти, які існують в області рішення (підсвічування вимикача в холі, вікно багатодокументного інтерфейсу або кабіна ліфта).

Коли актор-ініціатор ініціює певну поведінку, усі учасники дотримуються поведінки, яка задана на їх картках. Якщо процес переривається через брак інформації або якщо одній сутності необхідно переговорити з іншою, а взаємодія не визначена, то картки модифікуються і ролі розігруються знову.

Нижче пропонується зразок програвання одного з можливих варіантів використання [7]:

1) Управління включенням. Мій домовласник тільки що натиснув кнопку, яка керує набором ламп. Він все ще утримує кнопку в натиснутому стані. Я відправив Центральному блоку управління повідомлення у той момент, як тільки кнопка була натиснута, і збираюся надсилати йому повідомлення кожну секунду, поки кнопка натиснута.

2) Центральний блок керування. Коли я отримав перше повідомлення, то змінив стан виходу з «Викл.» на «Вкл.». Коли я отримав друге повідомлення, стало очевидно, що домовласник хоче змінити яскравість освітлення, тому при отриманні кожного повідомлення я збираюся змінювати яскравість на 10%.

3) Лампа. Я апаратно запрограмована на змінне накалювання. Я змінюю яскравість світла під час нашої розмови.

2.4.6 Швидке прототипування

Швидке прототипування – це часткова реалізація ПЗ, яка створена з метою допомогти розробникам, користувачам і замовникам краще зрозуміти вимоги до ПС.

Чим більш детальним є прототип, тим легше зрозуміти вимоги, які висуваються замовником. З іншого боку, будь-який прототип є окремою програмою. Це означає, що чим детальнішим є прототип, тим він дорожчий. З табл. 2.7 можна зрозуміти, що відносно недорогий прототип, який при цьому має велику цінність, необхідно створювати практично завжди. Велика цінність означає, що побудова прототипу дасть змогу замовнику краще зрозуміти, який саме ПП буде створено, а програмістам допоможе більш детально зрозуміти, які функціональні можливості треба розробити.

Таблиця 2.7. Груба оцінка вигоди від створення прототипу

	Очікувана цінність прототипу	
	Низька	Висока
Низька вартість прототипу	Можливо	Так
Висока вартість прототипу	Ні	Можливо

Існує багато способів розбиття прототипів на категорії. Яким повинен бути прототип у кожному конкретному випадку, залежить від того, яку проблему необхідно вирішити шляхом його побудови. Наприклад, можна виділити наступні прототипи:

- які відкидаються;
- які еволюціонують;
- операційні;
- вертикальні і горизонтальні;
- інтерфейсу користувача;

- алгоритмічні;
- тощо.

Питання прототипування та швидкого прототипування більш детально будуть розглянуті в наступних підрозділах.

2.5 Формалізація вимог

Традиційні технічні дисципліни, такі як цивільне будівництво або електротехніка, зазвичай легко адаптують найкращі математичні методи. Наприклад, застосування методів математичного аналізу у машинобудуванні не викликає труднощів. Однак, технології розроблення ПЗ не йдуть таким шляхом. Хоча у процесі створення ПЗ вже понад 25 років використовуються дослідження з використання математичних методів, їх вплив все ж обмежений. Так звані формальні методи розроблення ПС не використовуються. Багато компаній, які розробляють ПЗ, не вважають економічно вигідним застосування цих методів у процесі розроблення.

Термін «формальні методи» передбачає низку операцій, до складу яких входять створення формальної специфікації ПС, аналіз та вдосконалення специфікації, реалізація системи на основі перетворення формальної специфікації в ПЗ та верифікація програм. Усі ці дії залежать від формальної специфікації програмного забезпечення.

Формальна специфікація (ФС) – це системна специфікація, яка написана мовою, словник, синтаксис та семантику якої визначено формально. Необхідність формального визначення мови передбачає, що ця мова ґрунтується на математичних концепціях. Тут використовується область математики, яка називається дискретною математикою, в основі якої лежать алгебра, теорія множин та алгебра логіки.

У 1980-х роках багато дослідників вважали, що формальні специфікації та формальні методи є найбільш ефективним шляхом покращення якості ПЗ. Вони привели вагомі докази на користь того, що суворий та детальний аналіз, який є невід’ємною частиною формальних методів, призведе до створення ПЗ з малою

кількістю помилок. Вони передбачали, що до XXI століття більшість ПЗ розроблятиметься з використанням формальних методів.

Ці передбачення не справдилися з цілого ряду причин:

1) Успіхи технологій розроблення ПЗ. Використання в процесі проектування та розроблення ПС таких технологій інженерії програмного забезпечення, як структурні методи, управління конфігурацією, приховування інформації, тощо призвело до підвищення якості ПП. Це суперечить уявленням, які існували, що підвищення якості програм може бути досягнуто тільки шляхом доведення їх правильності.

2) Зміни на ринку ПП. У 1980-х роках якість була ключовою проблемою створення ПЗ. Нині головним критерієм розроблення багатьох класів ПЗ є не якість, а час постачання його на ринок. ПЗ має бути розроблене швидко, і клієнти готові прийняти його з деякими дефектами, якщо буде забезпечено швидке постачання. Методи швидкого розроблення не дуже добре узгоджуються з формальною специфікацією. Якість, без сумніву, є важливим показником, але вона повинна досягатися в контексті швидкого постачання на ринок.

3) Обмежена сфера застосування формальних методів. Формальні методи зазвичай погано підходять для опису взаємодій користувача та визначення інтерфейсів користувача. Інтерфейси користувача є складовою більшості сучасних систем, але користь від застосування формальних методів при цьому є обмеженою.

4) Обмежена масштабованість формальних методів. В успішних проектах формальні методи зазвичай використовувалися лише для розроблення невеликого критичного ядра системи. Проблема посилювалася нестачею засобів підтримки таких методів.

Ці фактори призвели до того, що ризики застосування формальних методів у більшості проектів перевищують можливі вигоди від їх використання. Витрати та проблеми введення формальних методів у процес розроблення дуже високі. Однак формальна специфікація є відмінним способом виявлення помилок у вимогах та засобом, який забезпечує однозначність системної специфіка-

ції. У всіх успішних проектах, які використовували формальні методи, повідомлялося про зменшення кількості помилок у готових ПС.

Таким чином, у системах, в яких можливе застосування формальних методів, воно може бути виправданим та навіть рентабельним. Використання формальних методів зростає у галузі розроблення критичних систем, де дуже важливі такі властивості, як: безпека, безвідмовність, захищеність. Для таких систем атестація потребує великих витрат, а вартість відмов дуже велика. І тут рентабельно використовувати формальні методи, оскільки вони можуть зменшити ці витрати.

Прикладами критичних систем, для розроблення яких успішно застосовувалися формальні методи, є системи сигналізації на залізниці, інформаційні системи управління повітряним транспортом, медичні системи управління і бортові системи космічних кораблів. Вони також були використані для специфікації пакетів програмних засобів, частини системи CICS (абонентська інформаційно-керуюча система) компанії IBM та ядра систем, які працюють у режимі реального часу.

На сьогоднішній день визначено три рівні специфікації ПЗ. Це вимоги системні та користувача, а також специфікація структури ПС. Вимоги користувача є найбільш узагальненими, а специфікація структури найбільш детальна. Формальні математичні специфікації знаходяться десь між системними вимогами та специфікацією структури. Вони не містять деталей реалізації ПС, але мають представляти її повну математичну модель.

У міру розроблення специфікації участь замовника зменшується, а участь підрядників і розробників ПЗ зростає. На ранніх стадіях розроблення специфікація має бути орієнтованою на замовника і написаною так, щоб він міг її зрозуміти. Однак на заключній стадії процесу розроблення повинна бути отримана специфікація, яка в основному призначена для підрядників та розробників, оскільки вона буде основою для реалізації ПС. Ця кінцева специфікація може бути формальною.

На рис. 2.13 показано етапи розроблення специфікації ПЗ та їх взаємозв'язок з процесом проектування. Етапи розроблення специфікації, які зо-

бражено на рис. 2.13, не є незалежними та не обов’язково розробляються у наведеній послідовності. На рис. 2.14 показано, що розроблення специфікації та проектування можуть виконуватися паралельно, коли інформація від етапів розроблення специфікації передається до етапів проектування та навпаки.

Рис. 2.13. Етапи розроблення специфікації

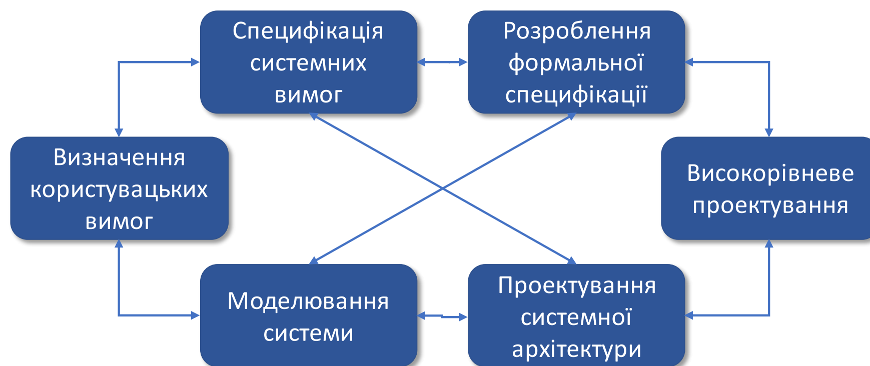

Рис. 2.14. Розроблення формальної специфікації

Створення ФС потребує детального аналізу системи, що дозволяє виявити помилки та невідповідності у специфікації неформальних вимог. Ця можливість виявлення помилок – найважливіший аргумент для використання ФС. Проблеми у вимогах, які залишаються невиявленими до останніх стадій процесу розроблення ПЗ, зазвичай вимагають великих витрат на внесення правок.

Розроблення та аналіз ФС потребують додаткових витрат. На рис. 2.15 показано вартість створення ПЗ при розробленні ФС та без неї. При звичайному розробленні ПЗ вартість атестації системи становить близько 50% всієї вартості, а вартість проектування та реалізації системи вдвічі більша за вартість розроблення специфікації. При використанні ФС вартості розроблення специфіка-

ції та реалізації системи можна вважати рівними, а вартість атестації значно знижується, оскільки в процесі розроблення ФС виявляються і усуваються недоліки у вимогах. Тим самим виключається необхідність перероблення ПС на останніх стадіях її створення.

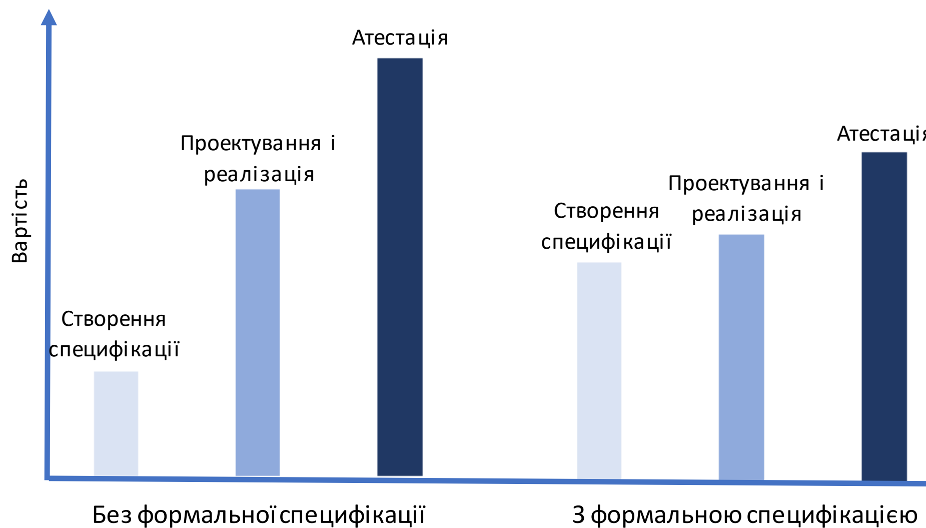

Рис. 2.15. Вартість розроблення ПЗ з формальною специфікацією

Існує два основних підходи до розроблення ФС, які використовуються для написання деталізованих специфікацій нетривіальних ПС:

1) Алгебраїчний підхід, у якому система описується у термінах операцій та їх відношень.

2) Заснований на моделях підхід, при якому модель ПС будується з використанням математичних конструкцій, таких як множини та послідовності, а системні операції визначаються тим, як вони змінюють стан системи.

Для розроблення ФС послідовних та паралельних систем на даний час створено декілька мов, які подані у табл. 2.8.

Таблиця 2.8. Мови розроблення формальних специфікацій

Тип мови	Послідовні системи	Паралельні системи
Алгебраїчний	Larch OBJ	Lotos

Продовження таблиці 2.8

Заснований на моделях	Z VDM B	CSP Мережі Петрі

2.6 Створення прототипу програмного забезпечення

Замовникам ПЗ та кінцевим користувачам зазвичай складно чітко сформулювати вимоги до ПС, яка розробляється. Важко передбачити, як система впливатиме на трудовий процес, як вона взаємодіятиме з іншими системами і які операції, що виконуються користувачами, необхідно автоматизувати. Ретельний аналіз вимог допомагає зменшити невизначеність щодо того, що ПС має робити. Проте реально перевірити вимоги, перш ніж їх затвердити, практично неможливо. У цій ситуації може допомогти макет або прототип системи.

Прототип – це початкова версія ПС, яка використовується для демонстрації концепцій, закладених у системі, перевірки варіантів вимог, а також пошуку проблем, які можуть виникнути як під час розроблення, так і під час експлуатації системи та можливих варіантів їх вирішення [1]. Швидке розроблення прототипу системи є дуже важливим, щоб користувачі могли почати експериментувати з ним якомога раніше.

Прототип ПЗ допомагає на двох етапах процесу розроблення системних вимог [1]:

1) Постановка вимог. Користувачі можуть експериментувати із системними прототипами, що дозволяє їм перевіряти, як працюватиме система. Користувачі отримують нові ідеї для встановлення вимог, можуть визначити сильні та слабкі сторони ПЗ. Унаслідок цього можуть сформуватися нові вимоги.

2) Перевірка існуючих вимог. Прототип дозволяє виявити помилки та упущення в раніше визначених вимогах. Наприклад, системні функції, визначені у вимогах, можуть бути корисними та потрібними (з точки зору користувача). Однак у процесі застосування цих функцій спільно з іншими функціями користувачі можуть змінити початкову думку про них. Унаслідок цього вимоги

до ПС зміняться, відображаючи змінене розуміння користувачами системних функцій.

Прототипування можна використовувати під час аналізу ризиків і на початковому етапі розроблення планів управління програмним проектом. Основною небезпекою при розробці ПЗ є помилки та упущення вимог. Витрати на усунення помилок у вимогах на більш пізніх стадіях процесу розроблення можуть бути дуже високими. Експерименти показують, що прототипування зменшує кількість проблем, пов’язаних із розробленням вимог, а також зменшує загальну вартість розроблення системи. З цих причин воно часто використовується у процесі розроблення вимог.

Проте різниця між прототипуванням, як окремим етапом процесу розроблення ПЗ, і розробленням основної ПС неочевидна. У даний час багато систем розробляються з використанням еволюційного підходу, коли швидко створюється початкова версія системи, яка потім поступово змінюється до остаточного варіанта. При цьому часто використовуються методи швидкого (гнучкого) розроблення, які можна використовувати при створенні прототипів.

Поряд з тим, що прототипи допомагають формувати вимоги, вони мають ряд інших переваг [1]:

1) Різне тлумачення вимог розробниками ПЗ та користувачами можна виявити при демонстрації діючого прототипу системи.

2) У процесі створення прототипу розробники можуть виявити неповні чи неузгоджені вимоги.

3) Працюючи, хоч і обмежено, у вигляді прототипу, ПС може продемонструвати свої слабкі та сильні сторони.

4) Прототип може бути основою написання специфікації високоякісної системи. Розроблення прототипу зазвичай веде до покращення специфікації системи.

Працюючий прототип може також використовуватися для інших цілей, наприклад [1]:

1) Навчання користувача. Прототип ПС можна використовувати для навчання персоналу перед поставкою остаточного варіанта системи.

2) Тестування ПС. Прототипи дозволяють моделювати тести. Один і той же тест запускається на прототипі та на системі. Якщо виходять однакові результати, це означає, що тест не виявив дефектів у системі. Якщо результати відрізняються, необхідно досліджувати причини відмінностей, що дозволяє виявити можливі помилки у системі.

На основі вивчення різних програмних проектів розробниками зроблено висновок, що ефективність застосування прототипів при розробці ПЗ полягає в наступному [1]:

1) Поліпшуються експлуатаційні якості системи.

2) Системна архітектура стає досконалішою.

3) Система більше відповідає потребам користувачів.

4) Скорочуються витрати на розроблення системи.

5) Супровід системи спрощується і стає зручнішим.

Ці дослідження показують, що покращення експлуатаційних якостей та збільшення відповідності ПС потребам користувача не вимагають збільшення загальної вартості розробки. Прототипування зазвичай підвищує вартість початкових етапів розроблення ПЗ, але знижує витрати більш пізніх етапах.

Модель процесу розроблення прототипу показано на рис. 2.16. На першому етапі даного процесу визначаються призначення прототипу та мета прототипування. Метою може бути розроблення макету інтерфейсу користувача, перевірка ФВ або демонстрація реалізованості системи для керівництва. Один і той самий прототип не може використовуватися для досягнення одночасно всіх цілей. Якщо цілі визначені неточно, функції прототипу можуть бути сприйняті неправильно.

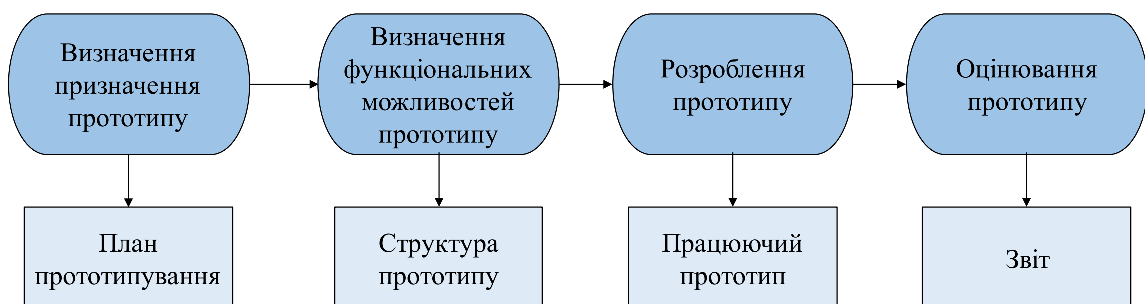

Рис. 2.16. Процес розроблення прототипу

На наступному етапі процесу розроблення прототипу визначаються його функціональні можливості, тобто приймається рішення про те, які властивості системи повинен відображати прототип, а які (що, можливо, важливіше) – ні. Для зменшення витрат на створення прототипу можна виключити деякі системні функції. Наприклад, можна послабити характеристики швидкодії та вимоги до використання пам’яті. Засоби управління та обробки помилок можуть ігноруватися або бути елементарними, якщо, звичайно, метою прототипування не є модель користувача. Також можуть бути знижені вимоги до надійності та якості ПС.

Завершальний етап процесу прототипування – оцінювання створеного прототипу. Є думка, що це найважливіший етап процесу прототипування. Тут перевіряється, наскільки створений прототип відповідає своїм призначенню та цілям. Також на його основі створюється план заходів щодо вдосконалення розроблюваної ПС.

2.6.1 Прототипування у процесі розроблення програмного забезпечення

Як уже зазначалося вище, кінцевим користувачам важко уявити, як вони використовуватимуть нову ПС у повсякденній роботі. Якщо система велика і складна, це неможливо зробити до створення та введення її в експлуатацію.

Один із способів подолання цієї проблеми полягає у використанні еволюційної стратегії ЖЦ. Це означає, що користувачеві надається незавершена система, яка потім змінюється і доповнюється доти, доки не стануть зрозумілими всі вимоги користувача. Як альтернативу, можна побудувати «експериментальний» прототип, який допоможе проаналізувати та перевірити вимоги. Після цього створюється ПС. На рис. 2.17 показано обидва підходи до використання прототипів.

Еволюційне прототипування починається з побудови відносно простої системи, яка реалізує найважливіші вимоги користувача. У міру виявлення нових вимог прототип змінюється та доповнюється. Зрештою, він стає тією ПС, яка потрібна замовнику. У цьому процесі не використовується детальна системна

специфікація, у багатьох випадках немає формального документа з системними вимогами. У даний час еволюційне прототипування є звичайною технологією розроблення ПС, яка широко використовується для розроблення Web-вузлів та додатків електронної комерції.

Рис. 2.17. Еволюційне та експериментальне прототипування

На противагу еволюційному підходу, метод експериментального прототипування призначений для розроблення та уточнення системної специфікації. Прототип створюється, оцінюється та модифікується. Дані оцінювання прототипу застосовуються для подальшої деталізації специфікації. Коли системні вимоги повністю сформовані, прототип перестає бути потрібним.

Існує різниця між метою еволюційного та експериментального прототипування [1]:

- метою еволюційного прототипування є постачання працюючої ПС кінцевому користувачу. Це означає, що необхідно розпочати створення системи, яка реалізує ті вимоги користувача, які найбільш зрозумілі та мають найвищий пріоритет. Вимоги з нижчим пріоритетом та нечіткі вимоги реалізуються за запитами користувачів;
- метою експериментального прототипування є перевірка та формування системних вимог. Тут спочатку створюється прототип, що реалізує вимоги, які сформульовані нечітко і які треба уточнити. Вимоги, які сформульовані чітко та зрозуміло, не потребують прототипування.

Інша важлива відмінність між цими підходами стосується управління якістю розроблюваної ПС. Експериментальні зразки мають дуже короткий термін життя. Вони швидко змінюються і для них висока експлуатаційна надійність не

потрібна. Для експериментального прототипу допускається знижена ефективність і безвідмовність, оскільки прототип повинен виконати лише основну функцію – допомогти у розумінні вимог.

На противагу цьому, еволюційні прототипи, які перетворюються в готову ПС, повинні бути розроблені з такими ж стандартами якості, що і будь-яке інше ПЗ. Вони повинні мати стійку структуру та високу експлуатаційну надійність, бути безвідмовними, ефективними і відповідати стандартам.

Розглянемо більш детально обидва типи прототипування.

Еволюційне прототипування. Як було сказано вище, у його основі лежить ідея розроблення початкової версії ПС, демонстрації її користувачам та подальшої модифікації аж до отримання системи, яка відповідає всім вимогам (рис. 2.18). Такий підхід спочатку використовувався для розроблення систем, які важко чи неможливо специфікувати (наприклад, систем штучного інтелекту). Нині він стає основною методикою розроблення ПС. Еволюційне прототипування має багато спільного з методами швидкого розроблення додатків і часто входить у ці методи як їхня складова частина.

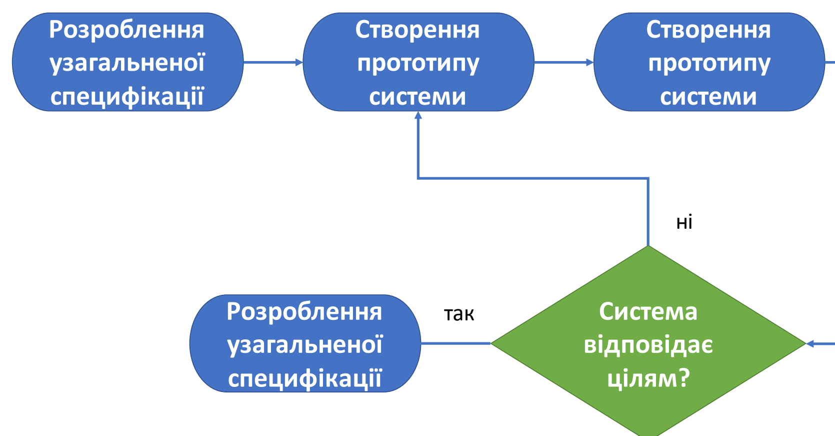

Рис. 2.18. Еволюційне прототипування

Цей метод прототипування має дві основні переваги [1]:

1) Прискорення розроблення системи. Сучасні темпи змін у діловій сфері потребують відповідних швидких змін у ПЗ. У деяких випадках швидке постачання ПЗ, зручність та простота його використання є важливішими, ніж повний

спектр функціональних можливостей ПС або довгострокові можливості її супроводу.

2) Взаємодія користувача із системою. Участь користувачів у процесі розроблення означає, що в системі повніше будуть враховані вимоги користувача.

Між окремими методами гнучкого розроблення ПЗ існують відмінності, але вони мають деякі загальні властивості [1]:

1) Етапи розроблення технічних вимог, проектування та реалізації перемежовуються. Немає детальної системної специфікації, проектна документація зазвичай залежить від інструментальних засобів, що використовуються для реалізації ПС. Вимоги користувача визначають лише найважливіші характеристики системи.

2) ПС розробляється покроково. Кінцеві користувачі та інші особи, які формують вимоги, беруть участь на кожному етапі проектування та оцінювання нової версії системи. Вони можуть пропонувати зміни та нові вимоги, які будуть реалізовані у наступній версії системи.

3) Застосування методів гнучкого розроблення систем. Вони можуть використовувати інструментальні CASE-засоби та мови четвертого покоління.

4) Інтерфейс користувача ПС зазвичай створюється за допомогою інтерактивних систем розроблення, які дозволяють швидко спроектувати та створити необхідний інтерфейс.

Еволюційне прототипування та методи, що базуються на використанні детальної системної специфікації, відрізняються підходами до верифікації та атестації систем.

Верифікація – процес перевірки ПЗ на відповідність специфікації НВ. Оскільки для прототипу немає докладної специфікації, його верифікація неможлива.

Атестація – процес, результати якого показують, що ПЗ відповідає тим цілям, для яких воно створювалося. Атестацію також важко провести без детальної специфікації, оскільки немає чітких формулювань цілей. Кінцеві користувачі, які беруть участь у процесі розроблення, можуть бути задоволені систе-

мою, тоді як інші користувачі – незадоволені, оскільки система не повністю відповідає тим цілям, які вони неявно перед нею поставили.

Верифікацію та атестацію системи, розробленої з використанням еволюційного прототипування, можна здійснити, якщо вона достатньо відповідає поставленій меті і своєму призначенню. Цю відповідність, звісно, не можна виміряти, а лише суб'єктивно оцінити. Такий підхід може породити проблеми, особливо якщо ПС створюється сторонніми компаніями-розробниками.

Існує три основні проблеми еволюційного прототипування, які необхідно враховувати, особливо при розробці великих систем із тривалим терміном ЖЦ [1]:

1) Проблеми керування. Структура управління процесом розроблення ПС будується відповідно до затвердженої моделі процесу створення ПЗ, де для оцінювання чергового етапу розроблення використовуються спеціальні контрольні проектні елементи. Прототипи еволюціонують так швидко, що створювати контрольні елементи стає не рентабельно. Крім того, швидке розроблення прототипу може вимагати застосування нових технологій. У цьому випадку може виникнути необхідність залучення спеціалістів із вищою кваліфікацією.

2) Проблеми супроводу системи. Через безперервні зміни у прототипах змінюється також структура системи. Це означає, що система буде важкою для розуміння всім, крім початкових розробників. Крім того, може застаріти спеціальна технологія гнучкого розроблення, яка використовувалася під час створення прототипів. Тому можуть виникнути труднощі при пошуку людей, які мають знання, потрібні для супроводу системи.

3) Проблеми укладання контрактів. Зазвичай договір на розроблення систем між замовником і розробниками ПЗ ґрунтується на системній специфікації. За її відсутності складно скласти договір на розроблення ПС. Для замовника може бути невигідним договір, за яким доводиться просто платити розробникам за час, витрачений на розроблення проекту; також малоймовірно, що розробники погодяться на контракт із фіксованою ціною, оскільки вони не можуть передбачити всі прототипи, які потрібно створити у процесі розроблення системи.

Із цих проблем випливає, що замовники повинні розуміти, наскільки ефективним є еволюційне прототипування як метод розроблення ПЗ. Цей метод дозволяє швидко створювати системи малого та середнього розміру, причому вартість ПС знижується, а її якість підвищується. Якщо до процесу розроблення залучаються кінцеві користувачі, то найчастіше система відповідатиме їх реальним потребам. Проте компанії-розробники, які використовують цей метод, повинні враховувати, що ЖЦ таких систем буде відносно коротким. У разі зростання проблем із супроводом систему необхідно замінити або повністю переписати. Для великих систем, коли до розроблення залучаються субпідрядники, на перший план виходять проблеми управління еволюційним прототипуванням. У цьому випадку краще використовувати експериментальне прототипування.

Покрокове розроблення (рис. 2.19) дозволяє уникнути деяких проблем, притаманних еволюційному прототипуванню. Загальна архітектура системи, визначена на ранньому етапі її розроблення, виступає у ролі системного каркасу. Компоненти системи розробляються покроково, потім входять у цей каркас. Якщо компоненти атестовані та включені до каркасу, ні архітектура, ні компоненти вже не змінюються (за винятком випадку, коли знаходяться помилки).

Рис. 2.19. Покроковий процес розроблення

Процесом покрокового розроблення можна керувати більше, ніж еволюційним прототипуванням, оскільки воно слідує звичайним стандартам розроблення ПЗ. Тут плани та документація створюються для кожного кроку розроблення ПС, що зменшує кількість помилок. Як тільки системні компоненти інтегровано в каркас – їх інтерфейси більше не змінюються.

Експериментальне прототипування. Модель процесу розроблення ПЗ, яка заснована на експериментальному прототипуванні, показано на рис. 2.20. У цій моделі розширено етап аналізу вимог для зменшення загальних витрат на розроблення. Основне призначення прототипу – зробити зрозумілими вимоги та надати додаткову інформацію для оцінки ризиків. Після цього прототип більше не використовується і не бере участь у процесі розроблення ПС.

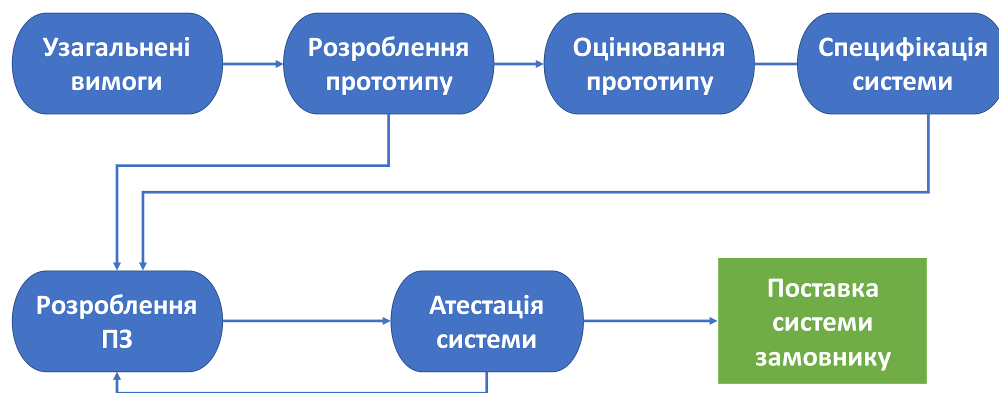

Рис. 2.20. Процес розроблення з використанням експериментальних прототипів

Цей метод прототипування зазвичай використовується для розроблення апаратних систем. Перш ніж буде розпочато дороге розроблення ПС, створюється макет (прототип), який використовується для перевірки структури цієї системи. Електронний макет системи будується з використанням готових компонентів, що дозволяє розробити версію ПС до того, як будуть вкладені кошти на розроблення спеціалізованих інтегральних схем.

Експериментальний прототип ПС не використовується для перевірки її архітектури, він допомагає розробити системні вимоги. Прототип часто зовсім не схожий на кінцеву систему, тому що для швидкого формування вимог та створення ПС використовується її спрощений прототип. В експериментальний

прототип закладаються лише обов'язкові системні функції, стандарти якості прототипу можуть бути знижені, критерії ефективності ігноруються. Мова програмування прототипу часто відрізняється від мови програмування, за допомогою якої буде створюватися остаточний варіант ПС.

У моделі процесу розроблення, показаної на рис. 2.20 передбачається, що прототип розробляється виходячи з узагальнених системних вимог, далі над прототипом проводяться експерименти і він змінюється доти, доки його функціональні можливості не задовольнять замовника. Після цього на основі прототипу деталізуються системні вимоги, реалізується звичайна для організації-розробника технологія розроблення ПЗ та система доводиться до остаточної версії. Деякі компоненти прототипу можуть використовуватись у ПС, тому вартість розроблення може бути знижена.

У даній моделі розроблення ПЗ основна проблема полягає в тому, що експериментальний прототип може не відповідати кінцевому ПП, який поставлятиметься замовнику. Спеціаліст, який тестує прототип, може мати власне бачення системи, яке не є типовим для її користувачів. Час тестування прототипу може бути недостатнім для повного оцінювання. Якщо прототип працює повільно, експерти можуть внести до нього зміни, які прискорюють роботу, але не співпадатимуть із засобами прискорення роботи кінцевої ПС.

Розробники іноді зазнають тиску менеджерів для прискорення роботи над прототипом, особливо якщо можлива затримка в поставці остаточної версії системи. Зазвичай таке прискорення породжує низку проблем [1]:

1) Неможливо швидко налаштувати прототип для виконання НВ, які ігнорувалися під час розроблення прототипу, таких, як: продуктивність, захищеність, стійкість до збоїв та безвідмовність.

2) Часті зміни вимог під час розроблення неминуче призводять до того, що зразок погано задокументований. Для розроблення прототипу використовується лише специфікація системної архітектури. Цього недостатньо для довготривалого супроводу системи.

3) Зміни, зроблені під час розроблення прототипу, можуть порушити архітектуру ПС, а її обслуговування у цьому випадку буде складним та дорогим.

4) Під час розроблення прототипу знижуються стандарти якості.

Щоб бути корисними у процесі розроблення вимог, експериментальні прототипи необов'язково мають виконувати роль реальних макетів систем. Паперові форми, які імітують інтерфейси систем, показали свою ефективність при формуванні вимог користувача, в уточненні проекту інтерфейсу і при створенні сценаріїв роботи кінцевого користувача. Вони дуже дешеві у розробці та можуть бути створені за декілька днів.

2.6.2 Технології швидкого прототипування

Ці технології розраховані головним чином на забезпечення швидкого розроблення прототипів, а не на такі їх системні характеристики, як: продуктивність, зручність експлуатації чи безвідмовність. Існує три основні методи швидкого прототипування:

1) Розроблення із застосуванням динамічних мов високого рівня.

2) Складання програм із повторним використанням компонентів.

3) Використання мов програмування БД.

Для зручності далі розглянемо ці методи окремо, але на практиці при розробленні прототипів систем вони часто використовуються спільно. Наприклад, мова програмування БД може застосовуватися для отримання даних з їх подальшою обробкою за допомогою повторно використовуваних компонентів. Інтерфейс користувача системи можна розробити за допомогою візуального програмування.

У даний час розроблення прототипів зазвичай спирається на набір інструментів, які підтримують, принаймні, два з цих методів. Наприклад, система Smalltalk VisualWorks підтримує мову дуже високого рівня та забезпечує повторне використання компонентів. Пакет Lotus Notes включає підтримку програмування БД за допомогою мови високого рівня та повторне використання компонентів, які можуть забезпечити операції над БД.

Більшість систем прототипування сьогодні також підтримують візуальне програмування, коли деякі частини або весь прототип розробляються в інтерактивному режимі. Замість послідовного написання програм розробник прототи-

пу віддає перевагу роботі з графічними піктограмами, що представляють функції, дані або компоненти інтерфейсу користувача, а також відповідні сценарії управління цими піктограмами. Програма, що готова до виконання, автоматично генерується з візуального представлення системи. Це спрощує розроблення ПЗ та зменшує витрати на прототипування.

Застосування динамічних мов високого рівня. Динамічні мови високого рівня дозволяють визначати типи даних, здійснювати синтаксичний аналіз та компіляцію безпосередньо на етапі виконання програми. Вони спрощують розроблення програм, оскільки зменшують кількість проблем, що пов'язані з розподілом пам'яті та управлінням нею. У таких мовах є засоби, які зазвичай мають бути побудовані з більш примітивних конструкцій у мовах, подібних до Ada або C. Приклади мов дуже високого рівня – Lisp (заснований на структурах списків), Prolog (заснований на алгебрі логіки) та Smalltalk (заснований на об'єктах).

Донедавна динамічні мови високого рівня широко не використовувалися для розроблення великих систем, оскільки вони потребують ґрунтовних засобів динамічної підтримки. Ці засоби збільшували обсяг необхідної пам'яті та зменшували швидкість виконання програм, написаних цими мовами. Однак зростання потужності та зниження вартості комп'ютерного обладнання зробило ці фактори не такими суттєвими.

Таким чином, для багатьох ділових додатків ці мови можуть замінити такі традиційні мови програмування, як, наприклад, C++, COBOL і Ada.

У табл. 2.9 представлені динамічні мови, які можуть використовуватися для розроблення прототипів [1].

При виборі мови для написання прототипу необхідно відповісти на низку запитань, а саме:

1) Який тип програми розробляється? Для кожного типу програми можна застосувати декілька різних мов. Якщо необхідний прототип програми, яка обробляє дані природною мовою, то мови Lisp або Prolog більше підходять, ніж Java або Smalltalk.

Таблиця 2.9. Приклади мов високого рівня, які можуть використовуватися при прототипуванні

Мова програмування	Тип мови	Тип програми
Smalltalk	Об'єктно-орієнтований	Інтерактивні системи
Prolog	Логічний	Системи обробки символьної інформації
Lisp	Заснований на списках	Системи обробки символьної інформації

2) Який тип взаємодії з користувачем? Різні мови забезпечують різні типи взаємодії з користувачем. Деякі мови, такі як Smalltalk і Java, добре інтегруються з Web-браузерами, тоді як мова Prolog найкраще підходить для розроблення текстових інтерфейсів.

3) Яке робоче середовище забезпечує мову? Розвинене робоче середовище підтримки мови зі своїми інструментальними засобами та легким доступом до компонентів, які можуть використовуватися повторно, спрощує процес розроблення прототипу.

Динамічні мови високого рівня для створення прототипу можна використовувати спільно, коли різні частини прототипу програмуються різними мовами.

Немає ідеальної мови для прототипування великих систем, оскільки зазвичай різні частини системи різнотипні. Перевага багатомовного підходу полягає в тому, що для створення кожного компонента можна підібрати потрібну мову і таким чином прискорити розроблення прототипу. Недоліком такого підходу є те, що важко розробити комунікаційні зв'язки для компонентів, написаних різнорідними мовами.

Програмування БД. Еволюційна стратегія розроблення в даний час є стандартною методикою для створення бізнес-додатків малого та середнього розміру. Більшість бізнес-додатків включають систему управління БД і обробки даних, які знаходяться в ній.

Для підтримки розроблення таких програм усі комерційні системи управління БД мають внутрішні засоби програмування. Програмування БД виконується на основі спеціалізованих мов, які мають вбудовану базу знань та засоби, необхідні для роботи з БД. Робоче середовище підтримки мови забезпечує інструментальні засоби для створення інтерфейсів, числових обчислень і звітів.

Мови четвертого покоління успішно застосовуються в практиці розроблення ПЗ, оскільки більшість сучасних додатків тією чи іншою мірою займаються обробкою інформації, що знаходяться у БД. Термін «мова четвертого покоління» застосовується як до мови програмування БД, так і до робочого середовища. Основні операції, які виконуються такими додатками, – це модифікація БД та створення звітів на основі інформації, отриманої з БД. Зазвичай для введення та виведення даних використовуються стандартні форми. Мови четвертого покоління мають засоби для створення інтерактивних програм, які дозволяють користувачам вносити зміни до БД. Інтерфейс користувача зазвичай складається з набору стандартних форм або електронної таблиці.

Зазвичай робоче середовище мов четвертого покоління включає наступні інструментальні засоби (рис. 2.21):

- мову програмування БД (точніше мову запитів до БД), наприклад, SQL;
- генератор інтерфейсів, який використовується для створення форм введення та відображення даних;
- електронну таблицю, яка застосовується для аналізу даних та виконання різних дій над числовою інформацією;
- генератор звітів, який призначений для створення звітів на основі інформації, що міститься у БД.

Більшість бізнес-додатків потребують структуровані форми для введення та виведення даних, тому мови четвертого покоління забезпечують потужні засоби для визначення екранних форм та створення звітів. Екранні форми часто визначаються як ряд взаємопов’язаних форм тому система, що генерує екрани, має забезпечувати наступне [1]:

1) Інтерактивне визначення форм, коли розробник визначає поля введення та їх організацію.

2) Зв'язування форм, коли розробник визначає певні дані, введення яких викликає відображення подальших форм.

3) Перевірка вхідних даних, коли розробник для формування полів форм визначає дозволений діапазон вхідних величин.

Рис. 2.21. Компоненти мови четвертого покоління

У даний час більшістю мов четвертого покоління підтримується розроблення інтерфейсів БД на основі Web-браузерів. Вони роблять БД доступною за допомогою мережі Internet. Це знижує вартість навчання та ПЗ і дозволяє зовнішнім користувачам мати доступ до БД. Однак обмеження протоколів мережі Internet та повільний перегляд Web-сторінок роблять цей метод не придатним для систем, у яких потрібна швидка взаємодія з користувачем.

Методи, які засновані на мовах четвертого покоління, можуть використовуватися для еволюційного прототипування або для генерування одноразового прототипу ПС. Структура, яку CASE-засоби накладають на ПЗ і супутню документацію, визначає більш зручний супровід прототипів, ніж пропонують прототипи, розроблені вручну. CASE-засоби можуть генерувати код SQL або код мовою нижчого рівня, наприклад, COBOL.

Хоча мови четвертого покоління підходять для розроблення прототипів, все ж таки вони мають ряд недоліків, які виявляються при розробці систем. Написане мовами четвертого покоління ПЗ зазвичай виконується повільніше і ви-

магає набагато більше пам'яті, ніж подібні програми, написані мовами програмування третього покоління. Наприклад, в одному з експериментів [1] відбулося перенесення ПЗ з мови четвертого покоління на мову програмування С++, що призвело до скорочення використання пам'яті на 50%. Програма на С++ також виконувалася в 10 разів швидше, ніж аналогічне ПЗ, написане з використанням мови четвертого покоління.

Незважаючи на те, що застосування мов четвертого покоління знижує вартість розроблення ПС, загальна сума витрат за повний ЖЦ таких систем поки що не зрозуміла. Їхні програми зазвичай погано структуровані і важкі у супроводі. Аналогічні проблеми можуть виникати при модифікації подібних систем. У даний час мови четвертого покоління не достатньо стандартизовані і не уніфіковані, тому при модифікації ПС, швидше за все, доведеться переписувати ПЗ, оскільки мова, якою воно було написано, вже застаріє.

Складання програм з повторним використанням компонентів. Час, який необхідний для розроблення ПС, можна зменшити, якщо багато частин такої системи буде використано з попередніх проектів компанії-розробника. Для швидкої побудови прототипу необхідно мати набір компонентів, які придатні для повторного використання, та механізм складання ПС з цих компонентів. Цей підхід показано на рис. 2.22.

Рис. 2.22. Складання повторно використовуваних компонентів

Прототипування з повторно використовуваними компонентами застосовується при розробленні вимог, якщо є відповідні компоненти. Якщо відповідних компонентів немає, то для реалізації деяких вимог буде необхідний комп-

ромісний підхід. Функціональні можливості доступних компонентів можуть не точно відповідати вимогам користувача. Але, з іншого боку, ці вимоги зазвичай досить гнучкі, тому в багатьох випадках можливе створення прототипу.

Розроблення прототипу з використанням компонентів можна реалізувати на двох рівнях:

1) Рівень програми, коли цілі прикладні системи інтегруються з прототипом так, щоб об'єднувалися їх функціональні можливості. Наприклад, якщо прототипу потрібні засоби обробки тексту, це можна забезпечити шляхом інтегрування в прототип стандартної системи текстового процесора. Варто зазначити, що, наприклад, програми Microsoft Office підтримують інтеграцію зі сторонніми системами.

2) Рівень компонентів, коли окремі компоненти поєднуються всередині структури, яка реалізує систему. Така структура може бути створена за допомогою однієї з мов опису сценаріїв, таких як: Visual Basic, TCL/TK, Python, Perl, тощо. В якості альтернативи можуть застосовуватися такі системи як CORBA, DCOM або JavaBeans.

Компоненти, які повторно використовуються, дають доступ до всіх своїх функціональних можливостей. Якщо компонент також забезпечує створення сценаріїв або засоби автоматизації (наприклад, макроси Excel), вони можуть використовуватися для розширення функціональних можливостей прототипу.

Для розуміння цього методу розроблення прототипу корисно було б скористатися складовим документом, який є схемою обробки даних прототипом і який можна розглядати як контейнер для декількох різних об'єктів. Ці об'єкти містять різні типи даних (наприклад, таблиця, діаграма, форма, тощо), які можуть бути оброблені різними додатками.

На рис. 2.23 представлений документ для прототипу системи, що включає текстові елементи, елементи електронної таблиці та звукові файли. Текстові елементи обробляються текстовим процесором, таблиці – електронною таблицею, а звукові файли – аудіо програвачем. Коли користувач системи звертається до об'єкта певного типу, викликається пов'язана з ним програма.

Розглянемо прототип системи, що підтримує управління розробленням вимог. Для цієї системи необхідні засоби фіксації вимог, їх зберігання, створення звітів, пошуку залежностей між вимогами та управління цими залежностями за допомогою матриць оперативного контролю. У прототипі має бути БД (для зберігання вимог), текстовий процесор (для введення вимог та створення звітів), електронна таблиця (для управління матрицями контролю) та спеціально написана програма для пошуку залежностей між вимогами.

Рис. 2.23. Зв’язування додатків за допомогою складеного документа

Основна перевага підходу до прототипування полягає в тому, що багато функціональних засобів прототипу можна реалізувати швидко і дешево. Якщо користувачі, які тестують прототип, знайомі з інтегрованими в прототип компонентами, їм не потрібно вчитися використовувати ці елементи прототипу. Проблеми при роботі з прототипом можуть виникнути лише при перемиканні з одного компоненту на інший. Але це значною мірою залежить від використовуваної операційної системи. Для організації перемикання між програмами найбільш широко [1] використовується механізм зв’язування та впровадження об’єктів OLE від Microsoft.

Не завжди можна або зручно використовувати великі готові компоненти. Для створення прототипів можна використовувати більш «тонкі» компоненти, а

саме: окремі функції або об’єкти, які виконують спеціальні дії, наприклад, сортування, пошук, відображення даних і т.д.

Прототипування починається з визначення загальної структури прототипу, потім компоненти інтегруються відповідно до цієї структури. Якщо немає компонентів, які виконують потрібні функції, замість них розробляється окреме ПЗ, яке також можна повторно використовувати в майбутньому.

Візуальні системи розроблення програм підтримують повторне використання компонентів. Розробники будують систему в інтерактивному режимі, визначаючи інтерфейс, як набір екранних форм, полів, кнопок і меню. Ці елементи інтерфейсу іменовані і кожен з них пов’язаний зі сценарієм обробки подій і даних. Ці сценарії можуть викликати програмні компоненти повторно.

На рис. 2.24 показаний екран програми, що містить меню, поля введення, поля виводу та кнопки, які представлені округленими прямокутниками праворуч на екрані. Після розташування цих графічних елементів на екрані розробник визначає які готові компоненти будуть пов’язані з ними або пише окремі програми, які виконуватимуть необхідні дії. Також на рис. 2.24 показано компоненти, які пов’язані з деякими елементами екрану.

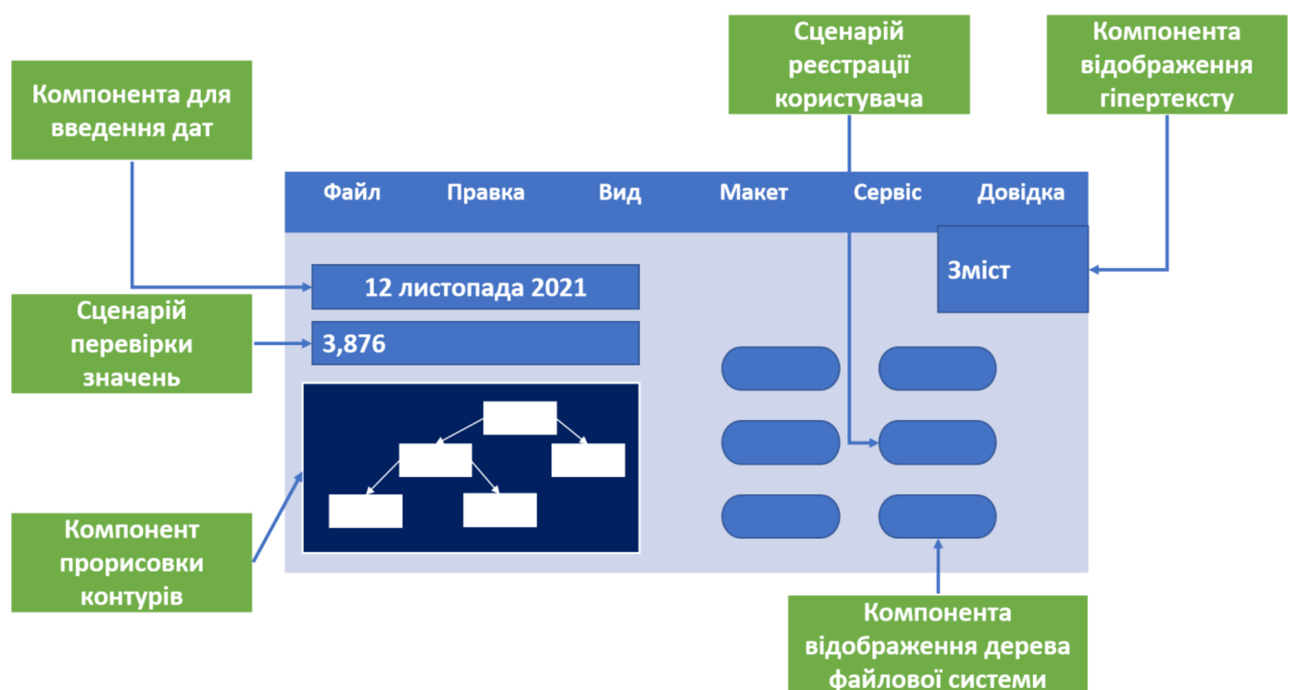

Рис. 2.24. Візуальне програмування із повторним використанням компонентів

Описаний підхід до розроблення систем надає можливість швидко розробити відносно мале та просте ПЗ, яке може бути створене однією особою чи невеликим колективом розробників. Для великих систем, які мають розроблятися великими колективами, подібний підхід організувати складніше, оскільки немає явної архітектури ПС і часто є складні залежності між різними компонентами системи. У цьому є причина труднощів при внесенні змін до системи. Крім того, через обмежений набір готових компонентів буває важко реалізувати нестандартні інтерфейси користувача. Компонентний метод розроблення більше підходить для створення великих ПС компаніями з достатнім досвідом роботи.

2.6.3 Прототипування інтерфейсів користувача

Використання графічних інтерфейсів користувача є стандартом для інтерактивних систем. Зусилля, що вкладаються у визначення, проектування та реалізацію такого інтерфейсу, становлять значну частину вартості розроблення ПЗ. Розробники не повинні нав'язувати користувачам свою точку зору на інтерфейс, який проектується. Користувачі повинні брати активну участь у процесі проектування інтерфейсу. Такий погляд на розроблення інтерфейсів привів до підходу, названого проектуванням, яке орієнтоване на користувача (англ. User-Centred Design), який ґрунтується на прототипуванні інтерфейсу та участі користувача у процесі його проектування.

Текстових описів і діаграм недостатньо для формування вимог до інтерфейсу користувача через його динамічну природу. Тому еволюційне прототипування з участю кінцевого користувача – єдиний прийнятний спосіб розроблення графічного інтерфейсу ПС.

Генератори інтерфейсів – це графічні системи проектування екранних форм, де інтерфейси компонуються з елементів типу меню, полів, піктограм та кнопок, які можна просто вибрати та помістити в екранну форму. Як згадувалося раніше, системи даного типу є необхідною частиною систем програмування БД. Генератори інтерфейсів створюють добре структуровану програму, яка згенерована за специфікацією інтерфейсу.

Іншим варіантом, який дозволяє створювати інтерфейси користувача, є мова розмітки гіпертексту HTML. Кнопки, поля, форми та таблиці можуть бути включені до Web-сторінок так само, як і засоби мультимедіа. Сценарії обробки подій та даних, пов'язані з об'єктами інтерфейсу, можуть виконуватися або на машині веб-клієнта, або на веб-сервері. Через широкі можливості Web-браузерів і мови HTML, у даний час все більше інтерфейсів будуються як Web-орієнтовані.

Для Web-орієнтованих інтерфейсів прототипи можна створювати за допомогою стандартних редакторів Web-сторінок, які будують інтерфейси користувача. Об'єкти на веб-сторінці, як і пов'язані з ними операції, визначаються за допомогою вбудованих засобів HTML (наприклад, зв'язування з іншою сторінкою) або за допомогою мови Java чи сценаріїв.

2.7 Управління персоналом при реалізації проектів

Люди, які працюють в компаніях з розроблення ПЗ, є їх найціннішим ресурсом. Саме вони представляють інтелектуальний капітал, тому що, наприклад, від менеджерів залежить чи отримає компанія найкращі з можливих дивіденди від інвестицій у людські ресурси. У компаніях і економічних структурах, які успішно розвиваються, це досягається в тому випадку, якщо компанія поважає своїх співробітників. Коло виконуваних ними обов'язків і рівень винагороди повинні відповідати їх вмінню, яке, у свою чергу, залежить від кваліфікації та досвіду.

Таким чином, **ефективний менеджмент** – це ефективне управління персоналом компанії. Керівники проектів, як менеджери, повинні вирішувати як технічні, так і далекі від технічних питання, використовуючи при цьому членів команди з оптимальною ефективністю. Вони повинні вміти мотивувати дії людей, організовувати їх роботу і гарантувати її якісне виконання. Слабкий менеджмент персоналу – одна з основних складових провалу ПП.

Людські можливості досить різноманітні. Перш за все це залежить від рівня інтелекту, освіти та особистого досвіду. Проте у всіх людей є щось спільне: у своїй розумовій діяльності люди підпорядковуються певним обмеженням. Ці

обмеження не що інше, як наслідок того, яким чином мозок людини зберігає і обробляє отриману інформацію. Саме знання меж людського мислення має виняткову важливість у розумінні того, чому деякі технології розроблення програмного забезпечення досягають високої ефективності.

2.7.1 Організація людської пам'яті

ПС є абстрактними об'єктами, при роботі з якими розробникам необхідно враховувати всі їх особливості. Наприклад, програміст повинен розуміти і завжди пам'ятати про взаємозв'язок між лістингом вихідного коду і динамічною поведінкою програми. Ці знання будуть корисні йому при розробленні програм у подальшому.

Організацію людської пам'яті можна представити у вигляді ієрархічної структури з трьома явно вираженими, пов'язаними між собою типами пам'яті (рис. 2.25) [22]:

1) Короткострокова пам'ять з швидким доступом, але обмеженими можливостями. Вхідний сигнал, який породжується органами чуттів, надходить саме сюди для подальшої обробки. Таку пам'ять можна порівняти з регістрами в обчислювальній машині, так як вона є місцем обробки інформації, а не її зберігання.

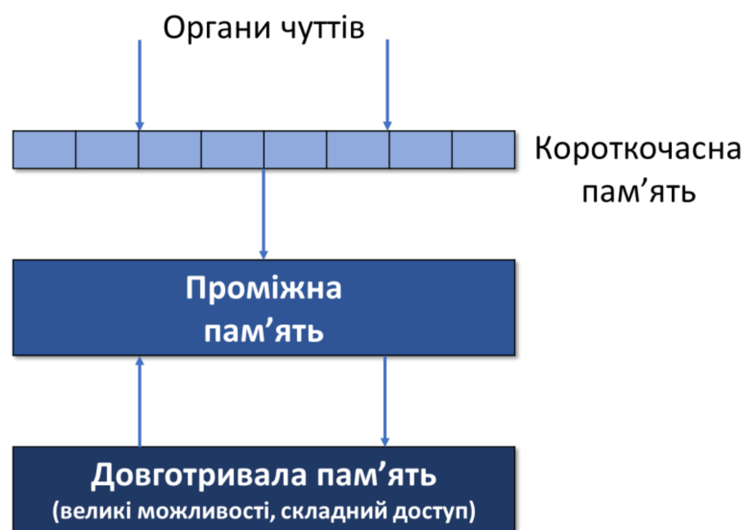

Рис. 2.25. Схема організації пам'яті людини

2) Проміжна пам'ять з високими можливостями. Доступ до такої пам'яті важчий, ніж до короткочасної. Вона теж служить для обробки інформації, проте може утримувати інформацію на довший термін, ніж короткочасна пам'ять. Не використовується для збереження інформації, призначеної для більш тривалого зберігання. За аналогією з комп'ютером така пам'ять нагадує оперативну, в якій інформація зберігається тільки на період проведення обчислювальних операцій.

3) Довготривала пам'ять. Це пам'ять з найширшими можливостями, важким доступом і вкрай ненадійними механізмами зберігання. Така пам'ять використовується для «постійного» зберігання інформації. Продовжуючи порівняння з комп'ютерною пам'яттю, можна зіставити довготривалу пам'ять з дисковими накопичувачами.

Короткочасна пам'ять отримує інформацію про поставлене перед людиною завдання через читання нею різних документів та контакти з іншими людьми. Далі отримана інформація, склавши єдине ціле з існуючими даними в довгостроковій пам'яті, надходить у проміжну пам'ять. Саме результат формування цього цілого і є основою вирішення поставленого завдання. Все це заноситься в довготривалу пам'ять для того, щоб використовуватися в майбутньому. При цьому рішення не обов'язково має бути вірним. Згодом, після надходження нової інформації, довготривала пам'ять змінюється. Незважаючи на це, неправильна інформація аж ніяк не видаляється, а навпаки, зберігається у видозміненій формі, адже найчастіше люди вчаться на власних помилках.

Пізнавальний процес людини певною мірою стримується обмеженим розміром короткочасної пам'яті. У одному класичному експерименті [22] визначається, що у короткочасній пам'яті може зберігатися близько семи квантів інформації. Квант інформації не являється твердо фіксованою величиною, це швидше певна інформаційна одиниця, якою може бути телефонний номер, сформульована мета або назва вулиці. Автор цього експерименту також пропонує опис процесу «утворення блоків», під час якого кванти інформації збираються в цілі блоки.

Якщо постановка завдання містить більше інформації, ніж той обсяг, з яким може впоратися короткочасна пам'ять, тоді ця інформація обробляти-

меться і перетворюватиметься паралельно надходженню. Такий процес може призвести до втрати інформації. Також велика ймовірність виникнення помилок, оскільки обробка інформації не встигає за її надходженням в пам'ять.

Формування блоків інформації використовується і під час читання програми програмістом. Той, хто читає програму, розбиває інформацію, що міститься у програмі, на блоки, які побудовані за принципом внутрішньої семантичної (смислової) структури. Сприйняття програми не відбувається послідовно від оператора до оператора, якщо тільки оператор не є логічним блоком. На рис. 2.26 показано, як проста програма сортування може бути розбита на блоки суб'єктом, що її читає.

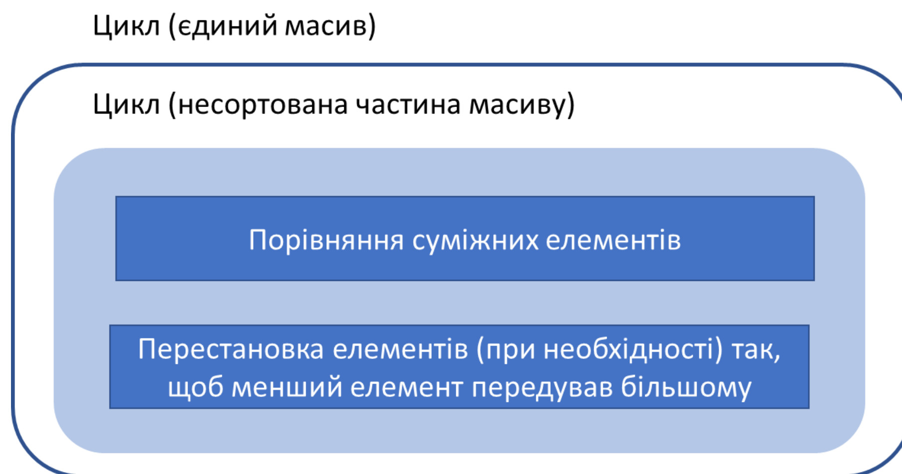

Рис. 2.26. Пізнавальні блоки у програмі сортування

Після визначення внутрішньої семантичної структури, яка представляє програму, ці знання передаються у довгострокову пам'ять. Якщо ця інформація використовується регулярно, зазвичай її важко забути. Без особливих зусиль таку інформацію можна відтворити у різних формах представлення. Тому людині легше запам'ятати абстрактні конструкції високого рівня, ніж подробиці низького рівня.

Знання, які здобуваються в процесі розроблення ПЗ і зберігаються в довгостроковій пам'яті, поділяються на два класи [22]:

1) Семантичні знання. Це знання про основні поняття, такі, наприклад, як функціонування оператора присвоєння, уявлення про клас об'єктів або струк-

туру організації програм. Ці знання набуваються через досвід та навчання та зберігаються у формі автономних уявлень.

2) Синтаксичні знання. Це деталізовані знання (подробиці) про окремі об’єкти та явища, наприклад, опис об’єкта в UML, які стандартні функції доступні в мові програмування, оператор присвоєння викликається за допомогою оператора «=» або оператора «:=» тощо. Ці знання зберігаються у неструктурованому вигляді.

Таку організацію знань показано на рис. 2.27. Тут передбачається, що семантичні знання і знання про завдання мають свою організацію і структуру, які показані на схемі у вигляді логічної структури взаємозв’язків між різними фрагментами знань. На противагу семантичним знанням, синтаксичні є довільними (певною мірою) і не мають чіткої організації. Тому саме для даного типу знань можливі помилки та втрата інформації.

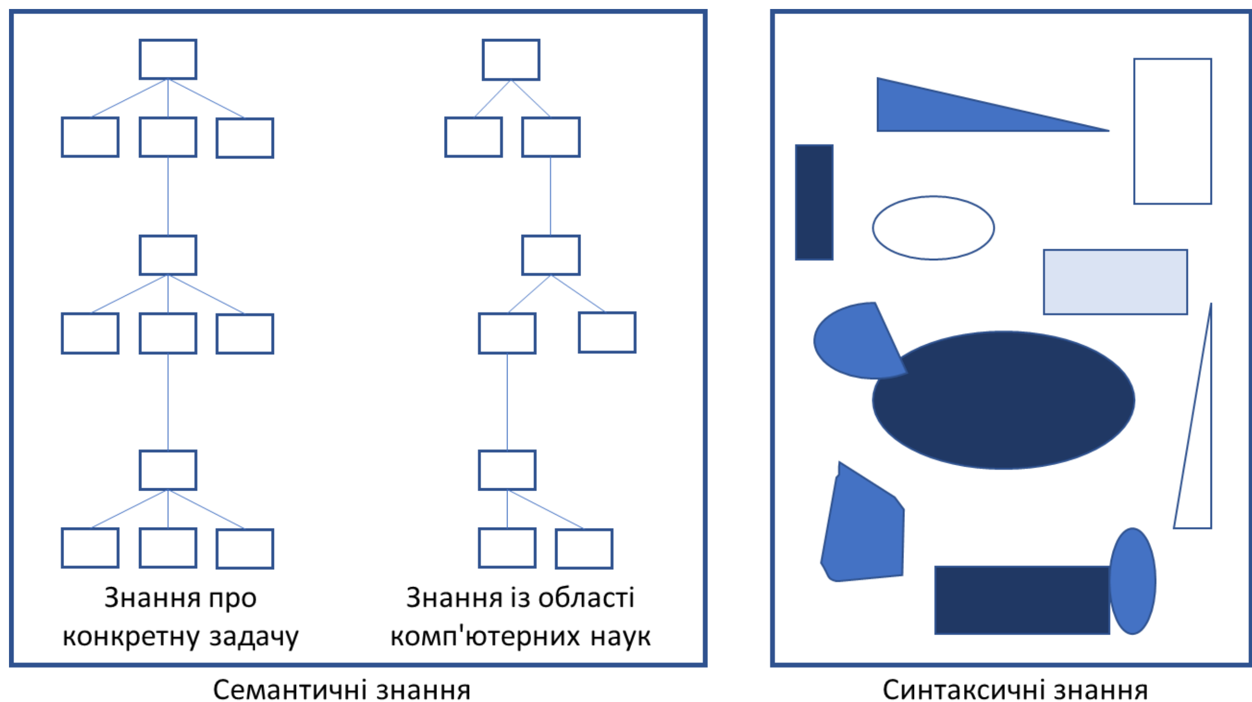

Рис. 2.27. Синтаксичні та семантичні знання

Семантичні знання набуваються за допомогою досвіду та шляхом активного навчання. Нова інформація свідомо інтегрується з семантичними структурами, що вже існують. І навпаки: синтаксичні знання набуваються шляхом про-

стого запам’ятовування. Хоча нові синтаксичні знання не поєднуються з тими, що існують в той же момент, однак можуть з ними взаємодіяти. Отже, ці знання забуваються частіше, ніж більш поглиблені семантичні знання.

Різні моделі набуття синтаксичних та семантичних знань допомагають зрозуміти, як досвідчені програмісти вивчають нову мову програмування. У них не виникає особливих труднощів у освоєнні основних понять мови, таких як присвоєння, цикл, умовні оператори тощо. Синтаксис нової мови, має тенденцію поєднуватися з синтаксисом вже знайомих мов. Тому програміст, що володіє мовою Delphi, при вивченні мови Java спочатку використовуватиме присвоєння швидше за допомогою оператора «:=», ніж «=», але у міру глибшого освоєння основні поняття (синтаксичні знання) закріплюються у пам’яті як семантичні знання. Останні зберігаються в пам’яті в абстрактній смисловій формі, але деталі основних понять можуть бути відтворені у різних конкретних уявленнях.

Розглянемо, наприклад, алгоритм двійкового пошуку, у якому здійснюється пошук конкретного елемента упорядкованої сукупності. Цей алгоритм включає перевірку середньої точки сукупності та застосування знань про взаємовідносини порядку для перевірки місцезнаходження елемента-ключа у одній або іншій частині сукупності. Програміст, знайомий з цим алгоритмом, легко напише його версію мовами Java, C++ або будь-якою іншою мовою програмування.

Ця модель добре пояснює і той факт, що для багатьох людей вміння програмувати приходить практично «одним махом», після довгого періоду навчання та труднощів. Уміння програмувати вимагає від людини розуміння семантичних концепцій та здатності розділяти семантичні та синтаксичні поняття.

Іноді викладачам не вдається зрозуміти проблеми студентів. Самі вони успішно розібралися в семантичній інформації та осмислили її, тому для них головне завдання полягає у обробці синтаксичної інформації. Тому часто викладачеві буває складно пояснити семантичні поняття на такому рівні, щоб це було доступно новачкам у програмуванні.

2.7.2 Рішення задач

Розроблення та написання ПЗ є процесом вирішення завдань. Щоб створити ПС, у першу чергу необхідно зрозуміти поставлене завдання (проблему), розробити стратегію пошуку рішення і перетворити рішення на програму.

Перший етап включає перехід постановки завдання з короткочасної пам'яті в проміжну. Далі проблема зіставляється та інтегрується зі вже наявними знаннями в довгостроковій пам'яті. Потім вона оброблюється з метою складання певного рішення. На закінчення, знайдене рішення переноситься у програму, яка виконується (рис. 2.28).

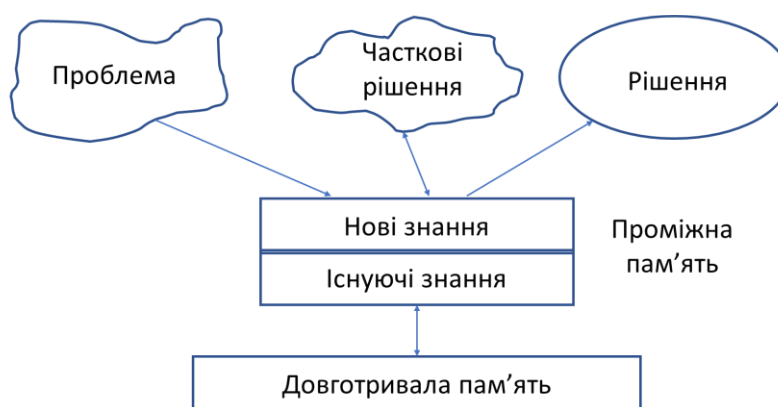

Рис. 2.28. Процес рішення задач

Процес вирішення завдань потребує інтеграції постановки задачі та комп'ютерних знань. Велике значення мають також організаційні моменти, такі як, наприклад, завершення роботи над рішенням у межах можливостей бюджету. Таким чином, користувач повинен розумітися на змістовній постановці завдання, розробник ПЗ повинен бути компетентним у обчислювальних системах, а менеджер повинен бути хорошим спеціалістом з організаційних питань. У процесі розроблення ПЗ всі ці знання об'єднуються і використовуються спільно.

Розроблення рішення (програми) включає побудову внутрішньої семантичної моделі завдання і моделі рішення, яке відповідає їй. Після формування моделі її необхідно подати у відповідній синтаксичній системі нотацій.

Нотація – сукупність позначень (умовних знаків) у системі правил для опису синтаксису мови програмування.

Таким чином, створення ПЗ є ітераційним процесом, який складається з трьох етапів [1]:

1) Інтеграція існуючих знань про комп’ютерні технології та про поставлене завдання для того, щоб створити нове знання і з його допомогою розібратися в проблемі. При цьому особливо важливим є практичний досвід вирішення прикладних завдань на цій стадії.

2) Створення семантичної моделі рішення, яка тестується і вдосконалюється до тих пір, поки вона не буде успішно справлятися з поставленим завданням.

3) Подання моделі будь-якою мовою програмування або в системі проектної нотації.

Якщо менеджерам необхідно визначити, кого включити у довгостроковий проект, насамперед слід оцінити здатність фахівця вирішувати всеосяжні проблеми та його досвід роботи у цій галузі і лише потім його майстерність програміста. Як тільки приходить розуміння поставленого завдання, у досвідчених програмістів виникають приблизно однакові труднощі у розробці програми, незалежно від того, яка при цьому використовується мова програмування. Безсумнівно, навички програмування необхідні, а для їх розвитку знадобиться досить багато часу (особливо це стосується таких порівняно складних мов, як C++). Але набагато легше освоїти певну мову програмування, ніж розвинути у собі здатність до вирішення завдань.

Переведення з семантичної моделі в програму виключає появу помилок, якщо обрана програмістом мова містить конструкції, які відповідають низькорівневим семантичним структурам. Незважаючи на різноманітність, вони у будь-якому разі повинні співвідноситися з такими поняттями мов програмування, як оператори присвоєння, цикли, умовні оператори, приховування інформації, об’єкти, спадковість тощо. Чим більше семантичні структури відповідають конструкціям мов програмування, тим легше написати програму.

Таким чином, програми, написані за допомогою мов програмування високого рівня (наприклад, Java), містять меншу кількість помилок, ніж ПЗ створене за допомогою мови асемблера, так як семантичні структури низького рівня можуть безпосередньо кодуватися виразами мов високого рівня. Однак, при цьому можуть виникнути проблеми, якщо чітко не розділити функціональні та об’єктно-орієнтовані аспекти вихідного завдання. Проблеми такого роду виникають у компаній, які звикли до стандартної методики аналізу, наприклад, SADT (*англ. Structured Analysis and Design Technique*), але використовують при цьому об’єктно-орієнтоване проектування та програмування.

SADT – метод структурного аналізу та проектування. Належить до групи методів, що реалізують структурний (а не об’єктно-орієнтований) підхід до проектування ПЗ. Дана модель також пояснює, чому структурне програмування є оптимальною альтернативою при організації управління програмою.

В основу структурного програмування покладено такі семантичні поняття, як цикли та умовні оператори. Короткочасна пам’ять програміста при цьому рідко буває перевантаженою, що знижує можливість виникнення помилок. Написані з використанням структурного підходу програми легші для розуміння, оскільки їх алгоритм можна чітко переглянути від початку до кінця. Структури, які необхідні для формування семантичних блоків, створюються послідовно, без участі інших частин програми. Таким чином, короткочасна пам’ять програміста повністю зайнята лише одним сеансом написання коду. При цьому не виникає необхідності вимагати з проміжної пам’яті інформацію про інші частини програми, які взаємодіють із створюваним кодом.

2.7.3 Мотивація

Координація діяльності людей для виконання певної роботи є одним з основних завдань менеджерів. Дуже часто мотивація людини спрямована на задоволення своїх потреб. Ці потреби мають ієрархічну структуру (рис. 2.29, [22]).

Найнижчий рівень ієрархічної структури представляють основні фізіологічні потреби у їжі, сні, самозбереженні тощо. Соціальні потреби пов’язані з необхідністю почуватися частиною соціальної групи. Потреби в оцінці асоцію-

ються з бажанням отримати певний рівень поваги в суспільстві, а потреба в самореалізації пов'язана з розвитком особистості. Природно, що пріоритетними у реалізації є нижчі потреби (наприклад, втамування голоду), а потім вже людина зосереджується на більш високих потребах.

Рис. 2.29. Ієрархічна структура людських потреб

Персонал компаній, які займаються розробленням ПЗ, як правило, не відчуває сильного голоду або спраги і знаходиться у відносній безпеці. Таким чином, в аспекті управління цими людьми головним завданням менеджера є задоволення їх потреб, пов'язаних з оцінкою, самореалізацією та необхідністю бути членом певної соціальної групи.

Тактика задоволення соціальних потреб ґрунтується на наданні людям можливості та часу для зустрічей із колегами, а також на тому, щоб забезпечити місце для таких зустрічей. Легкі у використанні засоби неформального спілкування (наприклад, месенджери чи електронна пошта) з цих позицій становлять виняткову цінність.

Для задоволення потреби в оцінці дуже важливо дати зрозуміти людям, наскільки важливою є їхня роль у компанії. Відкрите визнання їх досягнень – найпростіший та найефективніший спосіб задоволення цієї потреби. Крім того, люди повинні відчувати, що їхня робота оплачується на належному рівні, який визначається їх знаннями та досвідом.

Щоб задовольнити потреби персоналу в самореалізації, важливо надати кожному співробітнику певний рівень відповідальності за зроблену роботу. Це

досягається шляхом доручення їм досить важких завдань (але в жодному разі не нездійсненних), а також проведення навчання, у процесі якого можуть розвиватися їхні навички.

Розглядаючи мотивацію з психологічного погляду, можна виділити три типи професіоналів [22]:

1) Люди з цільовою орієнтацією, які мають достатньо мотивації від роботи, яку виконують. До цього типу належать випускники технічних спеціальностей, мотивація яких викликана інтелектуальними завданнями розроблення ПЗ.

2) Люди з само орієнтацією, мотивація яких заснована на особистому успіху та визнанні. Вони зацікавлені у розробленні ПЗ, переслідуючи свої особисті інтереси.

3) Люди із зовнішньою орієнтацією, мотивація яких потребує присутності та діяльності співробітників. Так як у наш час створення програм стає все більш орієнтованим на користувача, такі люди все частіше залучаються до розроблення ПЗ.

Люди із зовнішньою орієнтацією вважають за краще працювати в колективі, тоді як співробітники з само орієнтацією та цільовою орієнтацією прагнуть до роботи поодинці. При цьому жінки належать до типу людей із зовнішньою орієнтацією частіше, ніж чоловіки. Вони також краще виконують роботу з розповсюджування інформації.

Мотивація окремого працівника включає всі перераховані елементи, хоча у кожен конкретний період часу переважає лише один тип мотивації. Ніхто не стверджує, що особисті властивості характеру будь-якого програміста є постійними. Людині властиво змінюватися. Наприклад, у випускників технічних спеціальностей, які не задоволені оплатою своєї праці, може відбутися переоцінка цінностей, після чого особисті інтереси стануть для них вищими за зацікавленість у вирішенні технічних проблем.

Наведена піраміда потреб має також і недолік: у її основі лежить виключно мотивація, спрямована на досягнення виключно своїх інтересів. Тут не приділяється достатньо уваги тому факту, що люди почуваються частиною організації, професійної групи та частиною якоїсь культури.

Отже, у мотивації задіяні як особисті інтереси, так й інтереси цих розширених груп. Членство у згуртованій групі є надзвичайно високою мотивацією для багатьох. Люди, які виконують простіші завдання, часто люблять ходити на роботу тільки тому, що їм подобаються співробітники, з якими вони працюють, та завдання, які вони виконують.

2.7.4 Групова робота

Основна частина професійного ПЗ розробляється командами програмістів від двох людей до декількох сотень. В останньому випадку навряд чи хтось буде здатним ефективно працювати над одним завданням у такій великій команді, тож ці команди діляться ще й на підгрупи. Кожна підгрупа відповідає за певну частину проекту та працює над однією підсистемою. При грамотному підборі група складається не більше ніж з восьми осіб [1, 22]. У групах невеликого розміру легко знизити ризик виникнення проблем у взаємовідносинах між членами групи. Кожна група має бути забезпечена місцем для проведення зустрічей. Крім того, члени групи мають можливість зустрічатись в офісах. Для таких груп немає потреби застосовувати складні структури комунікації.

Організація команди, яка б ефективно працювати над створенням ПЗ, є досить складним завданням для менеджера. Необхідно, щоб у команді було рівне співвідношення технічних навичок, досвіду та вираження індивідуальності. Команда, що добре функціонує – це щось більше, ніж простий набір людей з необхідним співвідношенням навичок. У хорошій команді присутній дух товариства, який мотивує співробітників через успіхи всієї команди, включаючи досягнення власних цілей. Тому менеджери повинні стимулювати діяльність, спрямовану безпосередньо на «будівництво команди», щоб сприяти формуванню почуття відданості її інтересам.

Нижче пераховані чотири основні фактори, які тією чи іншою мірою впливають на групову роботу [22]:

1) Склад команди. Команда повинна мати правильне співвідношення навичок, досвіду та особистісних якостей.

2) Згуртованість команди. Члени робочої групи повинні сприймати себе як єдину команду, а не як просту сукупність індивідуумів, які працюють над проблемою.

3) Спілкування у команді. Між членами команди мають бути дружні стосунки.

4) Організація команди. Необхідно організувати команду таким чином, щоб кожен відчував свою цінність та був задоволений своєю роллю.

Створення команди. На початку практичної діяльності більшості фахівців основним мотиваційним чинником слугує робота. Тому в групи з розроблення ПЗ часто включаються люди, які мають власні міркування щодо того, як повинні вирішуватися завдання технічного плану. Такий принцип підбору робочих груп викликаний постійними скаргами на проблеми, що виникають внаслідок ігнорування стандартів інтерфейсу, переробки системи після створення програмного коду, непотрібних нововведень у систему тощо. Якщо ви хочете уникнути цих проблем, то потрібна умова – правильний вибір членів робочої групи.

Група, в якій співробітники доповнюють один одного, може працювати набагато ефективніше за групу, відбір в яку проводився виключно на основі навичок програмування. Люди, які люблять свою роботу (цільова орієнтація) можуть стати чудовими професіоналами. Люди із само орієнтацією на найкращий результат зможуть довести справу до кінця. Співробітники із зовнішньою орієнтацією успішно налагоджують спілкування всередині групи. Є думка [22], що співробітники із зовнішньою орієнтацією – необхідна складова будь-якої робочої групи. Вони налаштовані на спілкування і тому можуть визначити (і запобігти) виникненню будь-яких напруження та конфліктів на ранній стадії. Саме такі люди допоможуть вирішити особисті проблеми членів команди, перш ніж ті вплинуть на всю команду.

Іноді просто неможливо організувати команду так, щоб її члени взаємодоповнювали один одного за особистими якостями. Якщо таке трапляється, менеджер проекту зобов’язаний контролювати групу для того, щоб не дозволити особистим цілям співробітників стати вище за мету організації або групи. Дося-

гти такого контролю набагато легше, якщо члени команди братимуть участь у кожному етапі проекту. Прояв індивідуальної ініціативи буде вірогіднішим, якщо з членами групи проводити інструктаж, не згадуючи тієї ролі, яку їхня частина роботи грає у всьому проекті.

Наприклад, програміст, який отримав проект програми для кодування, бачить, що в ПЗ необхідно внести зміни, які роблять його досконалішим. Якщо впроваджувати ці зміни без розуміння логіки вихідного проекту, то це може призвести до конфлікту з іншими частинами ПС. Якщо вся група включена в розроблення проекту із самого початку, працівник зможе оцінити наслідки внесення таких змін до системи і це не призведе до конфлікту з тими фахівцями, які її проектували.

Важливе місце у команді займає лідер (*англ. Team Leader*). Він (або вона) відповідає за технічне керівництво та адміністративне управління. Лідери групи повинні бути в курсі її повсякденної діяльності, гарантуючи ефективну роботу команди та тісне співробітництво з менеджерами проекту під час планування діяльності з його реалізації.

Лідер – це, як правило, посада, він підзвітний головному менеджеру проекту. Керівник проекту (головний менеджер) може бути і лідером команди у сенсі цього терміну, –він веде групу лише у технічних питаннях. Члени групи можуть вибрати іншого лідера команди. Він може краще за призначеного лідера розбиратися в технічних питаннях або краще мотивувати членів групи до виконання роботи.

Іноді буває корисним взагалі розділити технічне лідерство та управління проектом. Часто трапляється, що компетентні у технічних питаннях працівники слабо розуміються на адмініструванні. Виконуючи роль адміністратора, такі лідери вже не можуть робити повноцінний внесок у технічну роботу команди. У цій ситуації краще призначити адміністратора, котрий звільнить лідера від щоденних адміністративних проблем.

Якщо в групу призначається лідер, який є неприйнятним для членів команди, це може призвести до досить напруженої ситуації. Члени групи не поважатимуть нового лідера і можуть віддати перевагу своїм інтересам на проти-

вагу інтересам групи. Ця проблема дуже істотна для такої області зі швидким розвитком, як технології розроблення ПЗ, де нові члени команди можуть володіти більш сучасними знаннями і досвідом, ніж лідери груп, які мають тільки практичний досвід. Крім того, деяких досвідчених співробітників може образити призначення молодого лідера, навіть якщо він здатний внести в роботу нові ідеї.

Згуртованість команди. Члени згуртованої команди віддані її інтересам більше, ніж своїм власним. У них високо розвинений командний стиль та ототожнення цілей групи з особистими інтересами. Вони також сприймають групу як єдину сутність і намагаються захистити її від зовнішнього втручання. Це зміцнює групу, вона стає здатною самостійно справлятися з проблемами та непередбачуваними ситуаціями. Члени групи підтримують одне одного у процесі різних змін, що допомагає їм долати труднощі.

Добре згуртована команда має низку переваг [22]:

1) Можливість становлення стандарту якості групи. Оскільки цей стандарт визначається всією групою одноголосно, його легше контролювати, ніж чужі стандарти, що нав’язуються групі ззовні.

2) Члени команди підтримують тісні контакти. Працюючи у групі, люди навчаються одне в одного. Скутість та затягування роботи, викликані незнанням чи непоінформованістю, зменшуються у міру того, як відбувається взаємне навчання.

3) Члени команди ознайомлені із діяльністю один одного. Цим досягається можливість продовження роботи навіть після звільнення одного зі співробітників.

4) Можливе впровадження в практику групи безособового програмування. Створена програма має бути власністю всієї команди, а не окремої особи.

Безособове програмування означає певний стиль роботи, при якому проекти, програми та документація вважаються власністю всієї групи, а не окремої людини, яка займалася їх розробленням. Якщо розробники сприймають свою роботу саме так, вони охочіше віддають її на перевірку іншим членам групи, легко сприймають критику, прагнуть працювати над удосконаленням системи.

При цьому команда розробників стає більш згуртованою, тому що кожен почувається відповідальним за розроблення всієї системи.

Крім того, що безособове програмування покращує якість системної архітектури, програм та документації, воно також удосконалює взаємовідносини всередині групи, стимулює невимушене обговорення завдання, незалежно від соціального становища, досвіду та статі. У ході виконання проекту члени групи активніше підтримують робочі відносини між собою. Усі ці чинники сприяють згуртованості. Саме тому кожен працівник сприймає себе частиною команди.

Багато факторів можуть вплинути на згуртованість групи, у тому числі організаційна культура та вираження особистісних якостей у групі. Менеджери можуть розвивати згуртованість кількома шляхами. Можна організовувати соціальні заходи для працівників та їхніх сімей. Можна прищепити групі почуття самобутності, для цього її треба назвати, визначити сутність команди та сферу її діяльності. Менеджери повинні проводити заходи (наприклад, ігри та спорт), спрямовані на створення команди.

Проте, найкращий спосіб виховати дух команди – дати можливість кожному відчути, що він несе певну частку відповідальності, показати, що йому довіряють, а також гарантувати доступ до проектної інформації для всіх членів групи. Іноді менеджерам здається, що вони не повинні розкривати певну інформацію. Проте така лінія поведінки постійно створюватиме у групі почуття недовіри. Простий обмін інформацією – найдешевший та найефективніший спосіб дати людям відчути себе частиною команди.

Незважаючи на все, у сильних та згуртованих групах також можуть виникнути проблеми [22]:

1) Неусвідомлений опір зміні лідера. Якщо лідера згуртованої групи потрібно замінити кимось зі сторони, члени групи можуть спільно виступити проти нового лідера. Працівники витрачають багато часу, чинячи опір змінам і нововведенням, пропонованим новим лідером, при цьому поступово падає продуктивність праці. Тому, по можливості, слід призначати нового лідера зі складу групи.

2) Групова думка. Це назва ситуації, в якій цінні робочі якості членів групи руйнуються внаслідок відданості команді. Висунення альтернативних пропозицій замінюється прихильністю до норм і рішень групи. Будь-яка ідея приймається, якщо більшості це вигідно, альтернативи при цьому не розглядаються.

Щоб уникнути проблеми групової думки, потрібно організовувати офіційні засідання, де члени команди будуть змушені так чи інакше висловлювати власні думки. Можна запросити незалежних експертів для аналізу рішення групи. Команда повинна складатися з людей, які здатні обговорювати, ставити питання і не звертати особливої уваги на усталене становище. Під час дискусій такі люди виступають у якості запеклих сперечальників, постійно ставлячи під сумнів рішення групи, спонукаючи цим членів групи обмірковувати і оцінювати свої дії.

Спілкування у групі. Для групи з розроблення ПП необхідний розвинений комунікаційний чинник, тобто спілкування та хороші засоби зв'язку між членами групи. Працівники повинні інформувати один одного про те, як іде їхня робота, про рішення, що робилися щодо проекту, та ті необхідні зміни, які вносилися до попередніх розпоряджень. Постійне спілкування також сприяє згуртованості, оскільки працівники краще починають розуміти мотивацію своїх колег, їхні слабкі та сильні сторони.

На ефективність спілкування можуть вплинути наступні показники [22]:

1) Розмір групи. Чим більша група, тим важче забезпечити постійне спілкування між її членами. Показник односторонніх зв'язків групи обчислюється за формулою (n - 1), де n – розмір групи. З цього можна зрозуміти, що у групі з 7-8 чоловік може бути людина з низьким показником зв'язків. Відмінність у соціальному становищі членів групи призводить до появи більшої кількості односторонніх зв'язків. Члени групи з вищим становищем у суспільстві частіше домінують у спілкуванні зі своїми колегами, які стоять нижче. Останні, у свою чергу, неохоче йдуть на контакт або висловлюють критичні міркування.

2) Структура групи. Працівники, які перебувають у групах із неформальною структурою, легше спілкуються між собою, ніж у групах, які мають певну

офіційну ієрархію у відносинах. В останніх спілкування відбувається чітко в ієрархічній послідовності (у той чи інший бік). Співробітники на одній і тій самій ієрархічній сходинці можуть зовсім не спілкуватися між собою. Це в основному проблема широкомасштабних проектів, що включають декілька груп розроблення ПП. Якщо у такому проекті співробітники різних відділів спілкуються лише через своїх менеджерів, це може призвести до запізнення у проведенні робіт та нерозумінні один одного.

3) Склад групи. Якщо в групі багато людей зі схожими особистісними характеристиками, вони можуть конфліктувати один з одним, унаслідок чого може значно знизитися рівень спілкування в групі. Найкраще люди спілкуються у змішаних різностатевих групах, ніж у однорідних по статі. Серед жінок переважає зовнішня орієнтація, у групі вони є свого роду регуляторами взаємовідносин, полегшуючи спілкування.

4) Робоче оточення. Правильна організація робочого місця – основний чинник у розвитку чи гальмуванні комунікаційних зв’язків у групі.

Організація групи. Невеликі команди програмістів найкраще організовувати у легкій неформальній атмосфері. Лідер бере участь у роботі над ПП поруч із іншими членами групи. Технічним лідером може стати людина, яка найкраще керуватиме процесом розроблення. Неформальна група завжди обговорює майбутню роботу з усіма членами колективу, а завдання призначаються відповідно до можливостей та досвіду конкретного співробітника. Високоврівневе проектування системи здійснюється старшими спеціалістами, проектування низького рівня надається тому співробітнику, який призначений на виконання конкретного завдання.

Робота неформальних груп може виявитися надзвичайно ефективною, якщо більшість членів групи досвідчені і кваліфіковані фахівці. Група функціонує як демократична система, де рішення приймаються більшістю голосів. У психологічному плані це породжує дух команди і тим самим призводить до зміцнення згуртованості та підвищення продуктивності праці. Неформальна організація групи може надати «ведмежу послугу», якщо група складається з недосвідченого або некомпетентного персоналу. Виникає нестача керівної ланки,

здатної керувати роботою, що призводить до нестачі координації робіт і, можливо, провалу проекту.

Вважається, що найбільше впливають на продуктивність праці особисті якості співробітників. Щоб використати висококваліфікований персонал із найбільшою віддачею, багато фахівців пропонують будувати групу навколо одного висококваліфікованого провідного програміста. Основний принцип такої організації полягає в тому, щоб компетентний та досвідчений співробітник відповідав за розроблення всього ПП. Провідного програміста не слід завантажувати рутинною роботою, йому навпаки потрібна хороша підтримка у вирішенні питань адміністративного та технічного плану. Такого співробітника також слід позбавити від зайвого спілкування з фахівцями поза групою (рис. 2.30).

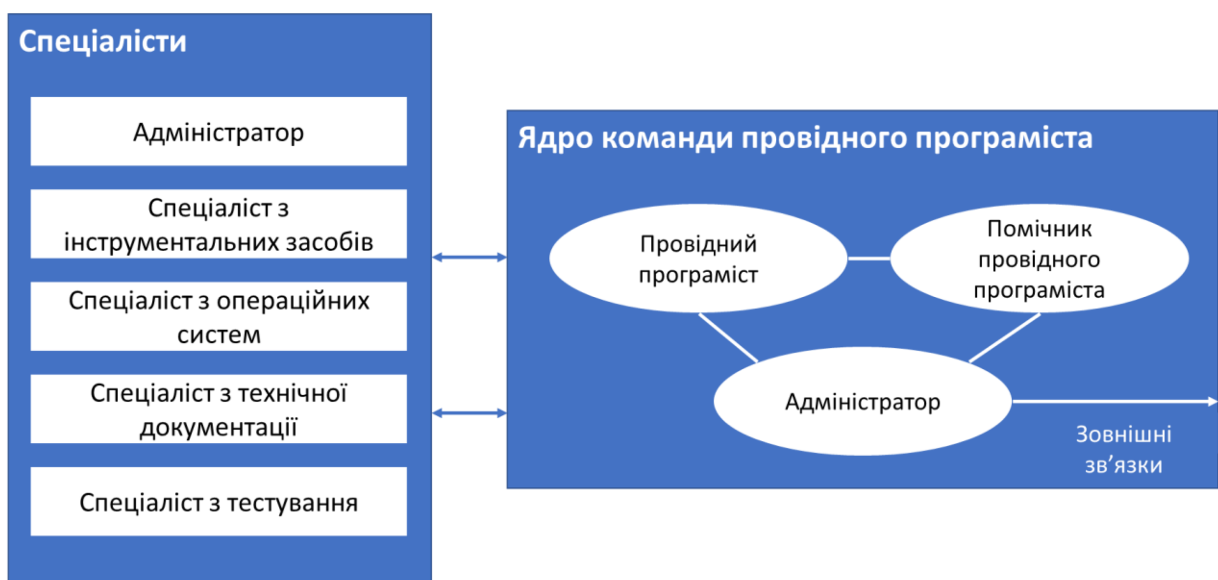

Рис. 2.30. Структура команди з провідним програмістом

Основними членами (ядром) команди провідного програміста є такі особи [22]:

1) Провідний програміст, який бере на себе основну відповідальність за розроблення, програмування, тестування та впровадження системи.

2) Досвідчений помічник (заступник) провідного програміста, чия роль полягає у підтримці провідного програміста та атестації ПС.

3) Адміністратор, який приймає на себе всю канцелярську роботу, пов'язану з проектом (наприклад, управління конфігурацією, заключні процедури з документацією, тощо).

Залежно від типу та розміру прикладного завдання, із сукупності фахівців можуть бути запрошені професіонали в якості тимчасового або постійного персоналу для роботи в команді. Це можуть бути адміністратор, спеціаліст з інструментальних засобів розроблення ПЗ, спеціаліст з операційних систем або мов програмування, спеціаліст з тестування систем.

Обґрунтуванням такого підходу може бути наступне твердження: скільки фахівців із розроблення ПЗ, стільки відмінностей у здібностях до програмування. Рівень продуктивності праці (за умовною шкалою від найкращих до найгірших програмістів) може різнитися у 10 разів і більше. Саме тому потрібно з найбільшою ефективністю використати можливості найкращих працівників, забезпечивши їм оптимальну підтримку. Хоча ідеї команди провідного програміста вже понад 25 років, вона все ще залишається одним із ефективних способів організації невеликих груп програмістів.

Якщо є можливість відбору потрібних людей, організація групи за таким принципом приведе до успіху. Проте й у таких групах можуть виникати проблеми, наприклад [1, 22]:

1) Талановиті розробники та програмісти зустрічаються нечасто. А організація групи заснована на компетентному провідному програмісті та його помічнику. Якщо вони роблять помилки, їх рішення немає з ким обговорити. У демократичній групі, навпаки, кожен може обговорити рішення і таким чином виявити помилки та уникнути їх.

2) Провідний програміст відповідає за повне виконання проекту і може також взяти на себе заслуги у разі успіху. Однак члени групи можуть з цим не погодитися, якщо їхня роль у проекті не буде визнана достатньою мірою. У такому разі незадоволені їх потреби у оцінці, оскільки всі досягнення будуть приписуватися лише провідному програмісту.

3) Можливість невиконання проекту у разі хвороби чи звільнення провідного програміста та його заступника. Керівники проектів можуть не погодитись на такий ризик.

4) Організаційна структура підприємства може виявитися не здатною забезпечити подібний тип групи. Великі компанії зазвичай мають добре розвинену службову ієрархію, тому призначення провідного програміста зі сторони може виявитися досить важким завданням. А у компаніях невеликого розміру практично неможливо виділити одного співробітника на виконання єдиного завдання.

Тому структура груп із провідним програмістом може бути для організації дуже ризикованою ідеєю. Проте з неї можна винести щось справді корисне: необхідно підтримувати талановитих програмістів, виділяючи їм помічників, адміністраторів, тощо. Таким чином, можна використовувати здібності обдарованих співробітників найбільш ефективно. Призначення вузькоспеціалізованого спеціаліста на короткі проміжки часу в окремі групи розробників може стати ефективнішим за використання програміста з великим досвідом протягом тривалого часу в роботі над одним проектом.

2.7.5 Підбір та збереження персоналу

Одним з основних функціональних обов'язків менеджера є підбір персоналу. Тільки у виняткових випадках менеджери мають право призначати найбільш підходящих для певної роботи співробітників незалежно від обов'язків, які вони виконують у даний момент, або від фінансових можливостей компанії. Здебільшого менеджер проекту не має права остаточного вибору персоналу. Він може бути змушений приймати в команду розробників будь-якого співробітника, який є в компанії і більш-менш підходить для цього. Як правило, знайти потрібну людину необхідно в максимально короткий термін. Також вільному підбору персоналу може перешкоджати обмежений бюджет проекту. Саме останній фактор може стати значною перешкодою у наймі кваліфікованих (але дорогих) програмістів, які могли б працювати над проектом.

У табл. 2.10 наведено основні фактори, які можуть вплинути на рішення менеджера у виборі майбутнього персоналу, якщо він має відносну свободу. Важко уявити ці фактори в порядку їх значущості, оскільки вони залежать від галузі застосування ПЗ, виду проекту, а також від кваліфікації та досвіду майбутніх членів групи.

Таблиця 2.10. Фактори, які впливають на вибір персоналу

Фактор	**Пояснення**
Знання про сферу застосування ПЗ	Для того, щоб розробити ПС, яка добре функціонує, програміст повинен мати чітке уявлення про ту прикладну область, де буде застосовуватися розроблюване ПЗ
Досвід роботи на різних комп’ютерних платформах	Цей фактор може бути важливим при низькорівневому програмуванні, у загальному випадку він не є вирішальним
Знання мови програмування	Цей фактор застосовується для короткострокових проектів, коли просто не вистачає часу для вивчення нової мови
Освіта	Освіта є своєрідним показником тих основних знань і умінь, якими має володіти кандидат, а також його здатність до навчання. Цей показник стає менш значущим пропорційно до досвіду, який отримується в роботі над різними проектами
Комунікабельність	Цей фактор є досить важливим, тому що в процесі реалізації проекту програмістам потрібно буде спілкуватися в усній та письмовій формах з іншими фахівцями, менеджерами та споживачами.
Здатність до адаптації	Показником може бути різноманітний досвід, отриманий раніше. Цей фактор також може показати здатність до навчання

Продовження таблиці 2.10

Життєва позиція	Люди, які працюють над проектом, повинні любити свою роботу та прагнути отримувати нові знання та навички. Це дуже показовий фактор, проте його важко оцінити
Особистісні якості	Це дуже важлива ознака, але також важка в оцінюванні. Адже члени групи мають бути сумісні для спільної роботи. Немає окремого типу особистості, який більшою чи меншою мірою відповідає фахівцю в галузі інженерії ПЗ

Рішення про призначення нового співробітника в проекті ґрунтується на трьох видах інформації:

1) Інформація про освіту та практичний досвід, що надається кандидатом на посаду (резюме або автобіографія).

2) Інформація, що отримується під час інтерв’ювання кандидата.

3) Рекомендації від інших людей, які мають досвід спільної роботи з кандидатом.

Деякі компанії користуються різноманітним набором тестів для оцінювання кандидатів. Це може бути перевірка здатності працювати програмістом та психологічні тести. Такі тести спрямовані на створення психологічної карти особистості, визначають ставлення опитуваного до роботи певного типу та здатність її виконати. Деякі менеджери переконані в абсолютній марності таких тестів, інші вважають, що за допомогою тестів вони отримують корисну інформацію для відбору персоналу. Як згадувалося раніше, здатність вирішувати завдання належить до сфери побудови семантичних моделей, що саме собою займає багато часу. Тести на професійну придатність та психологічні завдання засновані більшою мірою на швидкості відповіді на запитання. Проте складно знайти переконливий доказ визначення здатності вирішувати практичні завдання на основі пройдених тестів на професійну придатність.

Якщо менеджери проектів стикаються з труднощами у підборі відповідного персоналу з потрібними здібностями та досвідом, вони змушені складати

команди з недосвідчених програмістів, що, у свою чергу, викликає певні проблеми, пов'язані з непоінформованістю у сфері застосування ПЗ або технологій, які використовуються в проекті. Часто причина виникнення такої ситуації полягає у тому, що у деяких організаціях технічно обдаровані співробітники швидко досягають вершини кар'єри. Для подальшого просування таким працівникам необхідний статус менеджера. Якщо переводити їх у категорію менеджерів, це буде означати втрату цінного технічно кваліфікованого персоналу. Щоб уникнути такої ситуації, деякі компанії заснували у своїй структурі два паралельні види діяльності – технічний та управлінський, що мають однакову значимість. Досвідчений технічний персонал цінується на такому ж рівні, як і менеджери. З просуванням кар'єри фахівець може орієнтуватися на технічний чи управлінський рід діяльності і переходити в той чи інший вид діяльності без втрати статусу і зарплати.

2.7.6 Робоче середовище

Організація робочого місця має виняткове значення для продуктивності праці та отримання задоволення від роботи. Психологічні дослідження довели, що на поведінку працівників впливають розміри кімнати, якість меблів, обладнання, температура, вологість, яскравість, якість світла, рівень шуму та можливість залишитись наодинці з собою. Поведінка членів групи розробників також залежить від архітектурних особливостей приміщення та можливостей доступних засобів зв'язку.

Керівнику, як головному менеджеру, дорого обійдеться невміння забезпечити хороші робочі умови для співробітників. Якщо люди почуваються некомфортно на роботі, то пропорційно незадоволеності зростає коефіцієнт текучості робочої сили. Це, у свою чергу, вимагатиме великих витрат коштів на наймання та навчання нового персоналу. Також може сповільнитись виконання проекту, оскільки не вистачатиме кваліфікованих спеціалістів.

Команди з розроблення ПП часто працюють у великих відкритих офісах, іноді з перегородками, і старшим за посадою співробітникам можуть надаватися окремі кабінети. Деякі дослідження показали, що відкриті приміщення, які

використовуються багатьма компаніями, є абсолютно непопулярними і непродуктивними. При цьому було також визначено ключові чинники оформлення робочого простору:

1) Усамітнення. Програмістам потрібне певне місце, де вони можуть сконцентруватися та працювати без будь-якого втручання.

2) Огляд зовнішнього світу. Люди вважають за краще працювати при денному світлі, маючи перед собою привабливий вигляд із вікна.

3) Індивідуалізація. Люди засвоюють різні звички в організації робочого процесу та мають різні думки щодо оформлення приміщення. Тут велике значення має здатність облаштовувати робоче місце так, щоб воно відповідало різним смакам та виражало індивідуальні риси.

Таким чином, людина за своєю природою вважає за краще працювати в окремому приміщенні, яке вона може оформити за своїм смаком та бажанням. Такі кабінети, на відміну від відкритих офісів, створюють сприятливішу обстановку та знижують кількість пауз у роботі. У відкритому приміщенні співробітнику неможливо сконцентруватися, як це можна зробити в тихій робочій обстановці. Наслідком зниження концентрації стає неодмінне падіння рівня продуктивність праці.

Отже, окремі приміщення програмістів істотно впливають на продуктивність. За результатами порівняльного аналізу продуктивності праці програмістів у різних робочих умовах з’ясувалося, що можливість сконцентруватися та зниження рівня сторонніх втручань у роботу значно підвищує робочу активність. З двох груп програмістів з однаковими здібностями, ті, які працювали в сприятливих умовах, виявилися вдвічі продуктивнішими, ніж ті, що знаходилися в найгірших приміщеннях.

Групи з розроблення ПЗ також потребують приміщення, де всі члени групи можуть зібратися разом і обговорити проект (чи то офіційне засідання, чи просто неформальна зустріч). Кімната для зустрічей повинна вміщувати всіх членів групи та забезпечувати їм необхідний ступінь усамітнення. Два види потреб, усамітнення та спілкування всередині групи, можуть здатися взаємови-

ключними. Подібний конфлікт потреб можна вирішити шляхом розміщення індивідуальних кабінетів навколо однієї великої кімнати для зустрічей.

Схожу модель організації офісів пропонує Кент Бек в книзі [23]. Однак він все ж таки наполягає на збереженні відкритого типу офісів для спільної роботи з перегородками для тих співробітників, які хотіли б попрацювати на самоті. Ключовим фактором тут є надання двох видів приміщення (загальних та індивідуальних) для того, щоб члени групи обирали робоче місце на свій розсуд.

Такий тип організації робочого приміщення допомагає співробітникам вирішувати проблемні питання та обмінюватися інформацією неформальним, але ефективним способом. Розглянемо, наприклад, ситуацію, коли компанія намагається боротися з таким явищем, як витрата часу співробітників за розмовами навколо кавоварки. Припустимо, що керівництво дало розпорядження забрати у них кавоварку, але після цього негайно почали надходити численні запити на допомогу в програмуванні. Виявляється, що у той час, коли співробітники пліткували за чашкою кави, вони заодно допомагали один одному у вирішенні проблемних питань. Цей випадок незаперечно доводить, що кімнати для неформальних зустрічей також необхідні компанії, як і кімнати для робочих засідань.

2.7.7 Модель оцінки рівня розвитку персоналу

Інститут інженерії програмного забезпечення (англ. *Software Engineering Institute, SET*) у США тривалий час займався програмою вдосконалення процесу розроблення ПЗ. Модель оцінки рівня розвитку (*англ. Capability Maturity Model, CMM*) є частиною цієї програми. Вона увібрала в себе найкраще з практики технологій розроблення ПЗ. У розвиток цієї моделі інститут також пропонує модель оцінки рівня розвитку персоналу (*англ. People Capability Maturity Model, P-CMM*). Її можна використовувати як основу стратегії управління людськими ресурсами в організації.

Подібно до СММ, модель P-CMM має п’ять рівнів (рис. 2.31):

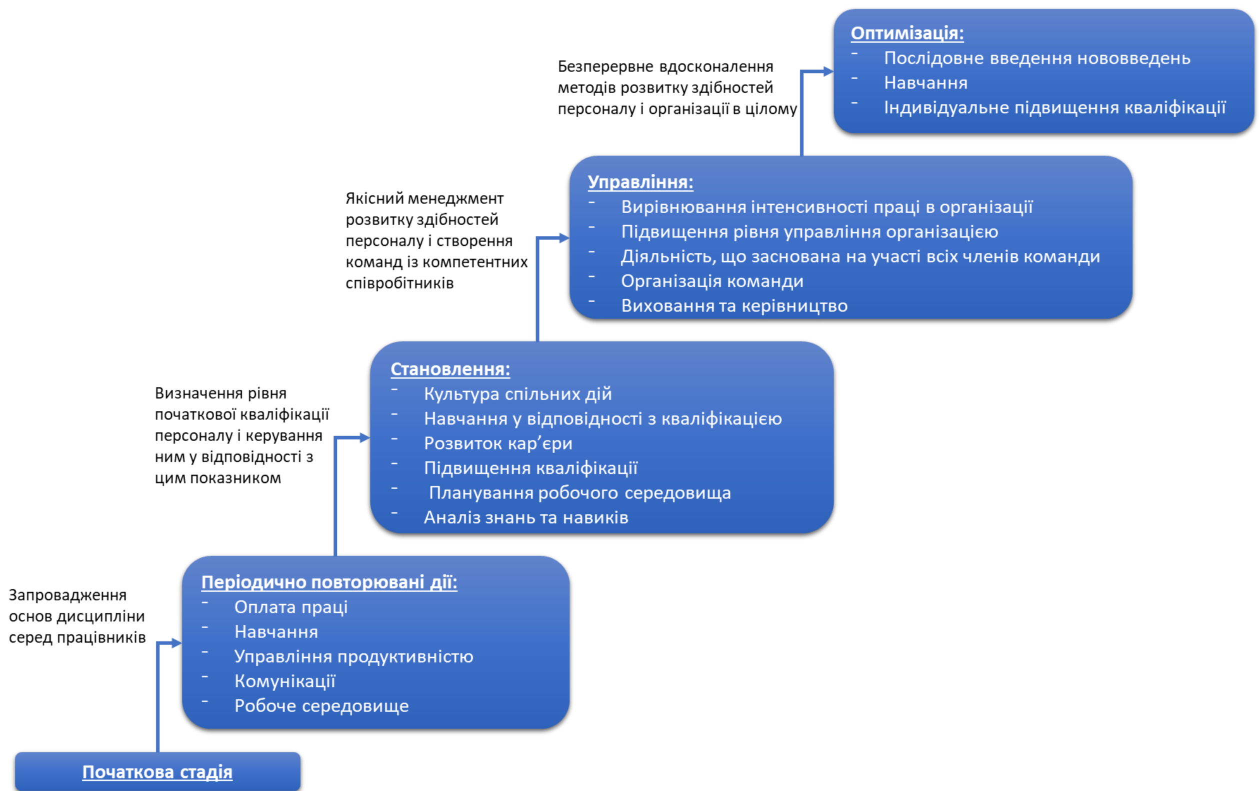

Рис. 2.31. Модель оцінки рівня розвитку персоналу

1) Початковий рівень. Практикується управління персоналом, модель якого не підпорядкована певним правилам.

2) Повторення. Проведення політики, спрямованої на розвиток здібностей персоналу.

3) Становлення. Уведення в компанії стандарту управління, заснованого на кращому досвіді управління персоналом.

4) Управління. Визначаються та вводяться кількісні цілі в управлінні персоналом.

5) Оптимізація. Центр уваги переноситься на безперервне підвищення кваліфікації та мотивації працівників.

Стратегічні цілі цієї моделі полягають у наступному:

1) Розширення можливостей компаній, які займаються розробленням ПЗ шляхом підвищення кваліфікації персоналу.

2) Гарантування того, що здатність високоякісного розроблення ПЗ є відмінною рисою всієї компанії, а не декількох людей, які цим займаються.

3) Забезпечення сумісності між мотивацією окремого індивідуума та мотивацією всієї компанії.

4) Збереження в компанії цінних людських ресурсів (наприклад, працівників, які мають рідкісні знання та навички).

Модель P-CMM – це відмінний та дієвий підхід до поліпшення якості управління персоналом, оскільки він дає основу для визнання, мотивування, нормалізації та вдосконалення досвіду в управлінні персоналом. Завдяки цьому менеджер усвідомлюватиме значущість працівника, як особистості, та необхідність його подальшого вдосконалення в професійній сфері.

Для застосування даної моделі в повному обсязі знадобиться значна кількість коштів, що не є рентабельним для багатьох компаній. Однак дана модель у будь-якому випадку є відмінним посібником, який допоможе багатьом компаніям удосконалити свої можливості у розробленні ПЗ високої якості.

Отже, можна зробити загальні висновки щодо управління персоналом у ході ЖЦ ПЗ:

1) Управління розробленням ПЗ безпосередньо пов’язане з управлінням людьми. Саме тому менеджер програмного проекту повинен мати певне уявлення про психологію та суб’єктивні фактори людської діяльності, щоб не ставити нездійсненних вимог для себе і для персоналу.

2) Людська пам’ять поділяється на короткочасну, проміжну та довготривалу. Знання можуть бути синтаксичними та семантичними. Процес вирішення проблеми включає інтеграцію семантичної інформації, яка зберігається в довгостроковій пам’яті, і нової інформації, яка надходить з короткочасної пам’яті.

3) Знання предметної області, у якій застосовуватиметься створюване ПЗ, здатність адаптуватися і особистісні якості є ключовими чинниками під час підбору персоналу.

4) Групи з розроблення ПЗ повинні бути невеликими та добре згуртованими. Лідери таких груп повинні мати високу технічну кваліфікацію та забезпечуватися адміністративною та технічною підтримкою усіх інших.

5) На взаємовідносини всередині групи впливають багато факторів, у тому числі: соціальне становище членів групи, її розмір, розподіл персоналу за статевою ознакою, особисті якості, можливість спілкування, тощо.

6) Шляхом забезпечення належних умов роботи, які включають наявність відповідних обчислювальних та комунікаційних засобів, можна підвищити продуктивність праці та почуття задоволеності роботою.

7) Модель оцінки рівня розвитку персоналу P-CMM забезпечує загальну основу та механізм підвищення продуктивності праці персоналу, а також можливість отримувати найкращий результат від інвестицій у людські ресурси.

Контрольні запитання

1. Чим відрізняється процес розроблення ПЗ від реалізації технічних проектів?

2. Який перелік робіт необхідно виконувати менеджеру програмного проекту?

3. Хто є основним менеджером проекту? Які завдання він виконує?

4. Що таке проектний (залізний) трикутник? Як він визначає якість ПЗ?

5. Які основні типи планів необхідно розробити менеджеру програмного проекту?

6. Які розділи входять до плану проекту?

7. Чим контрольні позначки (точки) відрізняються від контрольних проектних елементів?

8. Як складається графік робіт проекту?

9. Чим мережева діаграма відрізняється від діаграми Ґанта?

10. Що таке критичний шлях? Як він визначається?

11. У чому полягає суть часової діаграми зайнятості?

12. Що таке вимоги? Які є дві категорії вимог?

13. Що таке специфікація вимог? З чого вона складається?

14. Які є два основних типи вимог?

15. Що таке функціональні вимоги?

16. Які три частини має містити специфікація функціональних вимог у загальному випадку?

17. Які вимоги є нефункціональними? На які групи вони поділяються?

18. Які характеристики ПЗ входять до нефункціональних вимог?

19. Як виразити нефункціональні вимоги за допомогою кількісних показників?

20. Що таке вимоги предметної області? Чим вони відрізняються від функціональних та нефункціональних?

21. Які є методи визначення вимог? Чим вони відрізняються один від одного?

22. Які питання необхідно задавати на інтерв'ю з замовником?

23. Як проходить мозковий штурм?

24. Як проходить сеанс спільного розроблення додатків?

25. У чому полягає суть процесу розкадрування? Які є типи розкадровок?

26. Як проходить обігравання ролей?

27. Що таке формальна специфікація вимог? Як її розробити?

28. Що таке прототип? На якому етапі процесу розроблення ПЗ він є корисним?

29. Які переваги та недоліки використання прототипів ПС?

30. У чому полягає суть еволюційного прототипування?

31. У чому полягає суть експериментального прототипування?

32. Які є технології швидкого прототипування?

33. Як відбувається прототипування інтерфейсів користувача?

34. Якими трьома типами можна представити людську пам'ять?

35. Як проходить процес рішення задач?

36. Як підвищити мотивацію співробітників у відповідності до ієрархічної структури людських потреб?

37. Як правильно створити команду під час групового розроблення ПЗ?

38. Як підвищити згуртованість команди під час групового розроблення ПЗ?

39. Чому спілкування в групі є важливим під час групового розроблення ПЗ?

40. Як правильно організувати групу під час групового розроблення ПЗ?

41. Як правильно підібрати та зберегти персонал під час групового розроблення ПЗ?

42. Як впливає робоче середовище на групову роботу?

43. У чому полягає суть моделі оцінки рівня розвитку персоналу?

РОЗДІЛ 3

АРХІТЕКТУРНЕ ПРОЕКТУВАННЯ

3.1 Основи архітектурного проектування

Під словосполученням «архітектурне проектування» найчастіше мається на увазі розроблення детальної схеми перетворення вимог у готове ПЗ.

Проектування є тим процесом, який пов'язує визначення вимог з кодуванням і налагодженням програмної системи (ПС) – набору взаємозалежних компонентів у вигляді ПЗ, яке входить до складу комп'ютерної системи.

При роботі над невеликими проектами якісне проектування є корисним, а при роботі над великими – обов'язковим, тому що структура вдалого високорівневого проекту ПС може успішно охоплювати та включати в себе цілий ряд більш низькорівневих проектів.

Проектування завжди пов'язане з визначенням пріоритетів і компромісів. В ідеальному випадку всі ПС ніколи не містили б ніяких помилок, мали б нескінченну швидкодію, спричиняли б нульове навантаження на мережу, не висували б ніяких вимог до підсистеми зберігання даних і створювалися б без всяких витрат. Однак, у реальності одним з найважливіших аспектів роботи проектувальника є аналіз характеристик проекту (які не завжди ідеально узгоджуються між собою) і досягнення балансу між ними. Якщо швидкість відгуку ПС важливіша, ніж мінімізація часу її розроблення, проектувальник вибере один варіант. Якщо важливою є швидкість розроблення, то оптимальним може виявитися зовсім інший варіант.

Проектування обмежує можливості. Проектування передбачає не тільки забезпечення певних можливостей системи, але і їх обмеження. Наприклад, при конструюванні будівель обмежені обсяги ресурсів та технічних можливостей часто вимагають спрощення проекту. У кінцевому результаті це призводить до його поліпшення. Проектування ПЗ у загальному випадку нічим не відрізняється.

Проектування являється недетермінованим процесом. Якщо три різні архітектори спроектують одну і ту ж ПС, то скоріше за все, вони розроблять три абсолютно різних проекти. Проте кожен з цих проектів буде цілком прийнятним. Як правило, спроектувати ПЗ можна десятками різних способів, використовуючи різні шаблони та архітектурні рішення.

Проектування являється евристичним процесом. Як було сказано вище, проектування не є детермінованим процесом. Це означає, що використовувані методи найчастіше є наближеними. Отже, немає універсальних практичних правил або способів, які можуть спрацювати навіть у 75% проектів та привести до передбачуваних якісних результатів.

Проектування є методом спроб і помилок. Методи або інструменти проектування, які можуть бути ефективними в одному випадку, в іншому можуть призвести до значних проблем з ПС. Універсальних методик проектування ПЗ не існує.

Проектування є поступовим процесом. Проект ПЗ не виникає в уяві архітектора відразу в готовій формі. Кожен окремий проект розвивається і вдосконалюється під час обговорень з командою розроблення, написання коду, а також під час тестування. У переважній більшості випадків проект ПЗ дещо змінюється під час початкового розроблення системи, але найбільше – при її розширенні. Від особливостей створюваного ПЗ залежить чи буде ступінь прийнятих змін прийнятним.

Суттєві та несуттєві проблеми. Деякі програмісти стверджують, що складність розроблення будь-якого ПЗ пояснюється наявністю суттєвих та несуттєвих властивостей чи проблем.

Несуттєвими властивостями називають випадкові (другорядні, довільні, необов'язкові) властивості об'єкта – властивості, які не впливають на його сутність. Наприклад, літак може мати одну пару коліс на кожному шасі (рис. 3.1), дві пари, взагалі не мати шасі (наприклад, гідроплан) і все ж бути літаком. Тип двигуна, кількість крил, коліс чи дверей – усе це являється несуттєвими властивостями.

Рис. 3.1. Приклад несуттєвих властивостей

Деякі несуттєві проблеми розроблення ПЗ на сьогоднішній день вже давно вирішені. Наприклад, поступово втратили свою значимість через глобальну еволюцію мов програмування проблеми, які пов’язані з їх незручним синтаксисом. Інтегровані середовища розроблення позбавили велику кулькість програмістів від проблем, які були пов’язані з поганою взаємодією розрізнених інструментів. Проте, несуттєві проблеми розроблення ПЗ, які залишилися на сьогодні, потребують значно більшого часу на їх вирішення. Це пояснюється тим, що розроблення ПЗ вимагає досконалого аналізу всіх деталей дуже складного набору пов’язаних між собою концепцій.

У філософії суттєвими називають властивості об’єкта, якими він повинен володіти, щоб бути саме цим конкретним об’єктом. Літак, наприклад, повинен мати двигун, крила і двері. Якщо об’єкт не має будь-якої з цих істотних властивостей, то він не являється літаком.

Причиною виникнення суттєвих проблем розроблення ПЗ є необхідність аналізу складного та хаотичного реального світу, точного і повного визначення взаємозв’язків між різними сутностями, проектування абсолютно вірних рішень, тощо. При цьому взагалі не допускаються ніякі припущення. Навіть якби

існувала мова програмування з технологіями та термінологією проблемної області реального світу, розроблення ПЗ все одно було б складним завданням через необхідність точного визначення принципів функціонування реального світу. Вирішення все більш складних проблем реального світу призводить до необхідності аналізу все більш складної взаємодії між окремими сутностями. Це призводить до суттєвого підвищення складності ПС.

Отже, основним джерелом практично всіх суттєвих та несуттєвих проблем розроблення ПЗ є складність реального світу.

Важливість управління складністю. Як було сказано раніше, пов'язані з розробленням ПЗ проекти рідко вважаються провальними через технічні причини. Найчастіше провал проекту пояснюється неадекватним формуванням вимог, невдалим плануванням або неефективним управлінням, тобто проблемами управління. У випадках, коли провал проекту все-таки зумовлений переважно причинами технічного характеру, дуже часто цією причиною виявляється неконтрольований ріст складності проекту в процесі розроблення. Інакше кажучи, ПЗ стало настільки складним, що розробники перестали розуміти, що ж воно робить. Якщо робота над проектом досягає такого моменту – прогрес припиняється.

Управління складністю – це один з основних технічних аспектів розроблення ПЗ. Едсгер Дейкстра вже давно звертав увагу на те, що «комп'ютерні технології – це єдина галузь, яка змушує людський розум охоплювати діапазон, який тягнеться від окремих бітів до декількох сотень мегабайт інформації, що відповідає відношенню 1 до 10^9 або різниці в дев'ять порядків» (Dijkstra, 1989). З 1989 року складність розроблюваного ПЗ постійно зростала. Сьогодні вислів Дейкстри може характеризуватися не дев'ятьма, а навіть п'ятнадцятьма порядками і більше. Дейкстра також казав, що «жодна людина не володіє інтелектом, здатним вмістити всі деталі сучасної комп'ютерної програми» (Dijkstra, 1972), тому розробникам ПЗ не слід навіть намагатися охопити відразу всю програму. Замість того необхідно організовувати розроблення ПЗ так, щоб можна було безпечно працювати з її окремими фрагментами по черзі. Метою цього процесу

є мінімізація задачі, над рішенням якої потрібно думати в конкретний момент часу.

На рівні архітектури ПЗ складність проблеми можна знизити, провівши декомпозицію системи на окремі підсистеми. У декомпозиції складної задачі на більш прості частини і полягає основне завдання всіх сучасних методик проектування ПЗ. Чим більш незалежними будуть підсистеми, тим простіше та безпечніше зосередитися лише на одній задачі в конкретний момент часу. Добре визначені на етапі проектування ролі та об'єкти розділяють проблемні задачі так, щоб їх можна було вирішити по черзі.

Прагнення архітекторів та програмістів зробити функції (методи) розроблюваного ПЗ компактними та стислими допомагає знизити навантаження на інтелект розробника та персоналу супроводу. Цьому ж сприяє написання ПЗ у термінах проблемної області, а також використання самого високого рівня абстракції. Як наслідок, програмісти можуть компенсувати початкові обмеження людського розуму завдяки написанню більш зрозумілого коду, який містить меншу кількість помилок.

Найчастіше причинами неефективності є: просте, але неправильне рішення складної проблеми; складне рішення простої проблеми; неадекватне та складне рішення складної проблеми.

Як було сказано вище, складність сучасного ПЗ обумовлена його природою та природою реального світу. Це призводить до того, що у будь-якому випадку виникне складність, яка притаманна самій проблемній області реального світу. Саме тому підхід до управління складністю розроблюваного ПЗ має бути адаптивним: необхідно зводити до мінімуму обсяг складності суттєвих задач, з яким доведеться працювати в кожен момент часу; необхідно стримувати необов'язкове значне зростання складності несуттєвих задач. Необхідно зрозуміти та запам'ятати, що всі інші технічні задачі розроблення ПЗ вторинні у порівнянні з управлінням складністю.

3.2 Бажані характеристики проекту

Високоякісні проекти з розроблення ПЗ мають декілька загальних характеристик. Досягнення всіх наведених нижче цілей призведе до створення дуже якісного ПЗ.

Майже в будь-якому ПЗ є задачі, які суперечать одна одній. Саме тому однією з задач проектування архітектури є об'єднання конкуруючих задач у вдалому наборі компромісів. Деякі характеристики ПЗ, наприклад, надійність та продуктивність, описують його якість, тоді як інші є внутрішніми характеристиками або елементами бізнес-логіки. Розглянемо більш детально ряд внутрішніх характеристик проекту.

Мінімальна складність. Як було неодноразово сказано вище, головною метою проектування повинна бути мінімізація складності. Необхідно уникати розроблення заплутаних проектів. Замість цього необхідно створювати «прості» та «зрозумілі» проекти. Якщо при роботі над окремою частиною ПЗ проект не дозволяє безпечно ігнорувати більшість інших частин, то він скоріше за все є невдалим.

Простота супроводу. Проектуючи ПЗ, не треба забувати про персонал супроводу. Необхідно думати трохи наперед і уявляти які питання можуть виникнути при погляді на написаний код. Саме тому треба проектувати систему так, щоб її робота була очевидною та однозначною.

Слабке зчеплення (*англ., coupling*)**.** Воно передбачає зведення до мінімуму числа зв'язків між різними частинами ПЗ. Для проектування модулів з мінімальним числом зовнішніх взаємозв'язків необхідно використовувати принципи абстракції інтерфейсів, спадковість та інкапсуляцію інформації. Це дозволить максимально полегшити тестування, інтеграцію та супровід ПЗ.

Можливість розширення. Розширюваністю ПС називають властивість, яка дозволяє покращити систему, не порушуючи її основної структури та наявної функціональності. При такому підході зміна однієї частини ПС не повинна впливати на її інші частини. Внесення змін повинно вимагати найменших зусиль.

Можливість повторного використання. Будь-яку систему необхідно розбивати на структурні частини (модулі, підсистеми, тощо) таким чином, щоб їх можна було використати в інших проектах.

Можливість легкого перенесення на інші платформи. ПС необхідно проектувати таким чином, щоб її легко можна було адаптувати до іншого програмного середовища.

Мінімальна, але повна функціональність. Як правило, ПЗ вважається закінченим не тоді, коли в нього не можна більше нічого додати, а коли з нього не можна нічого прибрати. Це висловлювання є актуальним тому, що будь-який додатковий код необхідно спочатку розробити та проаналізувати, потім протестувати, переглядати кожного разу при зміні пов'язаних частин ПЗ, а також доведеться підтримувати зворотну сумісність у майбутніх версіях ПЗ.

Стратифікація. Стратифікація – це поділ рівнів декомпозиції, що дозволяє вивчити систему на будь-якому окремому рівні і отримати при цьому узгоджене уявлення. Це означає, що систему необхідно проектувати так, щоб її можна було вивчати на окремих рівнях, повністю ігноруючи інші рівні. Наприклад, при створенні сучасної системи, яка повинна використовувати великий обсяг погано спроектованого старого коду, необхідно створити проміжний рівень, який відповідає за взаємодію нового коду зі старим. Проміжний рівень у цьому разі має бути спроектованим таким чином, щоб він приховував погану якість старого коду і надавав узгоджений набір сервісів новим рівням системи. Такий підхід має декілька переваг: 1) ізолювання поганого коду; 2) у разі необхідності рефакторингу (покращення коду) чи повній заміні старого коду не потрібно буде змінювати новий код за винятком створеного проміжного рівня.

Відповідність стандартним методикам. Чим більш нестандартною є розроблювана програмна система, тим складніше іншим програмістам (наприклад, персоналу супроводу) в ній розібратися. Застосовуючи стандартні підходи, наприклад, шаблони чи патерни, можна надати системі звичний для розробників вигляд.

3.3 Системна архітектура

Системна архітектура [24] – це структура обчислювальної системи, що визначає її роботу на найвищому концептуальному рівні, включаючи апаратні і програмні підсистеми, видимі зовні властивості цих підсистем, відносини між ними, а також документування структури, властивостей та відносин. Завдяки затвердженню прийнятих на ранніх етапах проектування рішень про високорівневий дизайн ПС документування архітектури спрощує процес взаємодії між усіма учасниками проекту. Це також дозволяє зберегти та повторно використовувати елементи дизайну у інших проектах, як шаблони.

Для розроблення архітектури системи залучаються фахівці з наступними ролями: системний архітектор (проектує систему в цілому, а також окремі її компоненти), архітектор бази даних (займається проектуванням бази даних і її структури), системний аналітик (бере участь у проектуванні, готує документацію), адміністратори (беруть участь у проектуванні апаратної частини системи).

На архітекторів системи покладається велика відповідальність. Якщо розроблена архітектура не буде реалізовувати поставлені замовником вимоги, то це, наприклад, може збільшити терміни виконання проекту (за рахунок того, що необхідно буде робити доопрацювання та виправлення недоліків архітектури), а отже, знизити прибуток за розробку.

Як зазначалося вище, архітектурне проектування системи – перший етап процесу проектування, на якому розробляється базова структура системи, тобто визначаються основні підсистеми, а також їх структура керування і шляхи взаємодії між собою. Модель системної архітектури часто є відправною точкою для створення специфікації різних частин системи.

Підсистема – це окрема система, функції якої не залежать від сервісів, що надаються іншими підсистемами.

Сервіс – це закінчена сутність, яка виконує певний набір певних завдань у своїй області відповідальності та надає інтерфейс для взаємодії. Сервіс або мікросервіс з архітектурної точки зору можуть бути ідентичними поняттями з різницею лише в масштабі. Сервіс може складатися з набору мікросервісів, які

реалізують свої правила бізнес-логіки, або бути одним самостійним мікросервісом.

Підсистеми складаються з модулів і мають певні інтерфейси, за допомогою яких взаємодіють з іншими підсистемами.

Модуль – це зазвичай компонент підсистеми, який надає один або декілька сервісів для інших модулів. Модуль може використовувати сервіси, підтримувані іншими модулями. Як правило, модуль не розглядається як незалежна система. Модулі зазвичай складаються з ряду інших, більш простих компонентів.

Визначимо загальні для всіх процесів архітектурного проектування системи етапи [25]:

- **Структурування системи**. Проводиться декомпозиція ПС на сукупність незалежних (бажано) підсистем. Визначаються основні шляхи взаємодії між окремими підсистемами.
- **Моделювання керування**. Розробляється базова модель керування взаємовідносинами між окремими підсистемами.
- **Модульна декомпозиція**. Проводиться декомпозиція кожної підсистеми на окремі модулі. Визначаються типи модулів та основні шляхи взаємодії між окремими модулями.

Результатом процесу архітектурного проектування системи є документ, який відображає архітектуру системи. Він складається з набору графічних схем (блок-схем, структурних, функціональних схем, тощо) системи з відповідним описом. В описі обов'язково повинно бути вказано, з яких підсистем складається система і з яких модулів складається кожна підсистема. Графічні схеми моделей ПС дозволяють поглянути на архітектуру з різних сторін.

Проектування системної архітектури тісно пов'язане з вимогами до розроблюваної ПС.

3.3.1 Архітектурні моделі

Як правило, розробляються чотири архітектурні моделі [25]:

1. **Статична структурна модель.** У ній представлені підсистеми (та інші компоненти), які в подальшому розробляються незалежно одна від одної.

2. **Динамічна модель процесів.** У ній представлена організація протікання процесів під час роботи ПС.

3. **Інтерфейсна модель.** Визначає сервіси кожної підсистеми та інтерфейси, через які ці сервіси надаються.

4. **Модель відносин.** Зображає взаємовідносини між частинами системи, наприклад потоки даних між підсистемами.

Моделі архітектури можуть залежати від НВ до системи, наприклад, [25]:

1. **Продуктивність**. Якщо критичною вимогою є продуктивність системи, то необхідно розробити таку архітектуру, в якій за всі критичні операції відповідало б якомога менше підсистем з максимально обмеженою взаємодією між ними. Для зменшення взаємодії між компонентами краще не використовувати дрібні структурні елементи.

2. **Захищеність**. Для забезпечення цієї НВ архітектура повинна мати багаторівневу структуру, в якій найбільш критичні системні елементи захищені на внутрішніх рівнях, а перевірка безпеки цих рівнів здійснюється на більш високому рівні.

3. **Безпека**. У цьому випадку архітектуру слід спроектувати таким чином, щоб за всі операції, які впливають на безпеку ПС, відповідало якомога менше підсистем. Виконання даної НВ дозволяє знизити вартість розроблення і вирішує проблему перевірки надійності.

4. **Надійність**. У цьому випадку слід розробити архітектуру з включенням надлишкових компонентів. Надлишковість дає змогу замінювати і оновлювати компоненти, не перериваючи роботу системи.

5. **Зручність супроводу**. У цьому випадку архітектуру системи слід проектувати на рівні дрібних структурних компонентів, які можна легко змінювати. Підсистеми, які створюють дані, повинні бути відокремлені від підсистем, які використовують ці дані. Слід також уникати структури спільного використання даних.

На першому етапі процесу проектування архітектури ПС розбивається на декілька підсистем (рис. 3.2), які можуть взаємодіяти між собою. На самому абстрактному рівні архітектуру системи можна зобразити графічно за допомогою блок-схеми, в якій окремі підсистеми представлені окремими блоками. Часто кожну підсистему також можна розбити на декілька менших частин (модулів). На схемі ці частини зображуються прямокутниками всередині великих блоків (підсистем). Потоки даних і/або потоки керування між підсистемами позначаються стрілками. Така блок-схема дає найбільш загальне уявлення про структуру системи та її основний функціонал.

Проте лише схема не являється повноцінною архітектурою. Справедливим є твердження, що кожна архітектура може зображатися у вигляді схеми (блок-схеми, структурної чи функціональної схеми, тощо), але не кожна схема є архітектурою. Повноцінною архітектура стає лише в купі з документуванням структурних елементів, відносин, взаємозв'язків, тощо. Більш детально документування архітектурних рішень буде розглянуто пізніше.

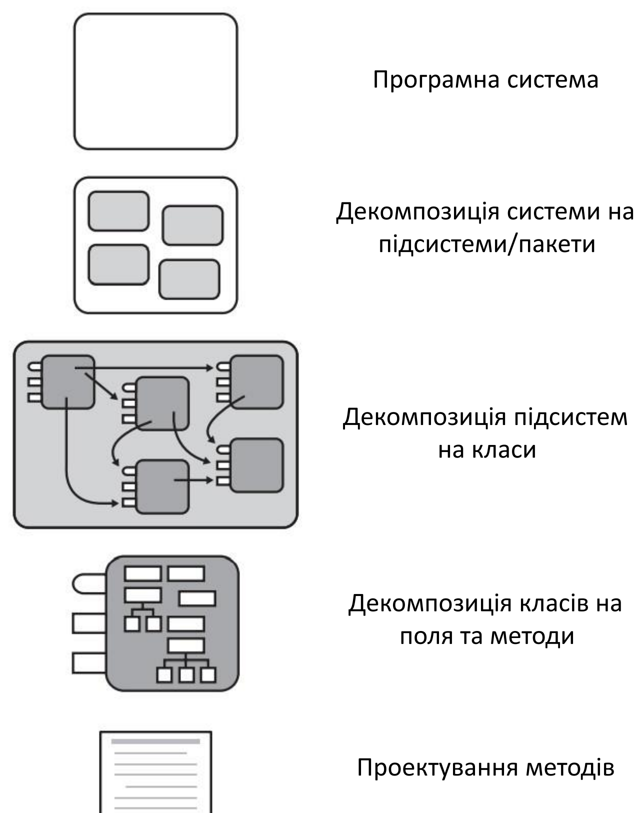

Рис. 3.2. Процес проектування системної архітектури

Проектування програмної системи вимагає декількох рівнів детальності. Деякі методи проектування архітектури використовуються на всіх рівнях, а інші – лише на одному-двох.

Рівень 1 (системний). На першому рівні проектування архітектури знаходиться вся ПС. У деяких випадках можна перейти з системного рівня відразу до проектування окремих класів, але найчастіше доцільно провести декомпозицію на підсистеми або пакети.

Рівень 2 (декомпозиція на підсистеми або пакети). Суть проектування полягає в поділі ПС на основні підсистеми (рис. 3.3). Зазвичай цей рівень наявний в проектах, які тривають більше декількох тижнів. При проектуванні окремих підсистем необхідно для кожної окремо обирати оптимальний підхід та методики. Не обов'язково застосовувати один і той же підхід для всіх підсистем, якщо є більш оптимальний.

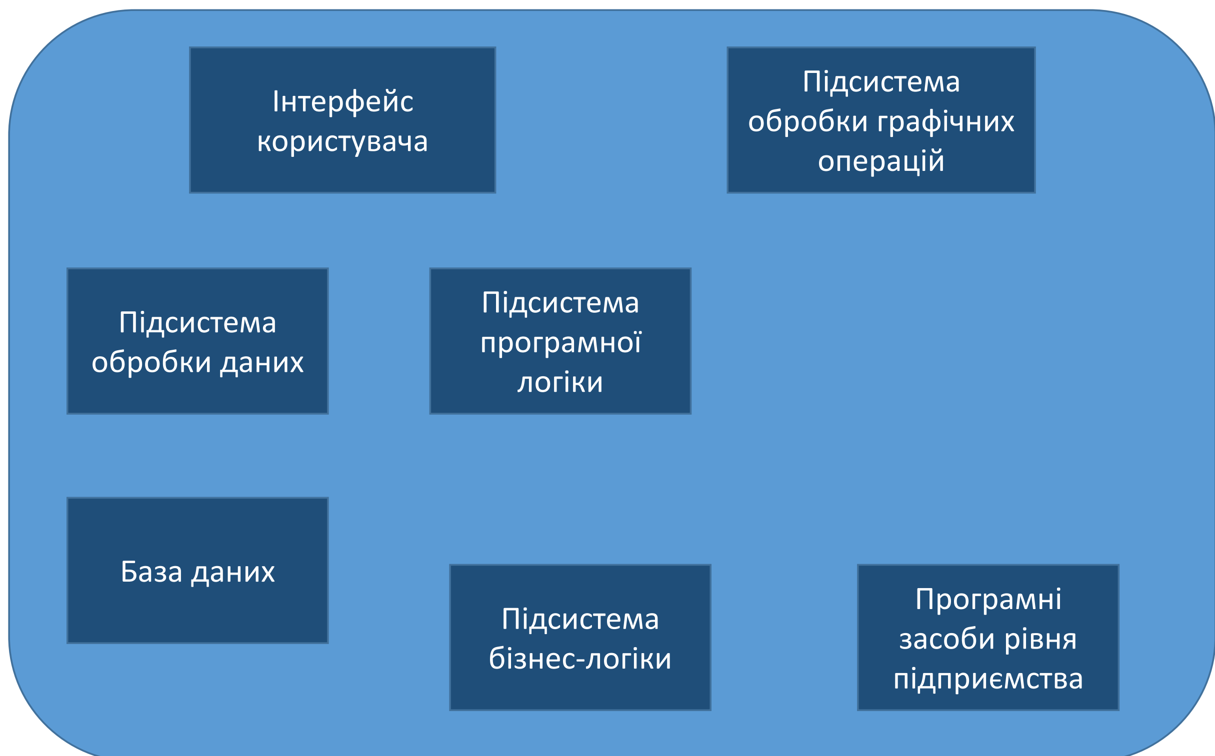

Рис. 3.3. Перший етап декомпозиції (основні підсистеми)

Як було сказано раніше, архітектура є не просто блок-схемою, а й має містити опис взаємодії між структурними елементами. Тому основною метою да-

ного рівня являється визначення правил взаємодії між підсистемами. Варто зазначити, що сенс декомпозиції зникає, якщо всі підсистеми можуть взаємодіяти між собою. Саме тому необхідно максимально обмежувати взаємодію різних підсистем. Якщо ж її не обмежити (рис. 3.4), то у відповідності до другого закону термодинаміки ентропія ПС повинна збільшитися.

У результаті взаємодії всіх підсистем постає ряд важливих питань:

- для внесення змін в підсистему обробки графічних операцій у скількох інших частинах системи має розібратися розробник?
- чи вдасться використати підсистему бізнес-логіки в інших системах?
- чи вдасться змінити в системі інтерфейс користувача без необхідності переробляти інші підсистеми?
- якими будуть наслідки перенесення підсистеми обробки або бази даних на віддалений комп'ютер?
- тощо.

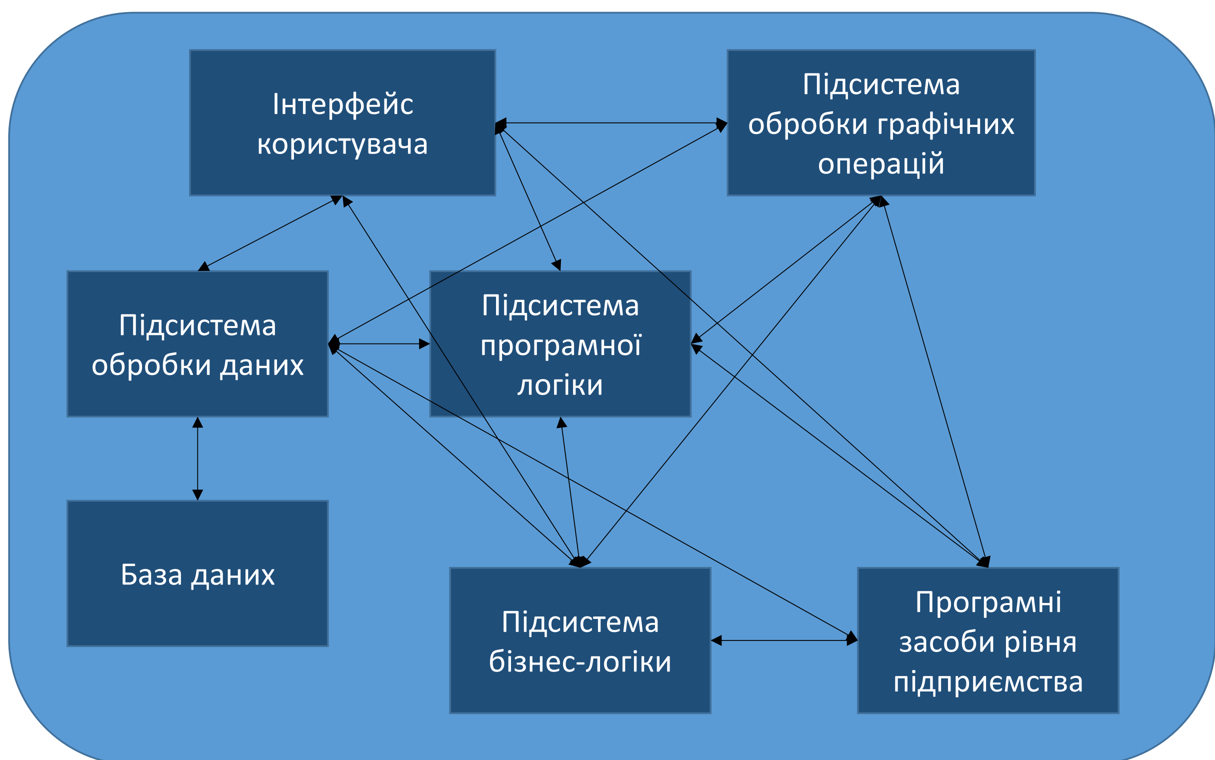

Рис. 3.4. Неправильно визначені взаємозв'язки

Отже, архітектура програмної системи має бути такою, щоб заміна будь-якої підсистеми викликала якомога менше змін в інших підсистемах. Це одне з ключових понять, яке не одноразово розглядається як у межах архітектурного проектування ПС, так і в межах детального проектування (low coupling).

Усі ці питання можна передбачити заздалегідь та вирішити, виконавши невелику додаткову роботу. Комунікація між підсистемами має бути реалізована на основі принципу «необхідного знання», тобто інформація має передаватися між підсистемами лише в необхідних для виконання відповідних функцій об’ємах. Треба завжди пам’ятати головне правило: краще на початку максимально обмежити взаємодію, а за необхідності – зробити доступ більш вільним. На рис. 3.5 показано, як декілька правил визначення взаємозв’язків можуть змінити архітектурне уявлення системи.

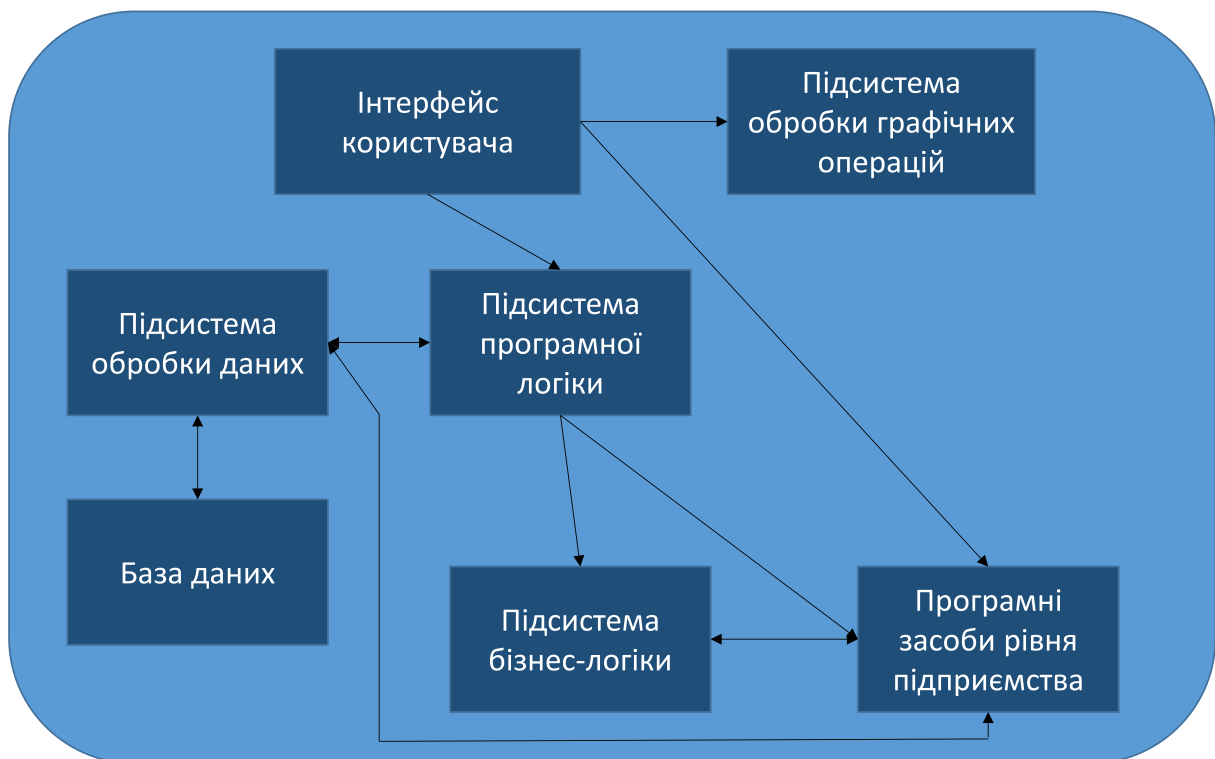

Рис. 3.5. Визначені з більшою якістю взаємозв’язки між підсистемами

Найпростішим взаємозв’язком є виклик однією підсистемою методу з іншої, більш складним – одна підсистема містить класи іншої, а найскладнішим – успадкування класами однієї підсистеми від класів іншої. Взаємозв’язки підси-

стем мають бути простими та зрозумілими також і для простоти супроводу. Цього легко досягнути, якщо запам'ятати просте правило: ПЗ не повинно містити циклічних зв'язків, при яких клас А використовує клас В, клас В, у свою чергу, використовує клас С, а клас С – клас А.

Деякі типи підсистем можна використовувати повторно у вигляді шаблонів для побудови майбутніх ПС. Визначимо ті, що зустрічаються найчастіше.

Підсистема бізнес-логіки. *Бізнес-логікою* називають закони, накази, директиви, процедури, політики та інші вимоги, які є важливими для проблемної області, висуваються замовником і мають бути реалізованими в ПЗ. Наприклад, при розробці автоматизованої системи розрахунку заробітної плати працівників бізнес-логікою можуть бути Податковий кодекс України та закони, які визначають види податків та випадки їх стягнення. Додатковим джерелом бізнес-логіки для даної системи може бути угода з профспілкою, у межах якої регламентується оплата понаднормової роботи чи відрядження.

Підсистема графічного інтерфейсу користувача. Ізольовані в окремій підсистемі компоненти інтерфейсу користувача дозволять змінювати його, не впливаючи на інші частини ПС. Як правило, підсистема графічного інтерфейсу користувача також включає в себе декілька підлеглих підсистем, наприклад, управління вікнами, довідкова система, тощо.

Підсистема доступу до БД. Підсистема, яка приховує деталі реалізації БД, забезпечує важливий рівень абстракції, що збільшує безпеку та знижує складність системи. Операції над БД концентруються в одному місці і знижують ймовірність виникнення помилок. Додатково можна легко забезпечити можливість зміни структури БД без зміни більшої частини системи.

Підсистема ізоляції залежностей від операційної системи. Залежності від операційної системи та від обладнання варто також ізолювати в окремій інтерфейсній підсистемі. Це дозволить пізніше перенести ПЗ на іншу операційну систему, наприклад, Mac OS або Linux. При цьому доведеться змінити лише поточну підсистему. Інтерфейсна підсистема є занадто великою і складною для самостійної реалізації, але існують комерційні рішення, які поставляються у вигляді готових бібліотек.

Рівень 3 (поділ підсистем на окремі класи). На даному рівні йде подальше проектування, але вже не системної, а програмної архітектури. Цей рівень проектування передбачає визначення всіх класів ПС та визначення деталей взаємодії кожного класу з її іншими елементами. Отже, можна сказати, що метою даного рівня є декомпозиція кожної підсистеми до такого ступеня детальності, який дозволить програмісту реалізувати частини наявних підсистем у формі окремих класів. Окрім класів на даному рівні також проектуються програмні інтерфейси.

Рівень 4 (поділ класів на окремі методи). Деякі методи вже будуть визначені під час проектування програмної архітектури на попередньому рівні. На четвертому рівні необхідно детально визначити закриті для доступу ззовні методи класів. Повне визначення методів кожного класу підсистеми часто дозволяє краще зрозуміти її інтерфейси. Це також може призвести до виникнення необхідності зміни існуючого інтерфейсу, тобто до повернення на третій рівень проектування.

Рівень 5 (проектування окремих методів). Мета даного рівня полягає в детальному визначенні функціональності кожного методу. Даний рівень може включати, наприклад, написання псевдокоду, коду, пошук необхідних алгоритмів, тощо. Цей рівень проектування виконується в усіх програмних системах, але не завжди якісно і усвідомлено.

3.3.2 Архітектурні парадигми

Сьогодні у книгах, які присвячені архітектурі ПЗ все частіше піднімаються питання про архітектурні стилі (парадигми), тобто сукупність фундаментальних термінів, установок та уявлень, які визначають основні підходи до вирішення проблем проектування архітектури ПС.

Особливу популярність отримали наступні парадигми [24]:

1. **Об’єктно-орієнтована.** В її межах використовується об’єктно-орієнтований підхід до структуризації ПЗ на всіх етапах його ЖЦ.

2. **Компонентно-орієнтована.** В основі даної парадигми закладено принцип використання програмних компонентів (підсистем, модулів, тощо), як еле-

ментів загальної складальної одиниці. Дана парадигма бере свій початок із принципів складального проектування, конструювання та виробництва.

3. **Сервісно-орієнтована.** Її використання засноване на ідеї масового сервісного обслуговування користувачів ПЗ за їх запитами.

Ці три архітектурні парадигми зазвичай не використовуються самостійно. У загальному випадку на практиці при розробці компонентів використовують об'єктно-орієнтований підхід (ООП) до структури та реалізації, а при розробці сервісів – складання з окремих компонентів.

Розглянемо дані парадигми більш детально.

Об'єктно-орієнтована парадигма. В основі цієї парадигми [24] лежать методи, ідеї та засоби об'єктно-орієнтованого аналізу та проектування. Найбільш послідовно ця парадигма розкрита в методології гнучкого розроблення RUP (Rational Unified Process).

В основу ООП покладено наступні принципи [24]: абстрагування, спадковість (ієрархічність), інкапсуляція (обмеження доступу), стійкість, модульність, типізація, паралелізм.

Розглянемо, що являє собою кожний з цих принципів окремо.

Абстрагування – це процес виділення окремих абстракцій у межах предметної області. **Абстракція** – це сукупність суттєвих для вирішення поставленого завдання характеристик деякого об'єкта, які відрізняють його від всіх інших та чітко визначають його особливості для подальшого розгляду і аналізу.

У відповідності до визначення, абстракція, яка застосовується до реального об'єкта, істотно залежить від задачі, яка вирішується: в одному випадку суттєвою буде форма предмета, у другому – його закон руху, у третьому – матеріали, з яких виготовлений об'єкт, у четвертому – вага, тощо. У сучасній практиці розроблення ПЗ абстракція передбачає об'єднання всіх її властивостей в єдину програмну одиницю, наприклад, абстрактний тип даних (клас). Ці властивості можуть стосуватися як стану аналізованого об'єкта (поля), так і його поведінки (методи).

У вузькому сенсі **інкапсуляція (обмеження доступу)** – це приховування певних елементів реалізації абстракції, які не зачіпають її суттєвих характерис-

тик. Необхідність обмеження доступу передбачає наявність чіткого розмежування двох наступних частин в описі абстракції [24]:

- реалізації, тобто сукупності недоступних ззовні елементів абстракції, наприклад, її внутрішньої організації та механізмів створення її поведінки;
- інтерфейсів, тобто сукупності доступних ззовні елементів абстракції, наприклад, основних характеристик стану та поведінки.

Обмеження доступу при об’єктно-орієнтованому підході дозволяє розробнику [24]:

- модифікувати реалізацію необхідних об’єктів (у правильно організованій системі не призведе до необхідності зміни інших об’єктів);
- виконувати конструювання системи поетапно й не відволікатися на особливості реалізації абстракцій, які використовуються.

Отже, у більш широкому сенсі **інкапсуляція** – це поєднання всіх властивостей об’єкта (складових його поведінки та стану) в єдину абстракцію з наступним обмеженням доступу до реалізації цих властивостей.

Модульність – принцип розроблення ПС, що передбачає її реалізацію у вигляді окремих частин або, інакше кажучи, модулів. При виконанні декомпозиції системи на окремі модулі бажано об’єднувати між собою логічно пов’язані частини. Таким чином можна забезпечити (по можливості) скорочення кількості зовнішніх зв’язків між модулями. Даний принцип успадкований від модульного програмування. Якщо дотримуватися цього принципу, можна спростити процеси проектування і налагодження ПС.

Ієрархія [24] – упорядкована система абстракцій.

В ООП найчастіше використовуються два види ієрархії:

1) Агрегація або інакше «ціле/частина» (рис. 3.6). Даний тип ієрархії показує, що деякі абстракції більш низького рівня включені в основну абстракцію як її складові частини. Наприклад, літак складається з шасі, крил, корпусу та двигуна. Такий варіант ієрархії використовується під час декомпозиції системи у двох основних випадках: на фізичному рівні – при декомпозиції системи на

окремі модулі; на логічному рівні – при декомпозиції предметної області на окремі об’єкти.

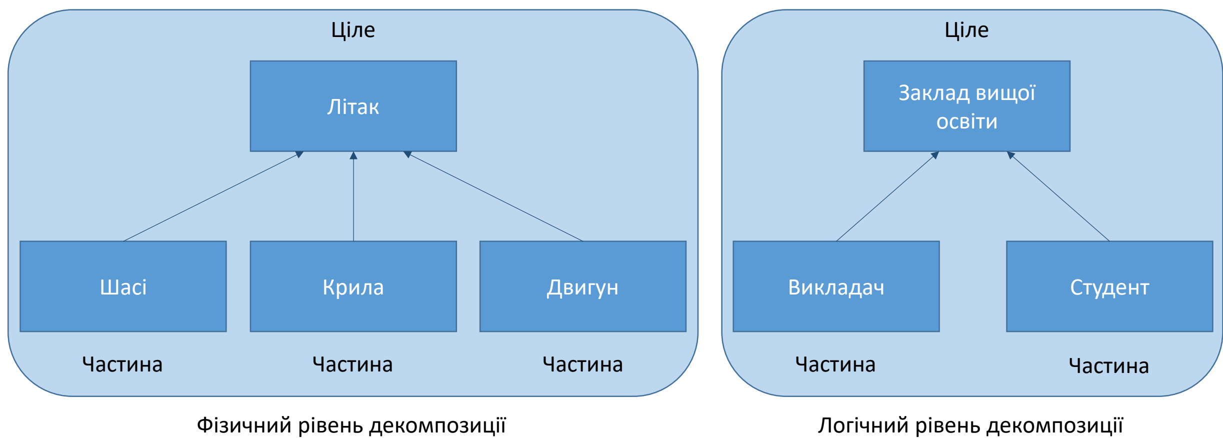

Рис. 3.6. Приклад агрегації

2) Ієрархія «загальне/приватне». Даний тип ієрархії показує, що абстракція більш низького рівня є окремим випадком абстракції більш високого рівня. Наприклад, винищувач (рис. 3.7) – є конкретним типом літака, а літак, у свою чергу, – конкретний тип літальних апаратів. Ця ієрархія найчастіше використовується при розробці структури класів, наприклад, при побудові складних класів на базі більш простих за рахунок додавання нових або уточнення наявних характеристик.

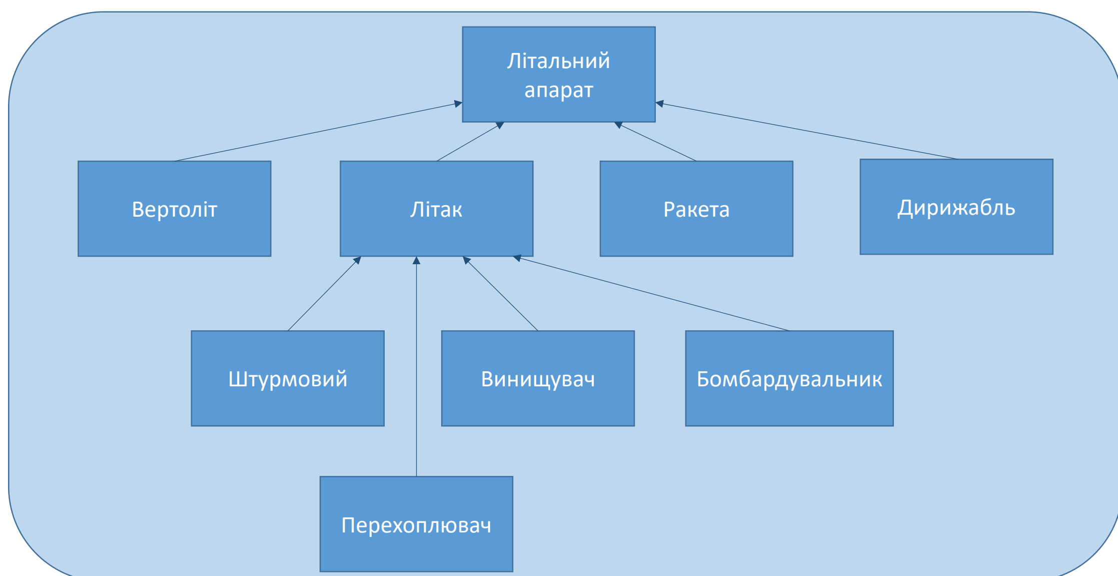

Рис. 3.7. Приклад ієрархії «загальне/приватне»

Одним з найважливіших механізмів ООП є **спадковість** – явище використання однією абстракцією функціональної або структурної частини іншої чи декількох інших абстракцій (відповідно просте і множинне успадкування). Інакше кажучи, **спадковість** – це передача ознак (стану та поведінки) від суперкласів до відповідних підкласів для автоматичного відтворення у нащадків спільних ознак зовнішньої та внутрішньої будови.

Типізація – це обмеження, які не дають змоги абстракціям різних типів динамічно замінювати одна одну (або сильно зменшує можливість такої заміни). Як правило, ці обмеження накладаються на властивості об'єктів. У мовах програмування, які мають жорстку типізацію, для кожного створюваного об'єкту (параметра, змінної, поля, тощо) оголошується тип даних, який визначає множину допустимих операцій над відповідним програмним об'єктом. Наприклад, мова програмування Java має жорстку типізацію, а деякі мови на основі C – середній рівень типізації.

Використання типізації забезпечує наявність наступних переваг [24]:

- можливість написання ПЗ з більш ефективною кодовою базою;
- спрощення процесу документування;
- раннє виявлення помилок на етапі компіляції при проведенні неприпустимих операцій над програмними об'єктами відповідних типів даних.

Тип даних може пов'язуватися з програмним об'єктом статично і динамічно. При статичному (ранньому) зв'язуванні тип об'єкта визначається на етапі компіляції, а при динамічному (пізньому) – тільки під час виконання програми. Динамічне зв'язування дозволяє реалізувати в межах мови програмування роботу зі змінними-вказівниками на певні об'єкти, які належать до різних класів, так звані, поліморфні об'єкти.

Паралелізм – це властивість декількох абстракцій одночасно виконувати певні операції, тобто перебувати в активному стані. Існує безліч завдань, вирішення яких вимагає паралельного (одночасного) виконання певних інструкцій. До таких завдань можна віднести, наприклад, необхідність автоматичного та одночасного управління декількома процесами. Реального паралелізму можна

досягти тільки за рахунок використання багатопроцесорних систем. При виконанні паралельних завдань кожне з них виконується окремим процесором. Проте це не означає, що одно процесорні системи повністю позбавлені такої Властивості. Як правило, вони імітують паралелізм шляхом поділу процесорного часу між окремими завданнями управління.

Стійкість [24] – це властивість абстракції існувати в часі (незалежно від процесу, який її породив) і/або в просторі, переміщаючись з адресного простору, в якому вона була створена.

У межах ООП розрізняють наступні види об’єктів [24]:

- **локальні об’єкти.** Їх час життя обчислюється від виклику підпрограми (функції, методу, блоку коду, тощо) до її завершення;
- **глобальні об’єкти.** Існують поки програма виконується (завантажена в пам’ять);
- **тимчасові об’єкти.** Вони зберігають проміжні результати деяких операцій, наприклад, обчислень;
- об’єкти, дані яких між сеансами роботи програми зберігаються в файлах зовнішньої пам’яті.

Усі зазначені вище принципи ООП в тій чи іншій мірі реалізовано в різних об’єктно-орієнтованих (інколи й інших) мовах.

Компонентно-орієнтована парадигма. Складність сучасного ПЗ і типова для нього форма створення в розподіленому середовищі колективом розробників призвели до ідеї складання ПЗ з розроблених незалежно, повноцінно покритих тестами та повторно-використовуваних (апаратних і програмних) компонентів. Це дуже схоже на складальне виробництво в інших галузях промисловості. Ця ідея може бути застосована для будь-яких етапів ЖЦ ПЗ, у тому числі і для етапу розроблення архітектури ПЗ.

Платформи, подібні до CORBA Component Model, Java Enterprise Edition і Microsoft.Net, а також промислове виробництво компонентів перевели компонентно-орієнтовану парадигму в практичне русло [24]. У межах цієї парадигми засновано принципи розроблення Component-base software development.

До числа основних характеристик компонентно-орієнтованої архітектури можна віднести [24]:

- компоненти є основними складовими одиницями моделювання, проектування та реалізації;
- основна увага спрямована на компонентні моделі (наприклад, CORBA Component Model);
- інтерфейси і взаємодія – ключові моменти розроблення архітектури ПЗ;
- інтерфейси створюються для розширення і надання конкретної спеціалізації для різних компонентів;
- широке використання патернів проектування і принципів поділу відповідальності.

Використання ООП призводить до утворення великої кількості дрібних класів, об’єктів та зв’язків. Знайти серед цих маленьких одиниць компоненти, які можна використовувати не одноразово, дуже важко. Ідея компонентно-орієнтованого підходу полягає в тому, щоб об’єднати частини коду, які пов’язані між собою певними зв’язками, та використовувати їх спільно. Ці інтегровані частини якраз і називають компонентами.

Методи розроблення на основі компонентів складаються з нетрадиційних процедур розроблення, включаючи пошук компонентів, їх оцінку, тощо, тому часто рекомендується застосовувати даний підхід у поєднанні з інфраструктурою проміжного програмного забезпечення (наприклад, Enterprise Java Beans) для підтримки процесу проектування.

Переваги компонентно-орієнтованого підходу до проектування архітектури включають у себе наступні пункти:

- мінімум часу та коштів на розроблення завдяки пошуку готових компонентів у каталогах комплектуючих;
- можливість швидкої перебудови ПС;
- висока якість, яка пов’язана з високою якістю готових компонентів.

Серед недоліків можна окремо виділити складність виявлення проблемного коду при тестуванні компонента.

Сервісно-орієнтована парадигма. Великий клас ПЗ, особливо реалізованого у вигляді WEB-додатків, розробляють як системи сервісів (мікросервісів), до яких клієнти мають оперативний доступ [24].

До числа основних характеристик та переваг сервісно-орієнтованої архітектури можна віднести:

- в архітектурному проектуванні ПЗ основна увага приділяється протоколам доступу, відкритті сервісу, якості наданих послуг, тощо;
- сервіс відкрито повідомляє про свої можливості за допомогою спеціального опису, який реєструється в базі сервісів у певному форматі;
- необхідний сервіс можна легко отримати після надходження запиту з будь-якого дозволеного джерела;
- між викликами сервіс не зберігає дані;
- сервіс може бути відкритий і використаний незалежно та динамічно.

До недоліків сервісно-орієнтованої архітектури можна віднести відсутність станів у сервісів, що призводить до накладних витрат при обміні даними між користувачами і провайдерами.

Сервіс може бути реалізований з використанням об'єктно-орієнтованих або компонентно-орієнтованих технологій.

Як було сказано вище, кожен з трьох розглянутих підходів надає різні можливості, а також вони можуть використовуватися доповнюючи один одного. Наприклад, сервіси можуть бути реалізовані за допомогою набору компонентів з урахуванням необхідних характеристик якості. У той же час, кожен компонент може бути реалізованим за допомогою об'єктно-орієнтованої мови програмування. Однак, кожна з розглянутих парадигм має свою специфіку в проектуванні архітектури ПЗ.

Отже, у результаті порівняння розглянутих парадигм можна виокремити декілька принципів, які допоможуть у побудові якісної архітектури:

1) Необхідно детально проаналізувати вимоги до ПЗ та окреслити його межі.

2) Необхідно визначити бажані атрибути якості розроблюваного ПЗ. Це дозволить окреслити перелік парадигм, які доцільно використовувати у даному випадку.

3) Необхідно побудувати перелік сценаріїв використання (Use Case сценаріїв), які є одними з ключових понять у кожній з наведених вище архітектурних парадигм. Сценарії допоможуть прояснити основні ролі, відповідальність, характеристики взаємодії, очікувані характеристики якості, тощо. Також ці сценарії можуть допомогти з вибором найбільш доцільного типу сутностей (об'єкти, компоненти, сервіси) для розроблюваного ПЗ.

Застосовуючи принципи архітектурного проектування в рамках певної стратегії розроблення ПЗ, необхідно пам'ятати, що сукупність парадигм, які можна використовувати в процесі розроблення, не обмежуються лише розглянутими варіантами і мають обиратися у відповідності до специфіки предметної області.

3.3.3 Сценарії використання

Розглянемо більш детально сценарії використання або інакше варіанти використання чи Use Case сценарії. По своїй суті **Use Case сценарій** – це письмовий опис того, що має робити ПС для того, щоб кінцевий користувач досяг поставленої цілі.

При цьому Use Case сценарій має не зачіпати деталей реалізації, тобто не вказувати системі як саме досягти бажаного результату, а лише повідомити про необхідність виконання певного завдання. Також Use Case сценарій не описує інтерфейси користувача та екрани, тому що це робиться на окремому етапі проектування ПС.

Залежно від складності та деталізованості Use Case сценарій може містити наступні елементи:

1) Назва або ім'я (Name) – коротке та зрозуміле ім'я, яке відбиває суть.

2) Короткий опис (Brief Description) – короткий текст, що описує поточний Use Case сценарій.

3) Учасники (Actors) – список усіх учасників взаємодії (дуже часто складається з однієї людини).

4) Передумови (Preconditions) – список умов, які мають бути виконані перед початком виконання поточного Use Case сценарію.

5) Тригер (Trigger) – умова або подія, яка змушує користувача приступити до виконання Use Case сценарію.

6) Базовий сценарій (Basic Flow) – послідовність дій, які виконує учасник для успішного досягнення цілі.

7) Альтернативні сценарії (Alternative Flows) – опис альтернативних варіантів виконання Use Case сценарію. Необхідно відмітити, що в результаті виконання альтернативного сценарію учасник обов'язково має успішно досягти кінцевої мети.

8) Виняткові сценарії (Exceptional Flows) – список того, що може привести відповідного учасника до невиконання Use Case сценарію.

9) Постумови (Post Conditions) – результат, який наступає після виконання Use Case сценарію.

Варто зазначити, що на практиці не завжди використовуються всі пункти з наведеного списку. Це залежить від кожного окремого випадку.

Розглянемо приклад формування Use Case сценарію (табл. 3.1). Як видно з наведеної таблиці, далеко не всі елементи було використано у даному прикладі.

Тепер складемо послідовність кроків для Basic Flow:

1) Користувач починає імпорт даних.

2) Користувач натискає імпорт даних з файлу XLS.

3) Система дає можливість вибрати файл з комп'ютера або перетягнути його для завантаження.

4) Користувач використовує вибір файлу з комп'ютера.

5) Користувач обирає файл XLS з необхідними даними.

6) Система обробляє обраний файл.

Таблиця 3.1. Приклад Use Case сценарію

Назва (Name)	Імпорт даних з файлу XLS
Учасники (Actors)	Будь-який користувач системи, система
Тригер (Trigger)	Користувач вирішує перенести файл з даними із зовнішнього джерела в систему. Користувач заходить у розділ обробки даних та починає імпорт.
Постумови (Post Conditions)	Дані з файлу додані та збережені в системі. Система інформує користувача про результат імпорту даних.

7) Система перевіряє файл на наявність помилок.

8) Система не знаходить помилок, що перешкоджають подальшій роботі.

9) Користувач підтверджує імпорт.

10) Система починає імпорт даних.

11) Система проводить всі необхідні перевірки даних.

12) Система успішно закінчує імпорт даних.

13) Система показує імпортовані дані.

14) Система сповіщає користувача про успішне завершення завдання.

15) Користувач бачить імпортований список даних.

Це можна вважати початковою версією базового сценарію (Basic Flow), в якій можна виявити та одразу усунути всі слабкі місця. У результаті спроектована взаємодія має вийти зрозумілою та логічною. Однією з найчастіших помилок при складанні базового (і не тільки) сценарію є відсутність опису дій самої системи. Тому про це варто не забувати.

Наступним етапом є обговорення розробленого сценарію з керівником проекту (інколи з аналітиком вимог) для того, щоб отримати зворотній зв’язок, подивитися чи така ідея є можливою для реалізації, обговорити деталі рішення та (хоча б частково) узгодити. Цей крок дозволить значно зменшити кількість непередбачених змін.

Після доопрацювання Basic Flow (у разі, якщо були до нього зауваження) необхідно ще раз обговорити його з керівником проекту. У результаті для написання остаточного Use Case сценарію пройде декілька ітерацій. У процесі

можна додавати нові виняткові та альтернативні сценарії. Проте не варто намагатися покрити всі можливі варіанти, треба знати міру. Зазвичай використання детальних та повних Use Case сценаріїв є дуже корисним, а інколи взагалі необхідним, при розробці дуже складних проектів.

Розглянемо ще один приклад Use Case сценарію, який представлений в табл. 3.2. Основна задача даного варіанту використання полягає в описі процесу авторизації кінцевого користувача в системі. Як видно з табл. 3.2, у даному випадку використовується більше елементів опису, ніж у попередньому прикладі.

Таблиця 3.2. Приклад Use Case сценарію

Назва (Name)	Авторизація користувача
Учасники (Actors)	Будь-який користувач системи, система
Тригер (Trigger)	Користувач намагається авторизуватися
Базовий сценарій (Basic Flow)	1) Користувач запускає систему. Система відкриває сесію користувача, пропонує ввести логін і пароль. 2) Користувач вводить логін і пароль. 3) Система перевіряє логін і пароль. 4) Система створює запис в історії авторизацій (наприклад, IP-адреса користувача, логін, дата, робоча станція). 5) Система сповіщає користувача про успішну авторизацію.
Виняткові сценарії (Exceptional Flows)	а*) Немає доступу до БД. Система видає повідомлення. Результат: користувач не може увійти. 1а) У налаштуваннях безпеки для даної IP-адреси існує заборона на вхід в систему. Результат: форма логіна не надається, система видає повідомлення користувачу. 2а) Користувач вибирає: «Нагадати пароль».

Продовження таблиці 3.2

	Викликається сценарій «Нагадати пароль». 3а) Користувача з уведеними логіном і паролем не знайдено. Результат: відмова в авторизації. Система видає повідомлення. Перехід на крок 2. 3б) Кількість невдалих спроб авторизуватися досягла максимальної, установленої в налаштуваннях. Результат: користувач не може увійти. Система видає повідомлення. Вхід з IP адреси Користувача заблокований на час, установлений в налаштуваннях.
Постумови (Post Conditions)	Користувач успішно авторизований

При правильному використанні написання Use Case сценаріїв допоможе вирішити поставлену задачу швидше та ефективніше. До переваг Use Case сценаріїв можна віднести:

- допомагає заощаджувати час на розроблення архітектури, прибираючи непотрібні частини ПЗ;
- графічний інтерфейс користувача виходить більш логічним і зрозумілим, підвищуючи ефективність навчання та роботи;
- проектування графічного інтерфейсу і дослідження взаємодії з ним користувача відбуваються швидко і просто;
- легше виносити на верхній рівень найбільші елементи інтерфейсу;
- з’являється чітке розуміння того, що може призвести до негативних наслідків у ході взаємодії користувача з ПЗ;
- допомагає архітектору пояснити іншим учасникам команди розроблення, як повинен вести себе кінцевий ПП;
- швидко виявляються помилки спроектованого досвіду взаємодії.

3.3.4 Приклади архітектурних стилів та моделей

Основною ідеєю програмної архітектури є зниження складності ПС шляхом абстрагування та розмежування повноважень.

З точки зору користі програмна архітектура дає напрямок для руху і вирішення завдань для різних груп користувачів, наприклад, розробників ПЗ, зацікавлених осіб, групи супроводу, фахівців з розгортання, тестувальників, тощо. Цей факт є аргументом на захист необхідності і доцільності обов’язкового створення архітектури ПЗ ще до етапу безпосереднього розроблення.

Серед основних глобальних класів архітектур ПЗ можна виділити наступні чотири:

- цілісна (монолітна) програма;
- комплекс автономно виконуваних програм;
- шарувата (багатошарова, багаторівнева) програмна система;
- колектив паралельно виконуваних програм.

Цілісна (монолітна) програма. Її основною відмінністю є те, що використання архітектури даного класу не потребує декомпозиції на підсистеми або пакети (п. 3.3.1), а виконується у вигляді єдиного програмного середовища. На відміну від компонентного або сервісного (мікросервісного) ПЗ, де необхідно зменшувати кількість взаємозв’язків між окремими модулями, монолітні програми виконуються таким чином, що майже всі їх компоненти (модулі, функції, класи, тощо) є взаємозалежними (рис. 3.8). У ПС, які використовують такий архітектурний стиль, для правильної компіляції або виконання коду без помилок мають бути присутніми всі пов’язані між собою компоненти.

Якщо, наприклад, ПЗ має виконувати одну яскраво виражену задачу низької складності, то найчастіше його проектують саме у вигляді цілісної програми. Ще однією перевагою такого архітектурного стилю є те, що майже ніколи не потрібно документувати архітектурні рішення, так як немає ніякої зовнішньої взаємодії, окрім роботи з кінцевим користувачем, яка описується вже у документації щодо застосування даного ПЗ (наприклад, у керівництві користувача).

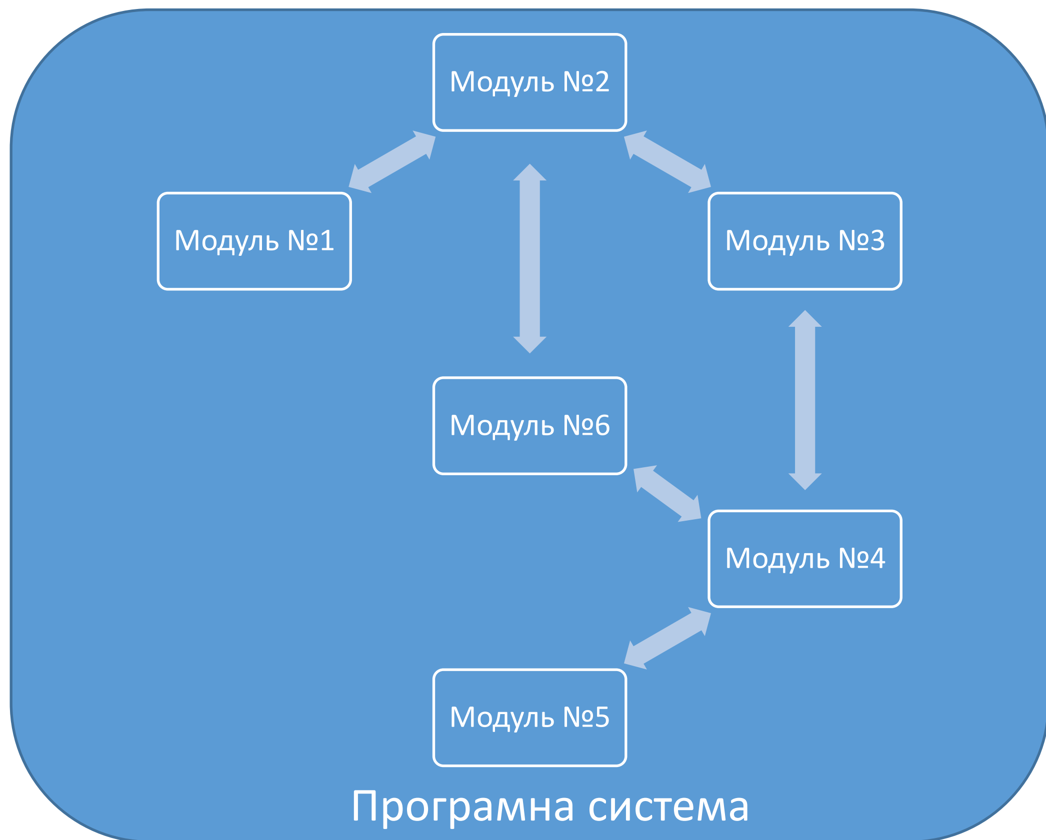

Рис. 3.8. Приклад цілісної архітектури

Серед переваг монолітної архітектури можна зазначити наступні:

1) Простота розроблення. Вона пов'язана з концентрацією усіх функціональних можливостей в одному місці, що призводить до полегшення інтеграції різних інструментальних засобів, які спрощують процес.

2) Простота модифікації. Оскільки всі компоненти знаходяться поруч, у них легше вносити зміни.

3) Простота запуску програми. Відсутність зв'язків з зовнішніми компонентами полегшує процес компіляції.

4) Відсутні проблеми обміну завданнями між компонентами ПЗ.

5) Покращена продуктивність у порівнянні, наприклад, з архітектурою на основі мікросервісів завдяки єдиному коду.

Проте даний архітектурний стиль не позбавлений і недоліків, а саме:

1) Дуже великий обсяг коду. Даний архітектурний стиль не бажано використовувати для великих проектів, оскільки з часом код розростеться до гранді-

озних розмірів. Це призведе до ускладнення його розуміння, втрати якості та недоцільності подальшого обслуговування.

2) Складність розширення. Якщо в ПЗ потрібно додати нові функціональні можливості, то при використанні цілісної архітектури можна зіткнутися з безліччю перешкод. Майже у всіх випадках для розширення можливостей доведеться переписати ПЗ майже повністю, а це довго і дорого.

3) Обмежена гнучкість. Аналогічна до попереднього випадку ситуація проявляється і під час виправлення багів та введення оновлень. Будь-яке виправлення помилок призводить до необхідності коригування великої кількості пов'язаних елементів, а будь-яке оновлення може викликати необхідність переписувати ПЗ з нуля.

4) Залежність між компонентами. Даний недолік випливає з попередніх. З одного боку, дана характеристика є саме перевагою такого архітектурного стилю, так як збільшує продуктивність, але з іншого, – якщо в якомусь компоненті ПЗ буде знайдено баг (помилку), то це сповільнить або взагалі зупинить роботу всієї програми, а не лише одного конкретного компонента.

Незважаючи на недоліки даний клас архітектур досі є актуальним і використовується в багатьох стартапах та невеликих розробках, наприклад:

- за відсутності досвіду роботи з проектування архітектури;
- якщо в команді до п'яти розробників;
- якщо заздалегідь відомо, що ПЗ не буде часто модифікуватися та сильно розширюватися;
- якщо потрібно швидко розгорнути ПЗ невеликого розміру;
- якщо потрібно швидко протестувати певний функціонал;
- якщо в пріоритеті розроблення знаходяться швидкість роботи ПЗ та його продуктивність.

Ще однією сферою використання цілісної архітектури є навчальний процес. Студенти дуже рідко ставлять перед собою запитання необхідності проектування архітектури ПЗ, тому більшість програм, які вони пишуть, за замовчуванням мають саме такий клас.

Іншим архітектурним стилем є **комплекс автономно виконуваних програм** (рис. 3.9). Він складається з набору програм, такого, що:

- користувач може активувати будь-яку з цих програм;
- доки працює активна програма, доти не може бути запущено жодну іншу програму з даного набору;
- використовується єдине інформаційне середовище для усіх програм комплексу.

Останній пункт є ключовою вимогою щодо використання такого класу архітектурних рішень.

Рис. 3.9. Приклад комплексу автономно виконуваних програм

Шарувата (багатошарова, багаторівнева) програмна система. Клас багатошарової архітектури ПЗ сьогодні є найбільш часто використовуваним. Метою багаторівневої архітектури є організація компонентів в горизонтальні логічні (програмні) шари та фізичні рівні.

Шар – це логічна одиниця, яка має певну роль та свою відповідальність у межах ПЗ.

Рівень – це фізична одиниця, у межах якої працює код (наприклад, веб-сервер, БД, тощо).

Іноді при проектуванні кожен шар розміщується у окремому рівні, але так робити не обов’язково. На одному рівні без проблем можна розмістити декілька пов’язаних між собою шарів. Якщо розділити рівні фізично, то це збільшить масштабованість, ремонтопридатність та стійкість, але при цьому зменшиться швидкодія за рахунок додаткового мережевого зв’язку.

Традиційна багаторівнева архітектура ПЗ має три рівні та чотири шари (рис. 3.10). Розглянемо більш детально таку архітектурну схему. У ній присутні чотири наступні стандартні шари:

1) **Шар графічного подання.** Цей шар містить усі графічні інтерфейси користувача (веб, десктопний, мобільний, тощо), які є відкритими для нього.

2) **Шар бізнес-логіки.** У межах даного шару проходить основна обробка та перевірка всієї бізнес-логіки та бізнес-процесів.

3) **Шар доступу до даних.** Цей шар відповідає за взаємодію з БД.

4) **Шар сховища даних.** Цей шар є фактичним сховищем даних для ПЗ.

Рис. 3.10. Приклад багаторівневої архітектури

Завдяки впровадженню різних рівнів ПЗ покращується простота, ремонтопридатність та здатність тестувати програмні компоненти.

Наведені рівні можна описати наступним чином:

1) **Рівень графічного подання.** На цьому рівні розміщена база коду графічних інтерфейсів користувача, тобто шар графічного подання. Це найвищий рівень ПЗ, і це єдиний рівень, до якого користувачі можуть отримати прямий доступ (в ідеальному випадку).

2) **Бізнес-рівень.** Цей рівень містить базу внутрішнього коду, тобто шар бізнес-логіки та шар доступу до даних.

3) **Рівень даних.** На цьому рівні розміщуються сховища даних, наприклад, БД, файлова система, хмарне сховище, БД документів, тощо.

Існує два способи реалізації багаторівневої архітектури програмного забезпечення, а саме: закрита багатошарова архітектура та відкрита багатошарова архітектура.

У **закритій багатошаровій архітектурі** (рис. 3.11) шар може викликати лише наступний шар, який знаходиться безпосередньо під ним. Розглянемо наведену нижче діаграму закритої шаруватої архітектури.

Рис. 3.11. Приклад закритої шаруватої архітектури

На наведеній діаграмі кожен шар позначений як закритий. Це означає, що для того, щоб запит пройшов до самого нижнього шару, він має пройти послідовно через кожен шар. Наприклад, коли запит ініціюється через рівень графічного подання, він спочатку буде проходити через рівень бізнес-логіки, потім через шар доступу до даних і, нарешті, дійде до шару сховища даних.

Мета закритої багатошарової архітектури – забезпечити повну ізоляцію шарів один від одного. Це дає змогу стверджувати, що зміни, внесені в один шар архітектури, не впливають на інші шари (низьке зчеплення).

У **відкритій багатошаровій архітектурі** (рис. 3.12) шар може передавати запит через один або декілька шарів за умови, що ці шари, які знаходиться безпосередньо під ним, позначені як відкриті. Розглянемо наведену нижче діаграму відкритої шаруватої архітектури.

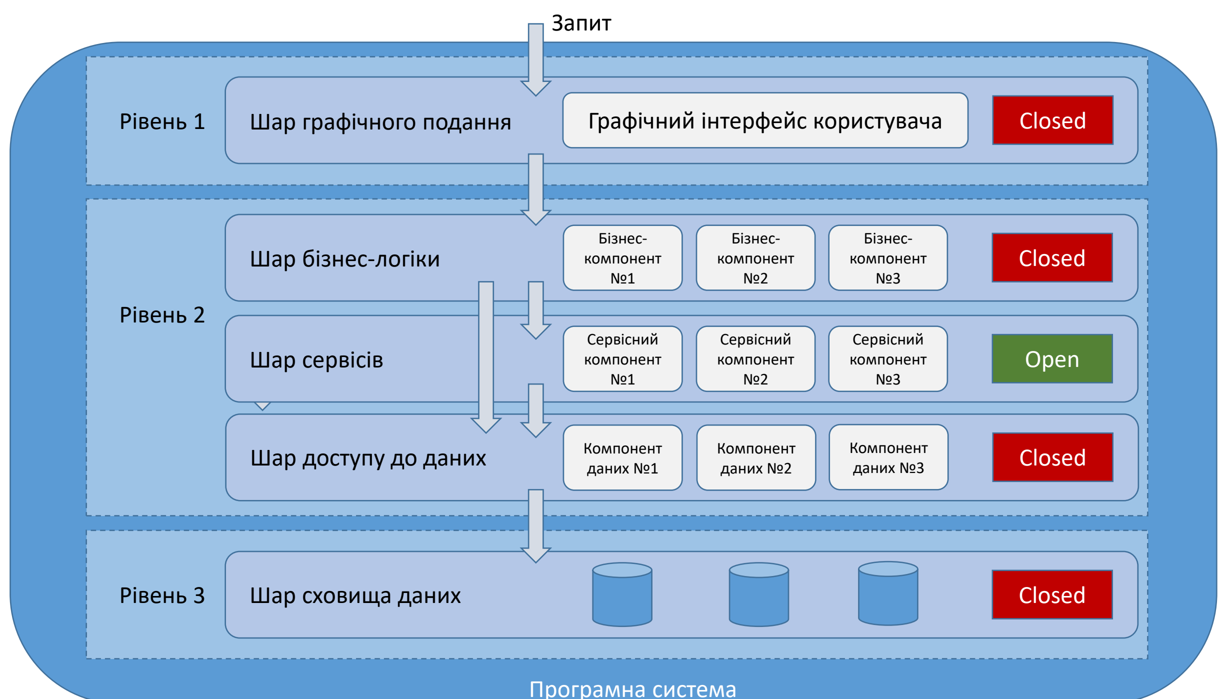

Рис. 3.12. Приклад відкритої шаруватої архітектури

На наведеній вище діаграмі було представлено новий шар сервісів. Цей шар забезпечує ряд сервісних компонентів, серед яких не всі взаємодіють з БД. Саме з цієї причини шар позначено як «Open» (відкритий). Це означає, що шар бізнес-логіки має доступ як до шару сервісів, так і до шару доступу до даних,

тобто запиту, який проходить через шар бізнес-логіки, не потрібно проходити шар сервісів, щоб використовувати шар доступу до даних.

Серед основних недоліків шаруватої архітектури можна зазначити, що при її реалізації часто створюється непотрібний мережевий трафік, оскільки кожен шар просто передає запити до наступного, часто не виконуючи ніякої програмної логіки для їх обробки.

Отже, підводячи підсумки, можна сказати, що для багаторівневої архітектури справедливі наступні твердження:

- нижчі шари не знають про властивості та (інколи) існування вищих;
- взаємодія між сусідніми шарами (у закритій архітектурі) або через один/декілька шарів (у відкритій архітектурі) відбувається через визначені заздалегідь інтерфейси;
- будь-який шар нічого не знає про внутрішню структуру шарів, з якими може взаємодіяти;
- неприпустимим є використання одних і тих же глобальних даних декількома різними шарами.

Отже, це означає, що шари ведуть взаємодію шляхом передавання абстракцій через визначені інтерфейси, обмежуючи зв'язки лише з суміжними знизу та зверху (для видачі результатів кожного звернення).

Останнім з розглянутих класів архітектур буде **колектив паралельно виконуваних програм**. Він являє собою набір програм, які можуть одночасно перебувати в стадії виконання та взаємодіяти між собою. Для синхронізації даних зазвичай такі програми взаємодіють між собою шляхом передачі одна одній деяких повідомлень. Такі визначення відразу розкривають одну з особливостей: щоб працювати одночасно, програми колективу мають використовувати декілька центральних процесорів (або використовувати один у порядку черги) та мають бути усі завантаженими в оперативну пам'ять.

Ці твердження приводять до декількох недоліків, а саме: в однопроцесорних системах, де немає явного паралелізму, колектив паралельно виконуваних програм працюватиме повільніше; необхідно використовувати більший об'єм пам'яті у порівнянні з іншими класами архітектури.

Отже, розглянемо найбільш поширені на сьогодні архітектурні моделі, які використовуються при розробленні ПЗ.

Архітектура файл-сервер (file-server). Найпростішою для реалізації архітектурою [24, 25] є «файл-сервер» (рис. 3.13). Проте така архітектура окрім простоти містить і найбільшу кількість недоліків, кожен з яких обмежує її область використання. У найпростішому випадку дані розташовуються фізично на тому ж комп'ютері, що і ПЗ.

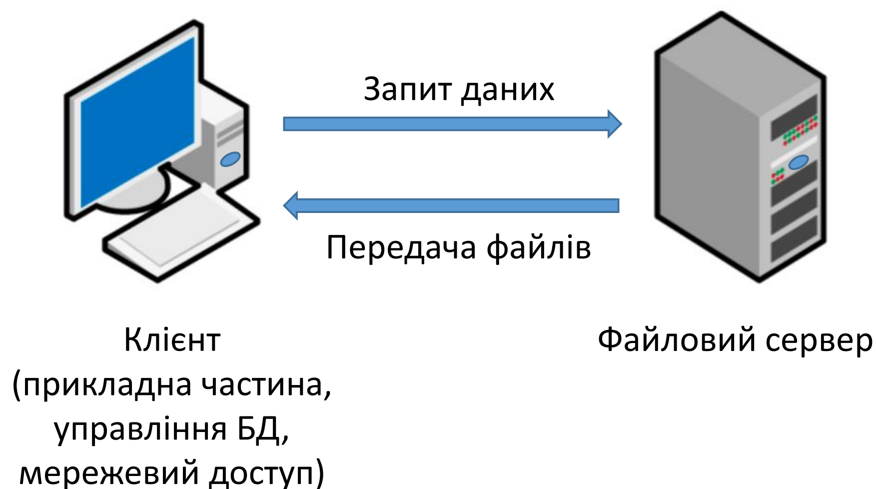

Рис. 3.13. Структура системи з архітектурою файл-сервер

До значних недоліків, що виникають при роботі з ПЗ, яке побудовано за такою архітектурою, можна віднести наступне [24, 25]:

1) При передачі даних суттєво навантажується мережа. При отриманні запиту на вибірку інформації вся БД копіюється на робочу станцію клієнта, а вибірка даних здійснюється вже безпосередньо на ній. У результаті зростає мережевий трафік і збільшуються вимоги до апаратних потужностей для користувача комп'ютера. Це також призводить до перевантаження мережі при збільшенні числа користувачів і обсягу даних.

2) Високі вимоги до апаратних ресурсів комп'ютерів. При використанні файл-серверної моделі архітектури керування даними відбувається за допомогою файлових сервісів операційної системи клієнта – усі бібліотеки доступу до

даних містятья в ПЗ на його стороні. Отже, усе обчислювальне навантаження повністю лягає на ПЗ клієнта.

3) Низький рівень безпеки. Оскільки клієнт напряму з'єднується з сервером і отримує доступ до БД, набагато простіше вносити зміни в окремі таблиці. Це призводить до низького рівня безпеки як з точки зору внесення помилкових змін, так і з точки зору завдання шкоди та викрадення даних.

4) Невисока швидкість обробки інформації. Також випливає з пояснення першого недоліку: постійна передача даних туди і назад через мережу призводить до її перевантаження та повільної роботи.

5) Важко забезпечити цілісність даних. Прямий доступ клієнта до БД загрожує неприємними наслідками, наприклад, руйнуванням її таблиць та індексів.

6) Декілька користувачів не можуть одночасно отримати доступ до однієї і тієї ж ділянки БД.

7) Дуже обмежена кількість користувачів, які можуть одночасно працювати з ПС, побудованою на основі даної моделі архітектури. Зазвичай число користувачів такої ПС без виникнення проблем її функціонування не може бути вищим за десять. При збільшенні кількості активних користувачів ПС може «захлинутися» через перевантаженість локальної мережі великими потоками необробленої інформації.

Незважаючи на велику кількість недоліків, низька вартість системи з такою архітектурою та простота реалізації є її важливими перевагами. Отже, якщо бізнес невеликий, то рішення використовувати подібну модель архітектури може бути найбільш оптимальним.

Великі корпоративні структури надають більшу перевагу рішенням по автоматизації обліку і управління, які передбачають організацію паралельних обчислень, розподілену обробку даних, можливість вибору різних операційних систем і серверних платформ, глибоке розмежування рівнів доступу, тощо. Для таких компаній вибір файл-серверної моделі архітектури є неприпустимим.

Клієнт-серверна модель (client-server). Недоліки архітектури «файл-сервер» вирішуються при перебудові ПЗ на базі архітектури «клієнт-сервер» (рис. 3.14).

Як правило, сервер – найпотужніший і найбільш надійний комп'ютер. Він обов'язково підключається через джерело безперебійного живлення для забезпечення стабільності роботи. Додатково в ньому передбачаються системи резервного дублювання (подвійного або навіть потрійного). В особливо відповідальних випадках одночасно підключаються разом декілька серверів. Це призводить до того, що при виході з ладу одного з серверів до роботи приступає інший. Таким чином, при концентрації обробки даних на сервері, надійність системи в цілому обмежується лише матеріальними засобами, які замовники готові вкласти в технічне оснащення.

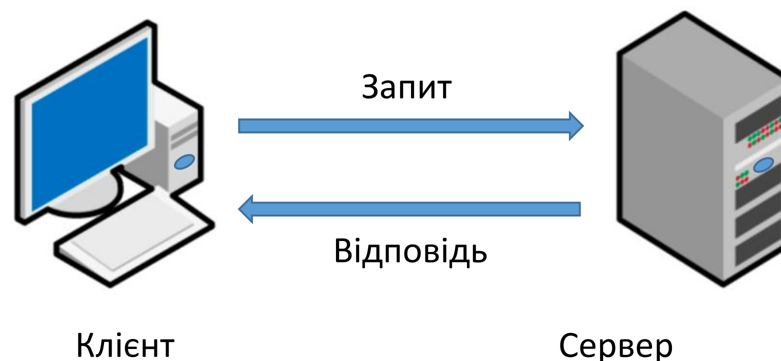

Рис. 3.14. Структура системи з архітектурою клієнт-сервер

При використанні такої моделі архітектури ПЗ, яке розташовано на сервері, очікує запити від клієнтських програм. Отримавши запит, ПЗ надає їм свої ресурси у вигляді відповідних даних (наприклад, завантаження файлів за допомогою FTP, HTTP, потокового мультимедіа, BitTorrent, тощо) або сервісних функцій (наприклад, спілкування за допомогою систем миттєвого обміну повідомленнями, робота з електронною поштою, тощо).

Як і при використанні архітектури «файл-сервер» БД міститься на мережевому сервері. Відмінність полягає в тому, що прямий доступ до БД з ПЗ клієнтської частини не відбувається. Пряме звернення до БД можливе через спеці-

альну керуючу програму, яка поставляється розробником системи управління БД, – сервер БД (SQL-сервер).

Отже, перенесення обчислювального навантаження на SQL-сервер і є характерною особливістю архітектури «клієнт-сервер». Це дозволяє максимально розвантажити ПЗ клієнта. Як наслідок, відбувається істотне збільшення безпеки даних як від виконання просто помилкових операцій з ними, так і від зловмисних дій.

Трирівнева архітектура. Дана архітектура (рис. 3.15, рис. 3.16) є однією з моделей шаруватої (багатошарової або багаторівневої) архітектури, а також розширеним варіантом «клієнт-серверної» моделі. Являє собою архітектурну модель ПЗ, що передбачає наявність у ньому трьох компонентів [24]: клієнтської програми, сервера додатків, до якого підключено клієнтську програму, і сервера БД, з яким працює сервер додатків.

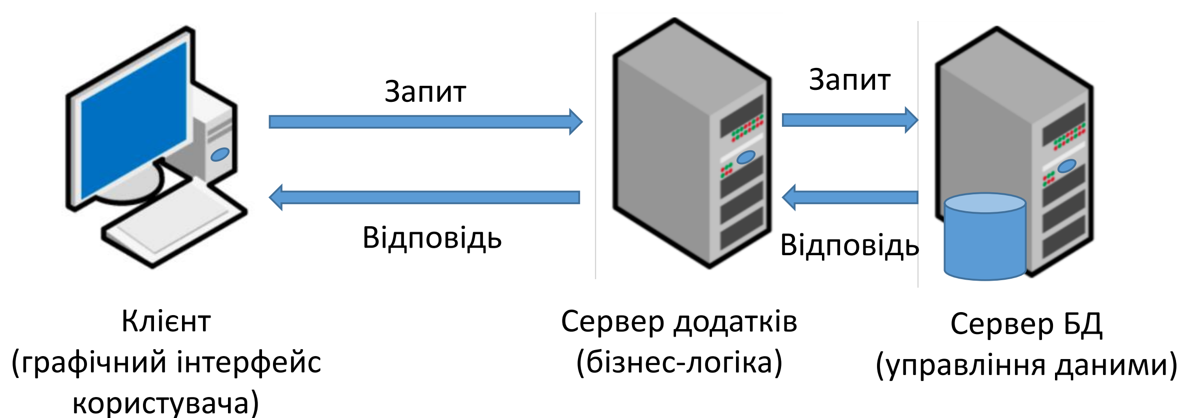

Рис. 3.15. Структура системи з трирівневою архітектурою

Вони можуть взаємодіяти один з одним за наступною схемою:

- на стороні клієнта відбувається відображення даних;
- на виділеному сервері додатків знаходиться прикладний компонент, який виконує функції проміжного ПЗ;
- сервер БД відповідає за управління ресурсами та представляє дані, на які приходить запит від користувача.

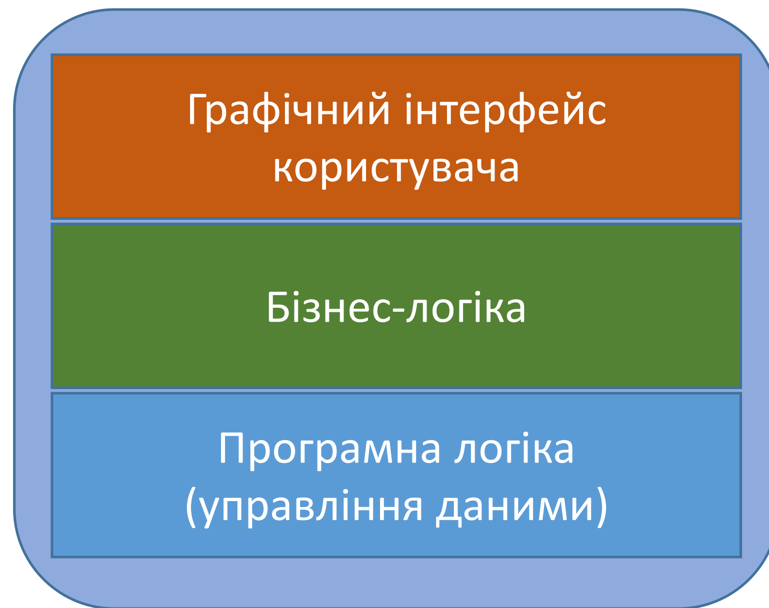

Рис. 3.16. Інше уявлення структури системи з трирівневою архітектурою

Трирівнева архітектура є наступним кроком в еволюції «клієнт-серверного» варіанту. У даному випадку додатковий компонент (сервер додатків) виступає у якості посередника між ПЗ клієнта та серверами БД. Варто зазначити, що один і той же сервер додатків можуть використовувати одночасно багато клієнтів, запити від яких обов'язково проходять через бізнес-логіку перш ніж надійти до БД. Відповідь від БД проходить той же самий шлях, але у зворотному напрямку.

Серед основних переваг такої моделі архітектури можна зазначити підвищення безпеки та збільшення кількості одночасних користувачів завдяки наявності серверу додатків у порівнянні з «файл-серверною» та «клієнт-серверною» архітектурами.

Peer-to-peer. Децентралізована, тимчасова, однорангова або пірінгова мережа (рис. 3.17) – це комп'ютерна мережа, учасники якої мають рівні права [24]. Кожен вузол (peer) такої мережі є одночасно клієнтом та сервером, тому дуже часто окремі виділені сервери відсутні. На відміну від архітектури «клієнт-сервер», така організація архітектури дозволяє зберігати працездатність мережі при будь-якій кількості вузлів та будь-якому їх поєднанні.

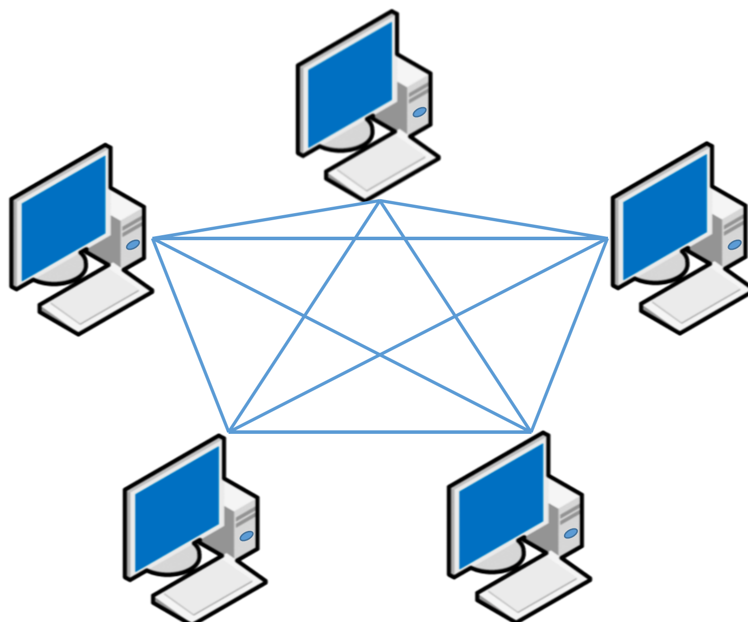

Рис. 3.17. Приклад пірінгової мережі

Архітектура труб та фільтрів (pipes and filters). Дана архітектура є однією з найбільш простих та дуже надійних структур з послідовним процесом оброблення інформації. Вона складається з набору компонентів, які називаються фільтрами. Кожен фільтр отримує інформацію з компонентів, які з'єднують сусідні фільтри. Дані компоненти називаються трубами (або трубопроводами, *англ. pipes*). Подібна архітектура дуже часто використовується в компіляторах, Unix-програмах та аналізаторах синтаксису [24].

У загальному випадку схема обробки інформації є послідовною (рис. 3.18), але можливі варіанти реалізації з розгалуженням потоків (рис. 3.19).

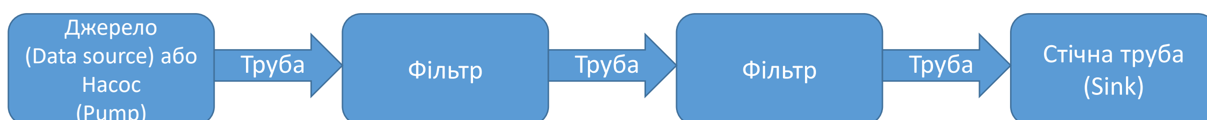

Рис. 3.18. Приклад послідовної архітектури труб та фільтрів

Джерело або насос (*англ. data source, pump*) – є джерелом даних системи.

Фільтр (*англ. filter*) – підсистема, яка отримує інформацію або від джерела (насосу), або від попереднього фільтра за допомогою спільних трубопроводів. Після обробки потік даних з фільтру відправляється далі.

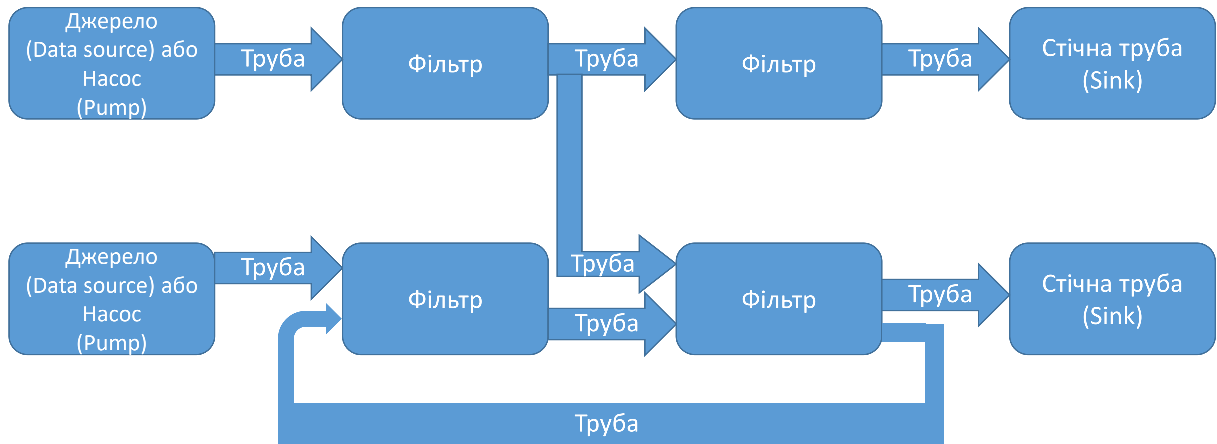

Рис. 3.19. Приклад архітектури труб та фільтрів з розгалуженням

Труба або трубопровід (*англ. pipe*) – здійснює передачу даних від джерела до фільтру або між окремими фільтрами в ПС.

Стічна труба (*англ. sink*) – приймач обробленої фільтрами інформації.

Незважаючи на переваги простої структури даної архітектурної моделі, їй притаманні також істотні недоліки:

- для того, щоб почався процес оброблення, фільтр має отримати весь об'єм даних повністю;
- можуть виникати суттєві часові затримки через накопичення даних у буфері;
- дані можуть бути втраченими у разі переповнення буфера;
- кожна труба проектується для передачі лише одного типу даних.

Сервісно-орієнтована архітектура (рис. 3.20). Суть даної архітектурної моделі полягає у використанні слабко зчеплених компонентів, які мають стандартні інтерфейси. Окремо варто відмітити, що ці компоненти можна легко замінювати, а також вони ведуть обмін даними за стандартизованими протоколами.

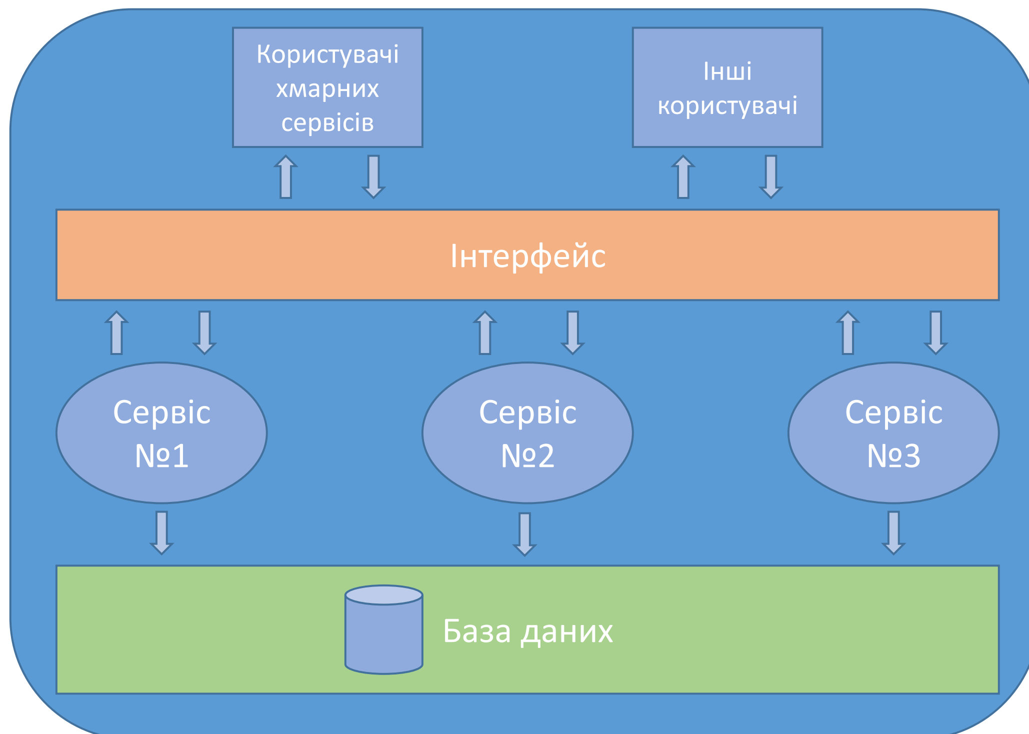

Рис. 3.20. Приклад сервісно-орієнтованої архітектури

Використання стандартних інтерфейсів дозволяє приховати деталі внутрішньої структури компонентів від інших елементів ПС. Також це дозволяє комбінувати та не однократно використовувати компоненти для побудови складних програмних комплексів. Дана архітектура забезпечує ПС незалежність від інструментів розроблення та використовуваних платформ, а також сприяє їх керованості і масштабованості.

Сервісно-орієнтована модель архітектури не прив'язана до якоїсь конкретної технології та може бути реалізована з використанням їх широкого спектру (наприклад, REST, RPC, DCOM, CORBA, тощо).

Розглянемо, наприклад, технологію REST (*англ. Representational State Transfer, передача репрезентативного стану*) [24]. Це метод взаємодії компонентів ПС в мережі «Інтернет», при якому виклик віддаленої процедури є HTTP-запитом, а необхідні дані передаються в якості параметрів цього запиту. Такі запити називають REST-запитами і найчастіше це HTTP-запити GET або POST. Дана технологія є альтернативою більш складних методів, таким як SOAP, CORBA та RPC.

У широкому сенсі REST означає концепцію побудови розподіленого додатка, при якій компоненти взаємодіють на зразок взаємодії клієнтів і серверів у мережі «Інтернет».

Хоча дана концепція лежить в самій основі Всесвітньої павутини, термін REST був введений Роєм Філдінгом (англ. Roy Fielding), одним із творців протоколу HTTP, лише в 2000 році [24]. Він описав концепцію побудови розподіленого ПЗ, при якій кожен REST-запит клієнта до сервера містить в собі вичерпну інформацію щодо бажаної відповіді, а сервер не зобов'язаний зберігати інформацію про стан клієнта.

Необхідні умови для побудови розподілених REST-додатків [24]:

- архітектура «клієнт-сервер» або її аналог;
- взаємодія клієнта з сервером відбувається через проміжні вузли, про існування яких клієнт може не знати;
- у кожному запиті клієнта має міститися явна вказівка про можливість кешування відповіді і отримання відповіді з існуючого кеша;
- сервер не зобов'язаний зберігати інформацію про стан клієнта;
- стандартизований програмний інтерфейс сервера.

ПЗ, яке не відповідає наведеним умовам, не може називатися REST-додатком. Якщо ж усі умови дотримуються, то таке ПЗ отримає наступні переваги [24]:

- надійність – не потрібно зберігати інформацію (яка може бути втрачена) про стан клієнта;
- простота пограмних інтерфейсів;
- портативність компонентів;
- прозора система взаємодії;
- продуктивність (за рахунок використання кеша);
- легкість внесення змін, тобто здатність еволюціонувати, пристосовуючись до нових вимог;
- масштабованість.

Отже, ключовим аспектом, який виділяє сервісно-орієнтовану архітектуру є використання незалежних сервісів з чітко визначеними програмними інтерфейсами. Ці сервіси для виконання своїх завдань можуть викликатися одним зі стандартних способів за умови, що сервіси нічого не знають заздалегідь про ПЗ, яке їх викличе. При цьому ПЗ, у свою чергу, не може знати того, як саме сервіси вирішують свої задачі.

Сервісно-орієнтована архітектура також може розглядатися як стиль архітектури інформаційних систем. Така архітектура дозволяє створювати ПЗ, побудоване шляхом комбінації взаємодіючих сервісів зі слабким зчепленням. Ці сервіси мають взаємодіяти на основі незалежного від платформи і мови програмування стандартизованого інтерфейсу. Реалізація інтерфейсу взаємодії дозволяє приховати залежну від мови програмування реалізацію кожного сервісу. Наприклад, сервіси, написані на Java, що працюють на платформах Java EE і сервіси на C#, що працюють на платформах .Net, можуть бути з однаковим успіхом викликані однією і тією ж програмою. ПЗ, яке працює на одних платформах, може викликати сервіси, які працюють на інших платформах. Такий підхід полегшує повторне використання компонентів.

Деякі мови високого рівня або певні специфікації можуть розширювати концепцію сервісу за рахунок надання методів та засобів для об'єднання дрібних сервісів у більш великі бізнес-сервіси. У свою чергу, бізнес-сервіси можуть бути включені в склад бізнес-процесів, які реалізовані у вигляді компонентів Інтернет-порталів або додатків.

Архітектура на основі подій (*англ. Event-driven architecture*). Це модель архітектури, яка дозволяє компанії (організації, фірмі, підприємству, тощо) виявляти «події» або важливі ділові моменти (наприклад, транзакції, відвідування сайту, відмова від кошика, тощо) і реагувати на них у режимі реального часу або майже в реальному часі. Архітектура на основі подій замінює традиційний підхід «запит/відповідь», коли службам доводиться чекати відповіді, перш ніж вони зможуть перейти до наступного завдання. При використанні даної архітектури загальний потік керується подіями, тобто він призначений для надання певної реакції на них або виконання певних дій у відповідь на кожну подію.

Подію можна визначити, наприклад, як суттєву зміну відповідного стану. Змоделюємо наступну ситуацію: купівля нового смартфону. Коли хтось купує смартфон, стан цього смартфону змінюється з «доступний для придбання» на «проданий». Системна архітектура продавця мобільних телефонів може розглядати цю зміну стану, як подію, що створюється, публікується, визначається та споживається різними додатками в складі системи.

Архітектуру на основі подій, часто називають «асинхронною комунікацією». Це означає, що відправник і одержувач не повинні чекати один одного, щоб перейти до наступного завдання. Наприклад, телефонний дзвінок вважається синхронним, бо виконується більше за традиційним принципом «запит/відповідь»: хтось телефонує та просить щось зробити; чекає, поки відповідач виконає завдання; потім обидві сторони кладуть слухавки; можна виконувати наступний дзвінок. Асинхронним прикладом можуть бути текстові повідомлення: коли надсилається повідомлення (інколи навіть невідомому отримувачу), не обов'язково чекати відповідь, щоб відправити ще одне.

Традиційно більшість систем працюють на основі моделей, які орієнтовані на дані. В їх межах дані є джерелом істини. У моделі архітектури, що керується подіями, дані все ще важливі, але події стають найважливішим елементом. Наприклад, у сервісно-орієнтованій моделі найвищим пріоритетом було переконатися, що не відбувається втрата будь-яких даних. З архітектурою на основі подій основним пріоритетом є необхідність переконатися, що система реагує на події. Коли подія виникає, то для неї існує закон спадної віддачі, а саме: чим старші за часом стають події, тим менше вони цінні. Однак, сьогодні сервіс-орієнтована архітектура та архітектура на основі подій дуже часто використовуються разом.

При використанні архітектури на основі подій є виробники подій, які генерують і надсилають повідомлення про події. Також при цьому може бути один або декілька споживачів, коли виникнення події запускає логіку її обробки. Наприклад, припустимо, що Netflix щойно завантажив новий фільм. Може бути декілька програм, які прослуховують (очікують) цю подію. Після отримання повідомлення вони також запускають їхні власні внутрішні системи, щоб

публікувати власну інформацію про цю подію для своїх користувачів. Це відрізняється від традиційного обміну повідомленнями «запит-відповідь» тим, що програми працюють постійно. Тобто, навіть якщо вони знаходяться в режимі очікування певної події, вони не паралізовані і можуть реагувати на публікацію повідомлення. Таким чином, багато різних служб можуть працювати паралельно.

Компоненти архітектури на основі подій можуть включати три частини (рис. 3.21): виробник, споживач, брокер. Брокер може бути необов'язковим, особливо якщо є лише один виробник та один споживач, які перебувають у безпосередньому спілкуванні один з одним, а виробник при цьому просто надсилає події споживачеві. Наприклад, виробник веде передачу даних лише до БД або сховища, щоб події збиралися та зберігалися для подальшого аналізу.

Брокер повідомлень відповідає за придбання, зберігання та доставку подій своїм споживачам. Брокер повідомлень повинен бути високонадійним, масштабованим і гарантувати, що він не втрачає події при системних збоях.

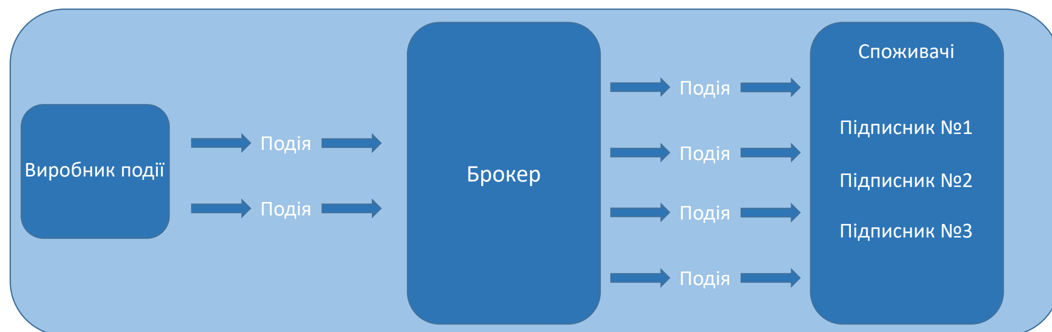

Рис. 3.21. Приклад архітектури на основі подій

Існує дві категорії брокерів повідомлень, що визначаються тим, як вони зберігають дані:

1) **Стандартні.** Ці брокери зберігають події в сховищі даних для обслуговування споживачів. Вони видаляють події зі свого магазину після доставки їх споживачам.

2) **На основі журналів**. Ці брокери зберігають події у журналах, навіть після їх споживання. Оскільки події не видаляються, брокери дозволяють споживачам повторювати події з попереднього моменту часу.

Переваги такої архітектурної моделі наступні:

1) Кращий досвід роботи в режимі реального часу. Інтерфейси, що керуються подіями, покращують інтерактивний досвід для кінцевого користувача. Це досягається за рахунок абстрагування від багатьох обов'язків, які раніше були покладені на користувачів. З іншого боку, це збільшує складність зі сторони виробника, але кінцевий результат того вартий.

2) Низьке зчеплення. ПЗ, яке використовує інтерфейси, потрібно знати лише формат даних подій. Йому не потрібно турбуватися про те, який сервер надсилає дані. Зі свого боку, сервери не зацікавлені в ідентифікації споживачів і просто керуються своєю внутрішньою логікою для створення подій. Отже, в результаті витрачаються менші кошти на управління змінами.

3) Чудове масштабування. Постачальникам інтерфейсів не потрібно відстежувати конкуруючих користувачів або чекати їх відповідей. Кількість підписок на споживчому кінці не має ніякого значення для виробника, якому потрібно лише знати, що те, що він надіслав брокеру, гарантовано буде передано підписникам, скільки б їх не було. Асинхронність – це та характеристика системи, яка ідеально підходить для масштабування, що є ще однією перевагою слабкго зчеплення.

4) Розширюваність і стійкість. Компоненти зі слабким зчепленням також за своєю природою легко розширюються і мають низький ризик регресії. Функціональні можливості можна додати як ще одного незалежного користувача, який ніяк не впливає на інших споживачів.

5) Більш послідовне використання паралельності. Незмінність подій та впорядкування повідомлень полегшують досягнення паралельності в розподілених системах.

6) Оптимізоване використання мережевих ресурсів. Синтез інтерфейсів на основі подій із традиційними підходами, орієнтованими на обслуговування, ча-

сто призводить до незручних конструкцій. Керовані подіями інтерфейси більш оптимально використовують обчислювальні та мережеві ресурси.

Серед недоліків даної архітектурної моделі можна відмітити наступні:

1) Підвищена складність для постачальника інтерфейсів. З інтерфейсами на основі подій бекенд інтерфейсу виконує всю роботу, не в останню чергу підтримуючи стан та ретельно відстежуючи всі зміни стану. Виробник бере на себе весь тягар організації процесу надання досвіду кінцевому користувачу. Після того, як поставка була обіцяна, події мають бути опубліковані з великим ступенем достовірності та цілісності. Реалізація системи таким чином, щоб збої залишалися статистично незалежними, є важкою інженерною проблемою.

2) Неоднозначність в аналітиці інтерфейсів. В архітектурі на основі подій складно відобразити метрику (наприклад, відомості про кількість подій, які надсилаються). Традиційні способи вимірювання ефективності інтерфейсів не можуть застосовуються у випадку використання такої архітектурної моделі.

3) Обмежені управління, стандартизація та досвід/підтримка розробників. Шлюзи інтерфейсів не підтримують протоколи та шаблони на основі подій. Також існує величезна різноманітність реалізацій та рішень, які заважають зусиллям зі стандартизації.

Архітектура на основі моделей (*англ. Model-driven architecture*) – це відкрита та нейтральна до постачальників архітектурна база, яка використовує відповідні стандарти і моделі у ЖЦ розвитку систем у різних сферах та технологіях.

Основною метою розробників даної архітектури було створення такого підходу, який дозволяє знизити ризик, спричинений різницею між технологіями розроблення ПС і платформами. Для вирішення цього завдання вводяться поняття залежної від платформи моделі (Platform Specific Model (PSM)) і незалежної від платформи моделі (Platform Independent Model (PIM)).

Архітектура на основі моделей передбачає створення незалежної від платформи моделі та подальший перехід до залежної від платформи моделі з використанням спеціалізованих засобів. Крім того можливий перехід від однієї залежної від платформи моделі до іншої.

Незалежна від платформи модель – це модель системи, яка не має жодної інформації щодо реалізації конкретної технології.

Залежна від платформи модель – це модель системи, яка містить інформацію про впровадження, специфічну для технології. Наприклад, «загальний» опис системи є незалежною від платформи моделлю, а опис системи з використанням технологій Java або Microsoft .NET – це модель, специфічна для платформи. Існує ряд методів, які перетворюють незалежні від платформи моделі, у конкретні моделі, які залежать від платформи та визначають деталі того, як ці системи використовують можливості своїх платформ для забезпечення необхідних операцій.

Платформа – це набір технологій, включаючи функціональні можливості, що надаються через інтерфейси та шаблони використання. Платформи можуть бути загальними, специфічними для технології (наприклад, платформа Java) або від конкретних постачальників (наприклад, платформа Microsoft .NET). Модель платформи описує платформу, включаючи її частини та послуги, які вона надає.

Поширена послуга – це послуга, доступна на широкому спектрі платформ. Наприклад, файлові послуги, служби безпеки тощо.

Система – це набір програм, підтримуваних набором платформ.

Додаток – функціональна частина системи.

Реалізація – це специфікація, яка містить всю інформацію для побудови та введення системи в експлуатацію. Наприклад, система магазину може бути набором програм Java, які надають додатки для фінансових транзакцій, що підтримуються платформою Java, яка описується моделлю платформи специфікації Java Virtual Machine (JVM).

Модель системи – це опис або специфікація системи та її середовища для певної мети, яка може бути представлена графічно, а також текстово. Модельний підхід зосереджується на моделях для роботи з системами, включаючи їх проектування, оцінку, конструювання, розгортання, експлуатацію, обслуговування та зміну.

Архітектура системи визначає, які елементи входять в систему і як вони працюють разом, щоб забезпечити її функціональність, включаючи ті компоненти та зв'язки, які є частиною системи і взаємодіють між собою.

Точка зору (*англ. viewpoint*) на систему включає перспективу, зосереджену на конкретних проблемах щодо системи, яка приховує деталі. Це необхідно для того, щоб створити спрощену модель, яка має лише ті елементи, що стосуються проблем точки зору.

Вигляд (*англ. view*, або модель точки зору) системи – це представлення системи з перспективи точки зору. Наприклад, точка зору безпеки зосереджена на проблемах безпеки, а її модель точки зору містить ті елементи, які стосуються безпеки з більш загальної моделі системи.

На рис. 3.22 зображено наступні точки зору:

1) Незалежна від обчислень точка зору (*англ. Computation Independent Viewpoint, CIV*) зосереджена на вимогах системи та її оточення.

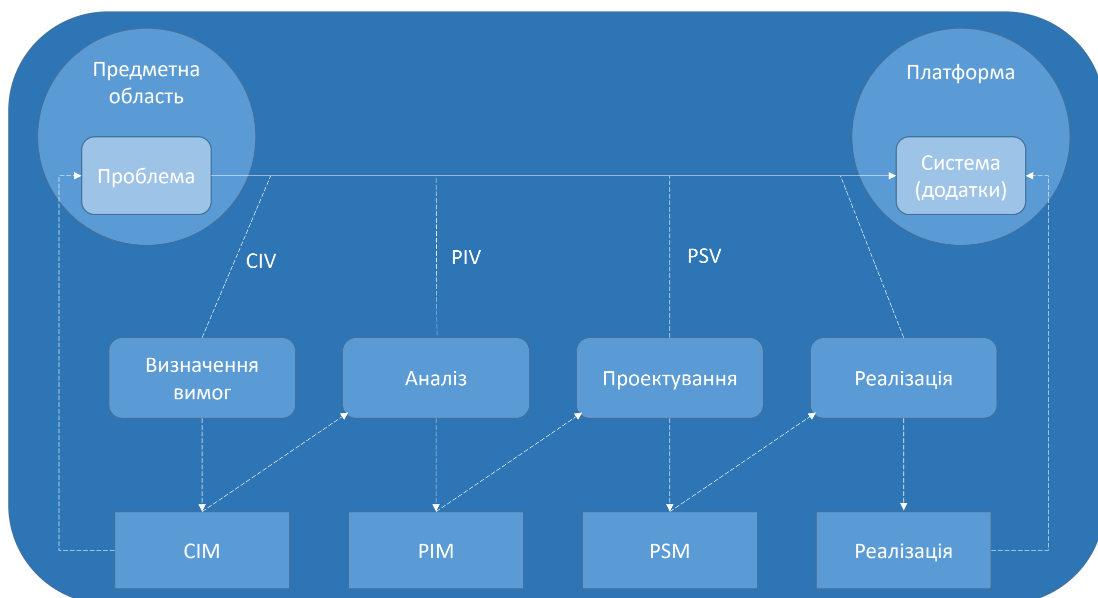

Рис. 3.22. Узагальнений приклад архітектури на основі моделей

2) Незалежна від платформи точка зору (*англ. Platform Independent Viewpoint, PIV*) зосереджена на роботі незалежної від будь-якої платформи системи. Саме тому не змінюється від однієї платформи до іншої. PIV відповідає аналізу та моделі незалежної від платформи системи.

3) Точка зору платформи (*англ. Platform Specific Viewpoint, PSV*) зосереджена на роботі системи на основі конкретної платформи, отже, змінюється від платформи до платформи.

На рис. 3.22 зображено наступні моделі:

1) Незалежна від обчислень модель (*англ. Computation Independent Model, CIM*) системи з незалежною від обчислень точкою зору описує область і вимоги системи. CIM може включати в себе модель інформаційної точки зору, яка фіксує інформацію про дані в системі.

2) Модель незалежної від платформи (*англ. Platform Independent Model, PIM*) системи з незалежною від платформи точкою зору описує роботу системи незалежно від будь-якої платформи. Дана модель може складатися з моделі інформаційної точки зору, яка фіксує інформацію про дані в системі, а також моделі обчислювальної точки зору, яка фіксує інформацію про обробку даних в системі незалежно від будь-якої платформи.

Щоб досягти незалежності від платформи, модель може орієнтуватися на технологічно нейтральну віртуальну машину.

Віртуальна машина – це сукупність компонентів і сервісів, які не залежать від будь-якої конкретної платформи і можуть бути реалізовані на декількох платформах. Віртуальна машина завжди залишається незалежною та не піддається впливу жодної базової платформи.

3) Модель конкретної платформи (*англ. Platform Specific Model, PSM*) системи з точкою зору платформи описує роботу системи, оскільки вона використовує одну або декілька конкретних платформ. Дана модель також може складатися з моделі інформаційної точки зору, яка фіксує інформацію про дані в системі, а також моделі обчислювальної точки зору, яка фіксує інформацію про обробку даних в системі на основі конкретної платформи.

Рис. 3.22 ілюструє лише теоретичні основи та загальні рекомендації щодо побудови системи з використанням архітектури на основі моделей. На сьогоднішній день існує декілька специфікацій (або специфікацій трансформації), включаючи правила та іншу інформацію, необхідну для перетворення незалеж-

них від платформи моделей у моделі для конкретних платформ, але в межах даного підручника вони розглядатися не будуть.

Архітектурні моделі MV(C або P, або VM). Метою цих архітектур є розділення обов’язків візуалізації, обробки та керування даними для додатків з графічним інтерфейсом користувача. Їхні цілі полягають у збільшенні: модульності, гнучкості, тестованості та ремонтопридатності.

Модель-Вигляд-Контролер (*англ. Model-View-Controller, MVC*) – це широко використовуваний шаблон проектування для розроблення програмних додатків. Даний шаблон був розроблений у 1979 та спочатку називався Model-View-Controller-Editor. MVC був детально описаний у «Шаблони проектування: елементи багаторазового об’єктно-орієнтованого програмного забезпечення» [6] у 1994 році, що зіграло певну роль у популяризації його використання. Шаблон розбиває програму на три компоненти (рис. 3.23).

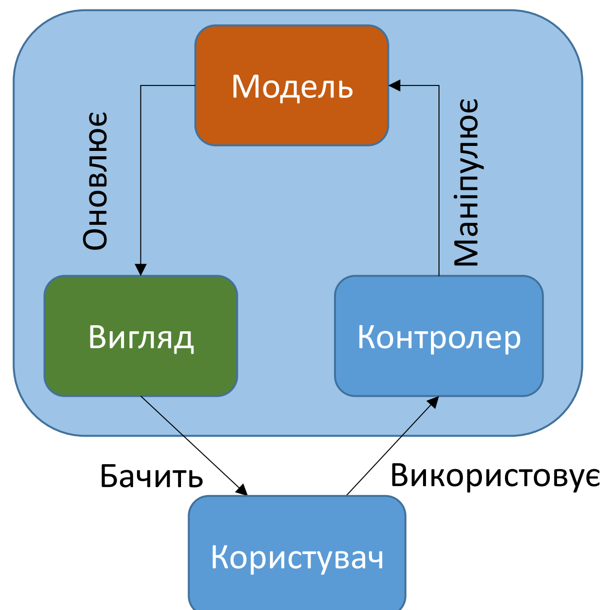

Рис. 3.23. Приклад архітектури MVC

Модель – це набір класів, які описують бізнес-логіку та дані. Вона також визначає бізнес-правила для маніпуляцій даними.

Вигляд (*англ. View*) має два важливих завдання: представлення даних користувачеві та обробка взаємодії з ним.

Контролер потрібен для обробки вхідних запитів. Він отримує дані користувача через модель для перегляду, а також виконує функцію посередника між моделлю та виглядом.

Модель-Вигляд-Ведучий (*англ. Model-View-Presenter, MVP*) – похідна від шаблону проектування MVC, яка фокусується на покращенні логіки відображення. Дана модель виникла у компанії під назвою Taligent на початку 1990-х, коли вони працювали над моделлю для середовища C++ CommonPoint.

Хоча MVP (рис. 3.24) є похідною від MVC, вони все ж мають невеликі відмінності.

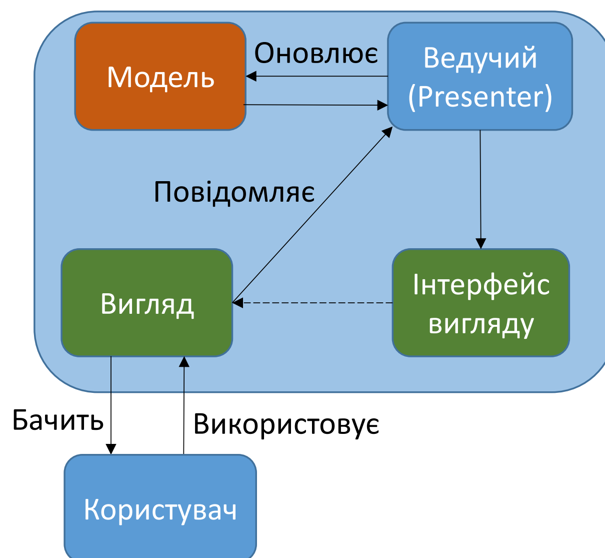

Рис. 3.24. Приклад архітектури MVP

Модель (як і у попередньому випадку) – це набір класів, які описують бізнес-логіку та дані, а також визначає основні бізнес-правила для маніпуляцій даними.

Вигляд визначає компоненти інтерфейсу користувача, такі як jQuery, HTML, CSS тощо. Даний елемент відповідає за відображення даних, отриманих від ведучого (*англ. Presenter*). Він також перетворює модель(і) в інтерфейс користувача.

Ведучий бере на себе відповідальність за вирішення всіх подій інтерфейсу користувача від імені вигляду. Вигляд надає вхідні дані від користувача, а потім користується допомогою моделі, щоб відфільтрувати дані та передати ре-

зультат назад до вигляду. Рівні View і Presenter спілкуються між собою через спільний інтерфейс.

Переваги шаблону MVP:

- View є пасивною частиною даної архітектури, щоб його можна було легко замінити;
- керувати кодом легше;
- на відміну від MVC є сумісним з керованим тестами розробленням (*англ. Test-driven development, TDD*).

Недоліки даної архітектурної моделі:

- не підтримує слабке зчеплення між рівнями View і Presenter;
- має дуже великий розмір коду.

Модель-Вигляд-МодельВигляду (*англ. Model-View-ViewModel,* MVVM) також є похідною моделлю від MVC. Шаблон MVVM (рис. 3.25) підтримує двостороннє зв'язування даних між View та ViewModel. Він дозволяє автоматично розповсюджувати зміни, які виникають усередині ViewModel до View.

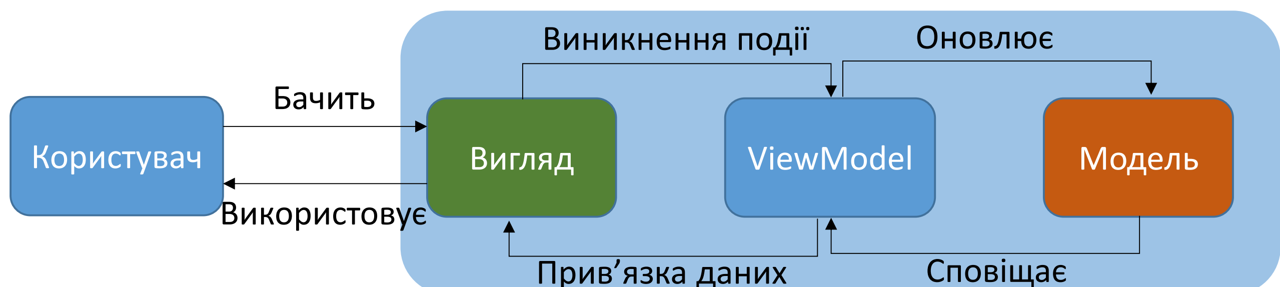

Рис. 3.25. Приклад архітектури MVVM

MVVM також складається з трьох ключових частин:

Модель (як і у попередніх двох випадках) – це набір класів, які описують бізнес-логіку та дані, а також визначає основні бізнес-правила для маніпуляцій даними. Це безпосередньо пов'язано з діяльністю БД.

Вигляд визначає компоненти інтерфейсу користувача, такі як: jQuery, HTML, CSS тощо. Вигляд відповідає за відображення даних, отриманих від ViewModel. Це єдина доступна для користувача інтерактивна частина програ-

ми. Він також перетворює модель(і) в інтерфейс користувача. Вигляд у шаблоні MVVM є активним на відміну від MVC та MVP.

ViewModel відповідає за представлення функцій, методів і команд для підтримки стану View, роботи з моделлю та активації подій у самому View. Його можна визначити як модель для перегляду програми.

Переваги шаблону MVVM:

- низьке зчеплення між моделлю ViewModel і View;
- забезпечує найкращу сумісність з керованим тестами розробленням.

Його недоліки:

- кожен компонент інтерфейсу користувача потребує елементи, які будуть викликати події;
- має великий розмір коду.

3.3.5 Документування архітектури

Будь-яке ПЗ має архітектуру, яка може бути простою або дуже складною, розробленою заздалегідь або результатом впровадження. Більше того, кожна архітектура постійно розвивається. Тому важливо не тільки передати команді розроблення все, що пов'язано з архітектурним уявленням ПС, але й підтримувати цю інформацію в актуальному стані.

Існує багато стандартів для документування архітектури ПЗ. Один архітектор програмного забезпечення використовуватиме для цього UML (*англ. Unified Modeling Language, уніфікована мова моделювання*), другий – зовсім інший підхід. Проте у використаних методологіях документування завжди будуть схожі елементи (діаграми, схеми, тощо). У реальності архітектурна документація потребує більш ніж одного типу діаграм, щоб бути повною в будь-якому випадку використання.

UML. Існує декілька діаграм, які створюються за допомогою UML. Ці діаграми можна умовно розділити на дві категорії [26-28]:

- Поведінкова діаграма UML, яка складається з:
  - діаграми діяльності;
  - діаграми варіантів використання;

- діаграми огляду взаємодії;
- часової діаграми;
- діаграми стану;
- схеми зв'язку;
- діаграми послідовності.

- Структурна діаграма UML включає в себе наступні частини:
  - діаграму класів;
  - діаграму об'єкта;
  - діаграму компонентів;
  - схему композитної структури;
  - схему розгортання;
  - схему пакету;
  - діаграму профілю.

Детальний опис кожної з зазначених вище діаграм та схем є дуже об'ємним, тому у даному підручнику цей матеріал розглядатися не буде. З ними можна більш детально познайомитися, прочитавши додаткову літературу.

Загалом, UML – це дуже потужний інструмент для документування архітектурних рішень, який підходить як для чорнового, так і для детального проектування архітектури.

Однак, щоб задокументувати всю архітектуру програми за допомогою UML, доведеться використовувати одночасно декілька типів діаграм. При спробі використати, наприклад, одну діаграму класів, щоб описати всю програму, виникнуть значні проблеми, пов'язані з розмірами цієї програми.

Прикладом доцільного та правильного використання діаграми класів UML є, наприклад, документування шаблонів проектування (рис. 3.26).

Дана діаграма може відображати класи, інтерфейси, зручність використання та відносини успадкування, дані та поведінку. Вона також стисла і легко читається. Її незначний розмір дозволяє значно зменшити час на створення.

Однак, наведений нижче приклад (рис. 3.27) ілюструє один з недоліків неправильного використання. Діаграма дуже об'ємна, тому вона сильно заплутана і складна в розумінні. Крім того, на її створення буде потрібно так багато

часу, що коли вона буде закінчена, то вже буде застарілою, тому що хтось (у процесі її побудови) вже встиг внести зміни в код.

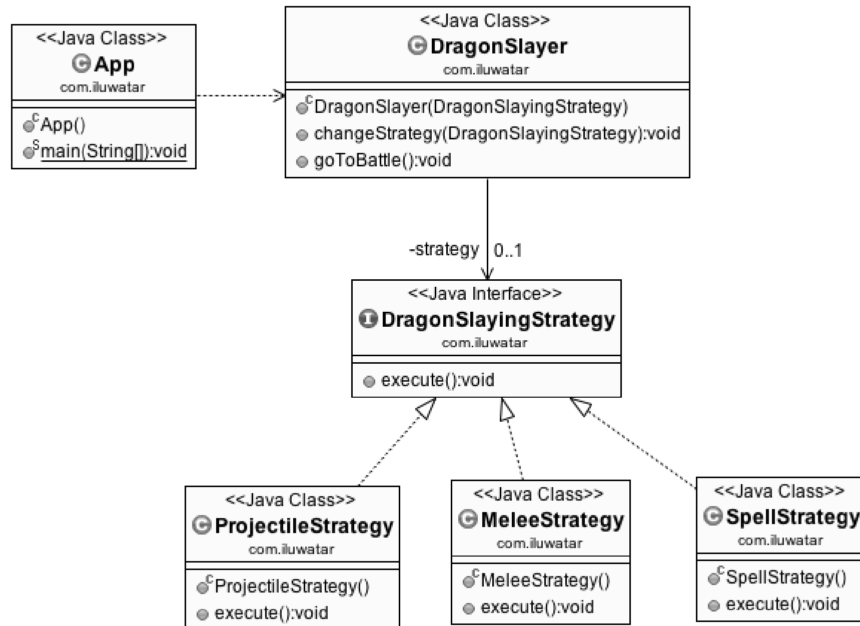

Рис. 3.26. Приклад діаграми класів для шаблону «Стратегія»

Рис. 3.27. Приклад невдалого використання діаграми класів

Отже, можна і необхідно використовувати UML лише в певних ситуаціях, наприклад: для детального опису шаблонів, невеликих частин програми або високої деталізації програми з низькою деталізацією.

При використанні UML проблема документування ПЗ залишається вирішеною не повністю.

Архітектурний вигляд моделі 4+1. Дана модель (рис. 3.28) була створена і опублікована ще в 1995 році в статті під назвою «Архітектурні креслення – модель для перегляду системної архітектури 4+1».

Цей спосіб візуалізації архітектури ПС заснований на п’яти уявленнях/перспективах програми та розповідає, які діаграми можна використовувати для документування кожного з цих уявлень.

1) Логічне/структурне подання. Турбується про функціональні можливості, які надає система, і про якість розробленого для забезпечення такої функціональності коду.

2) Подання реалізація/розробник. Відображає статичну організацію коду, компонентів, модулів і пакетів.

Концептульні/Логічні

Фізичні/Робочі

Функціональні

Логічне/структурне подання
Важливо для: аналітики, архітектори
Етап: визначення вимог
Фокус: об’єктно-орієнтована декомпозиція
Важливість: функціональність
Артефакти:
- діаграма класів;
- діаграма об’єктів;
- схема композитної структури.

Подання реалізація/розробник
Важливо для: розробники, менеджери
Етап: проектування
Фокус: декомпозиція на підсистеми
Важливість: управління життєвим циклом
Артефакти:
- діаграма компонентів;
- схема пакету.

Варіанти використання/сценарії
Важливо для: кінцеві користувачі
Етап: складання в єдине ціле
Фокус: декомпозиція на корисні особливості
Важливість: зрозумілість, зручність використання
Артефакти: - діаграма варіантів використання; - історії користувачів.

Нефункціональні

Подання процес/поведінка
Важливо для: системні інтегратори
Етап: проектування
Фокус: декомпозиція процесів
Важливість: продуктивність, масштабованість, пропускна здатність
Артефакти: - діаграма послідовності; - схема зв’язку; - діаграма діяльності; - діаграма стану; - часова діаграма; - оглядова діаграма взаємодії.

Розгортання/фізичний вигляд
Важливо для: системні інженери
Етап: проектування
Фокус: зіставити програмне забезпечення з апаратним забезпеченням
Важливість: топологія системи, доставка, встановлення, зв’язок (комунікація)
Артефакти: - схема розгортання; - топологія мережі.

Рис. 3.28. Елементи опису системної архітектури 4+1

3) Подання процес/поведінка. Воно зосереджене на поведінці системи під час її роботи, наприклад, на взаємодії системних процесів, паралельності, синхронізації, продуктивності, тощо.

4) Розгортання/фізичний вигляд. Ілюструє фізичну організацію програми. Показує який код виконується і на якому обладнанні.

5) Варіанти використання/сценарії. Архітектура в цілому пояснюється за допомогою декількох варіантів використання (Use Case сценаріїв), які є послідовністю взаємодій. Частина архітектури розвивається з таких варіантів використання.

Важливо зазначити, що архітектурний вигляд моделі 4+1 не вимагає використання всіх згаданих діаграм або уявлень. Завжди необхідно розуміти як працювати з інструментами та використовувати не більше (але і не менше) того, що необхідно для рішення поточної задачі.

Записи архітектурних рішень (*англ. Architecture Decision Records, ADR*). Основна ідея їх використання полягає не у документуванні поточного або майбутнього стану архітектури ПС, а натомість у документуванні причин, які привели до цього. Ці записи є особливо важливими, тому що мають намір розповісти іншим членам команди розроблення, чому архітектура саме така, яка вона є на даний момент.

ADR – це запис у журналі про рішення архітектора, які були прийняті і які призвели до поточного або майбутнього стану архітектури. Вони доповнюють діаграми, які описують архітектуру, тому що містять причину прийняття певного рішення.

Архітектурно-важлива вимога (*англ. Architecturally-Significant Requirement, ASR*): вимога, вплив якої на архітектуру ПС можна виміряти.

Архітектурне рішення (*англ. Architecture Decision, AD*): вибір проекту програмного забезпечення, який відповідає важливим вимогам.

Записи архітектурних рішень: документ, який фіксує важливе архітектурне рішення, прийняте разом з його контекстом і наслідками.

Журнал архітектурних рішень (*англ. Architecture Decision Log, ADL*): колекція всіх ADR, створених і підтримуваних для конкретного проекту (або організації).

Управління знаннями архітектури (*англ. Architecture Knowledge Management, AKM*): вища сфера всіх попередніх концепцій.

ADR – це не просто документ, написаний після обговорення та прийняття рішення. Найкращий спосіб його використовувати – як відправну точку для обговорення, як запит на коментарі (*англ. Request for Comments, RFC*), який є ідеєю/пропозицією, що надсилається іншим членам команди/відділу, запитуючи їх внесок/думку/схвалення.

Основна мета даного підходу – його використання, щоб почати обговорення, провести мозковий штурм, прийняти найкраще можливе рішення та використати сам документ пропозиції як запис у журналі рішень (ADR).

Проте, той факт, що ADR написаний заздалегідь, не означає, що він незмінний. Його потрібно оновлювати/удосконалювати в міру розгортання обговорення. Особливо важливо, щоб усі варіанти, які розглядаються, були записані з їх плюсами та мінусами, щоб розпалити дискусію та прийняти чітке рішення.

Модель С4 була представлена Саймоном Брауном, і це один з найкращих підходів до документування архітектури ПЗ.

Ідея полягає в тому, щоб використовувати чотири різні рівні деталізації (або масштабування) для документування архітектури ПЗ:

- діаграма системного контексту;
- діаграма контейнера;
- діаграма компонентів;
- кодова діаграма.

Рівень 1 (діаграма системного контексту). Це діаграма найвищої деталізації (рис. 3.29). У ній мало деталей, але головна мета – описати контекст, в якому знаходиться ПС. Таким чином, вона буде складатися з одного єдиного блоку для всієї програми, і буде оточеною іншими блоками, які посилаються на зовнішні системи та користувачів, з якими взаємодіє додаток.

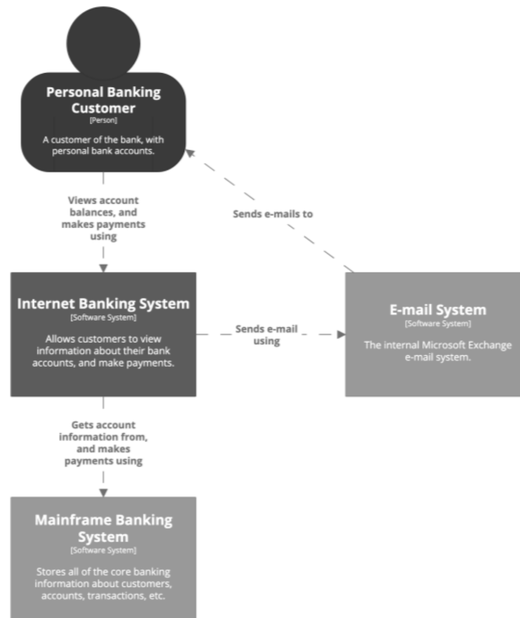

Рис. 3.29. Діаграма системного контексту

Рівень 2 (діаграма контейнера). На цьому рівні деталізації (рис. 3.30) можна побачити контейнери програми, де контейнером є будь-який незалежний технічний елемент програми, наприклад, мобільний додаток, програмний інтерфейс або БД. Дана діаграма також документує основні технології, які використовуються, та взаємодію контейнерів між собою.

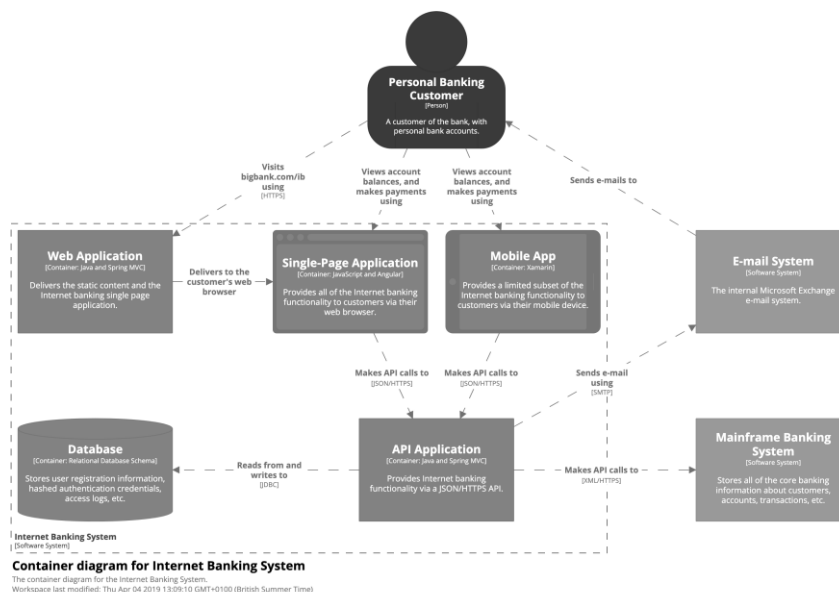

Рис. 3.30. Діаграма контейнера

Рівень 3 (діаграма компонентів). Діаграма компонентів (рис. 3.31) показує компоненти всередині одного контейнера. У цьому контексті кожен компонент є модулем програми, не обмежуючись модулями домену, але також включає суто функціональні модулі (наприклад, електронна пошта, sms, тощо). Отже, ця діаграма показує головні модулі контейнера та всі взаємозв'язки між цими модулями.

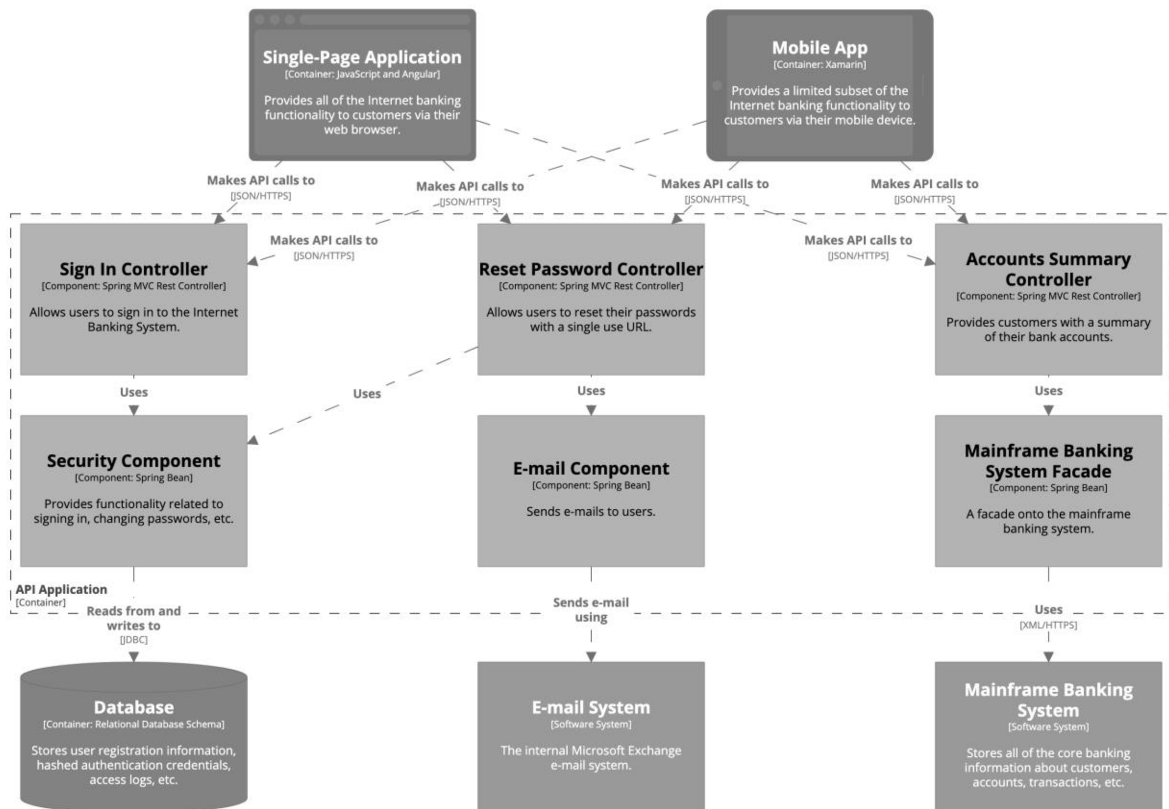

Рис. 3.31. Діаграма компонентів

Рівень 4 (кодова діаграма). Найбільш детальна діаграма, спрямована на опис структури коду всередині компонента. Для уявлення цього рівня на рис. 3.32 наведено діаграму мови UML.

Модель С4 – це чудовий спосіб задокументувати архітектуру програми та можливість зрозуміти архітектуру програми до певного рівня.

На наведених діаграмах можна побачити декілька обмежень:

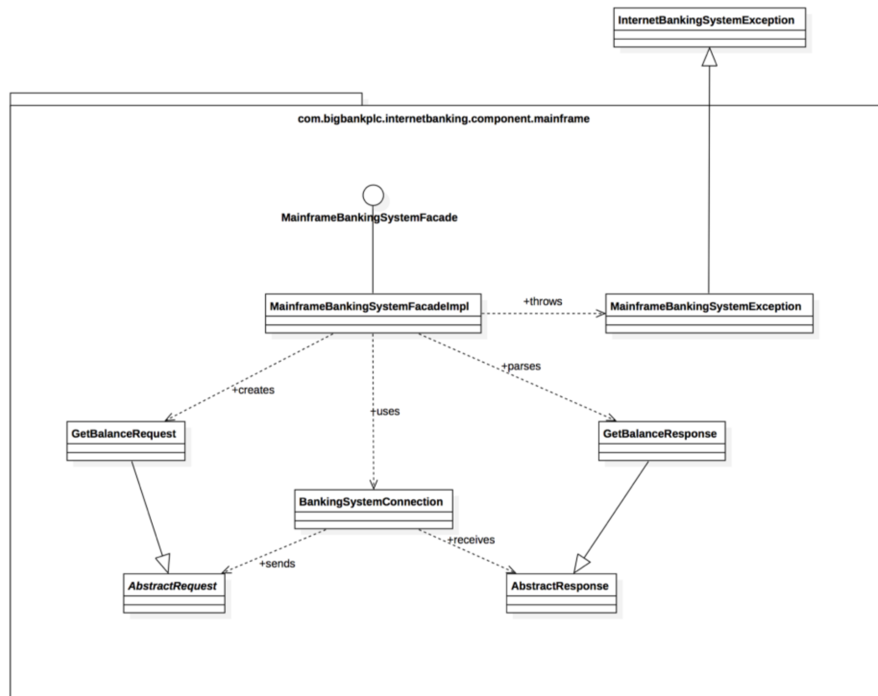

Рис. 3.32. Діаграма компонентів

1) Найчастіше їх потрібно створювати вручну, а не автоматизовано чи безпосередньо витягувати з коду. Це означає, що вони можуть не відображати фактичний код, а натомість лише поточне його розуміння.

2) Вони не зовсім допомагають побачити, що не так у кодовій базі додатків, безладних відносин коду та поганої структури, що саме впливає на модульність та інкапсуляцію, що є важливим для будь-якого ПП.

3) Вони не допомагають зрозуміти кодову базу в цілому, що можуть робити модулі програми та як вони взаємодіють один з одним.

Вибір того чи іншого способу документування архітектури ПЗ залежить від конкретного проекту та від навичок архітектора. Він може бути як вдалим, так і ні.

Контрольні запитання

1. Як суттєві та несуттєві властивості об’єктів впливають на процес проектування архітектури?

2. Чому важливо управляти складністю проекта?

3. Які є загальні характеристики якості розроблюваного ПЗ?

4. Що таке системна архітектура? З яких етапів складається процес її проектування?

5. Що таке функціональні та нефункціональні вимоги до програмного забезпечення?

6. Що таке декомпозиція? Які особливості її використання при проектуванні архітектури програмного забезпечення?

7. Які підсистеми найчастіше можна використовувати повторно у різних проектах?

8. Які є основні архітектурні парадигми? У чому полягають відмінності між ними?

9. Які принципи покладено в основу об'єктно-орієнтованого підходу?

10. Які переваги використання компонентно-орієнтованого та сервісно-орієнтованого підходів?

11. Чому в сучасній практиці розроблення програмного забезпечення об'єктно-, компонентно- та сервісноорінтований підходи використовуються в межах одного і того ж проекту?

12. Що таке варіант (сценарій) використання? З яких частин він складається та як використовується?

13. Які чотири архітектурні стилі використовуються найчастіше? Яка між ними різниця?

14. Чим відрізняються відкрита та закрита багаторівневі архітектури?

15. Чим відрізняються файл-, клієнт-сервер та трирівнева архітектури?

16. Які особливості використання сервісно-орієнтовної архітектури?

17. Чому архітектуру на основі подій називають «асинхронною комунікацією»?

18. Яка різниця між різними варіантами MV архітектур?

19. Які є методики документування архітектури?

20. У чому різниця між документуванням архітектури за допомогою UML та C4?

РОЗДІЛ 4

ПРИНЦИПИ SOLID ТА ШАБЛОНИ ОБ'ЄКТНО-ОРІЄНТОВАНОГО ПРОГРАМУВАННЯ

SOLID – це п'ять принципів (рис. 4.1) керування залежностями для об'єктно-орієнтованого програмування та проектування. Абревіатуру SOLID увів Роберт Сесіл Мартін [3]. Кожна літера позначає інший трибуквений акронім, який описує один принцип.

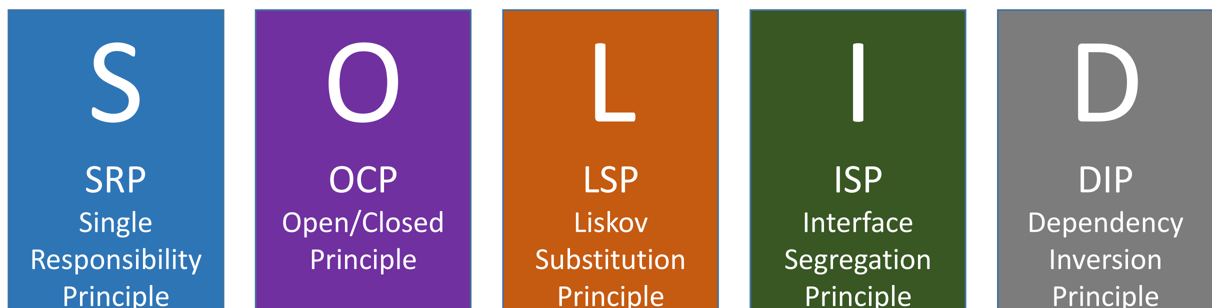

Рис. 4.1. Принципи SOLID

При роботі з ПЗ, в якому управління залежностями обробляється погано, код може стати жорстким, крихким і важко використовуватися повторно.

Жорсткий код (*англ. Rigidity*) – це код, який важко розширити, щоб змінити існуючу функціональність або додати нові функції. Тобто маленькі зміни можуть призводити до перебудови всієї системи.

Крихкість (*англ. Fragility*) – це тенденція ПЗ ламатися в багатьох місцях при кожній його зміні. Невелика зміна, внесена в одну частину програми, часто ламає інші частини програми, які не мають концептуального зв'язку з першою. У міру того, як крихкість зростає, збільшується ймовірність поломки. Таке ПЗ неможливо обслуговувати. Кожне виправлення погіршує ситуацію, створюючи більше проблем, ніж вирішується.

Нерухомість (*англ. Immobility*) – це неможливість повторного використання елементів ПЗ з інших проектів або частин того самого проекту.

Часто розробники усвідомлюють, що їм знадобиться деяка функціональність, яка вже реалізована в програмі. Однак дана функціональність має багато зв'язків, що ускладнює відокремлення та повторне використання. Після ретельного аналізу розробник визначає, що рефакторинг має більший ризик для життєво важливих функцій. Тому ПЗ просто переписується, а не використовується повторно.

Якщо дотримуватися принципів SOLID, то можна створити код, який є більш гнучким і надійним, а також має більшу можливість повторного використання.

Отже, наведемо деякі з найважливіших переваг дотримання принципів SOLID при написанні коду:

1) Легше швидко розширити систему новими функціями, не порушуючи існуючі. Якщо якісно не реалізувати нову функцію в програмі, це може вплинути на існуючу функціональність і викликати ненавмисні проблеми.

2) Код легший для читання та розуміння. Таким чином, витрачається менше часу на з'ясування того, що він робить.

3) Код стає більш зручним для підтримки.

4) Кожне ПЗ час від часу потребує оновлення. Тому ПЗ повинно створюватись, пам'ятаючи про можливість майбутніх змін. На початкових етапах розроблення реорганізувати код за принципами SOLID ще досить легко.

5) Якщо принципи SOLID застосовуються у коді, це також забезпечить легкий процес налагодження в цілому.

4.1 Принцип єдиної відповідальності

Принцип єдиної відповідальності (*англ. Single responsibility principle, SRP*) або принцип єдиної мінливості – простий, але вкрай складний для розуміння для будь-якого програміста-початківця.

Роберт Сесіл Мартін дав визначення цьому принципу у своїй книзі «Гнучке розроблення програмного забезпечення: принципи, структури та практики»

[18], а потім у книзі «Гнучкі принципи, структури та практики в C#» наступним чином: «у класу має бути лише одна причина для зміни».

Хоча визначення і виглядає досить просто, але досягнення цієї простоти може бути складним. Чому у класу має бути лише одна причина для зміни? Чому це так важливо?

У жорстко типізованих та мовах, які компілюються, декілька причин виникнення зміни можуть призвести до декількох небажаних перерозподілів. Коли є дві різні причини для зміни класу, то цілком може виявитися, що дві різні команди працюють над одним і тим самим кодом, але з різних причин.

Розглянемо наступний приклад (рис. 4.2): визначення кількості робочих днів для бухгалтерії, яка проводить розрахунок заробітної плати, та відділу кадрів, який рахує робочий стаж. У першому випадку для обох користувачів кількість робочих днів співпадає і програма працює коректно. Проте пізніше вийшов закон, за яким державні свята, незважаючи на статус «вихідний день», оплачуються як робочі дні. Це призводить до необхідності внесення змін у алгоритм розрахунку кількості робочих днів з боку бухгалтерії.

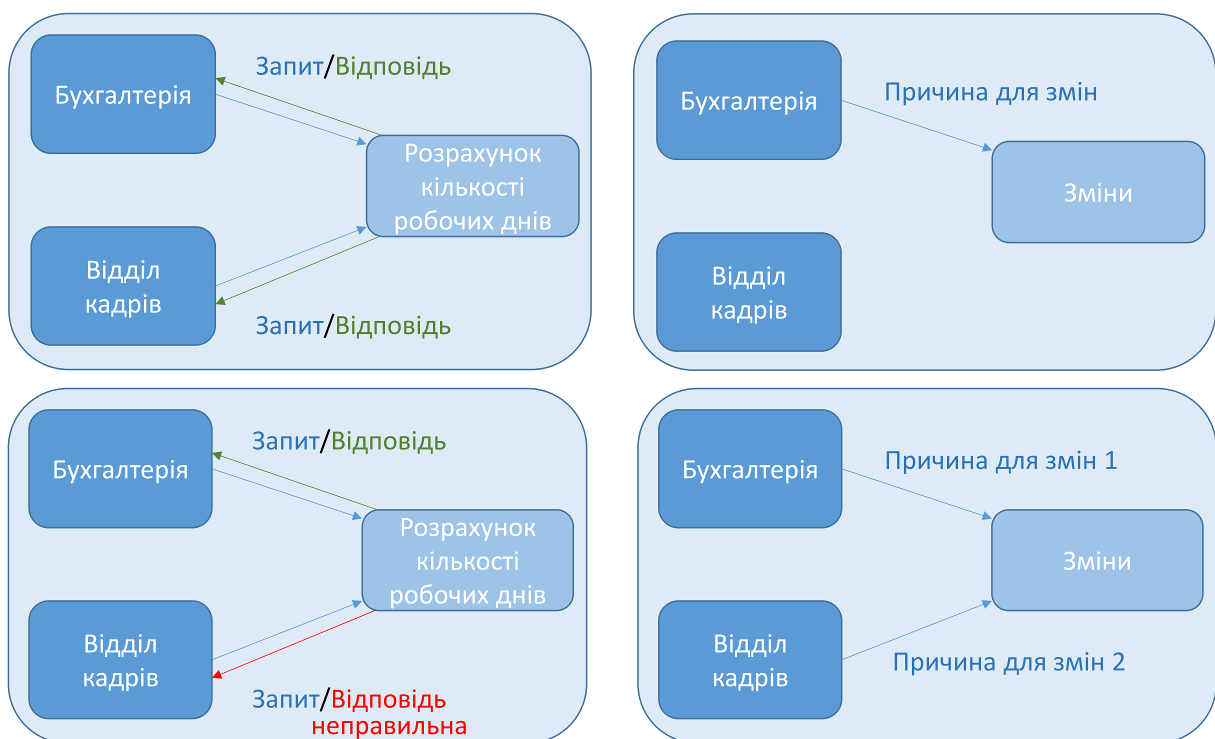

Рис. 4.2. Приклад порушення принципу єдиної відповідальності

Проте для відділу кадрів ці зміни не потрібні і кількість робочих днів, яку він отримує, буде неправильною. Дана помилка може бути помічена не відразу, тому що декілька днів на рік у межах глобального стажу не викликають значних проблем. Але, якщо виникнуть проблеми з заробітною платою, то даний недолік буде виявлено значно швидше і з'явиться вже дві причини, які викликають необхідність внесення змін у один і той же компонент.

Отже, кожна команда розробників може надати своє власне рішення, яке у випадку з мовами, що компілюються (наприклад, C++, C# або Java), може призвести до створення компонентів, несумісних з рештою частин розроблюваного ПЗ.

Навіть незважаючи на те, що може і не використовуватися мова, що компілюється, все одно може знадобитися повторне тестування класу або програмного модуля з різних причин. Це означає більше роботи над контролем якості, тобто більше часу та зусиль.

Отже, можемо розглянути три наступних (рис. 4.3) варіанти виникнення змін у компонентах програми:

1) Є одна причина для внесення змін, які стосуються лише частини змінюваного модуля. У даному випадку модуль відповідно до принципу єдиної відповідальності необхідно розділити на два окремі модулі. За необхідності, треба провести подальшу декомпозицію утворених модулів.

2) Є дві різні причини для внесення змін, які стосуються окремих частин змінюваного модуля. У даному випадку модуль також необхідно розділити на два окремі модулі, зміни яких викликані відповідними причинами. За необхідності, треба провести подальшу декомпозицію утворених модулів.

3) Є одна причина для внесення змін, які стосуються декількох окремих частин ПЗ. У даному випадку відповідні частини необхідно об'єднати в один модуль.

Визначення однієї персональної відповідальності, яку клас чи модуль повинен мати, є набагато складнішим завданням, ніж просто прохід по якомусь контрольному списку. Наприклад, щоб знайти можливі причини зміни, можна провести аналіз аудиторії розглянутого класу.

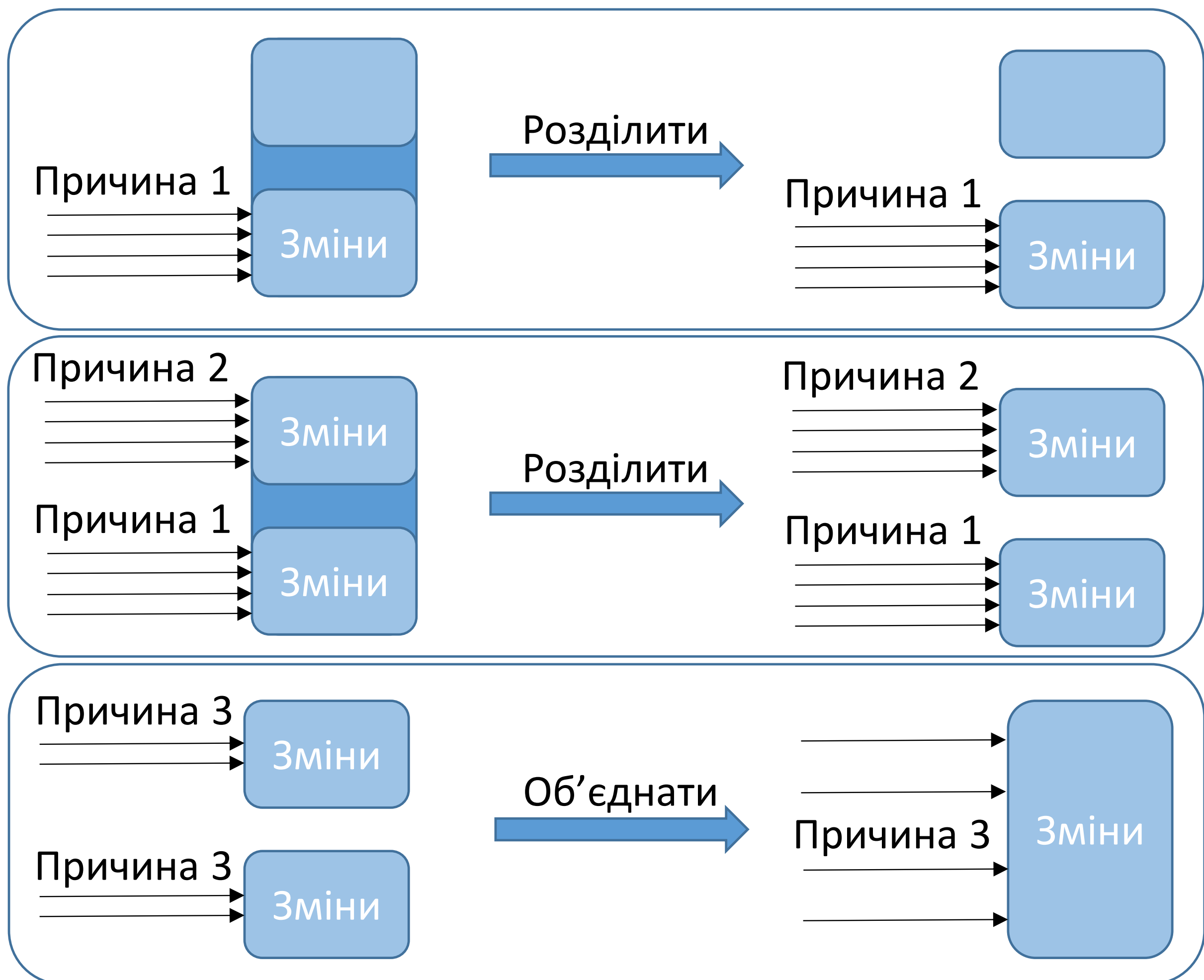

Рис. 4.3. Рекомендації щодо принципу єдиної відповідальності

Аудиторія – це користувачі розроблюваної програми або системи, які обслуговуються конкретним модулем та можуть запросити внесення в нього певних змін [3]. На практиці аудиторія дуже часто вимагає внесення змін. Наведемо декілька прикладів модулів та їх можливі аудиторії:

- модуль зберігання – аудиторія включає архітекторів та адміністраторів БД;
- модуль звітів – аудиторія складається із бухгалтерів, секретарів та інших;
- модуль обчислень оплати для системи заробітної плати – аудиторія може включати бухгалтерів, менеджерів та юристів;
- модуль пошуку книг для системи управління бібліотекою – аудиторія може містити бібліотекарів та/або самих клієнтів.

Порівняння конкретних осіб попри всі ці ролі може бути дуже складним. У маленькій компанії одна людина може мати декілька ролей. Натомість у великій компанії може бути декілька людей, прив'язаних до однієї ролі. Тож необхідно задуматися про розподіл ролей. Визначивши ролі, набагато простіше уявити акторів для них та пов'язувати аудиторію з цими акторами.

Виходить, що аудиторія визначає причини змін, а актори визначають аудиторію. Це значно допоможе скоротити концепції конкретних осіб, наприклад, «Олег – архітектор» відноситься до архітектури, а «Анна – референт» – до операцій.

Таким чином, відповідно до формулювання, яке надав Роберт Сесіл Мартін, **відповідальність** – це набір функцій, які обслуговують одного актора.

Актори можуть стати джерелом змін для набору функцій, які їх обслуговують. Щойно змінюються їхні потреби, також має змінитися певний набір функціоналу, щоб задовольнити нові вимоги.

За словами Роберта Мартіна: «Актор для відповідальності є єдиним джерелом змін цієї відповідальності» [18].

Розглянемо декілька класичних прикладів.

Об'єкти, які можуть друкувати самі себе. Припустимо, що є клас Book, який інкапсулює сутність книги та її функціонал.

```
public class Book {
   public String getTitle() {
      // інструкції для визначення назви книги
   }
   public String getAuthor() {
      // інструкції для визначення автора книги
   }
   public void turnPage() {
      // посилання на наступну сторінку
   }
   public void printCurrentPage() {
```

```
        // інструкції для друку сторінки
    }
}
```

Це може виглядати як цілком допустимий клас: є книга, можна отримати її назву, автора, можна переглянути її сторінку. Також вона може надрукувати поточну сторінку. Але тут є маленька проблема. Якщо подумати про суб'єктів, які можуть використовувати об'єкт типу Book, то можна виділити двох: управління книгами (наприклад, як бібліотекар) і механізм представлення даних (наприклад, як буде подано зміст користувачам – на екрані, використовуючи графічний інтерфейс або це може бути тільки текстовий інтерфейс користувача або друку). Тобто, існує два абсолютно різних суб'єкта.

Змішування бізнес-логіки з логікою відображення даних – погане рішення, тому що це порушує принцип єдиної відповідальності. Переглянемо наступний код:

```
interface Printer {
    void printPage(int page);
}

class TextPrinter implements Printer {
    void printPage(int page) {
        // інструкції для друку сторінки
    }
}

class HtmlPrinter implements Printer {
    void printPage(int page) {
        // інструкції для друку сторінки у вигляді html
    }
}
```

```
public class Book {
    public String getTitle() {
        // інструкції для визначення назви книги
    }
    public String getAuthor() {
        // інструкції для визначення автора книги
    }
    public void turnPage() {
        // посилання на наступну сторінку
    }
}
```

Навіть цей дуже простий приклад показує, як відокремлення відображення від бізнес-логіки, а також дотримання принципу єдиної відповідальності, дає великі переваги у гнучкості дизайну.

Об'єкти, які можуть зберігати себе. Такі об'єкти зазвичай можуть не тільки самі себе зберігати, але й потім діставати себе зі сховища. Приклад є аналогічним до попереднього випадку, тож код розглядатися не буде.

Для даного прикладу знову можна виділити двох суб'єктів: система управління книгами та сховище. Як тільки виникає необхідність змінити сховище, доведеться змінити і цей клас. Як тільки потрібно буде змінити спосіб переходу з однієї сторінки на іншу, потрібно також буде змінити цей клас. Тож тут є декілька причин для зміни.

Для даного випадку не потрібно створювати окремий інтерфейс, оскільки перенесення операцій зі сховищем до іншого класу достатньо сильно розділить обов'язки, і з'явиться можливість вільно змінювати механізми зберігання, без зміни класу Book.

Менш очевидний приклад вихідної задачі. Розглянемо клас Book з відображенням місця знаходження обраної книги.

Якщо бізнес вирішить відмовитися від послуг бібліотекаря та створити механізм самообслуговування в бібліотеці, то можна сказати, що даний приклад не порушує принцип єдиної відповідальності: читачі також є бібліотекарями, їм потрібно самим знаходити книги. Це цілком допустимо.

```
public class Book {
    public String getTitle() {
        // інструкції для визначення назви книги
    }
    public String getAuthor() {
        // інструкції для визначення автора книги
    }
    public void turnPage() {
        // посилання на наступну сторінку
    }
    public String getLocation() {
        // інструкції для визначення місця знаходження книги (номер стенду, полички, тощо)
    }
}
```

У разі наявності ролі бібліотекаря на перший погляд така конструкція класу може також здатися цілком допустимою. Відсутній метод, який взаємодіє зі збереженням чи відображенням даних, є стандартний функціонал *turnPage()* і декілька способів надання різної інформації про книгу.

Однак, і в даному випадку можуть виникнути проблеми. Щоб їх з'ясувати, необхідно детально проаналізувати програму. Проблема може бути у методі *getLocation()*.

Усі методи в класі Book відносяться до бізнес-логіки. Тому треба й дивитися з погляду бізнесу. Якщо додаток використовується реальними бібліотеками, які шукають книги та видають реальну фізичну книгу, то у даному випадку принцип єдиної відповідальності може бути порушено.

Можна помітити, що операціями актора-бібліотекаря є ті, які реалізовано в методах *getLocation(), getTitle()* та *getAuthor()*. Клієнти також можуть мати доступ до програми, щоб вибрати книгу і прочитати перші декілька сторінок. Це необхідно їм, щоб зрозуміти суть книги та потрібна вона їм чи ні. Таким чином, актори-читачі можуть бути зацікавлені у всіх методах, крім *getLocation()*. Для звичайного клієнта не має значення, де саме в бібліотеці зберігається книга, тому що вона буде передана клієнту бібліотекарем. Таким чином можна зробити висновок, що дійсно присутнє порушення принципу єдиної відповідальності.

Для вирішення даної проблеми можна реалізувати клас BookFinder, в якому буде зацікавлений лише актор-бібліотекар. Клієнту ж необхідний тільки клас Book. Звичайно, є декілька способів реалізувати клас BookLocator. Наприклад, він може використовувати автора та назву книги, щоб отримати необхідну інформацію. Це завжди залежить від бізнес-правил.

Важливим є те, що якщо бібліотека зміниться, а бібліотекарю доведеться шукати книги в організованому зовсім по-іншому місці, то сам клас Book при цьому змінюватися не буде. Так само, якщо буде вирішено надавати читачам короткий зміст книги замість можливості перегляду декількох перших сторінок, – це не торкнеться ні бібліотекаря, ні процесу пошуку книг на полицях.

```
public class Book {
   public String getTitle() {
      // інструкції для визначення назви книги
   }
   public String getAuthor() {
      // інструкції для визначення автора книги
   }
   public void turnPage() {
      // посилання на наступну сторінку
   }
}
```

```
class BookFinder {
    public String getLocation(Book book) {
        // інструкції для визначення місця знаходження книги (номер стенду, полички, тощо)
    }
}
```

Отже, важливо пам'ятати, що завжди необхідно ретельно розглядати вимоги бізнесу.

Якщо проаналізувати наведений код глибше та згадати усе, що було сказано вище про принцип єдиної відповідальності, то можна зробити висновок, що у класі Book все ж присутній «зайвий» метод – *turnPage()*.

Проте, щоб не розбивати програму на мільйон окремих класів, які реалізують по одному методу кожен, існує ще одне доповнення до принципу єдиної відповідальності: «Схожі для зміни сутності мають знаходитися в одному місці» або «Те, що змінюється разом, має знаходитися разом». Друге висловлювання відповідає третьому випадку, який зображений на рис. 4.3.

Іншими словами можна сказати, що принцип єдиної відповідальності вимагає не просто «дробити поки дробиться», але і не перестаратися – не розділити правильно з'єднані сутності.

Це значить, що метод *turnPage()* знаходиться на своєму місці, оскільки усі методи, які стосуються актора-читача, знаходяться також у класі Book, а також необхідність внесення змін у даний метод не зачіпатиме операції актора-бібліотекаря.

Основною цінністю ПЗ є простота його зміни. Вторинною є функціональність у сенсі максимально можливого задоволення вимог користувачів. Однак, щоб досягти другої мети, виконання первинної мети є обов'язковим. Щоб зберегти простоту зміни на високому рівні, необхідно мати архітектуру, яку легко змінювати, розширювати, додавати новий функціонал та забезпечувати дотримання принципу єдиної відповідальності:

1) Більша простота зміни призводить до більшого рівня функціональності.

2) Широка функціональність означає потреби користувачів.

3) Потреби користувачів визначають потреби суб'єктів.

4) Потреби суб'єктів визначають потреби зміни цих суб'єктів.

5) Потреби зміни суб'єктів визначають відповідальності.

Таким чином, при проектуванні ПЗ необхідно:

1) Знайти та виділити акторів (суб'єктів).

2) Визначити їх відповідальності.

3) Згрупувати функціональні можливості та класи таким чином, щоб кожен із них мав лише одну певну відповідальність.

Отже, для дотримання принципу єдиної відповідальності необхідно:

1) Розділяти компоненти таким чином, щоб кожен із них мав лише одну відповідальність.

2) Відповідальність – причина для зміни. Тобто кожен елемент має лише одну причину для зміни у термінах бізнес-логіки.

3) Потенційні зміни бізнес-логіки мають бути чітко локалізовані, тобто змінювані разом елементи мають бути поруч.

Необхідні критерії самоперевірки дотримання даного принципу:

1) Задайте собі наступне запитання «Що робить цей модуль/сервіс/клас/метод?». Відповідь має бути простою і не мати сполучників «і», «та» й інших. Проте цей критерій працює не завжди.

2) Виправлення деякого багу або додавання нових функціональних можливостей впливає на мінімальну кількість файлів/класів (в ідеальному випадку – одного). Так як відповідальність інкапсульована лише в одному файлі, завжди буде відомо де шукати та що виправляти.

3) Якщо декілька розробників працюють над різними функціональними можливостями в ПЗ, то ймовірність того, що той самий файл/клас буде змінений у декількох розробників одночасно – мінімальна.

4) При уточнюючому питанні про бізнес-логіку (наприклад, від розробника або менеджера) іде перевірка строго в одному класі/файлі й інформація отримується лише звідти. Тобто функціональні можливості, правила чи алгори-

тми компактно написані кожен на своєму місці, а не розкидані у всьому просторі кодової бази з відповідними змінними-флагами.

5) Зрозумілий неймінг (*англ. Naming*). За назвами файлів/класів/методів можна однозначно зрозуміти їх відповідальність. Наприклад, AllBookkeepersManagerService скоріше за все являється «божественним» класом, який уміє абсолютно все, а MessageSender – має чітко зрозумілу відповідальність.

Принцип єдиної відповідальності необхідно завжди розглядати під час написання коду. Архітектура класів та модулів може сильно залежати від цього. Правильне його використання може призвести до архітектури з низьким зчепленням та меншою кількістю залежностей.

Але, як і будь-яка монета, даний принцип має дві сторони. Буває дуже заманливо почати використовувати принцип єдиної відповідальності з самого початку при проектуванні архітектури програми. Також дуже привабливо спробувати відразу з'ясувати всіх акторів. Але, з погляду проекту архітектури, спробувати розглянути всі частини ПЗ з самого початку – небезпечно. Надмірна увага до принципу єдиної відповідальності може легко призвести до передчасної оптимізації і замість гарного проекту можна отримати архітектуру, в якій обов'язки класів та модулів дуже важко зрозуміти.

4.2 Принцип відкритості/закритості

Принцип відкритості/закритості (*англ. Open/Closed principle, OCP*) говорить про наступне: «програмні сутності повинні бути відкриті для розширення, але закриті для модифікації» [3, 18].

Тут йдеться про програмні сутності починаючи з класу і більше. Вони повинні бути закриті для модифікації, щоб дотримувався принцип інкапсуляції. Якщо один із тих, хто використовує програмну сутність, безконтрольно змінює її, то для інших робота з цією програмною сутністю стає непередбачуваною. З іншого боку, відкритість для розширення повертає те, що суворе дотримання інкапсуляції може забрати у розробника – гнучкість.

Якщо одна зміна в програмі спричиняє каскад змін у залежних модулях, то в програмі виявляються небажані ознаки «поганої» або «брудної» архітектури.

Програма стає крихкою, негнучкою, непередбачуваною та неперевикористовуваною. Принцип відкритості/закритості вирішує ці проблеми дуже прямолінійним шляхом. Він каже, що потрібно проектувати модулі, які ніколи не змінюються. Коли вимоги змінюються, потрібно розширювати поведінку таких модулів шляхом додавання нового коду, а не зміною старого коду, який вже працює.

Закрити програмну сутність від модифікації можна, наприклад, за допомогою інкапсуляції. Відкрити програмну сутність для розширення допоможуть поліморфізм та успадкування. Розглянемо як це робиться:

1) Необхідно чітко визначити точки, в яких буде доступним розширення програмної сутності. Оскільки не можна давати можливість змінювати все поспіль, тож необхідно визначити, що саме можна та/або потрібно, а що ні.

2) Визначитися з механізмами, за допомогою яких буде розширюватися програмна сутність. На рівні класів це може бути суперклас, доступний для успадкування, або поле/параметр методу/конструктора, куди буде покладено об’єкт з логікою, що розширює необхідний клас. Важливо відмітити, що у даному випадку часто виникає завдання вибору між успадкуванням та композицією. Загальна рекомендація полягає в тому, що в будь-якій незрозумілій ситуації необхідно робити вибір на користь композиції.

3) Якщо обрано успадкування, то треба пам’ятати про проблему крихких суперкласів. Предок має бути закритий від спадкоємців настільки, наскільки це можливо. Це означає, що навіть поля суперкласу мають бути закритими від підкласів (модифікатор доступу *private*), проте доступ до них все ще може бути отриманим за допомогою «гетерів» (методів, функціонал яких полягає в поверненні значень полів класу/екземпляру) та «сетерів» (методів, функціонал яких полягає у присвоєнні нових значень полям класу/екземпляру), які визначають чітку поведінку та правила доступу.

Причина такого підходу стає зрозумілою: коли поля екземплярів класу змінюються, кожен метод, який залежить від них, має змінитися. Тобто метод не закрито від змін цих полів.

При ООП очікується, що методи класу не закриті від змін полів цього класу. Однак, очікується, що будь-якому іншому класу, включаючи підкласи, доступ до зміни цих полів є закритим. Це якраз і розуміється під інкапсуляцією.

4) Якщо дається можливість для розширення, виділивши якусь частину програмної логіки в окремий об'єкт, треба класу, що розширюється, розкривати мінімум інформації про те, хто його розширює.

5) Необхідно визначитися за замовчуванням, яка поведінка має бути у точці розширення, якщо не вказано, за рахунок чого відбувається розширення. Можливо, ніякої. Можливо, варто змусити користувача програмної сутності задати її поведінку у цій точці. Деякі мови (Java у тому числі) мають механізм абстрактних класів, які спочатку неповні та вимагають для використання створення спадкоємців для заповнення точок, які відсутні.

6) Аргумент проти глобальних змінних той самий, що й аргумент проти публічних (модифікатор доступу *public*) полів екземплярів класу. Жоден модуль, який залежить від глобальної змінної, не може бути закритий від модуля, який може змінювати її. Будь-який модуль, який використовує цю змінну не передбаченим іншими модулями способом, зламає ці модулі. Це дуже ризиковано – мати безліч модулів, що залежать від примх якогось одного шкідливого модуля.

З іншого боку, у тих випадках, коли глобальні змінні мають невелику кількість залежних від них модулів або не можуть бути використані неправильно, вони не завдають шкоди. Проектувальник повинен оцінити, скільки закритості приноситися в жертву і визначити, чи варта того зручність, яку надає глобальна змінна.

Отже, модулі, що відповідають принципу відкритості-закритості, мають дві основні ознаки:

1) Відкриті для розширення. Тобто до модулю можна додати нову поведінку відповідно до зміни вимог до ПЗ або для задоволення потреб нових модулів.

2) Закриті для змін – вихідний код таких модулів недоторканний. Ніхто не має права вносити до нього зміни.

Здається, що дві ці ознаки одна з одною ніяк не в'яжуться, оскільки стандартний спосіб розширити поведінку модуля – це внести до нього зміни. Модуль, який не може бути змінений, зазвичай розглядається як модуль з фіксованою поведінкою.

Ці дві протилежні умови можна реалізувати, наприклад, використовуючи принципи ООП. Створюються фіксовані абстракції, які можуть представляти необмежений набір можливих поведінок.

У даному випадку абстракціями є абстрактні суперкласи, а необмежений набір можливих варіантів поведінки представлений усіма можливими підкласами. Модуль може маніпулювати абстракцією. Такий модуль закрито для змін, оскільки він залежить від фіксованої абстракції. Також поведінка модуля може бути розширена створенням нових спадкоємців абстракції.

Звичайно, завжди є зміни, які неможливо внести, не змінивши код якогось модулю – жодна система не може бути закрита на 100%. Тому під час проектування важливим є стратегічний підхід. Необхідно визначити, від яких змін та які саме модулі необхідно закрити. Це рішення слід приймати спираючись на досвід, а також на знання предметної області та користувачів ПС.

Порушення принципу відкритості/закритості призводить до ситуацій, коли зміна в одному модулі змушує змінювати інші, пов'язані з ним. Це, у свою чергу, порушує принцип єдиної відповідальності, тому що весь код, який змінюється з однієї причини, повинен бути зібраний в одному модулі (різні модулі – різні причини для зміни).

Розглянемо приклад. На рис. 4.4 сутність Client пов'язана безпосередньо з сутністю Server. Якщо раптом знадобиться, щоб Client міг працювати з різними сутностями Server, то доведеться змінити його код.

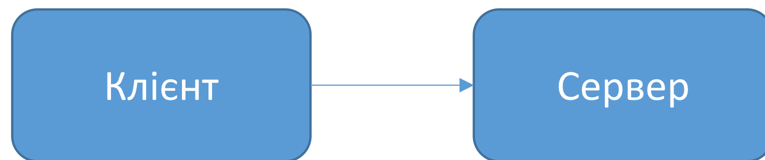

Рис. 4.4. Приклад порушення принципу відкритості/закритості

Щоб вирішити цю проблему, необхідно зв’язувати сутності не безпосередньо, а через абстракції. Якщо всі сутності Server реалізують інтерфейс AbstractServer (рис. 4.5), то не доведеться змінювати код сутності Client для заміни однієї сутності Server на іншу.

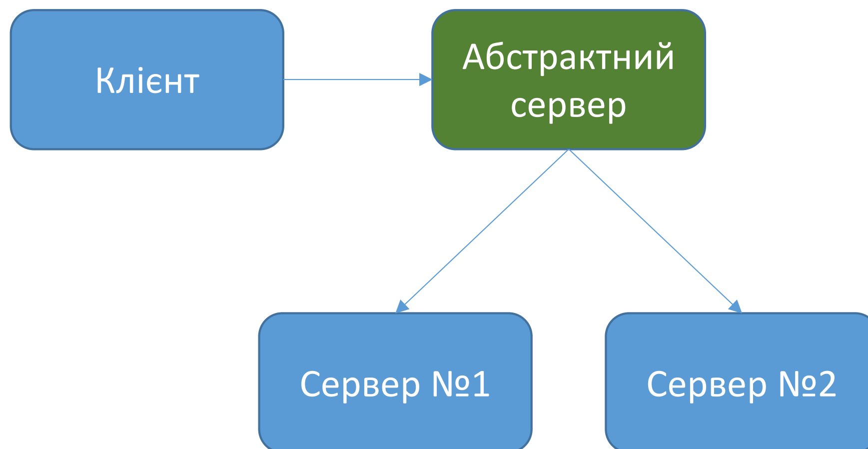

Рис. 4.5. Приклад правильного використання принципу відкритості/закритості

Отже, принцип відкритості/закритості:

- змушує проектувати модулі так, щоб вони робили лише одну справу;
- спонукає зв’язувати сутності через абстракції (а не через реалізацію) там, де можуть змінитися бізнес-вимоги;
- звертає увагу проектувальників на місця стику та взаємодію між сутностями;
- дозволяє скоротити кількість коду, який необхідно змінювати при внесенні змін до бізнес-вимог;
- робить внесення змін безпечним та відносно дешевим.

У кожного принципу є обмеження та сфера застосування. Для принципу відкритості/закритості характерні такі обмеження та підводні камені:

1) **Система не може бути закрита на 100%.** Завжди є зміни, які неможливо внести, не змінивши код модуля. Це призводить до того, що при проектуванні важливо використовувати стратегічний підхід, а саме: необхідно визначити, від яких змін та які саме модулі необхідно закрити.

Слід зважати на те, що люди дуже погано вміють прогнозувати зміни. Навіть маючи достатньо знань про систему та досвід, проектувальники не можуть бути впевнені, що передбачили всі можливі варіанти розвитку системи.

Принцип відкритості/закритості пропонує підхід Just-in-time design, при якому нові сутності додаються до системи в міру потреби, але не раніше. Це чимось схоже на відмову від ранньої оптимізації та раннього додавання абстракцій.

Ціль підходу в тому, щоб не створювати абстракції на порожньому місці. Один із критеріїв гарного дизайну – простота, тому використовувати принцип відкритості/закритості завжди слід з огляду на те, наскільки система вийде простою в результаті.

2) **Велика кількість сутностей.** Додавання функціональності менш ризиковане, ніж зміна існуючої, але натомість є ризик збільшити кількість сутностей. Безконтрольне і бездумне використання принципу відкритості/закритості може призводити до ситуацій, коли інтерфейсів стане занадто багато, а функціональність стане роздробленою.

Гарний дизайн ПС – це, насамперед, простий дизайн. Чим менше сутностей створюється для вирішення проблеми, тим вище ймовірність, що дизайн хороший. Саме тому використовувати принцип відкритості/закритості необхідно з обережністю.

3) **Не потрібен (найчастіше) для маленьких додатків.** Даний принцип націлений на швидке та дешеве додавання функціональності та масштабування системи. Він окупається, якщо система справді велика, і перевіряти, як вплинула зміна коду, дуже дорого чи довго (або неможливо).

Якщо ж додаток маленький, то принцип відкритості/закритості може перетворитися на принцип заради принципу, – коли розробники писатимуть шаблонний код для створення нових сутностей без видимої та відчутної користі.

Принцип відкритості/закритості слід застосовувати, якщо користь від нього значно вища, ніж витрати від його використання.

4.3 Принцип заміни (підстановки) Барбари Лісков

Принцип заміни (підстановки) Барбари Лісков (*англ. Liskov Substitution Principle, LSP*) каже що: «функції, які використовують базовий тип, повинні мати можливість використовувати підтипи базового типу, не знаючи про це» або «якщо базовий клас проходить певний юніт-тест, його також повинні проходити всі підкласи базового класу» [3, 24].

Якщо перефразувати простими словами, то підкласи не повинні суперечити суперкласу. Наприклад, вони не можуть надавати інтерфейс вужче базового. Поведінка спадкоємців має бути очікуваною для методів, які використовують суперклас.

Дещо зручніше думати про принцип підстановки Барбари Лісков у термінах «абстракція – реалізація». Абстрактний клас чи інтерфейс грають роль базового типу, але разом з цим – роль контракту поведінки.

Контракт – гарантія того, що екземпляри будь-якого конкретного класу матимуть відповідні поля та/або методи.

Це означає, що модуль, який використовує цей абстрактний клас або інтерфейс, зможе працювати з будь-якою його реалізацією. Наприклад, методу *showTitle()* буде неважливо, екземпляр якого конкретного класу передається:

```
public class Book {
   public String getTitle() {
      return "There is no title";
   }
}

class Alphabet extends Book{
   @Override
   public String getTitle() {
```

```
        return "Alphabet";
    }
}

class ABCbook extends Book{
    @Override
    public String getTitle() {
        return "ABC book";
    }
}

public class Main {
    public static void main(String[] args) {
        Book abc = new ABCbook();
        Book alphabet = new Alphabet();
        Book book = new Book();
        showTitle(book);
        showTitle(abc);
        showTitle(alphabet);
    }
    public static void showTitle(Book book) {
        System.out.println(book.getTitle());
    }
}
```

Виконувати принцип підстановки Барбари Лісков простіше, якщо не будувати великих і глибоких ієрархій. Зазвичай, для зв'язку модулів достатньо інтерфейсу та його реалізацій.

Довгі ланцюжки ієрархій типів – крихкі. Замість ієрархій типів найчастіше краще використовувати композицію, щоб збирати сутності із необхідної функціональності.

Наприклад, успадкування передбачає проектування від загального до приватного у вигляді ієрархії:

Тварини → Ссавці → Людина.

Але такі ієрархії не завжди правильно відображають зв'язки сутностей у проектованій ПС. Іноді ієрархій може бути одночасно декілька (множинне успадкування), тоді успадкування зайде в глухий кут – незрозуміло за якою ієрархією успадковуватися і як.

Композиція ж має на увазі проектування від приватного до загального у вигляді сукупності декількох відокремлених наборів функціональних можливостей:

Людина = Скелет + М'язи + Нервова система + Імунна система + ...

Таким чином, немає обмежень у рамках однієї ієрархії, а значить необов'язково підводити кожен аспект бізнес-логіки під цю ієрархію.

Більше того, часто у відносинах сутностей, які моделюються, немає жодної ієрархії. У такому разі успадкування сутностей навпаки лише зашкодить.

Проте, зловживання композицією сутностей також має негативні наслідки. Воно дуже часто призводить до неконтрольованого зростання обсягів коду, тож для кожного окремого випадку необхідно обирати, який варіант краще: спадковість чи композиція.

Отже, принцип підстановки Барбари Лісков:

- допомагає проектувати ПС, спираючись на поведінку окремих її модулів;
- вводить обмеження та правила успадкування об'єктів, щоб їх нащадки не суперечили базовій поведінці;
- робить поведінку модулів ПС послідовною та передбачуваною;
- допомагає уникати дублювання за рахунок виділення загальної для декількох модулів функціональності у спільний (загальний) інтерфейс;

- дозволяє виявляти при проектуванні проблемні абстракції та приховані зв'язки між різними сутностями.

Для даного принципу характерні наступні обмеження:

1) **Зміна ієрархії коштує дорого.** Даний принцип допомагає проектувати відносини між сутностями з огляду на можливі зміни у вимогах. Але вірогідність якісного проектування ПС з першого разу – низька.

Прагнення до успадкування може призвести до зміни ієрархії об'єктів у системі. Великі зміни в структурах – дорогі, як у часі, так і в коштах.

Також зміна структури стосується великої кількості компонентів. Якщо код погано покритий тестами, переписування може призвести до непрацюючої програми.

2) **Використання контрактів – ресурсозатратно**. Використання контрактів дозволяє уникнути посилення передумов та послаблення постумов. Однак контракт збільшує складність проекту, а також збільшує час розроблення.

Покриття методів контрактними специфікаціями є дуже довгим за часом. Контракти часто багатослівні та дублюють тести, а у деяких випадках навіть знижують продуктивність коду.

3) **Виділення суперкласу має межу вкладеності.** Неможливо виносити функціональність все вище та вище без наслідків – надмірне винесення функціональності нагору може призвести до появи «божественного» об'єкта (може виконувати абсолютно все, тобто порушує принцип єдиної відповідальності).

Хоча виділення суперкласу і дозволяє уникнути дублювання, але не гарантує правильність ієрархії об'єктів – може статися, що нові бізнес-вимоги додадуть сутність, яка не вписуватиметься у вже змінену ієрархію.

4.4 Принцип поділу інтерфейсів

Принцип поділу інтерфейсів (*англ. Interface Segregation Principle, ISP*) концептуально схожий на принцип єдиної відповідальності і вирішує схожі проблеми, які відносяться до більш високого рівня абстракції.

Проблема банана та джунглів – одна з тих, за які не люблять ООП. Вона полягає в тому, що необхідно використовувати об'єкт «Банан», але (у кінцево-

му підсумку) для цього треба імпортувати об'єкт «Горила», яка має тримати «Банан», а потім усі сутності, від яких залежить «Горила» такі, як: інші стани об'єктів та середовище, тобто «Джунглі».

Якщо сформулювати простими словами, то суть проблеми полягає в наступному: «при успадкуванні підклас отримує від суперкласу разом з потрібною функціональністю додатково купу не використовуваної і не потрібної».

Однак, проблема виникає не так через ООП за замовчуванням, як через неправильну модель системи.

Формулювання принципу поділу інтерфейсів звучить наступним чином: «сутності не повинні залежати від інтерфейсів, які вони не використовують» [3, 24].

Коли принцип порушується, модулі є залежними від всіх змін в інтерфейсах, від яких вони залежать. Це призводить до високого зчеплення модулів один з одним.

Даний принцип допомагає проектувати інтерфейси так, щоб зміни стосувалися тільки тих модулів, на функціональність яких вони справді впливають. Найчастіше це призводить до декомпозиції (розділення) інтерфейсів на значно менші.

Принцип поділу інтерфейсів містить правила та обмеження, які допомагають вирішувати дану проблему. Якщо розглядати у загальному, то дані правила та обмеження є дуже схожими на ті, які використовуються у принципі єдиної відповідальності, тож детально розглядатися в межах цього підручника не будуть. Єдиною відмінністю є те, що для даного принципу вони стосуються інтерфейсів, а не модулів.

Якщо підводити підсумки, то можна сказати, що принцип поділу інтерфейсів:

- допомагає боротися з успадкуванням чи реалізацією непотрібної функціональності;
- дає можливість спроектувати модулі так, щоб їх зачіпали зміни лише тих інтерфейсів, які вони справді реалізують;
- знижує зчеплення модулів;

- знищує спадкування задля успадкування;
- заохочує використання композиції;
- дозволяє виявляти вищі абстракції та знаходити неочевидні зв'язки між сутностями.

Як і у випадку з принципом єдиної відповідальності, проблеми з поділом інтерфейсів виникають із труднощів проектування та прогнозування:

1) **Сліпе слідування цьому принципу є небезпечним.** Сліпе слідування загрожує подрібненню взагалі всіх інтерфейсів на атомарні частинки з одним методом чи полем. Але суть принципу не в тому, щоб роздробити усі інтерфейси. Основне завдання полягає в тому, щоб виділити мінімально необхідну кількість методів для того, щоб модулі, які його реалізують, не залежали від непотрібної функціональності.

2) **«Брудний» інтерфейс.** У багатьох предметних областях сутності можуть мати дуже велику кількість властивостей і методів, через що розділяти інтерфейси може бути важко. Розробники можуть зіткнутися з ситуацією, коли не буде очевидно чи інтерфейс все ще підпорядковується даному принципу, а в якій – вже ні.

3) **Конфлікти ролей та ієрархій.** Інтерфейси можна умовно розділити на ролі: тип, поведінка, очікування (наприклад, у контракті) та інші.

Ієрархія ролей може конфліктувати з ієрархією сутностей та модулів, які реалізують інтерфейси. Це робить структуру ПС складною для розуміння і може вводити в оману під час читання коду.

4.5 Принцип інверсії залежностей

ПС складаються з модулів, які можна умовно поділити на низько-рівневі та високо-рівневі.

Низько-рівневі містять утилітарну функціональність, наприклад: рендеринг елементів на сторінці, звернення до БД, запити на сервер, тощо.

Високо-рівневі містять складну, більш абстрактну бізнес-логіку. Вони досить абстрактні, щоб їх можна було використовувати повторно у інших проектах: валідація форм, авторизація користувачів, надсилання повідомлень, тощо.

У стійких ПС високорівневі модулі зазвичай не вимагають оновлення при зміні низькорівневих. Досягти такої стійкості допомагає принцип інверсії залежностей (*англ. Dependency Inversion Principle, DIP*) передбачає, що [3, 24]:

- високорівневі модулі не повинні залежати від низькорівневих;
- обидва типи повинні залежати від абстракцій;
- абстракції не повинні залежати від деталей;
- деталі повинні залежати від абстракцій.

Таким чином, даний принцип також допомагає знизити зчеплення модулів.

Коли модулі жорстко зчеплені, вони дуже багато знають один про одного і не функціонують окремо. У такій ситуації зміни в одному з них вимагатимуть змін в інших, що порушує принцип відкритості/закритості.

Зчеплення не варто плутати зі зв’язністю.

Зчеплення (*англ. coupling*) – ступінь взаємозалежності різних модулів. Чим вище зчеплення, тим крихкішою виходить система, і тим складніше вносити зміни.

Зв’язність (*англ. cohesion*) – ступінь зв’язаності задач певного модуля одна з одною. Чим вище зв’язність, тим суворіше модулі відповідають принципу єдиної відповідальності, тим вище сфокусований модуль на конкретній задачі.

Відповідно до даного принципу, модулі повинні залежати від інших модулів не безпосередньо, а через абстракції.

У прикладі з регулятором температури на рис. 4.6 структура ПС порушує даний принцип. Модулі залежать безпосередньо від інших модулів, а це збільшує зчеплення.

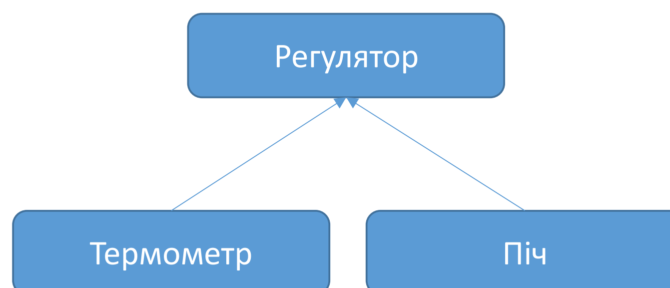

Рис. 4.6. Приклад порушення принципу інверсії залежностей

Виправлений варіант (рис. 4.7) вводить прошарок між сутностями у вигляді абстракцій – інтерфейсів.

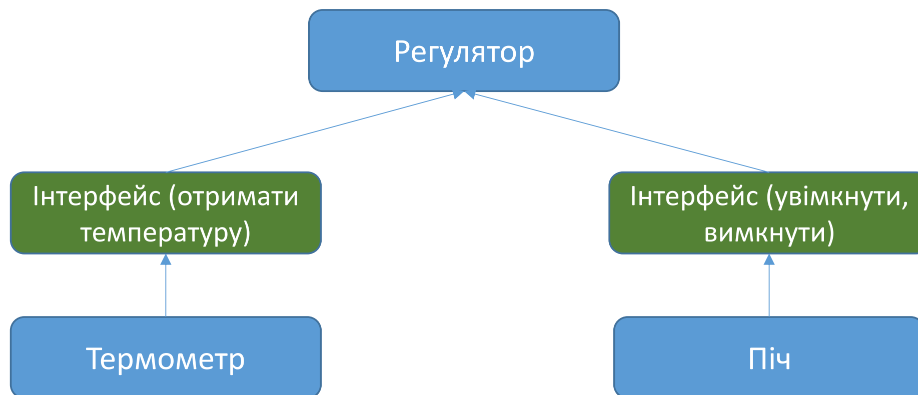

Рис. 4.7. Приклад правильного використання принципу інверсії залежностей

Модулям тепер не обов’язково працювати з конкретними модулями, вони можуть працювати з будь-якою сутністю, яка реалізує зазначений інтерфейс. Так знижується зчеплення.

При тестуванні модуля, який залежить від інших модулів, потрібно створювати екземпляр кожної залежності, або створити «заглушки». Якщо ж модулі залежать від інтерфейсів, достатньо створити «заглушку», що реалізує цей інтерфейс. Таким чином, даний принцип полегшує тестування системи.

Отже, принцип інверсії залежностей:

- вводить правила та обмеження на залежності одних модулів від інших;
- знижує зчеплення модулів;
- робить тестування модулів простішим;
- дозволяє проектувати ПС так, щоб модулі можна було без проблем замінити на інші.

Принцип інверсії залежностей може призводити до ситуацій, коли потік управління програми заданий неявно. Проте явне – краще неявного. Розглянемо обмеження даного принципу:

1) **Безграмотне використання**. При безграмотному та безконтрольному використанні принципу логіка взаємодії компонентів може виявитися розмаза-

ною по різних частинах ПЗ. Це знижує придатність до покриття тестами та збільшує вартість внесення змін.

Крім того, не всі залежності варті того, щоб їх інвертувати. Відрізнити такі залежності від інших буває важко через, наприклад, специфіку предметної області.

2) **Проблеми з ін'єкцією залежностей** (*англ. Dependency Injection, DI*). При ін'єкції через конструктор багатьох залежностей сильно збільшується сам конструктор, який в ідеальному випадку має залишатися порожнім (або дуже простим/коротким). При безграмотному використанні ін'єкції залежностей можуть виникнути циклічні залежності.

В обох випадках варто подивитися у бік принципу єдиної відповідальності. Швидше за все, клас робить занадто багато, і його треба розбивати на дрібніші.

4.6 Шаблони проектування

У програмній інженерії **шаблон проектування** – це загальне повторюване рішення поширеної проблеми в розробці ПЗ [6].

Тобто шаблон проектування є описом або шаблоном для вирішення проблеми, який можна використовувати в багатьох різних ситуаціях, але це не готовий проект, який можна перетворити безпосередньо в код.

Шаблони проектування можуть прискорити процес розроблення, надаючи апробовані, перевірені часом парадигми. Ефективний дизайн ПЗ вимагає розгляду проблем, які можуть стати видимими лише пізніше в процесі впровадження. Повторне використання шаблонів проектування допомагає запобігти виникненню серйозних проблем, а також покращує читабельність коду для програмістів і архітекторів, знайомих із шаблонами.

Часто люди розуміють лише те, як застосувати певні методи розроблення ПЗ до конкретних проблем. Ці методи важко застосувати до більш широкого кола проблем. Шаблони ж надають загальні рішення, задокументовані у форматі, який не вимагає конкретики, пов'язаної з певною проблемою.

Крім того, шаблони дозволяють розробникам спілкуватися, використовуючи добре відомі, добре зрозумілі імена для взаємодії компонентів ПЗ. Звичайні шаблони проектування можна вдосконалювати з часом, роблячи їх більш надійними, ніж спеціальні проекти.

Отже, використання шаблонів проектування має наступні переваги [6, 11]:

1) Спрощують проектування та підтримку ПС.

2) Код стає більш передбачуваним та інтуїтивно зрозумілим.

3) У коді менше помилок, тому що використовуються типові уніфіковані рішення, в яких давно знайдено всі приховані проблеми.

4.6.1 Шаблони GRASP

У даному розділі багато уваги приділяється принципам SOLID, тому що це основа ООП та дизайну. Для розробників об'єктно-орієнтованих мов знання принципів SOLID є вимогою для написання коду, який характеризується хорошою якістю.

З іншого боку, існує інший, менш відомий набір правил щодо ООП. Вони називаються GRASP (*англ. General Responsibility Assignment Software Patterns*) – шаблони програмного забезпечення для розподілу загальної відповідальності. Дані шаблони (рис. 4.8) користуються меншою популярністю ніж SOLID, але також варті уваги.

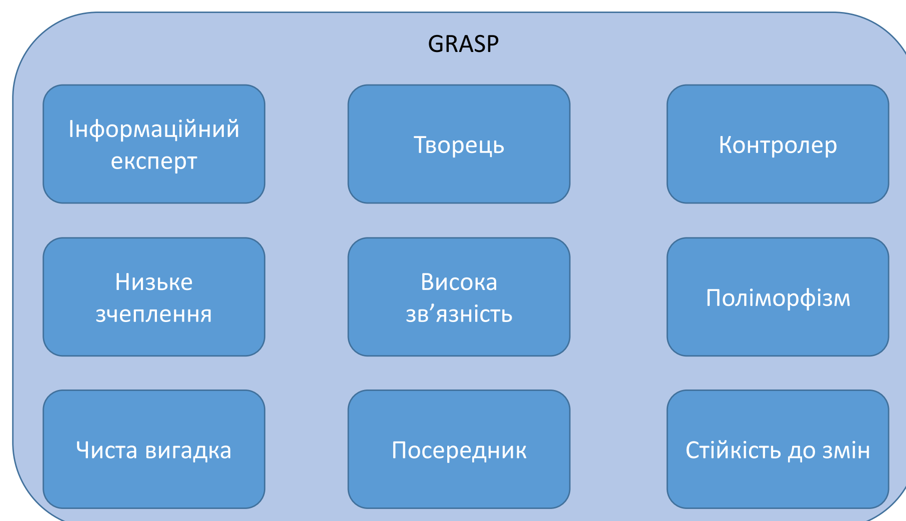

Рис. 4.8. Шаблони GRASP

Існує дев'ять шаблонів GRAPS, які були спочатку описані у книзі Крейга Лармана «Застосування UML і шаблонів проектування» [26]. Шаблони GRAPS не мають вираженої структури, чіткої області застосування та конкретної проблеми, що вирішується з їх допомогою, а лише являють собою узагальнені підходи/рекомендації/принципи, що використовуються при детальному проектуванні ПС.

GRASP складається з п'яти основних та чотирьох додаткових шаблонів [26]:

- Основні:
  - інформаційний експерт (*англ. Information Expert*);
  - творець (*англ. Creator*);
  - контролер (*англ. Controller*);
  - низьке зчеплення (*англ. Low Coupling*);
  - висока зв'язаність (*англ. High Cohesion*).
- Додаткові:
  - чиста вигадка (*англ. Pure Fabrication*);
  - посередник (*англ. Indirection*);
  - поліморфізм (*англ. Polymorphism*);
  - стійкість до змін (*англ. Protected Variations*).

Інформаційний експерт – це принцип, який використовується для визначення місця делегування обов'язків і звучить наступним чином [26]: «відповідальність має бути призначена тому, хто має максимум необхідної інформації для виконання відповідних функцій, тобто інформаційному експерту».

Використовуючи принцип інформаційного експерта, загальний підхід до розподілу обов'язків полягає в тому, щоб подивитися на певну відповідальність, визначити інформацію, необхідну для її виконання, а потім визначити, де ця інформація зберігається.

Творець. Створення об'єктів є одним із найпоширеніших видів діяльності в ООП. Фундаментальною властивістю зв'язку між об'єктами певних класів є розуміння того, який клас відповідає за створення об'єктів.

У загальному, клас B має бути відповідальним за створення екземплярів класу A, якщо одне (або бажано більше) з наступних тверджень виконується [26]:

1. Екземпляри B містять або комбіновано об'єднують екземпляри A.

2. Екземпляри B записують екземпляри A.

3. Екземпляри B тісно використовують екземпляри A.

4. Екземпляри B мають інформацію про ініціалізацію для екземплярів A та передають її при створенні.

Отже, суть відповідальності творця – створення інших об'єктів.

Контролер. Даний шаблон призначає відповідальність за роботу з системними подіями класу, який не містить елементів інтерфейсу користувача та представляє загальну систему або варіант використання (Use Case сценарій).

Об'єкт Controller – це об'єкт, який відповідає за отримання або обробку системної події і не пов'язаний зінтерфейсом користувача. Контролер варіантів використання має застосовуватися для роботи з усіма системними подіями Use Case сценаріїв. Наприклад, для варіантів використання «Створити користувача» та «Видалити користувача» можна мати один UserController замість двох окремих.

Він визначається як перший об'єкт за межами рівня графічного інтерфейсу користувача, який приймає та координує (керує) системні операції. Контролер повинен делегувати іншим об'єктам роботу, яку необхідно виконати; він координує або контролює діяльність. Сам по собі він не повинен виконувати багато роботи. Контролер GRASP можна розглядати як частину рівня програми/служби в об'єктно-орієнтованій ПС із загальними шарами за умови, що програма чітко розрізняє рівень додатків/сервісів і рівень домену.

Низьке зчеплення, висока зв'язність. У минулих розділах неоднократно було озвучено рекомендацію, в якій йдеться про те, що необхідно прагнути до досягнення низького зчеплення (low coupling) та високої зв'язаності (high cohesion) при роботі над кодом. Розглянемо більш детально, що насправді означає ця рекомендація, а також проведемо більш чіткий кордон між цими двома ідеями.

У той час як зчеплення є досить інтуїтивним поняттям (майже ніхто не має труднощів з ним), то зв'язаність важче зрозуміти. Понад те, різниця між ними часто здається незрозумілою. Це не дивно: ідеї, що лежать в основі цих термінів, схожі. Проте вони справді відрізняються.

Зв'язаність є показником ступеня утворення логічної єдиної атомарної одиниці (модуля) певною частиною кодової бази.

Вона також може вказати на кількість зв'язків усередині блоку коду. Якщо число є малим, то, ймовірно, межі для модуля вибрано неправильно, – код усередині цього блоку логічно не пов'язаний. Модуль не обов'язково є класом. Це може бути метод, клас, група класів, тощо.

З іншого боку, зчеплення є ступенем взаємозв'язку між окремими модулями. Іншими словами, це кількість з'єднань між двома (чи більше) модулями. Чим менше число, тим нижче зчеплення.

По своїй суті, висока зв'язність означає зберігання пов'язаних один з одним частин коду в одному місці. У той же час низьке зчеплення полягає в максимально можливому розділенні незв'язаних частин кодової бази.

Теоретично рекомендації виглядають досить просто, однак на практиці необхідно досить глибоко зануритися в предметну область розроблюваного ПЗ, щоб зрозуміти, які частини кодової бази насправді пов'язані.

Це означає, що ступінь зв'язності та зчеплення не може бути виміряна безпосередньо – вона залежить від семантики коду.

Можливо, відсутність об'єктивності в цій рекомендації є причиною того, що часто так важко її дотримуватися.

Крім коду, який одночасно має високу зв'язність та слабке зчеплення, існує принаймні ще три типи (рис. 4.9):

1) Ідеальним є код, який виконує рекомендації. У нього слабке зчеплення та висока зв'язність. Такий код зображено на рис. 4.10, де квадрати одного кольору позначають пов'язані частини кодової бази.

2) «Божественний» об'єкт є результатом запровадження високого зв'язування і високого зчеплення. Цей анти-шаблон в основному визначає ная-

вність одного фрагменту коду, який виконує всю роботу відразу (рис. 4.11). Інша назва подібного коду – велика груда бруду (*англ. Big Ball of Mud*).

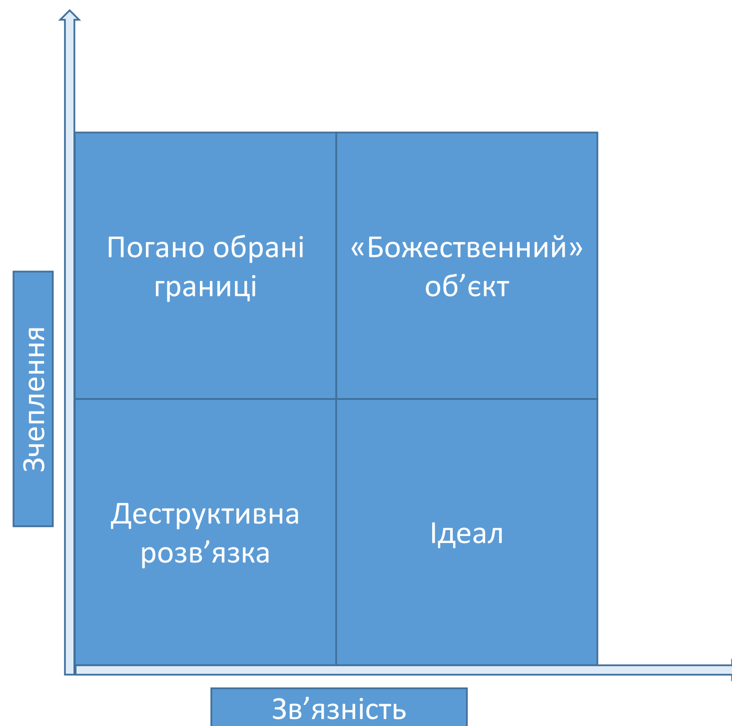

Рис. 4.9. Ступінь відношення між зчепленням та зв’язністю

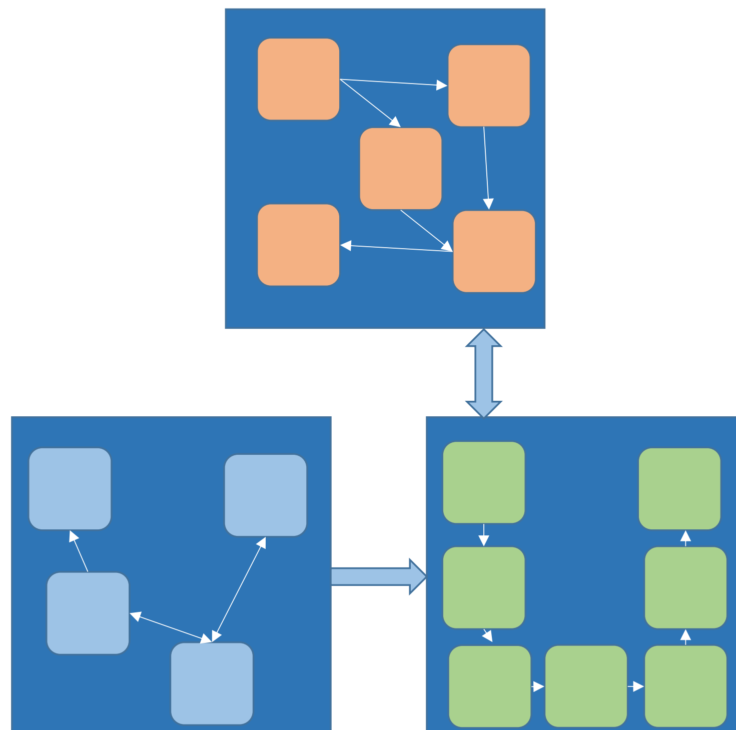

Рис. 4.10. Ідеальне відношення між зчепленням та зв’язністю

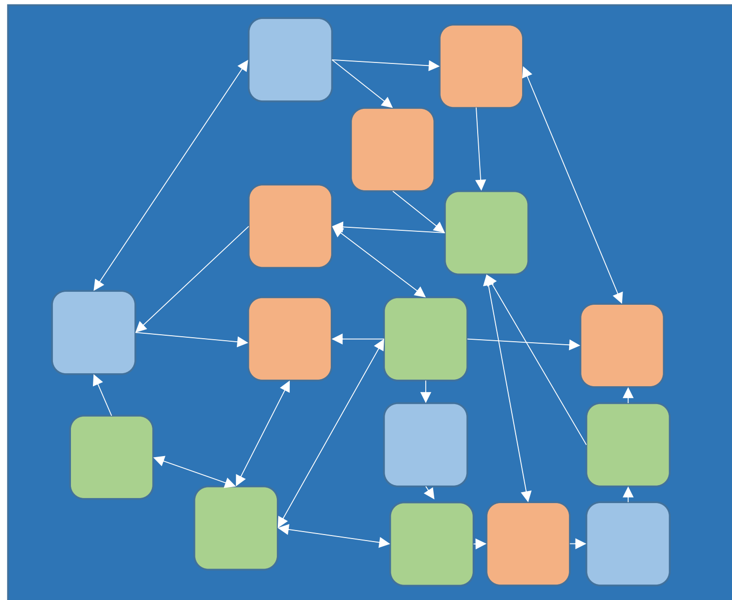

Рис. 4.11. «Божественний» об'єкт

3) Третій тип має місце, коли межі між різними класами чи модулями вибрано погано (рис. 4.12). На відміну від «божественного» об'єкту код цього типу має межі. Проблема полягає в тому, що вони обрані неправильно і часто не відображають фактичну семантику домену. Такий код часто порушує принцип єдиної відповідальності.

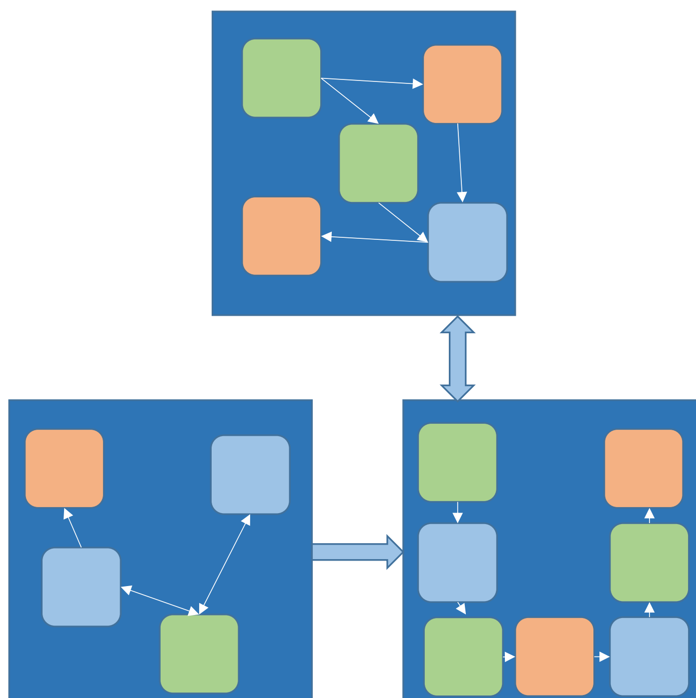

Рис. 4.12. Порушення меж модулів

4) Деструктивна розв'язка (*англ. Destructive Decoupling*) – найцікавіший випадок. Це може статися, коли програміст намагається так розв'язати кодову базу, що вона повністю втрачає фокус (рис. 4.13).

Часто, коли розробник намагається реалізувати рекомендації щодо низького зчеплення та високої зв'язності, він прикладає надто багато зусиль до реалізації першої рекомендації (низьке зчеплення) і повністю забуває про іншу. Це призводить до ситуації, коли код справді розділений, але у нього водночас немає чіткої спрямованості. Його частини настільки відокремлені одна від одної, що стає важко чи навіть неможливо зрозуміти їхнє значення. Ця ситуація якраз і називається деструктивною розв'язкою.

Деструктивна розв'язка часто йде пліч-о-пліч з підходом «інтерфейси всюди», тобто спокусою замінити кожен конкретний клас інтерфейсом, навіть якщо цей інтерфейс не є абстракцією.

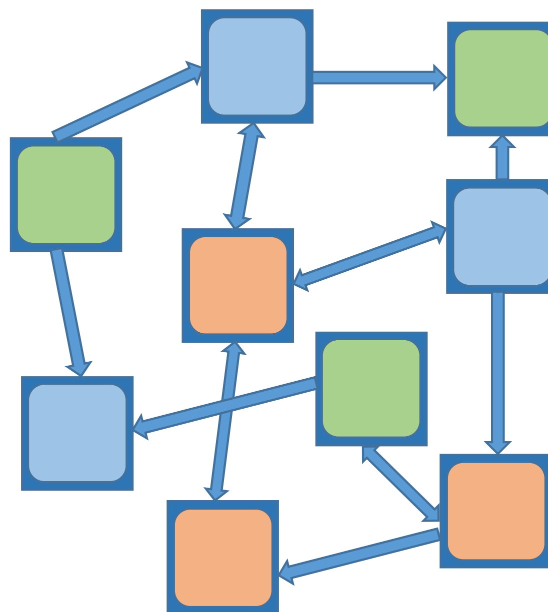

Рис. 4.13. Приклад деструктивної розв'язки

Підведемо підсумки для двох розглянутих шаблонів:

1) Зв'язність є показником утворення частиною кодової бази логічно єдиної атомарної одиниці.

2) Зчеплення є ступенем незалежності одного блоку від інших.

3) Неможливо досягти повного поділу (decoupling) без порушення цілісності (зв'язності, cohesion), і навпаки.

4) Необхідно намагатися дотримуватися принципу «висока зв'язність, низьке зчеплення» на всіх рівнях кодової бази.

5) Не треба потрапляти у пастку деструктивної розв'язки.

Поліморфізм. У цьому контексті даний шаблон вирішує проблему обробки альтернативних варіантів поведінки на основі типу. Вирішувати цю проблему варто з використанням поліморфних операцій, а не за допомогою перевірки типу та умовної логіки. Крім того, за допомогою поліморфізму легко створювати компоненти, що підключаються. У результаті отримуємо такі переваги:

- легше розширювати систему, додаючи нові функціональні можливості;
- новий функціонал можна вводити без модифікації клієнтської частини програми.

Проте, надмірне використання поліморфізму призводить до ускладнення коду і, як правило, не рекомендується.

Чиста вигадка – це клас, який спеціально створений для досягнення низького зчеплення та високої зв'язності, щоб отримати потенціал повторного використання (коли рішення, представлене шаблоном інформаційного експерта, не відповідає цим принципам). Це суто штучний об'єкт, аналогів якому немає в предметній області.

Загалом, будь-який фасад – це чиста вигадка, якщо це не архітектурний фасад у відповідному додатку.

Для людини є природним представляти в коді об'єкти, які відображають реальність предметної області. Наприклад, якщо йде робота над фінансовою програмою можна, ймовірно, очікувати, що в коді зустрінуться деякі класи, які називаються Salary або Billing.

Але іноді доводиться призначати обов'язки, які погано вписуються в усі класи предметної області. Відповідно до вищенаведеного принципу високої зв'язності, не потрібно нав'язувати відповідальність класу, який вже займається чимось іншим.

Саме тоді вступає в дію принцип чистої вигадки: створюється клас, який не відповідає об'єктам предметної області. Йому надається можливість злагоджено досягти цієї нової відповідальності.

Посередник. Шаблон вирішує проблему прямого зв'язку між компонентами ПЗ. Рішенням цієї проблеми є введення проміжного об'єкта для забезпечення зв'язку між іншими компонентами або службами, які безпосередньо не пов'язані між собою.

Прикладом цього є введення компонента контролера (Controller) для посередництва між даними (Model) та їх відображенням (View) у шаблоні MVC.

Захищеність від змін. Основна проблема, що розв'язується даним шаблоном звучить наступним чином: як спроектувати об'єкти (підсистеми та систему в цілому), щоб їх зміна не мала небажаного впливу на інші елементи? Для цього треба ідентифікувати точки можливих варіацій чи нестійкості ПС та розподілити обов'язки в такий спосіб, щоб забезпечити стійкий інтерфейс.

Даний шаблон описує ключовий принцип, на основі якого реалізуються механізми чи шаблони програмування та проектування з метою забезпечення гнучкості та захисту ПС від впливу змін зовнішніх систем.

Інкапсуляція даних, інтерфейси, поліморфізм, перенаправлення – всі ці принципи реалізуються у рамках шаблону захищеності від змін. Існує багато принципів проектування, які так чи інакше є проявом цього шаблону, наприклад:

- проектування на основі даних;
- пошук служб;
- проектування на основі інтерпретаторів;
- рефлексивне проектування чи проектування на мета-рівні;
- уніфікований доступ.

Яким би не був метод реалізації, постійним залишається основний принцип шаблону: забезпечити стійкість інтерфейсу.

У підсумку можна коротко охарактеризувати шаблони GRASP наступним чином:

1) Інформаційний експерт – інформація обробляється там, де вона міститься.

2) Творець – об’єкти створюються там, де вони потрібні.

3) Контролер – логіка багатопотокової обробки переміщається в окремий клас або компонент.

4) Низьке зчеплення та 5) Висока зв’язність – модулі проектуються з однорідною бізнес-логікою та мінімальною кількістю зв’язків між собою.

6) Поліморфізм – організовуються різні варіанти поведінки системи у вигляді поліморфних викликів, якщо це необхідно.

7) Чиста вигадка – створення класів, які не мають аналогів у предметній області, якщо необхідно дотримуватися низького зчеплення та високої зв’язності.

8) Посередник – виклик будь-якого класу відбувається через його інтерфейс.

9) Захищеність від змін – якщо застосовувати все вищесказане, то буде отримано стійкий до змін код.

4.6.2 Шаблони «Банди Чотирьох»

Як було сказано раніше, шаблонами проектування називають стандартні рішення проблем, які часто зустрічаються у галузі розроблення ПЗ. У даному випадку передбачається, що є певний набір загальних формалізованих проблем, які досить часто зустрічаються, і шаблони надають низку принципів для вирішення цих проблем.

Концепцію шаблонів вперше описав Крістофер Александер у книзі «Мова шаблонів. Міста. Будинки. Будівництво». Дана ідея здалася привабливою авторам Еріху Гамму, Річарду Хелму, Ральфу Джонсону та Джону Влісідесу. У 1995 році вони написали книгу «Design Patterns: Elements of Reusable Object-Oriented Software» [6], в якій застосували концепцію типових шаблонів у програмуванні. До книги увійшли двадцять три шаблони, які вирішують різні проблеми ООП. Пізніше цих авторів стали називати «Банда Чотирьох», а запропоновані ними шаблони – шаблонами «Банди Чотирьох».

Ці шаблони діляться на три великі категорії [6]:

1) **Твірні шаблони** – абстрагують процес створення класів і об'єктів. Серед них виділяються наступні [6]:

- абстрактна фабрика (*англ. Abstract Factory*) дає інтерфейс для створення сімейства взаємопов'язаних або споріднених об'єктів (*англ. dependent or related objects*), не визначаючи їх конкретних класів;
- будівельник (*англ. Builder*) відокремлює конструювання складного об'єкта від його уявлення, так що в результаті того самого процесу конструювання можуть виходити різні уявлення;
- фабричний метод (*англ. Factory Method*) визначає інтерфейс для створення об'єкта, але залишає підклас для прийняття рішення про те, екземпляр якого класу створювати (дозволяє класу делегувати створення об'єктів);
- прототип (*англ. Prototype*) дозволяє створювати нові об'єкти шляхом клонування існуючих;
- одинак (*англ. Singleton*) гарантує, що клас має лише один екземпляр, і надає глобальну точку доступу до нього.

2) **Структурні шаблони** – розглядають те, як класи та об'єкти утворюють більші структури (більш складні за характером класи та об'єкти). До таких шаблонів належать [6]:

- адаптер (*англ. Adapter*) перетворює інтерфейс одного класу в інтерфейс іншого, на який очікують клієнти, тобто адаптер робить можливою спільну роботу класів з несумісними інтерфейсами;
- міст (*англ. Bridge*) – структурний шаблон проектування, який дозволяє відокремити абстракцію від реалізації таким чином, щоб і абстракцію, і реалізацію можна було змінювати незалежно одна від одної.
- компонувальник (*англ.Composite*) компонує об'єкти в деревоподібні структури для представлення агрегації (ієрархій «ціле/частина»);

- декоратор (*англ. Decorator*) динамічно додає об'єкту нові обов'язки, є гнучкою альтернативою створенню підкласів з метою розширення функціональності;
- фасад (*англ. Facade*) дає уніфікований інтерфейс замість набору інтерфейсів деякої підсистеми; визначає інтерфейс вищого рівня, який полегшує використання підсистеми; об'єднує групу об'єктів у рамках одного спеціалізованого інтерфейсу та переадресує виклики його методів до цих об'єктів;
- пристосуванець (*англ. Flyweight*) дозволяє використовувати об'єкти, що розділяються, відразу в декількох контекстах; використовується переважно для оптимізації роботи з пам'яттю;
- заступник (*англ. Proxy*) є сурогатом іншого об'єкта та контролює доступ до нього.

3) **Поведінкові шаблони** – визначають алгоритми та взаємодію між класами та об'єктами, тобто їхню поведінку. Серед таких шаблонів можна виділити наступні [6]:

- ланцюжок обов'язків (*англ. Chain of responsibility*) дозволяє уникнути прив'язки відправника запиту до одержувача, даючи шанс обробити запит декільком об'єктам; зв'язує об'єкти-одержувачі в ланцюжок і передає запит уздовж цього ланцюжка, доки його не оброблять;
- команда (*англ. Command*) інкапсулює запит як об'єкт, дозволяючи цим задавати параметри клієнтів для обробки відповідних запитів, ставити запити в чергу або протоколювати їх, а також підтримувати скасування операцій;
- інтерпретатор (*англ. Interpreter*) визначає: подання граматики для заданої мови та інтерпретатор речень цієї мови; як правило, даний шаблон проектування застосовується для операцій, які часто повторюються;

- ітератор (*англ. Iterator*) представляє доступ до всіх елементів зібраного об'єкта, не розкриваючи його внутрішнього уявлення;
- посередник (*англ. Mediator*) представляє такий шаблон проектування, який забезпечує взаємодію багатьох об'єктів без необхідності посилатися один на одного, тим самим досягається слабке зчеплення взаємодіючих об'єктів;
- зберігач (*англ. Memento*) дозволяє виносити внутрішній стан об'єкта за його межі для подальшого можливого відновлення об'єкта без порушення принципу інкапсуляції;
- спостерігач (*англ. Observer*) визначає залежність типу «один до багатьох» між об'єктами таким чином, що при зміні стану одного об'єкта всі, що залежать від нього, сповіщаються про це і автоматично оновлюються;
- стан (*англ. State*) передбачає виділення базового класу або інтерфейсу для всіх допустимих операцій та спадкоємця для кожного можливого стану;
- стратегія (*англ. Strategy*) інкапсулює певну поведінку з можливістю її заміни;
- шаблонний метод (*англ. Template method*) визначає основу алгоритму та дозволяє підкласам перевизначати деякі кроки алгоритму, не змінюючи його загальної структури;
- відвідувач (*англ. Visitor*) описує операцію, що виконується з кожним об'єктом із деякої ієрархії класів; дозволяє визначити нову операцію, не змінюючи класів цих об'єктів.

4.6.3 Інші принципи програмування

Більшість ПС необґрунтовано перевантажена практично непотрібними функціями, що погіршує зручність їх використання кінцевими користувачами, а також ускладнює їх підтримку та розвиток розробниками. Саме тому було

створено та стандартизовано декілька принципів, які дозволяють вирішити ці проблеми.

KISS. Дотримання принципу KISS дозволяє розробляти рішення, які прості у використанні та у супроводі.

Основною метою KISS є забезпечення простоти ПС. Є два варіанти розшифровки цієї абревіатури, які користуються популярністю, а саме: «keep it simple, stupid!» і більш коректний для загального використання «keep it short and simple».

У проектуванні дотримання принципу KISS виражається в тому, що:

- ускладнення ПЗ лише завдає шкоди;
- необхідно реалізовувати тільки ті функціональні можливості та інтерфейси, які точно будуть використовуватись;
- не варто ускладнювати інтерфейси;
- безглуздо робити реалізацію складної бізнес-логіки, яка враховує абсолютно всі можливі варіанти поведінки ПС, користувача та навколишнього середовища – по-перше, це просто неможливо, а по-друге, така фанатичність змушує збирати «міжгалактичний космічний корабель», що найчастіше ірраціонально з комерційної точки зору.

У програмуванні дотримання принципу KISS можна описати наступним чином:

- немає сенсу безмежно збільшувати рівень абстракції, треба вміти вчасно зупинитися;
- безглуздо закладати в проект надлишкові функції «про запас», які може колись комусь знадобляться;
- не варто підключати велику бібліотеку, якщо від неї потрібна лише пара функцій;
- декомпозиція чогось складного на прості складові частини – це архітектурно правильний підхід;

- абсолютна математична точність або гранична деталізація потрібні не завжди – дані можна і потрібно обробляти з тією точністю, яка достатня для якісного вирішення завдання, а деталізацію видавати в потрібному, а не в максимально можливому обсязі.

Також KISS має багато спільного із принципом поділу інтерфейсів з принципів SOLID.

YAGNI (*англ. You aren't gonna need it*, *тобі це не знадобиться*). Якщо спрощено, то дотримання цього принципу полягає в тому, що можливості, які не описані у вимогах до системи, просто не повинні реалізовуватися. Це дозволяє вести розроблення, керуючись економічними критеріями – замовник не повинен оплачувати непотрібні йому функції, а розробники не повинні витрачати час на реалізацію того, що не потрібно.

Основна проблема, яку вирішує принцип YAGNI – це усунення жаги програмістів до використання зайвих абстракцій, до експериментів «з цікавості» та до реалізації функціоналу, який зараз не потрібен, але, на думку розробника, може або незабаром знадобитися, або просто буде корисним.

Безкоштовних функцій у ПП просто не буває. Якщо розглядати матеріальну сторону, то будь-які непотрібні, але фактично реалізовані функціональні можливості оплачуються замовником (до бюджету закладаються витрати на ті функції, які не потрібні), або виконавцем із прибутку за проектом. І той, і інший варіанти з погляду бізнесу – неправильні. Якщо ж говорити про нематеріальні витрати, то будь-які «бонусні» можливості ускладнюють супровід, збільшують ймовірність помилок та ускладнюють взаємодію з ПП – між обсягом кодової бази та описаними характеристиками є пряма залежність. Більше написаного коду – важче супроводжувати і вище ймовірність появи «багів», тут дуже доречна приказка: «найкращим є той код, якого немає».

Принципи YAGNI дуже схожий на KISS. KISS в основному націлений на спрощення кодової бази. YAGNI є трохи більш категоричним і застосовується для захисту проектів з розроблення ПЗ від «розмивання» їх рамок.

Підхід до реалізації проектів за технічним завданням є правильним з декількох причин:

1) Замовник не повинен платити за те, що йому не потрібно на даний момент.

2) ПП має бути максимально якісним без інтеграції непотрібних функцій.

3) Кодова база має забезпечувати достатньо простий супровід.

DRY (*англ. Don't Repeat Yourself, не повторюй себе*). Дотримання принципу програмування DRY дозволяє досягти високої супроводжуваності проекту, простоти внесення змін та якісного тестування.

Якщо код не дублюється, то для зміни логіки достатньо внесення виправлень всього в одному місці. Використання принципу DRY завжди призводить до декомпозиції складних алгоритмів на прості функції, які можна використовувати повторно. У свою чергу, декомпозиція складних операцій на більш прості значно спрощує розуміння і супроводжуваність програмного коду. Повторне використання функцій, винесених із складних алгоритмів, дозволяє скоротити час розроблення та тестування нового функціоналу.

Дотримання принципу DRY призводить до модульної архітектури ПЗ та до точного поділу відповідальності за бізнес-логіку між програмними класами. А це – запорука архітектури, яку можна легко супроводжувати. Хоча найчастіше саме модульність забезпечує саму можливість дотримання цього принципу у великих проектах.

У рамках одного програмного класу (або модуля) дотримуватися DRY і не повторюватися зазвичай досить просто. Також не вимагає титанічних зусиль робити це в рамках невеликих проектів, де всі розробники працюють з усім кодом ПЗ.

У великих проектах ситуація з DRY дещо складніша – повторення найчастіше з'являються через відсутність у розробників цілісної картини чи через неузгодженість дій у рамках кожної команди. Від розробників потрібно ретельне планування архітектури так само, як від архітектора чи керівника проекту потрібна наявність загального бачення ПС і чітка постановка завдань розробникам.

У процесі проектування DRY також може використовуватися – уніфікований та згрупований за якимось принципом доступ до конкретного функціо-

налу має бути в одному місці, а не розкиданий по ПС у довільних варіаціях. Цей підхід перетинається з принципом єдиної відповідальності з принципів SOLID.

Контрольні запитання

1. Які переваги надає дотримання принципів SOLID при написанні коду?
2. Які наслідки може викликати наявність двох чи більше причин для внесення змін в один і той же компонент програми?
3. Що таке аудиторія та відповідальність?
4. Які переваги дотримання принципу єдиної відповідальності?
5. У чому полягає ідея принципу відкритості/закритості?
6. Які є обмеження при використанні принципу відкритості/закритості?
7. У чому полягає ідея принципу підстановки Барбари Лісков?
8. Які є обмеження при використанні принципу підстановки Барбари Лісков?
9. У чому полягає ідея принципу інверсії залежностей?
10. Які є обмеження при використанні принципу інверсії залежностей?
11. Які шаблони входять до GRASP?
12. У чому полягає основна мета досягнення низького зчеплення та високої зв'язності?
13. Що таке «божественний» об'єкт та деструктивна розв'язка?
14. Які шаблони входять до твірних? Які завдання вони вирішують?
15. Які шаблони входять до структурних? Які завдання вони вирішують?
16. Які шаблони входять до поведінкових? Які завдання вони вирішують?
17. У чому полягає зміст принципу KISS?
18. У чому полягає зміст принципу YAGNI?
19. У чому полягає зміст принципу DRY?

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ

1. Sommerville I. Software Engineering / I. Sommerville. – London: Pearson, 2015. – 816 p.

2. Wiegers K. Software Requirements (Developer Best Practices) / K. Wiegers. – London: Pearson, 2013. – 672 p.

3. Martin R.C. The Clean Coder: A Code of Conduct for Professional Programmers / R.C. Martin. – London: Pearson, 2011. – 256 p.

4. Fowler M. Refactoring: Improving the Design of Existing Code / M. Fowler. – Boston: Addison-Wesley Professional, 2018. – 448 p.

5. Richards M. Fundamentals of Software Architecture: An Engineering Approach / M. Richards, N. Ford. – Sebastopol: O'Reilly Media, 2020. – 432 p.

6. Gamma E. Design Patterns: Elements of Reusable Object-Oriented Software / E. Gamma, R. Helm, R. Johnson, J. Vlissides. – Boston: Addison-Wesley Professional, 1994. – 416 p.

7. Якунін Ю.Ю. Технології розроблення програмного забезпечення: електрон. навч. посібник / Ю.Ю. Якунін. – Красноярськ: ІПК СФУ, 2008. – 225 с.

8. Савенко І.І. Технології розроблення програмного забезпечення: конспект лекції / І.І. Савенко. – Томськ: Видавництво Томського політехнічного університету, 2014. – 67 с.

9. Рудаков А.В. Технології розроблення програмних продуктів. Практикум: навч. посібник. для студ. установ середовищ. проф. освіти / А.В. Рудаков, Г.М. Федорова. – 4-те видання. – М: Видавничий центр «Академія», 2014. – 192 с.

10. Бахтізін В.В. Технології розробки програмного забезпечення: навч. посібник / В.В. Бахтізін, Л.А. Глухова. – Мінськ: БДУІР, 2010. – 267 с.

11. Axelrod A. Complete Guide to Test Automation: Techniques, Practices, and Patterns for Building and Maintaining Effective Software Projects / A. Axelrod. – NY: Apress, 2018. – 588 p.

12. Tamres L. Introducing Software Testing / L.Tamres. – Boston: Addison-Wesley Professional, 2002. – 304 p.

13. Макаровських Т. Документування програмного забезпечення. На допомогу технічному письменнику / Т. Макаровських. – М: Ленанд, 2016. – 264 с.

14. Pulse of the Profession [Електронний ресурс]. – Режим доступу: https://www.pmi.org/learning/thought-leadership/pulse (дата звернення: 10.02.2022) – Назва з екрана.

15. International Data Corporation [Електронний ресурс]. – Режим досту-пу: https://www.idc.com/ (дата звернення: 10.02.2022). – Назва з екрана.

16. Gartner [Електронний ресурс]. – Режим доступу: https://www.gartner.com/en (дата звернення: 10.02.2022). – Назва з екрана.

17. Новіков Ф.А. Навчально-методичний посібник з дисципліни Техно-логічні підходи до розробки програмного забезпечення/ Ф.А. Новіков. – СПб.: СПбГУ ІТМО, 2007. – 137 с.

18. Martin R.C. Agile Software Development, Principles, Patterns, and Practices / R.C. Martin. – London: Pearson, 2002. – 552 p.

19. Алексенко О.В. Технології програмування та створення програмних продуктів: конспект лекцій для студ. напряму підготовки 6.050101 «Комп'ютерні науки» усіх форм навчання / О.В. Алексенко. – Суми: СумДУ, 2013. – 133 с.

20. Орлов С. Програмна інженерія: технологи розробки програмного забезпечення. Підручник для вузів / С. Орлов. – СпБ: Пітер, 2017. – 640 с.

21. Гагаріна Л. Технології розробки програмного забезпечення: навча-льний посібник / Л. Гагаріна, Є. Кокорєва, Б. Віснадул. – М.: ВБ «ФОРУМ»: ІНФРА-М, 2008. – 400 с.

22. Гудов А.М. Технології розроблення програмного забезпечення: на-вчальний посібник / А.М. Гудов, С.Ю. Завозкін, С.М. Трофімов. – Кемерове: КДУ, 2009. – 138 с.

23. Beck K. Test Driven Development: By Example / K. Beck. – Boston: Addison-Wesley Professional, 2002. – 240 p.

24. Гагаріна Л.Г. Введення до архітектури програмного забезпечення: навчальний посібник / Л.Г. Гагаріна, А.Р. Федоров, П.А. Федоров. – Москва: ІНФРА-М, 2018. – 320 с.

25. Гудов А.М. Технологія розробки програмного забезпечення: навч. Посібник / А.М. Гудов, С.Ю. Завозкін, С.М. Трофімов. – Кемерово: «Кемеровський державний університет», 2009. – 138 с.

26. Larman C. Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and Iterative Development / C. Larman. – London: Pearson, 2004. – 736 p.

27. Fowler M. UML Distilled: A Brief Guide to the Standard Object Modeling Language / M. Fowler. – Boston: Addison-Wesley Professional, 2018. – 191 p.

28. Rumbaugh J. UML 2.0. Object-Oriented Modeling and Design with UML / J. Rumbaugh, M. Blaha. – London: Pearson, 2012. – 496 p.