МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

Тернопільський національний технічний університет імені Івана Пулюя

Кафедра програмної інженерії

Конспект лекцій

з дисципліни

«Об’єктні технології та інструментальні засоби конструювання програмного забезпечення»

для здобувачів другого (магістерського) рівнявищої освіти ОПП «Інженерія програмногозабезпечення» всіх форм навчання

Тернопіль — 2025

УДК 004.41(076.5)

Укладачі:
Михалик Д.М. к.т.н., доцент
Цебрій О.Р., к.ф.-м.н., доцент каф. ПІ
Стефанишин В.М., ас. каф. ПІ

Рецензент:
Загородна Н. В., к.т.н, доцент,
завідувач кафедрою кібербезпеки ТНТУ ім. Івана Пулюя

Розглянуто й затверджено на засіданні кафедри програмної інженерії. Протокол № 1 від 29.09.2025.

Розглянуто й затверджено на засіданні методичної комісії факультету комп'ютерно-інформаційних систем та програмної інженерії Тернопільського національного технічного університету імені Івана Пулюя.
Протокол № 1 від 1.09.2025.

Конспект лекцій з дисципліни «Об'єктні технології та інструментальні засоби конструювання програмного забезпечення» для здобувачів другого (магістерського) рівня вищої освіти ОПП «Інженерія програмного забезпечення» всіх форм навчання / Укладач: Михалик Д.М., Цебрій О.Р., Стефанишин В.М.– Тернопіль: ТНТУ ім. І. Пулюя, 2025 – 239 с.

Відповідальний за випуск: ас. каф. ПІ, Стефанишин В.М.

АНОТАЦІЯ

Пропонований конспект лекцій призначений для здобувачів другого (магістерського) рівня вищої освіти спеціальності 121/F2 «Інженерія програмного забезпечення» і містить систематизований виклад теоретичного матеріалу з дисципліни «Об’єктні технології та інструментальні засоби конструювання програмного забезпечення».

Головною метою видання є формування системного розуміння принципів і методів побудови якісного об’єктно-орієнтованого програмного забезпечення, здатного до розвитку, повторного використання та супроводу. Матеріал курсу побудовано від фундаментальних понять якості й модульності програмного забезпечення до практичних принципів організації системи класів, визначення їх відповідальностей, побудови контрактів, використання універсалізації, успадкування та інших механізмів об’єктно-орієнтованого конструювання.

У конспекті послідовно розглянуто питання якості програмного забезпечення, модульності, повторного використання програмних компонентів, абстрактних типів даних, управління пам’яттю, універсалізації, проєктування за контрактом, оброблення порушень контрактів, принципів і механізмів успадкування, правильного формування ієрархій класів, визначення класів та їх відповідальностей, використання глобальних об’єктів і констант. Окрему увагу приділено сучасним підходам до організації розроблення програмного забезпечення та практичному використанню інструментальних засобів програмної інженерії.

Особливістю видання є поєднання фундаментальних принципів об’єктно-орієнтованого конструювання з практичними проблемами сучасної розробки програмного забезпечення. Теоретичні положення розглядаються з позицій їх впливу на модульність, зв’язність і зчепленість компонентів, повторне використання, розширюваність, тестованість, надійність та підтримуваність програмних систем. Значна увага приділяється не лише способам застосування окремих механізмів об’єктно-орієнтованого програмування, а й визначенню умов, за яких їх використання є обґрунтованим або, навпаки, може призвести до погіршення структури програмного забезпечення.

Вивчення матеріалів конспекту спрямоване на формування у майбутніх інженерів програмного забезпечення здатності аналізувати структуру програмних систем, критично оцінювати прийняті проєктні рішення та обґрунтовано обирати методи й інструментальні засоби їх удосконалення. Конспект лекцій слугує теоретичним підґрунтям для:

- опрацювання та систематизації лекційного матеріалу з основних принципів об’єктно-орієнтованого конструювання програмного забезпечення;

- підготовки до виконання лабораторних робіт, пов’язаних із системами контролю версій, аналізом модульної структури програмних проєктів, розробленням абстрактних типів даних та використанням багатопотоковості;
- організації самостійної роботи студентів, зокрема поглибленого опрацювання професійної літератури, аналізу програмного коду та дослідження сучасних інструментальних засобів;
- підготовки та виконання курсового проєкту, що передбачає аналіз існуючого програмного продукту, обґрунтування його вдосконалення, реалізацію запропонованих змін та оцінювання отриманих результатів;
- підготовки до поточного, модульного та підсумкового контролю знань з дисципліни.

Конспект лекцій орієнтований на формування професійного інженерного мислення, за якого програмне забезпечення розглядається не лише як сукупність програмного коду, а як система взаємопов’язаних абстракцій, класів, модулів і компонентів із чітко визначеними відповідальностями та контрактами. Засвоєння викладеного матеріалу створює основу для прийняття обґрунтованих проєктних рішень, підвищення якості програмних продуктів та подальшого професійного розвитку фахівців у галузі інженерії програмного забезпечення.

ЗМІСТ

ВСТУП 9
ТЕМАТИЧНИЙ ПЛАН НАВЧАЛЬНОЇ ДИСЦИПЛІНИ 13
Лекція 1 16
1.1 Передумови забезпечення якості програмного забезпечення 18
1.2 Основні складові забезпечення якості програмного забезпечення 19
1.3 Завдання, цілі та метрики забезпечення якості програмного забезпечення 21
1.4 Міжнародні стандарти забезпечення якості програмного забезпечення 22
Висновки 24
Лекція 2 25
2.1 Поняття модульності 25
2.2 Критерії модульності 26
2.3 Правила модульності 29
2.4 Принципи модульності 31
Висновки 34
Лекція 3 35
3.1. Поняття повторного використання програмного забезпечення 35
3.2 Повторне використання програмних бібліотек і фреймворків 38
3.3. Повторне використання за допомогою шаблонів проєктування 39
3.4. Компонентна архітектура та компонентне програмування 45
3.5 Переваги та обмеження повторного використання програмного забезпечення 56
Висновки 57
Лекція 4 59
4.1 Поняття абстрактного типу даних 59
4.2 Варіанти реалізації абстрактних типів даних 63
4.3 Формальна специфікація абстрактних типів даних 66
4.4 Абстрактний тип даних STACK як формальна модель обчислень 71
4.5 Перехід від абстрактних типів даних до класів 74
Висновки 76
Лекція 5 78
5.1 Життєвий цикл об’єктів і способи розміщення даних у пам’яті 78
5.2 Проблема вивільнення пам’яті та ручне керування пам’яттю 89
5.3 Керування пам’яттю на рівні програмних компонентів 94
5.4 Автоматичне керування пам’яттю 98
5.5 Практичні аспекти й оптимізація автоматичного керування пам’яттю 102
Висновки 108
Лекція 6 110
6.1 Поняття універсалізації та способи узагальнення типів 110

6.2 Необхідність параметризації типів 114
6.3 Узагальнені класи та їх параметри 119
6.4 Масиви як приклад узагальненого класу 124
6.5 Переваги, вартість та обмеження універсалізації 128
Висновки 133
Лекція 7 134
7.1 Поняття проєктування за контрактом 134
7.2 Передумови та постумови 136
7.3 Інваріанти класу 140
7.4 Контракти в описі класів та операцій 145
7.5 Перевірка контрактів і оброблення порушень 152
7.6 Контракти, успадкування та повторне визначення операцій 153
7.7 Переваги та практичне застосування проєктування за контрактом 154
Висновки 155
Лекція 8 156
8.1 Основні поняття винятку та відмови 156
8.2 Причини виникнення винятків і порушень контракту 157
8.3 Принципи дисциплінованого оброблення винятків 159
8.4 Механізм оброблення винятків rescue і retry 160
8.5 Приклади оброблення винятків 161
8.6 Завдання секції rescue та відновлення інваріанта 164
8.7 Розширене оброблення винятків і запобігання повторним порушенням 165
Висновки 167
Лекція 9 168
9.1 Поняття та призначення успадкування 168
9.2 Батьківські класи, класи-нащадки та успадковані компоненти 171
9.3 Поліморфізм і сумісність типів 174
9.4 Повторне визначення компонентів і динамічне зв’язування 177
9.5 Відкладені компоненти та класи 179
9.6 Значення, переваги й обмеження успадкування 182
9.7 Правила правильного застосування успадкування 184
Висновки 188
Лекція 10 189
10.1 Поняття відкату в інтерактивних системах 189
10.2 Команда як об’єкт і базовий клас COMMAND 191
10.3 Успадкування, поліморфізм і динамічне зв’язування в механізмі Undo 193
10.4 Багаторівневий механізм Undo/Redo та історія команд 194
10.5 Практичні аспекти реалізації та інтерфейс користувача 196

10.6 Переваги, обмеження та загальні принципи побудови механізму відкату 198
Висновки 199
Лекція 11 200
11.1 Поняття методології розроблення програмного забезпечення 200
11.2 Agile та основні принципи гнучкого розроблення 201
11.3 Scrum: структура, команда, події та артефакти 202
11.4 Kanban та керування потоком робіт 203
11.5 Порівняння Agile, Scrum і Kanban 204
11.6 Застосування гнучких підходів під час об’єктно-орієнтованого розроблення 207
Висновки 208
Лекція 12 209
12.1 Необхідність правильного застосування успадкування 209
12.2 Вибір між успадкуванням і клієнтським зв’язком 210
12.3 Типові помилки використання успадкування 212
12.4 Основні допустимі форми успадкування 213
12.5 Успадкування для розширення та повторного використання 214
12.6 Побудова та розвиток ієрархій класів 215
12.7 Практичні правила правильного використання успадкування 216
Висновки 217
Лекція 13 218
13.1 Поняття ідентифікації класів 218
13.2 Аналіз вимог як джерело потенційних класів 219
13.3 Критерії виокремлення самостійного класу та класів в об’єктно-орієнтованій системі 220
13.4 Ознаки помилкового визначення класів 223
13.5 Додаткові джерела класів та повторне використання 224
13.6 Послідовність формування системи класів 226
Висновки 227
Лекція 14 228
14.1 Глобальна інформація в об’єктно-орієнтованій системі 228
14.2 Символічні та явні константи 229
14.3 Константи класових типів і одноразові функції 231
14.4 Спільні об’єкти та одноразова ініціалізація 232
14.5 Рядкові константи та фіксовані набори значень 233
14.6 Проблеми надмірного використання глобальних даних 234
14.7 Практичні правила використання глобальних об’єктів і констант 235
Висновки 236

СПИСОК ВИКОРИСТАНОЇ ЛІТЕРАТУРИ 238

ВСТУП

Сучасна інженерія програмного забезпечення характеризується постійним зростанням складності програмних систем, швидкою зміною технологій та підвищенням вимог до якості, надійності, підтримуваності й можливості подальшого розвитку програмних продуктів. За таких умов професійна підготовка інженера програмного забезпечення не може обмежуватися лише знанням синтаксису мов програмування або володінням окремими фреймворками. Важливого значення набуває розуміння фундаментальних принципів побудови програмних систем, організації програмного коду та взаємодії між його складовими.

Дисципліна «Об'єктні технології та інструментальні засоби конструювання програмного забезпечення» є складовою підготовки здобувачів другого (магістерського) рівня вищої освіти спеціальності F2 «Інженерія програмного забезпечення». Мета курсу полягає у вивченні основних принципів об'єктних технологій конструювання програмного забезпечення, опануванні прийомів роботи з програмним кодом, а також розумінні причин і наслідків проблем, що виникають під час розроблення складних програмних систем.

Об'єктно-орієнтований підхід залишається одним із фундаментальних способів організації складних програмних систем. Його ефективне використання передбачає значно більше, ніж механічне застосування класів, об'єктів, успадкування чи поліморфізму. Якісне об'єктно-орієнтоване програмне забезпечення вимагає правильної декомпозиції системи, чіткого розподілу відповідальностей, мінімізації залежностей, забезпечення модульності, повторного використання та формального визначення поведінки програмних компонентів.

Мета вивчення дисципліни полягає у формуванні у студентів системного інженерного мислення та здатності приймати обґрунтовані рішення щодо структури програмного забезпечення, способів організації класів і модулів, використання механізмів абстракції, універсалізації, успадкування та контрактного проєктування, а також вибору відповідних інструментальних засобів розробки.

Матеріал конспекту побудовано послідовно: від загальних характеристик якості програмного забезпечення та принципів модульної побудови — до абстрактних типів даних, управління пам'яттю, універсалізації та проєктування за контрактом. Значний блок присвячено успадкуванню, його можливостям, обмеженням і правилам коректного застосування. Завершальні лекції розглядають методи визначення класів, використання глобальних об'єктів і констант та місце сучасних методологій організації розроблення програмного забезпечення. Така послідовність відповідає тематичному плану робочої програми дисципліни.

Особлива увага в курсі приділяється взаємозв'язку між структурою програмного забезпечення та його якістю. Модульність, приховування інформації,

абстракція, низька зв’язність компонентів, повторне використання та контрольоване успадкування розглядаються не як ізольовані теоретичні поняття, а як взаємопов’язані механізми побудови програмних систем, які повинні залишатися зрозумілими, тестованими та придатними до модифікації протягом тривалого життєвого циклу.

Важливою складовою курсу є проєктування за контрактом (Design by Contract). Використання передумов, постумов та інваріантів дозволяє формалізувати взаємні зобов’язання програмних компонентів, чіткіше визначати їх поведінку та локалізувати відповідальність за порушення встановлених умов. Розгляд ситуацій порушення контрактів формує розуміння зв’язку між коректністю локальних програмних компонентів і надійністю системи в цілому.

Окремий блок лекцій присвячено успадкуванню як одному з найбільш потужних і водночас потенційно небезпечних механізмів об’єктно-орієнтованого програмування. Студенти розглядають принципи побудови ієрархій класів, повторного використання та розширення поведінки, аналізують обмеження успадкування і ситуації, у яких композиція або клієнтський зв’язок можуть бути кращою альтернативою. Робоча програма передбачає декілька послідовних лекцій, присвячених цій проблематиці.

Практична спрямованість дисципліни забезпечується поєднанням теоретичного матеріалу з лабораторними роботами, які передбачають використання сучасних систем контролю версій, аналіз модульної структури відкритих програмних проєктів, розроблення бібліотеки абстрактних типів даних та застосування багатопотоковості. Таким чином, теоретичні принципи курсу безпосередньо пов’язуються з аналізом і конструюванням реального програмного забезпечення.

Основні завдання курсу:

- сформувати розуміння сутності та характеристик якості програмного забезпечення і факторів, що впливають на неї;
- ознайомити студентів із принципами модульності, декомпозиції та повторного використання програмних компонентів;
- сформувати розуміння абстрактних типів даних як основи побудови програмних абстракцій;
- дослідити принципи управління пам’яттю та життєвим циклом програмних об’єктів;
- опанувати підходи до універсалізації та створення повторно використовуваних параметризованих програмних компонентів;
- сформувати навички застосування проєктування за контрактом для специфікації поведінки класів і програмних компонентів;
- поглибити розуміння принципів успадкування, поліморфізму та побудови коректних ієрархій класів;

- навчити обґрунтовано вибирати між успадкуванням, композицією та іншими видами взаємодії об’єктів;
- сформувати здатність визначати класи, їх відповідальності та взаємозв’язки на основі аналізу предметної області;
- ознайомити студентів із принципами правильного використання глобальних об’єктів і констант;
- показати взаємозв’язок фундаментальних принципів об’єктно-орієнтованого конструювання із сучасними підходами до організації командного розроблення програмного забезпечення;
- сформувати навички використання сучасних інструментальних засобів для розроблення, аналізу, контролю версій і забезпечення якості програмного коду.

Опанування матеріалу дисципліни повинно забезпечити здатність студентів оцінювати та обирати методи розроблення програмного забезпечення, аналізувати й обґрунтовувати проєктні рішення, модифікувати архітектуру програмних систем, свідомо вибирати парадигми та мови програмування і забезпечувати якість програмного забезпечення із застосуванням сучасних засобів розробки, тестування та верифікації.

Конспект містить 14 лекцій, тематика яких відповідає робочій програмі навчальної дисципліни.

Тема 1. Якість, модульність та повторне використання програмного забезпечення

- Лекція 1. Якість програмного забезпечення. Поняття якості програмного забезпечення, її характеристики, критерії оцінювання та роль забезпечення якості у процесі створення й супроводу програмних систем.
- Лекція 2. Модульність. Принципи модульної побудови програмного забезпечення, декомпозиція програмних систем, взаємозв’язок модульності з повторним використанням, розвитком та супроводом програмного забезпечення.
- Лекція 3. Підходи до повторного використання. Основні механізми повторного використання програмного забезпечення, використання бібліотек, компонентів, фреймворків, шаблонів та інших повторно використовуваних програмних абстракцій.

Тема 2. Абстракція, управління ресурсами та контрактне проєктування

- Лекція 4. Абстрактні типи даних. Поняття абстрактного типу даних, відокремлення специфікації від реалізації, операції над абстрактними структурами даних та їх роль у побудові модульних систем.
- Лекція 5. Управління пам’яттю. Життєвий цикл програмних об’єктів, ручне й автоматичне управління пам’яттю, збирання сміття та проблеми ефективного використання ресурсів.

- Лекція 6. Універсалізація. Побудова універсальних програмних компонентів, параметризація типів, обмежена й необмежена універсалізація та її зв’язок із повторним використанням.
- Лекція 7. Проектування за контрактом. Контракти програмних компонентів, передумови, постумови, інваріанти класів та розподіл відповідальності між клієнтом і постачальником.
- Лекція 8. Коли контракт порушується. Причини та наслідки порушень контрактів, виявлення некоректних станів і поведінки та принципи роботи з винятковими ситуаціями.

Тема 3. Успадкування та об’єктно-орієнтовані підходи

- Лекція 9. Принципи наслідування. Основні механізми успадкування, його переваги, обмеження та роль у побудові об’єктно-орієнтованих програмних систем.
- Лекція 10. Наслідування: відкат в інтерактивних системах. Використання механізмів успадкування в інтерактивних системах та підтримка змін стану й поведінки програмних об’єктів.
- Лекція 11. ООП методології. Сучасні підходи до організації розроблення програмного забезпечення, зокрема Agile, Scrum і Kanban, та їх вплив на командну роботу.
- Лекція 12. Використовуйте наслідування правильно. Практичні правила обґрунтованого застосування успадкування, типові помилки, вибір між успадкуванням і композицією та побудова гнучких ієрархій класів.

Тема 4. Формування системи класів та глобальна інформація

- Лекція 13. Як знайти класи. Методики ідентифікації класів, визначення їх відповідальностей, властивостей та операцій, формування системи класів на основі аналізу програмної системи.
- Лекція 14. Глобальні об’єкти і константи. Принципи представлення глобальної інформації, використання символічних і класових констант, спільних об’єктів та проблеми надмірного глобального стану.

Опанування всіх чотирнадцяти лекцій формує цілісне уявлення про об’єктно-орієнтоване конструювання програмного забезпечення як систему взаємопов’язаних інженерних принципів, а не лише як сукупність мовних механізмів. Отримані знання створюють основу для аналізу існуючих програмних систем, обґрунтованого вибору способів їх структурної організації, підвищення якості програмного коду та проєктування програмного забезпечення, придатного до повторного використання, тестування, супроводу та подальшого розвитку.

ТЕМАТИЧНИЙ ПЛАН НАВЧАЛЬНОЇ ДИСЦИПЛІНИ

Дисципліна: «Об’єктні технології та інструментальні засоби конструювання програмного забезпечення».

Рівень вищої освіти: другий (магістерський).

Спеціальність: F2 «Інженерія програмного забезпечення».

Форма підсумкового семестрового контролю: залік.

Обсяг дисципліни: 4 кредити ECTS (120 годин). Для денної форми навчання передбачено 42 години аудиторної роботи, з них 28 годин лекційних та 14 годин лабораторних занять, а також 78 годин самостійної роботи. Для заочної форми навчання передбачено 24 години аудиторної та 96 годин самостійної роботи.

Метою вивчення дисципліни є формування у здобувачів системного розуміння принципів і технологій конструювання програмного забезпечення на основі об’єктно-орієнтованого підходу, а також розвиток практичних навичок аналізу, проєктування, реалізації та вдосконалення програмних систем із використанням сучасних інструментальних засобів.

У межах дисципліни розглядаються фундаментальні питання якості та модульності програмного забезпечення, повторного використання програмних компонентів, абстрактних типів даних, управління пам’яттю, універсалізації, проєктування за контрактом, успадкування, визначення класів, глобальних об’єктів і констант. Особлива увага приділяється правильному розподілу відповідальностей між програмними компонентами, побудові гнучких ієрархій класів, зменшенню надлишкових залежностей та забезпеченню можливості подальшого розвитку і супроводу програмних систем.

Мета проведення лекційних занять полягає у формуванні теоретичної бази з об’єктно-орієнтованого конструювання програмного забезпечення та розвитку інженерного мислення, необхідного для прийняття обґрунтованих проєктних рішень. Під час лекцій студенти вивчають принципи модульної побудови програмних систем, способи створення й повторного використання програмних абстракцій, механізми універсалізації, контрактного проєктування та успадкування, а також підходи до визначення класів і організації глобальної інформації в програмному забезпеченні.

Лекційний матеріал спрямований не лише на опанування окремих механізмів об’єктно-орієнтованого програмування, а й на розуміння їх впливу на такі характеристики програмного забезпечення, як підтримуваність, модифікованість, повторне використання, тестованість, надійність і якість.

Мета проведення лабораторних занять полягає у формуванні практичних навичок застосування принципів об’єктно-орієнтованого конструювання та сучасних інструментальних засобів під час роботи з реальними програмними проєктами. Лабораторний практикум передбачає використання сучасних систем контролю версій,

аналіз модульної структури програмних проєктів із відкритим вихідним кодом, розроблення бібліотеки абстрактних типів даних та застосування багатопотоковості під час роботи з графічними об’єктами.

При цьому основними завданнями дисципліни є:

- сформувати у студентів розуміння принципів побудови якісного, модульного та підтримуваного програмного забезпечення;
- навчити аналізувати структуру програмних систем, визначати їх модулі, класи, відповідальності та взаємозв’язки;
- поглибити знання щодо повторного використання програмного забезпечення та побудови універсальних програмних компонентів;
- сформувати навички застосування абстрактних типів даних та принципів інформаційного приховування;
- ознайомити з принципами управління пам’яттю та життєвим циклом програмних об’єктів;
- навчити використовувати проєктування за контрактом для формалізації поведінки класів і програмних компонентів;
- сформувати розуміння можливостей, обмежень і правил правильного використання успадкування;
- навчити обґрунтовано вибирати між успадкуванням, композицією та іншими видами взаємодії між об’єктами;
- розвинути здатність визначати класи на основі аналізу предметної області та функціонального призначення програмної системи;
- сформувати навички використання сучасних систем контролю версій та інших інструментальних засобів конструювання програмного забезпечення;
- розвинути здатність аналізувати існуючі програмні рішення, виявляти їх недоліки та пропонувати обґрунтовані шляхи вдосконалення.

У результаті опанування дисципліни студент повинен бути здатним не лише реалізовувати окремі програмні компоненти, а й оцінювати структуру програмного забезпечення в цілому, обґрунтовувати вибір методів і засобів його конструювання, модифікувати архітектурні та програмні рішення й забезпечувати якість програмного продукту.

За результатами вивчення дисципліни ” Управління IT-проектами” студент повинен продемонструвати такі результати навчання (***кінцеві, підсумкові та інтегративні результати навчання, що визначають нормативний зміст підготовки):***

1. РН02 Оцінювати і вибирати ефективні методи і моделі розроблення, впровадження, супроводу програмного забезпечення та управління відповідними процесами на всіх етапах життєвого циклу.

2. РН06 Розробляти і оцінювати стратегії проєктування програмних засобів; обґрунтовувати, аналізувати і оцінювати варіанти проектних рішень з точки зору якості кінцевого програмного продукту, ресурсних обмежень та інших факторів.

3. РН08 Розробляти і модифікувати архітектуру програмного забезпечення для реалізації вимог замовника.

4. РН09 Обґрунтовано вибирати парадигми і мови програмування для розроблення програмного забезпечення; застосовувати на практиці сучасні засоби розроблення програмного забезпечення.

5. РН11 Забезпечувати якість на всіх стадіях життєвого циклу програмного забезпечення, у тому числі з використанням релевантних моделей та методів оцінювання, а також засобів автоматизованого тестування і верифікації програмного забезпечення

Вивчення навчальної дисципліни передбачає формування та розвиток у студентів компетентностей:

інтегральна компептентність:

Здатність розв'язувати складні спеціалізовані завдання або практичні проблеми інженерії програмного забезпечення, що характеризуються комплексністю та невизначеністю умов, із застосуванням теорій та методів інформаційних технологій.

загальні компетентності:

1. ЗК01 Здатність до абстрактного мислення, аналізу та синтезу.
2. ЗК03 Здатність проводити дослідження на відповідному рівні.
3. ЗК05. Здатність генерувати нові ідеї (креативність)

(фахові) компетентності:

4. СК02. Здатність розробляти і реалізовувати наукові та/або прикладні проєкти у сфері інженерії програмного забезпечення.
5. СК03. Здатність проєктувати архітектуру програмного забезпечення, моделювати процеси функціонування окремих підсистем і модулів.
6. СК08. Здатність розробляти і координувати процеси, етапи та ітерації життєвого циклу програмного забезпечення на основі застосування
7. учасних моделей, методів та технологій розроблення програмного забезпечення.
8. СК09. Здатність забезпечувати якість програмного забезпечення.
9. СК010. Здатність розробляти інноваційне програмне забезпечення.

Лекція 1

Тема: Якість програмного забезпечення

Мета: Сформувати у студентів розуміння поняття якості програмного забезпечення, її ролі у процесі розроблення сучасних програмних систем, ознайомити з основними моделями якості, міжнародними стандартами оцінювання програмного забезпечення, характеристиками якості відповідно до ISO/IEC 25010, а також сучасними підходами до забезпечення якості на всіх етапах життєвого циклу програмного забезпечення.

План лекції

1. Передумови забезпечення якості програмного забезпечення.
2. Основні складові забезпечення якості програмного забезпечення.
3. Завдання, цілі та метрики забезпечення якості програмного забезпечення.
4. Міжнародні стандарти забезпечення якості програмного забезпечення.
5. Висновки.

Підхід до програмної інженерії спрямований на досягнення єдиної мети — створення високоякісного програмного забезпечення у визначені терміни. Проте закономірно виникає запитання: що таке якість програмного забезпечення?

Філіп Кросбі у своїй фундаментальній праці, присвяченій питанням якості, дає дотепну відповідь на це запитання:

Проблема управління якістю полягає не в тому, що люди про неї не знають. Проблема полягає в тому, що вони думають, ніби знають достатньо.

У цьому сенсі якість має багато спільного із сексом. Усі її підтримують (звичайно, за певних умов). Кожен вважає, що розуміє її сутність (хоча навряд чи зміг би чітко її пояснити). Більшість переконана, що досягти успіху дуже просто — потрібно лише діяти відповідно до природних схильностей (адже якось усе працює). І, звичайно, майже всі вважають, що проблеми в цій сфері виникають через інших людей (якби вони лише виконували свою роботу належним чином).

Дійсно, якість є складним і багатогранним поняттям.

Недостатньо лише говорити про важливість якості програмного забезпечення. Необхідно:

- чітко визначити, що розуміється під поняттям якості програмного забезпечення;
- створити комплекс заходів, які забезпечуватимуть високу якість кожного результату роботи, отриманого в процесі розроблення програмного забезпечення;
- виконувати заходи із забезпечення та контролю якості в межах кожного програмного проєкту;

- використовувати метрики якості для розроблення стратегій удосконалення процесів розроблення програмного забезпечення та, як наслідок, підвищення якості кінцевого програмного продукту.

Відповідальність за якість програмного забезпечення покладається на всіх учасників процесу програмної інженерії.

Якість має важливе значення, оскільки дотримання вимог до неї зменшує обсяг повторних робіт, скорочує витрати на розроблення та прискорює вихід програмного продукту на ринок.

Перед початком заходів із забезпечення якості необхідно визначити поняття якості програмного забезпечення на різних рівнях абстракції. Після цього команда розробників повинна сформувати набір заходів із забезпечення якості, які дозволять своєчасно виявляти та усувати помилки ще до передачі результатів роботи на наступні етапи розроблення.

Основним документом, що визначає стратегію забезпечення якості проєкту, є план забезпечення якості програмного забезпечення (Software Quality Assurance Plan). Під час моделювання та програмування основними результатами діяльності із забезпечення якості є матеріали технічних оглядів. На етапі тестування формуються тестові плани та процедури тестування. Крім того, можуть створюватися й інші документи, пов'язані з удосконаленням процесів розроблення програмного забезпечення.

Основною метою забезпечення якості є виявлення помилок до того, як вони перетворяться на дефекти програмного забезпечення. Для цього необхідно постійно підвищувати ефективність процесів виявлення та усунення дефектів, що дає змогу зменшити обсяг повторних робіт під час розроблення програмного забезпечення.

Деякі розробники й досі помилково вважають, що питання якості виникають лише після завершення програмування. Насправді це не відповідає сучасним принципам програмної інженерії. Забезпечення якості програмного забезпечення (Software Quality Assurance, SQA) є наскрізною діяльністю, яка виконується протягом усього процесу розроблення програмного забезпечення.

Забезпечення якості програмного забезпечення охоплює:

1. процес забезпечення якості програмного забезпечення;
2. заходи із забезпечення та контролю якості, включаючи технічні огляди та багаторівневу стратегію тестування;
3. застосування сучасних методів і засобів програмної інженерії;
4. контроль усіх програмних артефактів та змін, що вносяться до них;
5. забезпечення відповідності процесу розроблення встановленим стандартам;
6. використання механізмів вимірювання, аналізу та звітності щодо показників якості.

Подальший матеріал присвячений питанням управління якістю та процесам, які дозволяють організації забезпечити виконання необхідних робіт у потрібний час, належним способом і відповідно до встановлених вимог.

1.1 Передумови забезпечення якості програмного забезпечення

Контроль та забезпечення якості є невід'ємними складовими діяльності будь-якої організації, яка створює продукцію для використання іншими людьми. До початку XX століття відповідальність за якість продукції повністю покладалася на майстра, який її виготовляв. Із розвитком промисловості та впровадженням масового виробництва контроль якості став окремим видом діяльності, який виконували спеціалісти, не залучені безпосередньо до виготовлення продукції.

Перша формалізована система забезпечення та контролю якості була впроваджена в компанії Bell Laboratories у 1916 році. Надалі цей підхід швидко поширився у виробничій сфері. У 1940-х роках були запропоновані більш формальні методи контролю якості, що базувалися на використанні вимірювань, статистичного аналізу та безперервного вдосконалення виробничих процесів як основних складових управління якістю.

Сьогодні практично кожна компанія застосовує механізми забезпечення якості своєї продукції. Високий рівень якості став не лише вимогою до продукції, а й важливою конкурентною перевагою підприємства.

Розвиток забезпечення якості програмного забезпечення відбувався аналогічно розвитку систем контролю якості у виробництві апаратного забезпечення. На початкових етапах розвитку обчислювальної техніки (1950–1960-ті роки) відповідальність за якість програмного забезпечення повністю покладалася на програміста. У 1970-х роках стандарти забезпечення якості програмного забезпечення почали активно застосовуватися під час розроблення програмного забезпечення для військових проєктів, а згодом були впроваджені й у комерційній розробці.

Забезпечення якості програмного забезпечення (Software Quality Assurance, SQA) — це сукупність запланованих і систематичних заходів, спрямованих на досягнення та підтримання високої якості програмного забезпечення. Основною метою SQA є впевненість у тому, що процес розроблення та його результати відповідають установленим вимогам, стандартам і очікуванням замовника.

Відповідальність за забезпечення якості програмного забезпечення покладається не лише на програмістів. У цьому процесі беруть участь розробники, керівники проєктів, фахівці із забезпечення якості, замовники, представники бізнесу та інші учасники процесу розроблення програмного забезпечення.

Група забезпечення якості програмного забезпечення (SQA Group) фактично виступає внутрішнім представником інтересів замовника. Її основним завданням є

оцінювання програмного продукту з позиції кінцевого користувача та перевірка відповідності програмного забезпечення встановленим вимогам щодо якості.

До основних завдань групи SQA належать:

- перевірка відповідності програмного забезпечення встановленим характеристикам якості;
- контроль дотримання стандартів і нормативних документів під час розроблення;
- перевірка правильності виконання процесів програмної інженерії;
- оцінювання результатів технічних оглядів, тестування та інших заходів контролю якості;
- забезпечення підтримання необхідного рівня якості програмного забезпечення протягом усього життєвого циклу його розроблення.

Таким чином, забезпечення якості є безперервним процесом, який охоплює всі етапи життєвого циклу програмного забезпечення та спрямований на створення програмного продукту, що відповідає встановленим вимогам, стандартам і потребам користувачів.

1.2 Основні складові забезпечення якості програмного забезпечення

Забезпечення якості програмного забезпечення (Software Quality Assurance, SQA) охоплює широкий спектр заходів і процесів, спрямованих на управління якістю програмного забезпечення. Основні складові забезпечення якості наведено нижче.

Стандарти. Міжнародні організації зі стандартизації, зокрема IEEE, ISO та інші, розробили значну кількість стандартів у галузі програмної інженерії та супровідної документації. Використання стандартів може бути добровільним або визначатися вимогами замовника чи інших зацікавлених сторін. Одним із основних завдань SQA є забезпечення дотримання прийнятих стандартів і відповідності їм усіх результатів розроблення програмного забезпечення.

Технічні огляди та аудити. Технічні огляди є одним із найважливіших засобів контролю якості та виконуються фахівцями з програмної інженерії для виявлення помилок ще до початку тестування. Аудит є різновидом перевірки, яку здійснюють спеціалісти із забезпечення якості з метою контролю дотримання встановлених вимог, процедур і стандартів під час розроблення програмного забезпечення. Наприклад, аудит процесу проведення технічних оглядів дозволяє переконатися, що вони виконуються відповідно до встановлених правил і забезпечують ефективне виявлення помилок.

Тестування. Тестування є одним із ключових засобів контролю якості програмного забезпечення. Його основною метою є виявлення помилок і дефектів програмного продукту. Одним із завдань SQA є забезпечення належного планування,

організації та виконання тестування для досягнення максимальної ефективності процесу перевірки.

Збирання та аналіз інформації про помилки й дефекти. Постійне вдосконалення процесів розроблення неможливе без аналізу отриманих результатів. Тому SQA передбачає накопичення, класифікацію та аналіз інформації про помилки й дефекти з метою визначення причин їх виникнення та розроблення заходів щодо їх запобігання у майбутньому.

Управління змінами. Зміни є невід'ємною складовою будь-якого програмного проєкту. За відсутності належного управління змінами виникає ризик неузгодженості між компонентами програмної системи, що негативно впливає на її якість. Одним із завдань SQA є забезпечення впровадження ефективних процедур управління змінами протягом усього життєвого циклу програмного забезпечення.

Навчання персоналу. Підвищення якості програмного забезпечення значною мірою залежить від професійної підготовки розробників, керівників проєктів та інших учасників процесу розроблення. З цією метою організація повинна забезпечувати постійне навчання персоналу та впровадження сучасних практик програмної інженерії.

Управління постачальниками програмного забезпечення. Значна частина програмних продуктів або їх компонентів може постачатися сторонніми організаціями. Завданням SQA є контроль якості таких компонентів, визначення вимог до постачальників та включення вимог щодо забезпечення якості до відповідних договорів.

Управління безпекою. Зі зростанням кількості кіберзагроз питання інформаційної безпеки набувають особливої актуальності. SQA забезпечує використання відповідних процесів, методів і технологій, спрямованих на захист програмного забезпечення та даних від несанкціонованого доступу, модифікації або втрати.

Забезпечення безпечності програмного забезпечення. У програмних системах, що використовуються в транспорті, медицині, авіації, енергетиці та інших критично важливих сферах, навіть незначні дефекти можуть призвести до серйозних наслідків. Тому SQA передбачає оцінювання можливих ризиків, пов'язаних із відмовами програмного забезпечення, та розроблення заходів щодо їх мінімізації.

Управління ризиками. Аналіз, оцінювання та мінімізація ризиків є важливою складовою забезпечення якості програмного забезпечення. SQA контролює правильність виконання процедур управління ризиками та забезпечує розроблення планів реагування на можливі ризикові ситуації.

Окрім наведених напрямів діяльності, забезпечення якості програмного забезпечення поширюється також на процеси супроводу програмного забезпечення,

підготовку технічної документації, роботу служб підтримки користувачів та інші допоміжні процеси, для яких якість є одним із ключових критеріїв ефективності.

1.3 Завдання, цілі та метрики забезпечення якості програмного забезпечення

Забезпечення якості програмного забезпечення (Software Quality Assurance, SQA) охоплює комплекс заходів, які виконуються двома основними групами учасників: розробниками програмного забезпечення та спеціалістами групи забезпечення якості (SQA Group). Розробники відповідають за якість програмного продукту шляхом застосування сучасних методів програмної інженерії, проведення технічних оглядів, використання програмних метрик та виконання комплексного тестування. Група SQA забезпечує планування процесів контролю якості, здійснює нагляд за їх виконанням, проводить аналіз отриманих результатів, веде відповідну документацію та готує звіти щодо стану якості програмного забезпечення. Основною метою діяльності групи SQA є сприяння команді розробників у створенні програмного продукту високої якості. Для досягнення цієї мети група забезпечення якості виконує низку основних завдань. До основних завдань групи SQA належать:

- розроблення плану забезпечення якості програмного забезпечення (Software Quality Assurance Plan), який визначає перелік заходів із забезпечення якості, стандарти, процедури контролю, порядок проведення аудитів, технічних оглядів, а також правила реєстрації та супроводу помилок;
- участь у формуванні опису процесу розроблення програмного забезпечення та перевірка його відповідності корпоративним стандартам, міжнародним нормативним документам і вимогам проєкту;
- контроль виконання процесів програмної інженерії відповідно до встановлених процедур, виявлення відхилень та перевірка їх усунення;
- проведення аудитів програмних артефактів з метою перевірки їх відповідності вимогам, стандартам і прийнятим правилам розроблення;
- документування всіх виявлених відхилень у процесах розроблення та програмних артефактах, а також контроль їх усунення відповідно до встановлених процедур;
- реєстрація випадків невідповідності встановленим вимогам та інформування керівництва проєкту про результати проведених перевірок;
- участь в управлінні змінами програмного забезпечення, а також збирання, аналіз і використання програмних метрик для оцінювання якості процесу розроблення.

Реалізація зазначених заходів спрямована на досягнення низки практичних цілей забезпечення якості програмного забезпечення.

Якість вимог. Якість моделі вимог суттєво впливає на всі подальші етапи розроблення програмного забезпечення. Вимоги повинні бути повними, однозначними, несуперечливими та зрозумілими. Одним із головних завдань SQA є перевірка правильності сформованих вимог ще до початку проєктування програмної системи.

Якість проєктування. Усі компоненти архітектури та проєктних моделей повинні відповідати встановленим вимогам і забезпечувати високий рівень якості майбутнього програмного продукту. Під час аналізу проєктних рішень оцінюються характеристики архітектури, повнота моделей, складність інтерфейсів і доцільність використання шаблонів проєктування.

Якість програмного коду. Вихідний код повинен відповідати прийнятим стандартам програмування, бути зрозумілим, підтримуваним, придатним до повторного використання та супроводжуватися необхідною документацією. Для оцінювання якості програмного коду використовуються показники складності, підтримуваності, зрозумілості, повторного використання компонентів і якості документування.

Ефективність контролю якості. Заходи контролю якості повинні забезпечувати максимально ефективне використання ресурсів проєкту. Для цього аналізується правильність розподілу часу між технічними оглядами, тестуванням та іншими видами перевірок, а також оцінюється результативність проведених заходів із виявлення та усунення дефектів.

Для оцінювання рівня якості використовуються спеціальні **метрики**, які дозволяють кількісно визначити окремі характеристики програмного забезпечення. Залежно від поставленої мети можуть застосовуватися такі групи метрик:

- для оцінювання якості вимог — кількість неоднозначних формулювань, повнота специфікації, кількість змін вимог, рівень простежуваності вимог;
- для оцінювання якості проєктування — наявність архітектурної моделі, повнота компонентів, складність інтерфейсів, використання шаблонів проєктування;
- для оцінювання якості програмного коду — цикломатична складність, підтримуваність, зрозумілість програмного коду, повторне використання компонентів, якість документації;
- для оцінювання ефективності контролю якості — розподіл трудових ресурсів, виконання запланованих термінів, результативність технічних оглядів, ефективність тестування, кількість виявлених дефектів та витрати часу на їх усунення.

1.4 Міжнародні стандарти забезпечення якості програмного забезпечення

Ефективне забезпечення якості програмного забезпечення неможливе без використання міжнародних стандартів, які визначають вимоги до процесів

розроблення, оцінювання та супроводу програмних систем. Застосування міжнародних стандартів дозволяє уніфікувати процеси розроблення, підвищити якість програмних продуктів, забезпечити їх відповідність вимогам замовників, а також сприяє впровадженню сучасних практик програмної інженерії.

Одним із найбільш поширених стандартів є ISO 9001, який встановлює вимоги до системи управління якістю організації. Стандарт визначає принципи побудови процесів управління, орієнтацію на потреби замовника, постійне вдосконалення діяльності організації, управління ризиками та прийняття рішень на основі об'єктивних даних. Незважаючи на те, що ISO 9001 не є спеціалізованим стандартом для програмної інженерії, його вимоги широко застосовуються під час організації процесів розроблення програмного забезпечення.

Важливе місце у програмній інженерії займає стандарт ISO/IEC 25010, який визначає модель якості програмного забезпечення та якості під час його використання. Відповідно до цього стандарту якість програмного продукту оцінюється за такими характеристиками: функціональна придатність, продуктивність, сумісність, зручність використання, надійність, безпека, супроводжуваність і переносимість. Використання моделі ISO/IEC 25010 дає можливість комплексно оцінити програмний продукт та визначити напрями його вдосконалення.

Для регламентування процесів створення програмного забезпечення широко використовується стандарт ISO/IEC/IEEE 12207, який визначає структуру життєвого циклу програмного забезпечення. Стандарт описує основні, допоміжні та організаційні процеси, що виконуються під час розроблення, експлуатації, супроводу та виведення програмного забезпечення з експлуатації. Дотримання вимог ISO/IEC/IEEE 12207 забезпечує системність виконання робіт та сприяє підвищенню якості кінцевого програмного продукту.

Важливим стандартом у сфері забезпечення якості програмного забезпечення є IEEE 730, який встановлює вимоги до розроблення плану забезпечення якості програмного забезпечення (Software Quality Assurance Plan). Стандарт визначає структуру документа, порядок планування заходів із забезпечення якості, проведення аудитів, технічних оглядів, контролю змін, управління документацією та оцінювання результатів діяльності із забезпечення якості.

Комплексне використання міжнародних стандартів дозволяє забезпечити єдиний підхід до організації процесів розроблення програмного забезпечення, підвищити рівень керованості проєктів, зменшити кількість дефектів, забезпечити відповідність програмних продуктів вимогам замовників та сучасним міжнародним практикам програмної інженерії. Саме тому міжнародні стандарти є основою сучасних систем забезпечення якості програмного забезпечення та широко застосовуються під час реалізації програмних проєктів різної складності.

Висновки

Забезпечення якості програмного забезпечення є одним із ключових напрямів сучасної програмної інженерії, що базується на системному підході до планування, контролю та постійного вдосконалення процесів розроблення програмних систем. Ефективна система забезпечення якості передбачає комплексне застосування організаційних, технічних і методичних заходів, спрямованих на запобігання виникненню дефектів, своєчасне їх виявлення та підвищення надійності програмного забезпечення. Важливу роль у забезпеченні якості відіграють спеціалізовані процеси Software Quality Assurance, використання програмних метрик для кількісного оцінювання характеристик програмного продукту, а також дотримання міжнародних стандартів, що регламентують процеси розроблення, оцінювання та супроводу програмного забезпечення. Комплексне застосування зазначених підходів забезпечує створення якісних, надійних, безпечних і супроводжуваних програмних систем, що відповідають сучасним вимогам програмної інженерії та потребам користувачів.

Лекція 2

Тема: Модульність у програмному забезпеченні. Принципи побудови модульних програмних систем

Мета: Сформувати у студентів цілісне розуміння принципів модульної побудови програмного забезпечення, ознайомити з основними властивостями програмних модулів, принципами декомпозиції програмних систем, поняттями зв'язності та зв'язування модулів, а також показати вплив модульності на якість, супроводжуваність, повторне використання та розвиток програмного забезпечення.

План лекції

1. Поняття модульності та її роль у програмній інженерії.
2. Критерії модульності.
3. Правила модульності.
4. Принципи модульності.
5. Висновки.

2.1 Поняття модульності

Необхідність використання модульності випливає з прагнення забезпечити розширюваність і повторне використання програмного забезпечення, які належать до основних характеристик його якості. Саме тому сучасні програмні системи будуються на основі гнучких архітектур, що складаються з автономних програмних компонентів. Поняття модульності охоплює поєднання цих двох характеристик якості та є одним із фундаментальних принципів конструювання програмного забезпечення.

Традиційно модульне програмування розглядалося як побудова програм із невеликих частин, найчастіше підпрограм. Проте такий підхід не забезпечує реальних переваг щодо розширюваності та повторного використання програмного забезпечення, якщо відсутній механізм, який гарантує, що створені модулі є достатньо автономними та організованими у стабільну архітектуру. Будь-яке повноцінне визначення модульності повинно враховувати саме ці властивості.

Метод конструювання програмного забезпечення можна вважати модульним тоді, коли він дозволяє створювати програмні системи, що складаються з автономних елементів, поєднаних між собою простою, логічною та узгодженою структурою. Такий підхід застосовується не лише на етапі проєктування, а й під час аналізу, специфікації, реалізації та супроводу програмного забезпечення.

Поняття модульності не може бути повністю визначене одним формулюванням. Подібно до поняття якості програмного забезпечення, модульність необхідно розглядати з кількох взаємодоповнювальних позицій. Для цього використовуються критерії, правила та принципи модульності, які в сукупності визначають основні вимоги до методів модульного проєктування програмного забезпечення.

Критерії модульності є незалежними один від одного, тому певний метод може задовольняти одні критерії та не відповідати іншим. Правила модульності логічно випливають із критеріїв, а принципи — із сформульованих правил, утворюючи цілісну систему вимог до побудови модульних програмних систем.

Під час розгляду питань модульності поняття модуля використовується як базова одиниця декомпозиції програмної системи незалежно від конкретної форми його реалізації. У різних методах програмування такими модулями можуть бути підпрограми, пакети або інші програмні конструкції. У сучасному об'єктно-орієнтованому програмуванні основною формою програмного модуля є клас, який найбільш повно реалізує вимоги модульного конструювання програмного забезпечення.

2.2 Критерії модульності

Метод проєктування можна вважати модульним лише за умови, що він відповідає сукупності фундаментальних вимог, які визначають ефективність модульної побудови програмного забезпечення. До основних критеріїв модульності належать:

- декомпозованість (Decomposability);
- компонованість (Composability);
- зрозумілість (Understandability);
- безперервність (Continuity);
- захищеність (Protection).

Ефективність модульного проєктування програмного забезпечення визначається не лише поділом системи на окремі програмні модулі, а й властивостями, якими повинні характеризуватися ці модулі та їх взаємодія. Для оцінювання якості модульної побудови програмних систем використовуються критерії модульності, які визначають основні вимоги до методів проєктування програмного забезпечення. Метод проєктування може вважатися модульним лише за умови, що він забезпечує ефективну декомпозицію системи, можливість повторного використання окремих модулів, простоту їх розуміння, стійкість архітектури до внесення змін та локалізацію наслідків виникнення помилок. До основних критеріїв модульності належать декомпозованість, компонованість, зрозумілість, безперервність та захищеність. Кожен із цих критеріїв характеризує окремий аспект модульного проєктування, а їх сукупність формує основу побудови гнучких, надійних та супроводжуваних програмних систем.

2.2.1 Декомпозованість (Decomposability)

Критерій декомпозованості визначає здатність методу проєктування розділяти складну програмну задачу на невелику кількість простіших підзадач, пов'язаних між собою простою структурою та достатньо незалежних одна від одної. У процесі

проєктування декомпозиція може повторюватися багаторазово, оскільки кожна отримана підзадача також може виявитися достатньо складною і потребувати подальшого поділу. Такий підхід дозволяє зменшити складність розроблення програмного забезпечення, полегшити аналіз системи та забезпечити можливість незалежного розроблення окремих її компонентів. Одним із важливих наслідків декомпозиції є можливість розподілу роботи між різними виконавцями або командами розробників. При цьому залежності між підсистемами повинні бути мінімальними та чітко визначеними, оскільки надмірна взаємозалежність ускладнює паралельне розроблення, а невраховані зв'язки можуть призвести до появи компонентів, які коректно працюють окремо, але не утворюють єдиної працездатної програмної системи. Класичним прикладом реалізації цього критерію є метод проєктування «згори донизу» (Top-Down Design), відповідно до якого система послідовно деталізується від найвищого рівня абстракції до окремих програмних модулів.

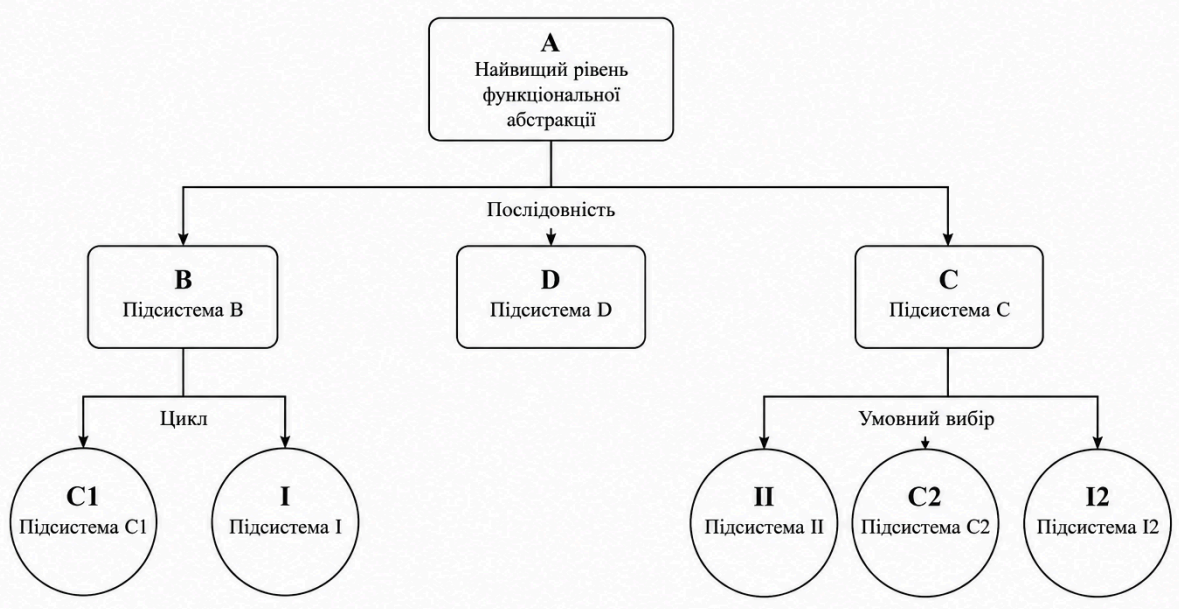


Рисунок 2.1 – Ієрархія декомпозиції програмної системи (Top-Down Design)

2.2.2 Компонованість (Composability)

Якщо декомпозованість характеризує процес поділу системи на окремі модулі, то компонованість описує зворотний процес — можливість повторного використання вже існуючих програмних компонентів під час створення нових програмних систем. Метод проєктування повинен забезпечувати розроблення програмних модулів, які є достатньо автономними та не залежать від конкретної задачі, для якої вони були створені. Це дозволяє використовувати їх у різних програмних продуктах без істотних модифікацій. Компонованість безпосередньо пов'язана з повторним використанням програмного забезпечення, оскільки саме вона забезпечує можливість створення нових програмних систем шляхом комбінування готових компонентів. Прикладами

реалізації цього критерію є бібліотеки підпрограм та програмні засоби операційної системи UNIX, які можуть вільно поєднуватися між собою завдяки єдиним правилам взаємодії. Водночас модулі, створені виключно для виконання вузькоспеціалізованих функцій без урахування можливості повторного використання, як правило, не відповідають вимогам компонованості.

2.2.3 Зрозумілість (Understandability)

Однією з найважливіших вимог до модульної архітектури є можливість зрозуміти роботу кожного окремого модуля без необхідності детального аналізу всієї програмної системи. Це особливо важливо під час супроводу програмного забезпечення, коли виникає потреба аналізувати або модифікувати вже існуючі програмні компоненти. Метод проєктування можна вважати таким, що забезпечує зрозумілість, якщо кожен модуль може бути досліджений окремо або із залученням лише невеликої кількості пов'язаних модулів. Надмірна кількість залежностей між компонентами значно ускладнює аналіз системи та збільшує витрати часу на її супровід. З цієї причини модулі повинні мати чітко визначене призначення, зрозумілий інтерфейс і мінімальну кількість зовнішніх залежностей.

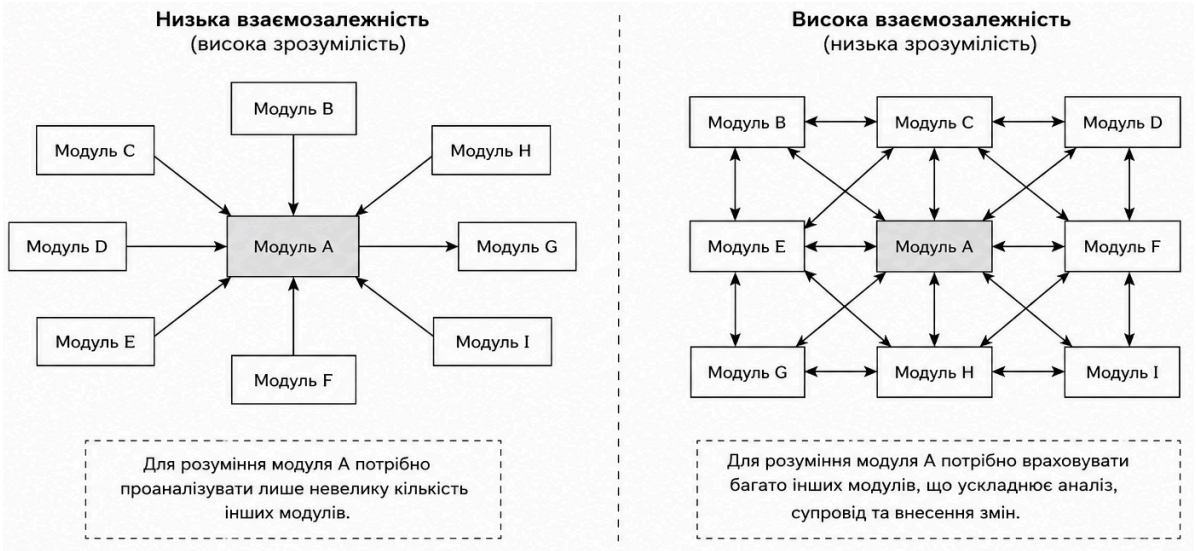


Рисунок 2.2 – Взаємозалежність програмних модулів та її вплив на зрозумілість системи

2.2.4 Безперервність (Continuity)

Під час розроблення програмного забезпечення зміни вимог є практично неминучими, тому архітектура програмної системи повинна бути стійкою до їх внесення. Критерій безперервності передбачає, що незначна зміна специфікації повинна спричиняти зміну лише одного або невеликої кількості програмних модулів без необхідності перебудови всієї системи. Такий підхід забезпечує простоту розвитку програмного забезпечення, зменшує витрати на його модернізацію та знижує ризик появи нових помилок. Досягненню безперервності сприяють використання

символічних констант, абстрагування від фізичного представлення даних, застосування єдиних принципів доступу до компонентів та інші прийоми модульного проєктування, що забезпечують локалізацію змін у межах окремих модулів.

2.2.5 Захищеність (Protection)

Критерій захищеності визначає здатність модульної архітектури локалізувати наслідки виникнення помилок під час виконання програмного забезпечення. У разі виникнення відмови або аварійної ситуації її вплив повинен обмежуватися одним програмним модулем або невеликою кількістю безпосередньо пов'язаних модулів, не порушуючи роботу всієї системи. Досягнення цього критерію забезпечується чітким розподілом відповідальності між модулями, перевіркою коректності вхідних даних безпосередньо в місці їх отримання, контрольованою взаємодією між компонентами та використанням механізмів оброблення виняткових ситуацій. Такий підхід дозволяє підвищити надійність програмного забезпечення, спростити пошук причин помилок та зменшити ризик поширення їх наслідків на інші компоненти програмної системи.

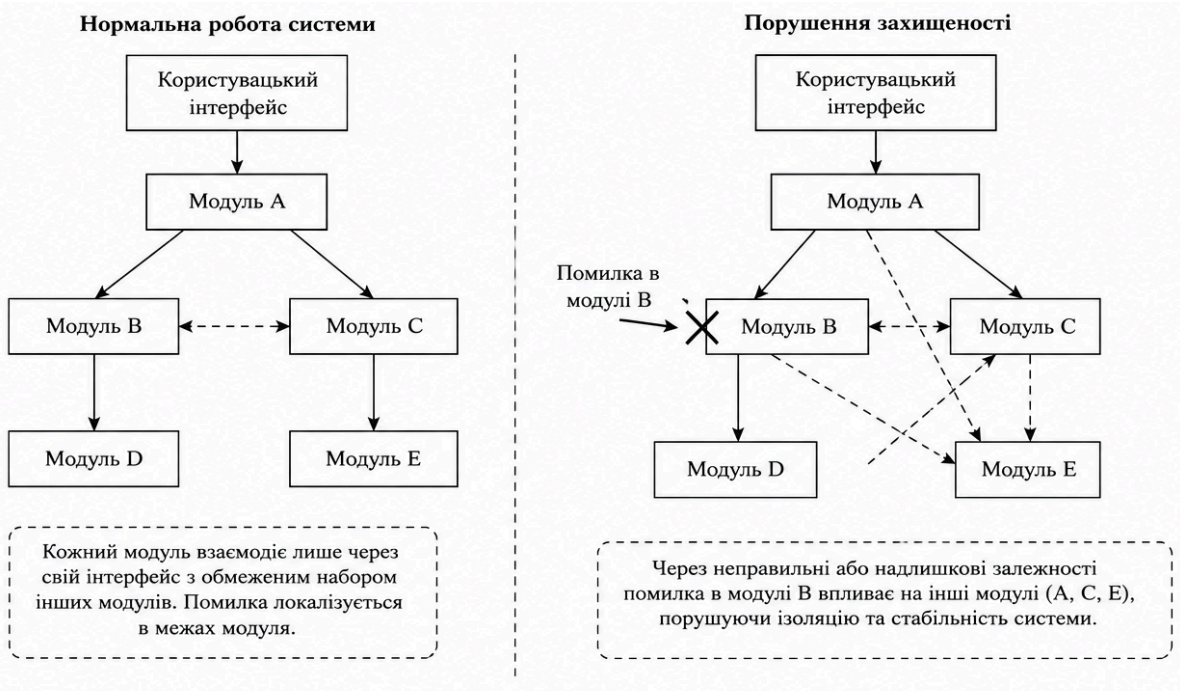


Рисунок 2.3 – Локалізація поширення помилок у модульній архітектурі

2.3 Правила модульності

Критерії модульності визначають основні вимоги, яким повинен відповідати метод проєктування програмного забезпечення. На їх основі формуються правила модульності, дотримання яких забезпечує побудову програмних систем із простою, зрозумілою та ефективною архітектурою. На відміну від критеріїв, які є незалежними один від одного, правила безпосередньо випливають із них і визначають практичні рекомендації щодо організації взаємодії програмних модулів. Основна увага при цьому приділяється способам обміну інформацією між модулями та організації їх

взаємодії. Правильна модульна архітектура передбачає, що взаємодія між програмними компонентами повинна бути контрольованою, обмеженою та зрозумілою. До основних правил модульності належать правило прямого відображення (Direct Mapping), правило мінімальної кількості інтерфейсів (Few Interfaces), правило малих інтерфейсів або слабкого зв'язування (Small Interfaces, Weak Coupling), правило явних інтерфейсів (Explicit Interfaces) та правило приховування інформації (Information Hiding).

2.3.1 Правило прямого відображення (Direct Mapping)

Будь-яка програмна система створюється для розв'язання задач певної предметної області. Якщо структура предметної області вже описана за допомогою моделі, то структура програмної системи повинна максимально відповідати цій моделі. Інакше кажучи, модульна структура програмного забезпечення має відображати модульну структуру задачі, яку вона реалізує. Такий підхід значно спрощує розроблення, супровід та розвиток програмного забезпечення, оскільки зміни у предметній області легко співвіднести зі змінами у відповідних програмних модулях. Крім того, результати аналізу предметної області можуть бути безпосередньо використані як основа для декомпозиції програмної системи.

2.3.2 Правило мінімальної кількості інтерфейсів (Few Interfaces)

Одним із найважливіших правил модульності є обмеження кількості взаємозв'язків між програмними модулями. Кожний модуль повинен взаємодіяти лише з мінімально необхідною кількістю інших модулів. Надмірна кількість зв'язків ускладнює архітектуру системи, збільшує залежність між компонентами та підвищує ризик поширення помилок або наслідків змін на значну частину програмної системи. Невелика кількість взаємозв'язків, навпаки, сприяє повторному використанню модулів, покращує їх зрозумілість і забезпечує простіший супровід програмного забезпечення.

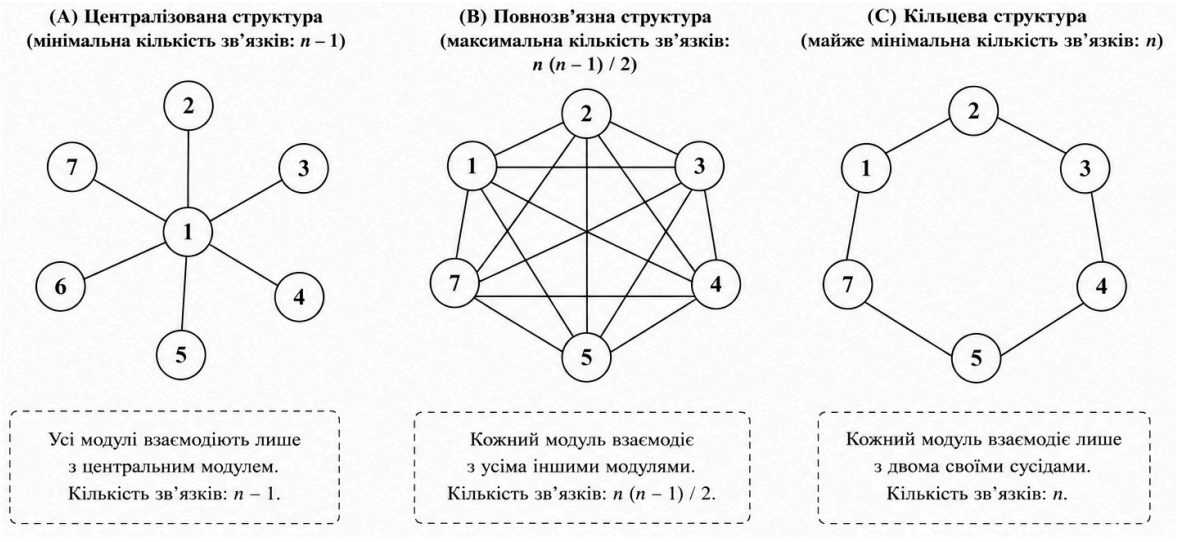


Рисунок 2.4 – Варіанти структур взаємозв'язків між програмними модулями

2.3.3 Правило малих інтерфейсів (Small Interfaces, Weak Coupling)

Окрім кількості зв'язків між модулями, важливе значення має також обсяг інформації, якою вони обмінюються. Якщо два модулі взаємодіють між собою, вони повинні передавати лише мінімально необхідний набір даних. Надлишковий обмін інформацією призводить до сильного зв'язування компонентів, унаслідок чого будь-які зміни в одному модулі можуть вимагати модифікації інших модулів. Зменшення обсягу інформації, що передається через інтерфейси, підвищує незалежність компонентів, покращує їх повторне використання та спрощує модернізацію програмного забезпечення.

2.3.4 Правило явних інтерфейсів (Explicit Interfaces)

Усі взаємозв'язки між програмними модулями повинні бути чітко визначені та очевидні для розробників. Будь-яка взаємодія між двома модулями повинна бути безпосередньо відображена у програмному коді або проєктній документації. Особливо важливо уникати прихованих залежностей, які виникають через використання спільних структур даних або глобальних змінних. Такі приховані зв'язки значно ускладнюють аналіз системи, підвищують ризик появи помилок та негативно впливають на можливість незалежного розвитку окремих програмних компонентів.

2.3.5 Правило приховування інформації (Information Hiding)

Одним із фундаментальних правил сучасної програмної інженерії є приховування внутрішньої реалізації програмного модуля від інших компонентів системи. Кожний модуль повинен надавати лише офіційно визначений інтерфейс, який описує його функціональні можливості, тоді як усі деталі реалізації залишаються прихованими. Такий підхід забезпечує незалежність модулів один від одного, дозволяє змінювати внутрішню реалізацію без впливу на клієнтські компоненти та значно спрощує супровід і розвиток програмного забезпечення. Основною метою приховування інформації є відокремлення специфікації програмного модуля від способу її реалізації, що забезпечує високу гнучкість архітектури програмної системи та мінімізує вплив внутрішніх змін на інші компоненти.

2.4 Принципи модульності

Критерії та правила модульності визначають загальні вимоги до побудови програмних систем, однак їх практична реалізація забезпечується дотриманням принципів модульного проєктування. Принципи модульності встановлюють рекомендації щодо організації програмних модулів, їх документування, способів взаємодії та розвитку програмного забезпечення. Їх застосування сприяє створенню програмних систем, які легко розширюються, підтримуються та повторно

використовуються. До основних принципів модульності належать принцип мовних модульних одиниць, принцип самодокументування, принцип уніфікованого доступу, принцип відкритості-закритості та принцип єдиного вибору.

2.4.1 Принцип мовних модульних одиниць (Linguistic Modular Units Principle)

Програмні модулі повинні відповідати синтаксичним конструкціям мови, яка використовується під час аналізу, проєктування або реалізації програмного забезпечення. Це означає, що кожен модуль має бути окремою структурною одиницею, яка підтримується відповідною мовою програмування або мовою специфікації. Використання методологій, які не підтримуються мовними засобами, призводить до необхідності ручного перетворення моделей у програмний код, що значно ускладнює розроблення та супровід програмного забезпечення. Крім того, окремі модулі повинні мати можливість незалежної компіляції та тестування, що забезпечує можливість паралельної роботи різних розробників над окремими компонентами системи.

2.4.2 Принцип самодокументування (Self-Documentation Principle)

Документація програмного модуля повинна бути невід'ємною частиною самого модуля, а не існувати у вигляді окремих документів. Такий підхід забезпечує постійну узгодженість між програмним кодом і його описом, оскільки будь-які зміни автоматично супроводжуються оновленням документації. Зберігання документації окремо від програмного коду часто призводить до втрати їх відповідності, що ускладнює супровід програмного забезпечення. Самодокументування передбачає використання зрозумілих назв класів, методів і змінних, а також опису призначення модулів безпосередньо в їх реалізації. Таким чином програмний модуль стає єдиним джерелом інформації про власне призначення, структуру та функціональність.

2.4.3 Принцип уніфікованого доступу (Uniform Access Principle)

Усі сервіси або функції програмного модуля повинні використовуватися однаковим способом незалежно від особливостей їх внутрішньої реалізації. Користувач програмного модуля не повинен знати, чи отримується певне значення безпосередньо із поля об'єкта, чи воно обчислюється за допомогою відповідного методу. Єдина форма доступу дозволяє змінювати внутрішню реалізацію без необхідності модифікації клієнтських модулів, що підвищує гнучкість архітектури програмного забезпечення та забезпечує локалізацію змін у межах одного компонента системи.

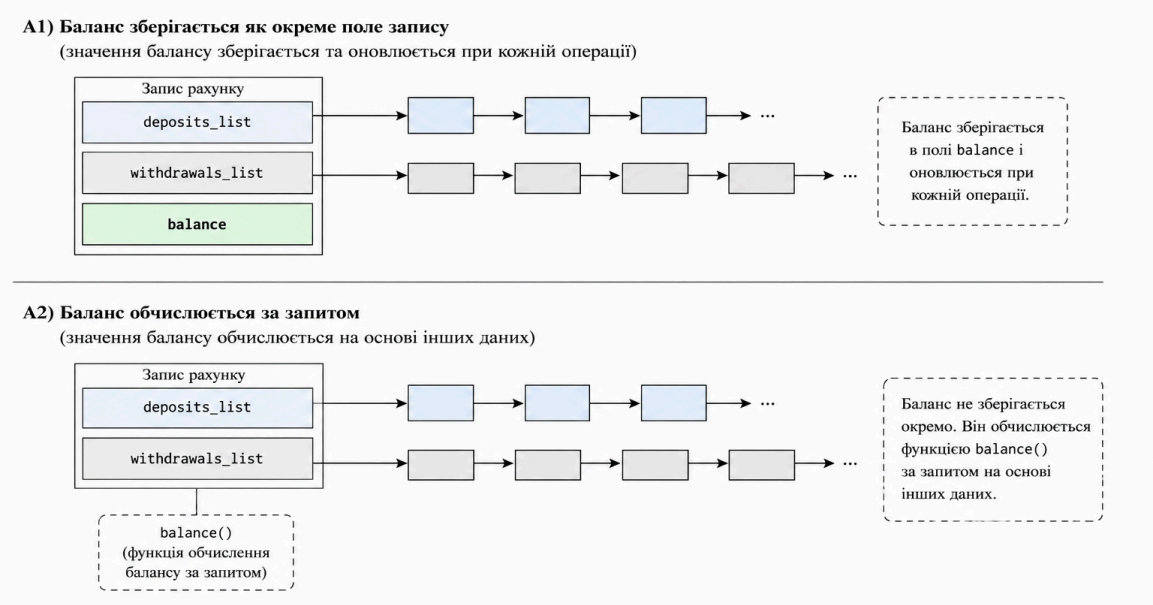


Рисунок 2.5 – Два способи реалізації доступу до балансу банківського рахунку

2.4.4 Принцип відкритості-закритості (Open-Closed Principle)

Програмний модуль повинен одночасно бути відкритим для розширення та закритим для модифікації. Відкритість означає можливість додавання нової функціональності без зміни вже існуючого програмного коду, тоді як закритість передбачає стабільність інтерфейсу та можливість безпечного використання модуля іншими компонентами системи. Такий підхід дозволяє уникнути ситуацій, коли зміни в одному модулі викликають необхідність модифікації значної кількості інших модулів. В об'єктно-орієнтованому програмуванні реалізація цього принципу забезпечується насамперед механізмом успадкування, який дозволяє створювати нові класи на основі вже існуючих без зміни їх початкової реалізації.

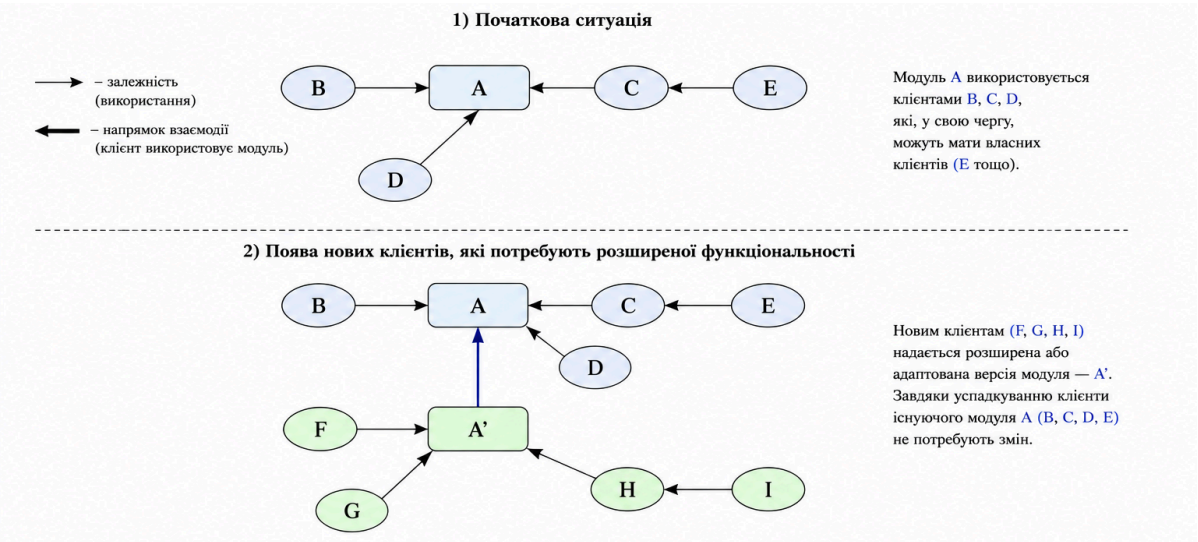


Рисунок 2.6 – Реалізація принципу відкритості-закритості за допомогою успадкування

2.4.5 Принцип єдиного вибору (Single Choice Principle)

Якщо програмна система підтримує декілька альтернатив або варіантів реалізації певної сутності, повний перелік таких альтернатив повинен бути відомий лише одному програмному модулю. Решта компонентів системи повинні працювати через визначений інтерфейс, не залежачи від кількості чи типів можливих варіантів. Такий підхід значно спрощує розвиток програмного забезпечення, оскільки додавання нового варіанта потребує внесення змін лише до одного модуля без необхідності модифікації всіх його клієнтів. Принцип єдиного вибору тісно пов'язаний із принципами приховування інформації та відкритості-закритості, забезпечуючи мінімізацію залежностей між компонентами програмної системи та підвищуючи її розширюваність.

Висновки

Модульність є одним із фундаментальних принципів сучасної програмної інженерії, який забезпечує ефективну побудову, розвиток і супровід програмних систем. Застосування критеріїв модульності дає змогу оцінити якість архітектури програмного забезпечення, а правила модульності визначають практичні підходи до організації взаємодії між програмними компонентами. Дотримання принципів модульності сприяє створенню програмних систем із низьким рівнем зв'язності, високою внутрішньою цілісністю модулів, можливістю повторного використання програмних компонентів та локалізацією змін під час розвитку програмного забезпечення. Сукупне використання критеріїв, правил і принципів модульного проєктування створює основу для розроблення якісних, надійних, масштабованих і зручних у супроводі програмних систем, що відповідають сучасним вимогам програмної інженерії.

Лекція 3

Тема: Підходи до повторного використання

Мета: Сформувати у студентів цілісне уявлення про сучасні підходи до повторного використання програмного забезпечення, ознайомити з основними механізмами повторного використання програмних компонентів, бібліотек, фреймворків, шаблонів проєктування та компонентної архітектури, а також показати їх вплив на якість, масштабованість і супровід програмних систем.

План лекції

1. Поняття повторного використання програмного забезпечення.
2. Повторне використання програмних бібліотек і фреймворків.
3. Повторне використання за допомогою шаблонів проєктування:
4. Компонентна архітектура та компонентне програмування:
5. Переваги та обмеження повторного використання програмного забезпечення.
6. Висновки

3.1. Поняття повторного використання програмного забезпечення

Повторне використання програмного забезпечення (Software Reuse) є одним із ключових напрямів сучасної програмної інженерії, який передбачає використання вже існуючих програмних компонентів, модулів, бібліотек, сервісів або навіть цілих програмних систем під час створення нових програмних продуктів. Основною ідеєю такого підходу є не повторна розробка однакової функціональності, а використання вже перевірених рішень, що дозволяє скоротити витрати часу, зменшити вартість розроблення, підвищити якість програмного забезпечення та прискорити його впровадження.

До початку 2000-х років систематичне повторне використання програмного забезпечення застосовувалося досить рідко. Проте зі зростанням складності інформаційних систем, розвитком відкритого програмного забезпечення (Open Source), появою великої кількості програмних бібліотек, компонентів, сервісів та корпоративних платформ повторне використання стало одним із базових принципів сучасної розробки програмного забезпечення. Сьогодні практично кожен програмний проєкт використовує певну кількість готових компонентів або зовнішніх програмних продуктів.

Повторне використання програмного забезпечення розглядається як стратегія розроблення, у якій процес створення нової системи максимально орієнтований на використання вже існуючих програмних рішень. При цьому повторно використовувані елементи можуть мати різний рівень деталізації — від окремих функцій і класів до повністю готових інформаційних систем.

У програмній інженерії виділяють кілька основних рівнів повторного використання програмного забезпечення:

- повторне використання програмних систем — використання готових інформаційних систем або їх окремих підсистем як складових нової програмної системи;
- повторне використання застосунків — використання готового програмного застосунку без змін або після його налаштування відповідно до вимог конкретного замовника;
- повторне використання компонентів — використання окремих модулів, підсистем або сервісів, які інтегруються до нового програмного продукту;
- повторне використання об'єктів і функцій — використання класів, бібліотек, алгоритмів або окремих функцій, що реалізують певну функціональність.

Окрім повторного використання готового програмного коду, у програмній інженерії широко застосовується повторне використання концепцій (Concept Reuse). У цьому випадку повторно використовуються не програмні компоненти, а архітектурні рішення, алгоритми, моделі, шаблони проєктування або підходи до розв'язання типових задач. Такі концепції можуть бути адаптовані до різних програмних систем незалежно від конкретної реалізації.

Найбільш очевидною перевагою повторного використання є зменшення витрат на розроблення програмного забезпечення, оскільки значна частина функціональності вже створена та протестована. Однак економія коштів є лише одним із позитивних наслідків застосування повторного використання. Використання перевірених програмних компонентів також сприяє скороченню часу розроблення, підвищенню надійності програмного забезпечення, ефективнішому використанню досвіду спеціалістів, зменшенню ризиків реалізації проєкту та забезпеченню дотримання стандартів програмної розробки.

Таблиця 3.1 – Переваги повторного використання програмного забезпечення

Перевага	**Опис**
Прискорення розроблення	Повторне використання програмного забезпечення дозволяє скоротити час створення нової системи, оскільки значна частина функціональності вже реалізована та перевірена. Це особливо важливо, коли швидке виведення продукту на ринок є пріоритетом.
Ефективне використання знань фахівців	Замість багаторазового створення однакових компонентів спеціалісти можуть розробляти універсальні програмні модулі, які накопичують професійний досвід і можуть використовуватися у багатьох проєктах.
Підвищення надійності	Повторно використовувані компоненти вже були

	протестовані в реальних програмних системах, тому більшість помилок їх реалізації вже виявлено та усунено.
Зменшення вартості розроблення	Використання готових компонентів дозволяє скоротити обсяг нового програмного коду, що безпосередньо зменшує витрати на розроблення програмного забезпечення.
Зниження ризиків проєкту	Вартість і характеристики готових програмних компонентів вже відомі, що дозволяє точніше оцінювати ресурси та строки виконання проєкту і зменшує ризики планування.
Відповідність стандартам	Повторно використовувані компоненти можуть реалізовувати вимоги галузевих стандартів, забезпечуючи однаковий стиль взаємодії користувача із системою та зменшуючи кількість помилок під час її використання.

Разом із перевагами повторне використання має і певні недоліки. Основними проблемами є складність пошуку відповідних компонентів, необхідність їх детального аналізу та адаптації, витрати на підтримку бібліотек компонентів, можливе збільшення вартості супроводу програмного забезпечення, недостатня підтримка окремих інструментів розробки, а також так званий синдром «Not Invented Here», коли розробники віддають перевагу створенню власних компонентів замість використання вже існуючих.

Таблиця 3.2 – Проблеми повторного використання програмного забезпечення

Проблема	**Опис**
Створення та підтримка бібліотеки компонентів	Формування бібліотеки повторно використовуваних компонентів, її супровід та забезпечення зручного доступу для розробників потребують значних організаційних і фінансових витрат.
Пошук, аналіз та адаптація компонентів	Перед використанням компонент необхідно знайти, оцінити його відповідність поставленій задачі, зрозуміти принципи роботи та, за необхідності, адаптувати до нового середовища використання.
Зростання витрат на супровід	Якщо вихідний код повторно використовуваного компонента недоступний, його оновлення або сумісність із новими версіями програмного забезпечення можуть ускладнювати супровід системи.
Недостатня підтримка інструментальних засобів	Деякі середовища розроблення та програмні інструменти не забезпечують належної підтримки процесів повторного використання компонентів або інтеграції з їх бібліотеками.

Синдром «Not Invented Here» («Не створено тут»)	Частина розробників віддає перевагу створенню власних програмних компонентів замість використання вже існуючих, вважаючи їх менш якісними або бажаючи реалізувати власне рішення.

3.2 Повторне використання програмних бібліотек і фреймворків

Одним із найбільш поширених способів повторного використання програмного забезпечення є застосування програмних бібліотек і фреймворків. Вони дозволяють використовувати вже реалізовані програмні компоненти для виконання типових задач, що значно скорочує обсяг нового програмного коду, підвищує якість програмного забезпечення та прискорює процес його розроблення. Використання готових програмних компонентів є стандартною практикою сучасної програмної інженерії, оскільки більшість програмних систем створюється не «з нуля», а шляхом інтеграції вже існуючих рішень.

Програмна бібліотека (Software Library) — це впорядкований набір функцій, класів або програмних модулів, які реалізують певну функціональність і можуть використовуватися в різних програмних проєктах. Бібліотеки забезпечують багаторазове використання вже розробленого програмного коду без необхідності його повторної реалізації. Зазвичай бібліотека виконує окремі спеціалізовані функції, наприклад роботу з файловою системою, математичні обчислення, оброблення графічної інформації, мережеву взаємодію або доступ до баз даних.

Основними перевагами використання програмних бібліотек є скорочення часу розроблення програмного забезпечення, підвищення надійності програмних компонентів завдяки їх попередньому тестуванню, зменшення кількості помилок у програмному коді та можливість використання перевірених алгоритмів і структур даних. Крім того, застосування бібліотек забезпечує стандартизацію процесу розроблення та спрощує супровід програмних систем.

На відміну від бібліотеки, **фреймворк (Application Framework)** являє собою програмну платформу, яка визначає загальну архітектуру програмного застосунку та надає готовий набір компонентів для реалізації його функціональності. Якщо бібліотека використовується за ініціативою розробника лише тоді, коли виникає потреба у виконанні певної операції, то у випадку фреймворку саме він визначає структуру програми та керує процесом її виконання. Такий підхід отримав назву інверсії **керування (Inversion of Control)**, оскільки виклик користувацького коду здійснюється самим фреймворком.

Фреймворки містять готову архітектуру програмного забезпечення, яка може бути адаптована до конкретної предметної області шляхом реалізації окремих компонентів або розширення базової функціональності. Завдяки цьому розробники можуть зосередитися на реалізації специфічної бізнес-логіки, не витрачаючи час на

створення типових механізмів взаємодії між компонентами системи. Найбільш поширеними прикладами є веб-фреймворки, фреймворки для мобільних застосунків, графічних інтерфейсів користувача, роботи з базами даних та корпоративних інформаційних систем.

Використання фреймворків забезпечує високу швидкість розроблення, єдині принципи побудови програмного забезпечення, повторне використання архітектурних рішень та спрощує супровід програмних продуктів. Водночас використання фреймворків може вимагати від розробників додаткового часу на їх вивчення, а також накладати певні обмеження щодо структури програмного застосунку та способів реалізації окремих компонентів.

Таким чином, програмні бібліотеки та фреймворки є важливими засобами повторного використання програмного забезпечення. Якщо бібліотеки забезпечують повторне використання окремих функцій або програмних модулів, то фреймворки дозволяють повторно використовувати вже сформовану архітектуру програмної системи, що значно підвищує ефективність розроблення сучасного програмного забезпечення.

3.3. Повторне використання за допомогою шаблонів проєктування

3.3.1 Що таке патерн проєктування

Шаблони проєктування (Design Patterns) є одним із найбільш ефективних засобів повторного використання перевірених рішень у програмній інженерії. На відміну від програмних бібліотек або готових компонентів, шаблон проєктування не є готовою реалізацією програмного коду. Він описує типову проблему, що виникає під час проєктування програмного забезпечення, і пропонує загальний спосіб її розв'язання, який може бути використаний в різних програмних системах.

Крістофер Александер визначає шаблон як опис проблеми, що багаторазово виникає у певному середовищі, та основної ідеї її розв'язання таким чином, щоб це рішення можна було застосовувати необмежену кількість разів, кожного разу реалізовуючи його по-різному. Ця ідея була адаптована до програмної інженерії, де замість будівельних конструкцій використовуються об'єкти, класи та інтерфейси, однак основна концепція залишається незмінною — шаблон описує перевірене рішення типової задачі в певному контексті.

Кожний шаблон проєктування складається з чотирьох основних елементів:

1. **Назва шаблону (Pattern Name)** — коротка назва, яка дозволяє однозначно ідентифікувати шаблон. Наявність назви формує спільну термінологію серед розробників, полегшує обговорення архітектурних рішень, спрощує документування програмного забезпечення та дозволяє розглядати проєктування на більш високому рівні абстракції.

2. **Проблема (Problem)** описує ситуацію, у якій доцільно використовувати шаблон. Вона визначає контекст застосування, пояснює, які труднощі виникають під час проєктування, та за яких умов використання конкретного шаблону є виправданим. У деяких випадках також зазначаються передумови, які повинні бути виконані для його застосування.

3. **Рішення (Solution)** містить узагальнений опис структури майбутнього програмного рішення. Воно визначає основні елементи проєктування, їх взаємозв'язки, відповідальність і принципи взаємодії. При цьому шаблон не описує конкретну реалізацію, а виступає своєрідним проєктним шаблоном, який може бути адаптований до різних програмних систем.

4. **Наслідки (Consequences)** характеризують результати використання шаблону та компроміси, які виникають під час його застосування. Вони дозволяють оцінити переваги та недоліки запропонованого рішення, зрозуміти його вплив на гнучкість, розширюваність, повторне використання та супровід програмного забезпечення, а також допомагають порівнювати різні варіанти архітектурних рішень.

Слід зазначити, що поняття шаблону залежить від рівня абстракції, на якому розглядається програмна система. Те, що для одного розробника є повноцінним шаблоном, для іншого може бути лише окремим будівельним елементом програмної архітектури. Шаблони проєктування не описують окремі структури даних або алгоритми, які можна безпосередньо реалізувати у програмному коді, а також не визначають архітектуру цілої інформаційної системи. Вони описують типові способи взаємодії класів і об'єктів, які можуть бути адаптовані для розв'язання широкого кола задач проєктування.

Таким чином, шаблон проєктування являє собою узагальнений опис перевіреного архітектурного рішення, яке визначає основні елементи програмної структури, їх ролі, взаємозв'язки та розподіл відповідальності між ними. Застосування шаблонів дозволяє використовувати накопичений досвід проєктування програмного забезпечення, підвищувати якість архітектури програмних систем, спрощувати їх розвиток і супровід, а також забезпечувати повторне використання перевірених проєктних рішень.

3.3.2 Опис шаблонів проєктування

Для ефективного використання шаблонів проєктування недостатньо лише графічного представлення структури класів або об'єктів. Хоча діаграми є важливим засобом опису архітектури програмного забезпечення, вони відображають лише кінцевий результат процесу проєктування. Для повторного використання шаблону необхідно також описати прийняті проєктні рішення, можливі альтернативи та наслідки їх застосування. Практичні приклади використання шаблонів також відіграють важливу роль, оскільки дозволяють краще зрозуміти особливості їх застосування.

Для забезпечення єдиного підходу до опису шаблонів використовується стандартизована структура, яка полегшує їх вивчення, порівняння та практичне використання. Кожний шаблон проєктування містить такі основні складові.

Назва та класифікація шаблону (Pattern Name and Classification). Назва шаблону коротко відображає його призначення та дозволяє сформувати єдину термінологію під час проєктування програмного забезпечення. Класифікація визначає місце шаблону серед інших шаблонів проєктування.

Призначення (Intent). Містить короткий опис проблеми, яку розв'язує шаблон, його основної мети та сфери застосування.

Інші назви (Also Known As). Якщо шаблон має альтернативні назви, вони також наводяться для спрощення його пошуку в різних джерелах.

Мотивація (Motivation). Описує приклад типової ситуації, у якій виникає проблема проєктування, та демонструє спосіб її вирішення за допомогою шаблону.

Застосовність (Applicability). Визначає ситуації, у яких використання шаблону є доцільним, а також ознаки програмних систем, що свідчать про необхідність його застосування.

Структура (Structure). Містить графічне представлення взаємозв'язків між основними класами та об'єктами шаблону за допомогою діаграм класів і діаграм взаємодії.

Учасники (Participants). Наводить перелік класів або об'єктів, що входять до складу шаблону, а також визначає їх функції та відповідальність.

Взаємодія (Collaborations). Описує взаємодію між учасниками шаблону під час реалізації його функціональності.

Наслідки (Consequences). Визначає результати використання шаблону, його переваги, недоліки та компроміси, що виникають під час його застосування.

Реалізація (Implementation). Містить рекомендації щодо практичного впровадження шаблону, можливі труднощі реалізації та особливості використання в різних мовах програмування.

Приклад програмного коду (Sample Code). Наводить фрагменти програмного коду, які демонструють один із можливих варіантів реалізації шаблону.

Відомі приклади використання (Known Uses). Містить приклади застосування шаблону в реальних програмних системах.

Пов'язані шаблони (Related Patterns). Наводить шаблони, які мають схоже призначення або часто використовуються разом із розглянутим шаблоном, а також пояснює основні відмінності між ними.

Таким чином, стандартизований опис шаблонів проєктування забезпечує єдине представлення інформації про кожний шаблон, полегшує їх аналіз, порівняння та вибір під час проєктування програмного забезпечення.

3.3.3 Як шаблони проєктування допомагають вирішувати проблеми проєктування

Шаблони проєктування дозволяють розв'язувати значну кількість типових проблем, з якими щоденно стикаються розробники об'єктно-орієнтованого програмного забезпечення. Вони не лише пропонують перевірені способи побудови програмних систем, але й допомагають знаходити оптимальні архітектурні рішення, підвищувати гнучкість програмного забезпечення та забезпечувати можливість його повторного використання.

Існує багато способів класифікації шаблонів проєктування. Одним із них є аналіз взаємозв'язків між окремими шаблонами, які описуються в розділі «Related Patterns» для кожного шаблону. Графічне представлення таких взаємозв'язків дозволяє краще зрозуміти місце кожного шаблону серед інших і спрощує вибір найбільш доцільного рішення під час проєктування програмного забезпечення.

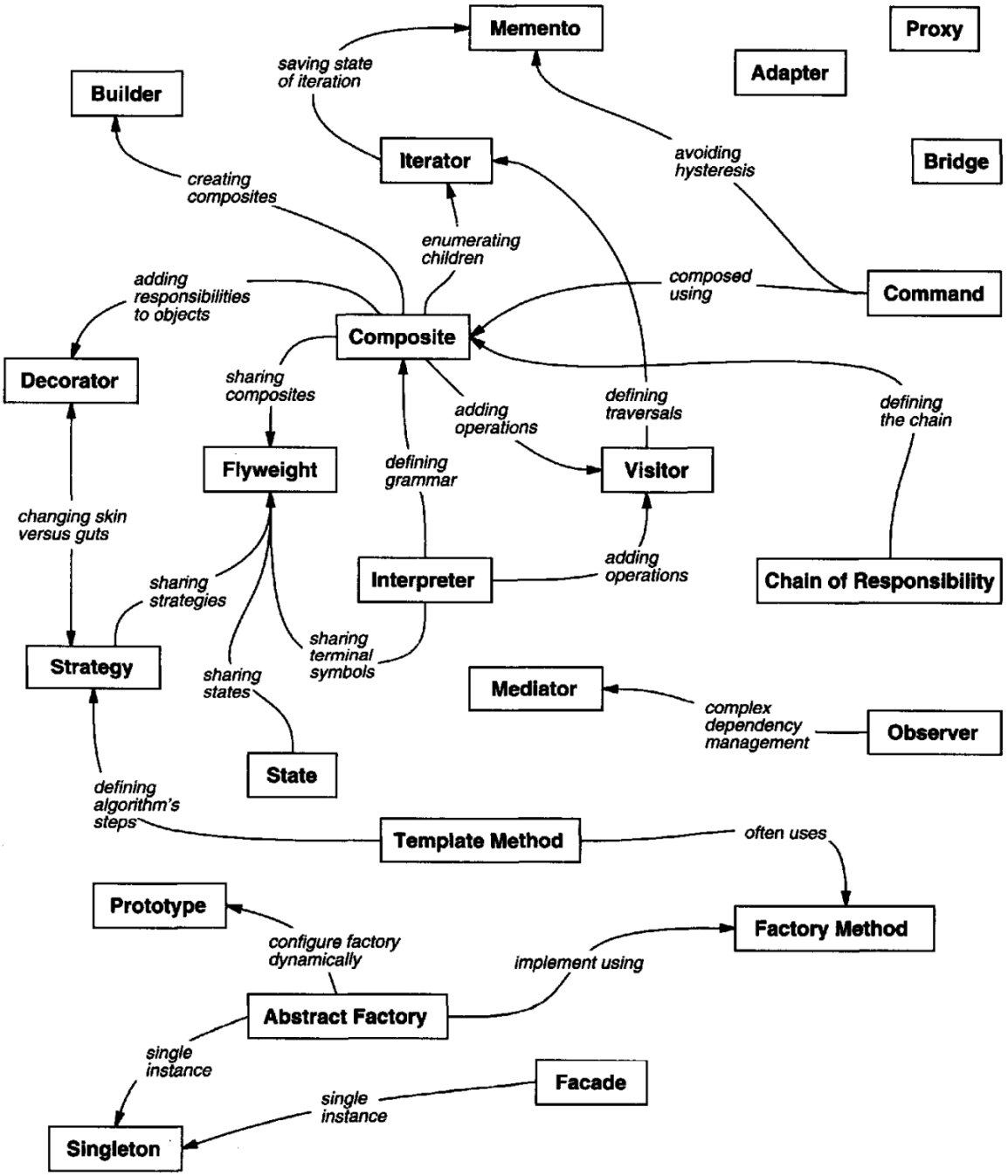


Рисунок 3.1 – Взаємозв'язки між шаблонами проєктування (Design Pattern Relationships).

Шаблони проєктування можуть розглядатися з різних точок зору. Використання декількох підходів до їх класифікації дозволяє глибше зрозуміти призначення шаблонів, відмінності між ними та особливості їх практичного застосування.

Однією з найважливіших задач об'єктно-орієнтованого проєктування є правильне визначення набору об'єктів, з яких складатиметься програмна система. Об'єкти поєднують дані та операції, що виконуються над цими даними. Виконання операцій здійснюється лише після отримання відповідного запиту від іншого об'єкта.

Такий підхід забезпечує інкапсуляцію, оскільки внутрішній стан об'єкта недоступний безпосередньо і може змінюватися лише через визначений інтерфейс.

Найскладнішим етапом об'єктно-орієнтованого проєктування є декомпозиція системи на окремі об'єкти. На цей процес впливають численні чинники, серед яких інкапсуляція, рівень деталізації, взаємозалежність компонентів, гнучкість архітектури, продуктивність, можливість розвитку програмної системи та повторне використання програмного забезпечення. Часто ці вимоги суперечать одна одній, тому вибір оптимальної структури системи є складним проєктним завданням.

Різні методології об'єктно-орієнтованого проєктування пропонують різні підходи до визначення об'єктів. Один із них ґрунтується на аналізі предметної області та виділенні основних іменників і дієслів, які перетворюються на класи та операції. Інший підхід акцентує увагу на відповідальності об'єктів і їх взаємодії. Також широко використовується моделювання об'єктів реального світу з подальшим перенесенням отриманої моделі у програмну систему. Універсального підходу не існує, і вибір конкретного способу залежить від особливостей програмного проєкту.

Більшість об'єктів програмної системи формується на основі моделі предметної області. Проте в процесі проєктування часто виникають класи, які не мають безпосередніх аналогів у реальному світі. До них належать як низькорівневі структури, так і абстрактні програмні компоненти, що забезпечують гнучкість архітектури. Саме такі абстракції часто стають основою повторного використання програмного забезпечення.

Шаблони проєктування допомагають виявляти подібні абстракції та визначати об'єкти, які доцільно використовувати для їх реалізації. Наприклад, окремі об'єкти можуть представляти алгоритми, стани або процеси, хоча вони не існують як фізичні об'єкти предметної області. Використання таких абстракцій забезпечує більш гнучку, розширювану та придатну до повторного використання архітектуру програмного забезпечення.

Наступною важливою проблемою є визначення рівня деталізації об'єктів. Об'єкти можуть представляти як окремі програмні компоненти, так і цілі підсистеми або навіть програмні застосунки. Вибір оптимального рівня деталізації безпосередньо впливає на складність архітектури, можливість її супроводу та повторного використання.

Шаблони проєктування також допомагають визначати інтерфейси об'єктів. Інтерфейс описує набір операцій, які може виконувати об'єкт, не розкриваючи деталей їх реалізації. Завдяки цьому різні класи можуть мати однаковий інтерфейс, але реалізовувати його різними способами. Такий підхід забезпечує поліморфізм, зменшує залежність між компонентами програмної системи та підвищує її гнучкість.

Крім цього, шаблони визначають принципи взаємодії між класами та об'єктами, допомагають відокремлювати інтерфейси від реалізації, зменшують зв'язність між

компонентами та спрощують модифікацію програмної системи. Вони також дозволяють проєктувати архітектуру таким чином, щоб окремі її частини могли змінюватися незалежно одна від одної.

Таким чином, шаблони проєктування є універсальним інструментом розв'язання типових задач об'єктно-орієнтованого проєктування. Вони допомагають визначати склад програмної системи, обирати оптимальний рівень деталізації об'єктів, формувати ефективні інтерфейси взаємодії між компонентами та створювати програмне забезпечення, яке є гнучким, розширюваним і придатним до повторного використання.

3.4. Компонентна архітектура та компонентне програмування

3.4.1 Поняття компонентного програмування

Компонентно-орієнтована програмна інженерія (Component-Based Software Engineering, CBSE) сформувалася наприкінці 1990-х років як підхід до розроблення програмного забезпечення, що ґрунтується на повторному використанні програмних компонентів. Її поява була зумовлена тим, що об'єктно-орієнтоване програмування не забезпечило очікуваного рівня повторного використання програмного коду. Окремі класи були надто деталізованими, залежали від конкретного застосунку та часто вимагали доступу до вихідного коду для їх використання. Це значно ускладнювало повторне використання та розповсюдження програмних компонентів.

На відміну від окремих об'єктів, програмні компоненти є більш високим рівнем абстракції. Вони визначаються своїми інтерфейсами, приховують деталі реалізації та можуть незалежно інтегруватися до інших програмних систем. Компонентно-орієнтована програмна інженерія передбачає визначення, реалізацію та композицію незалежних слабозв'язаних компонентів для побудови програмних систем.

Сьогодні компонентне програмування є одним із основних підходів до створення великих корпоративних програмних систем, для яких висуваються підвищені вимоги щодо продуктивності, надійності та інформаційної безпеки. Використання готових програмних компонентів дозволяє значно скоротити час розроблення, підвищити якість програмного забезпечення та прискорити його впровадження.

Основними складовими компонентно-орієнтованої програмної інженерії є:

- незалежні програмні компоненти, повністю визначені своїми інтерфейсами;
- стандарти компонентів, що забезпечують сумісність і взаємодію між компонентами;
- програмне проміжне забезпечення (middleware), яке підтримує взаємодію компонентів;

- процес розроблення, орієнтований на ефективне повторне використання програмних компонентів.

Однією з ключових особливостей компонентного програмування є чітке розділення інтерфейсу та реалізації компонента. Завдяки цьому реалізацію компонента можна змінювати або замінювати без необхідності внесення змін до інших частин програмної системи, якщо його інтерфейс залишається незмінним.

Компонентно-орієнтована програмна інженерія базується на кількох фундаментальних принципах. По-перше, компоненти є незалежними один від одного, а їх внутрішня реалізація прихована від інших компонентів. По-друге, взаємодія між компонентами здійснюється виключно через чітко визначені інтерфейси, що забезпечує можливість заміни одного компонента іншим без зміни архітектури системи. По-третє, компонентні платформи надають набір стандартних сервісів, які використовуються всіма компонентами, що дозволяє скоротити обсяг програмного коду, який необхідно реалізовувати під час створення нових програмних систем.

Таким чином, компонентне програмування є сучасним підходом до розроблення програмного забезпечення, який ґрунтується на повторному використанні незалежних програмних компонентів із чітко визначеними інтерфейсами. Такий підхід забезпечує гнучкість архітектури, спрощує супровід програмних систем, підвищує їхню надійність та дозволяє істотно скоротити витрати на розроблення програмного забезпечення.

Таблиця 3.3 – Основні характеристики програмного компонента

Характеристика компонента	**Опис**
Компонованість (Composable)	Для забезпечення можливості компонування всі зовнішні взаємодії компонента повинні здійснюватися через відкрито визначені інтерфейси. Крім того, компонент має надавати зовнішній доступ до інформації про себе, зокрема до своїх методів і атрибутів.
Розгортання (Deployable)	Компонент повинен бути самодостатнім, тобто здатним функціонувати як окрема програмна одиниця на компонентній платформі, що реалізує відповідну модель компонентів. Зазвичай компонент постачається у вигляді готового двійкового модуля і не потребує компіляції перед розгортанням. Якщо компонент реалізовано як сервіс, його розгортання виконує постачальник сервісу, а не користувач.
Документованість (Documented)	Компоненти повинні бути повністю документовані, щоб потенційні користувачі могли визначити, чи відповідають вони поставленим вимогам. Для всіх інтерфейсів компонента мають бути визначені синтаксис і, за можливості, семантика їх використання.
Незалежність	Компонент повинен бути незалежним, тобто його має бути можливо

(Independent)	компонувати та розгортати без використання інших конкретних компонентів. Якщо для роботи компонента необхідні зовнішні сервіси, вони повинні бути явно визначені в специфікації інтерфейсу «Requires».
Стандартизованість (Standardized)	Стандартизація означає, що компонент, який використовується в процесі компонентно-орієнтованої розробки програмного забезпечення, повинен відповідати стандартній компонентній моделі. Така модель може визначати інтерфейси компонента, метадані, документацію, правила компонування та механізми розгортання.

3.4.2 Компоненти та моделі компонентів

У спільноті програмної інженерії загальноприйнято вважати, що програмний компонент є незалежною програмною одиницею, яка може бути поєднана з іншими компонентами для створення програмної системи. Водночас існують різні підходи до визначення поняття програмного компонента. Одне з найбільш поширених визначень розглядає компонент як програмний елемент, що відповідає стандартній компонентній моделі та може бути незалежно розгорнутий і скомпонований без модифікації відповідно до визначених правил компонування. Інший підхід визначає програмний компонент як одиницю композиції із контрактно визначеними інтерфейсами та лише явно визначеними зовнішніми залежностями, яка може незалежно розгортатися та використовуватися іншими розробниками. Незважаючи на відмінності у формулюваннях, обидва визначення підкреслюють дві основні властивості компонентів — їх незалежність і можливість композиції.

Корисно розглядати програмний компонент як постачальника одного або декількох сервісів. Коли програмній системі необхідно виконати певну функцію, вона звертається до компонента, який надає відповідний сервіс, не враховуючи місце його виконання або мову програмування, якою він був реалізований. Наприклад, у бібліотечній інформаційній системі окремий компонент може реалізовувати сервіс пошуку книг у каталозі.

Взаємодія між компонентами здійснюється через інтерфейси двох типів:

- інтерфейс «Provides» визначає сервіси, які компонент надає іншим компонентам системи;
- інтерфейс «Requires» визначає сервіси, які необхідні компоненту для його коректної роботи і повинні бути надані іншими компонентами.

Таким чином, інтерфейс «Provides» описує функціональні можливості компонента, тоді як інтерфейс «Requires» визначає його зовнішні залежності. Такий підхід забезпечує слабке зв'язування компонентів і дозволяє змінювати їх реалізацію без впливу на інші елементи системи за умови незмінності інтерфейсів.

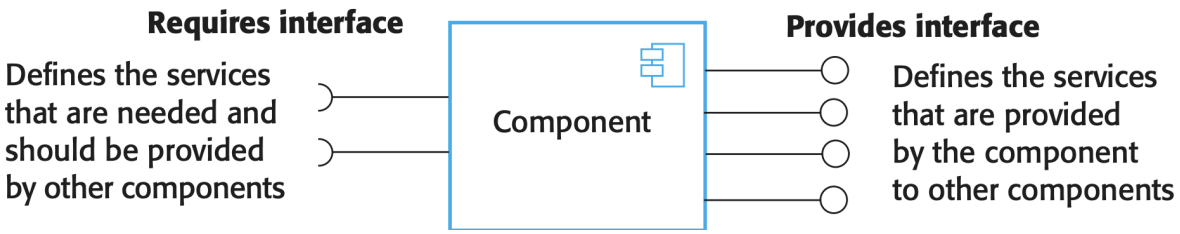


Рисунок 3.2 – Інтерфейси взаємодії компонентів (Component interfaces).

Приклад використання такого підходу наведено на моделі компонента Data Collector, призначеного для збору та оброблення інформації від групи датчиків. Компонент автономно збирає дані протягом визначеного часу та за запитом повертає узагальнену інформацію іншому компоненту системи. Інтерфейс «Provides» містить операції додавання, видалення, запуску, зупинки та тестування датчиків, а також формування звітів і отримання списку підключених датчиків. Інтерфейс «Requires» використовується для взаємодії з датчиками та передбачає використання сервісів керування датчиками і доступу до даних, незалежно від конкретного типу обладнання.

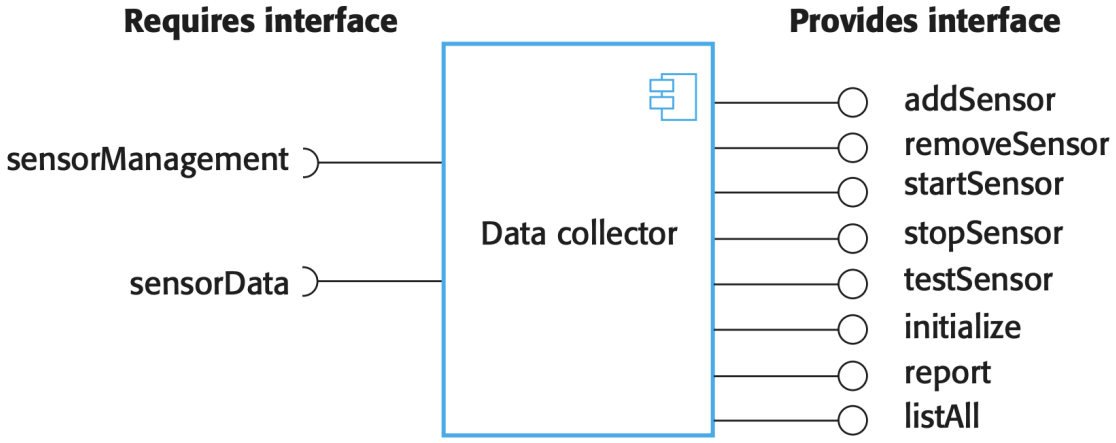


Рисунок 3.3 – Модель компонента Data Collector (A model of a Data Collector component).

Компоненти можуть взаємодіяти між собою за допомогою механізму віддаленого виклику процедур (Remote Procedure Call, RPC). Кожний компонент має унікальний ідентифікатор, за допомогою якого до нього можуть звертатися інші компоненти системи, навіть якщо вони виконуються на іншому комп'ютері. Викликаний компонент аналогічним способом використовує сервіси, визначені у власному інтерфейсі «Requires». Для компонентів, реалізованих як зовнішні сервіси, підтримка необхідних залежностей забезпечується всередині самого сервісу, тому інші програми можуть використовувати його без реалізації додаткових допоміжних компонентів.

Для забезпечення сумісності програмних компонентів використовується компонентна модель (Component Model). Компонентна модель визначає стандарти реалізації, документування та розгортання компонентів. Вона забезпечує можливість взаємодії компонентів незалежно від способу їх реалізації та визначає вимоги як до розробників компонентів, так і до програмної інфраструктури, що забезпечує їх виконання. Для сервісно-орієнтованих компонентів найпоширенішими є моделі вебсервісів, а серед компонентних платформ широкого використання набули Enterprise Java Beans (EJB) і Microsoft .NET.

Основними складовими компонентної моделі є:

- інтерфейси, які визначають набір операцій, параметрів, винятків та мову опису інтерфейсів;
- метадані та інформація про використання, що містять відомості про інтерфейси, атрибути, способи конфігурації та унікальні ідентифікатори компонентів;
- механізми розгортання, які визначають правила пакування компонентів, їх встановлення, налаштування та подальшої заміни.

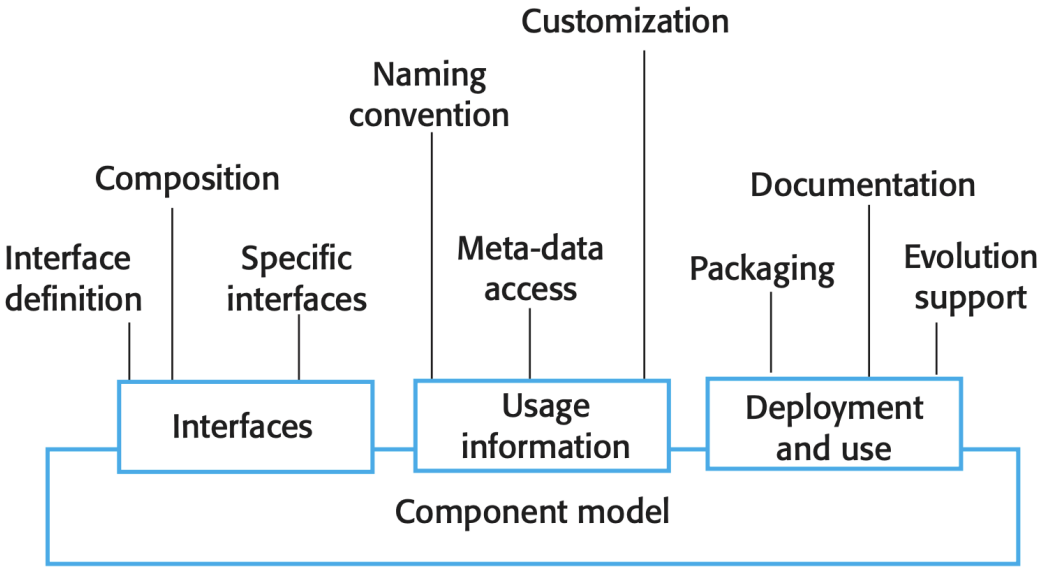


Рисунок 3.4 – Основні елементи компонентної моделі (Component model).

Таким чином, використання компонентних моделей забезпечує стандартизований підхід до створення, документування, розгортання та взаємодії програмних компонентів. Це є необхідною умовою побудови сучасних компонентно-орієнтованих програмних систем, у яких компоненти можуть незалежно розроблятися, повторно використовуватися та інтегруватися в різні програмні продукти.

3.4.3 Компонентно-орієнтований процес розроблення

Компонентно-орієнтовані процеси розроблення програмного забезпечення (Component-Based Software Engineering Processes, CBSE Processes) підтримують

створення програмних систем на основі повторного використання компонентів. Вони враховують можливість використання вже існуючих програмних компонентів та відрізняються від традиційних процесів розроблення тим, що охоплюють як створення компонентів для повторного використання, так і розроблення нових програмних систем із використанням готових компонентів.

На найвищому рівні виділяють два основні процеси компонентно-орієнтованої розробки:

- розроблення компонентів для повторного використання (Development for Reuse) — процес створення компонентів або сервісів, які в подальшому використовуватимуться в інших програмних системах. Зазвичай він передбачає узагальнення вже існуючих компонентів для розширення сфери їх застосування;
- розроблення програмних систем із повторним використанням компонентів (Development with Reuse) — процес створення нових програмних систем шляхом інтеграції вже існуючих компонентів і сервісів.

Обидва процеси мають різні цілі, тому включають різні види діяльності. Під час розроблення компонентів для повторного використання основною метою є створення одного або кількох універсальних компонентів. Розробник має доступ до їх вихідного коду та може модифікувати його для забезпечення більшої універсальності. У процесі розроблення із повторним використанням ситуація інша: спочатку необхідно знайти придатні компоненти, оцінити можливість їх використання та спроєктувати систему таким чином, щоб максимально використати вже існуючі програмні рішення. При цьому доступ до вихідного коду компонентів може бути відсутній.

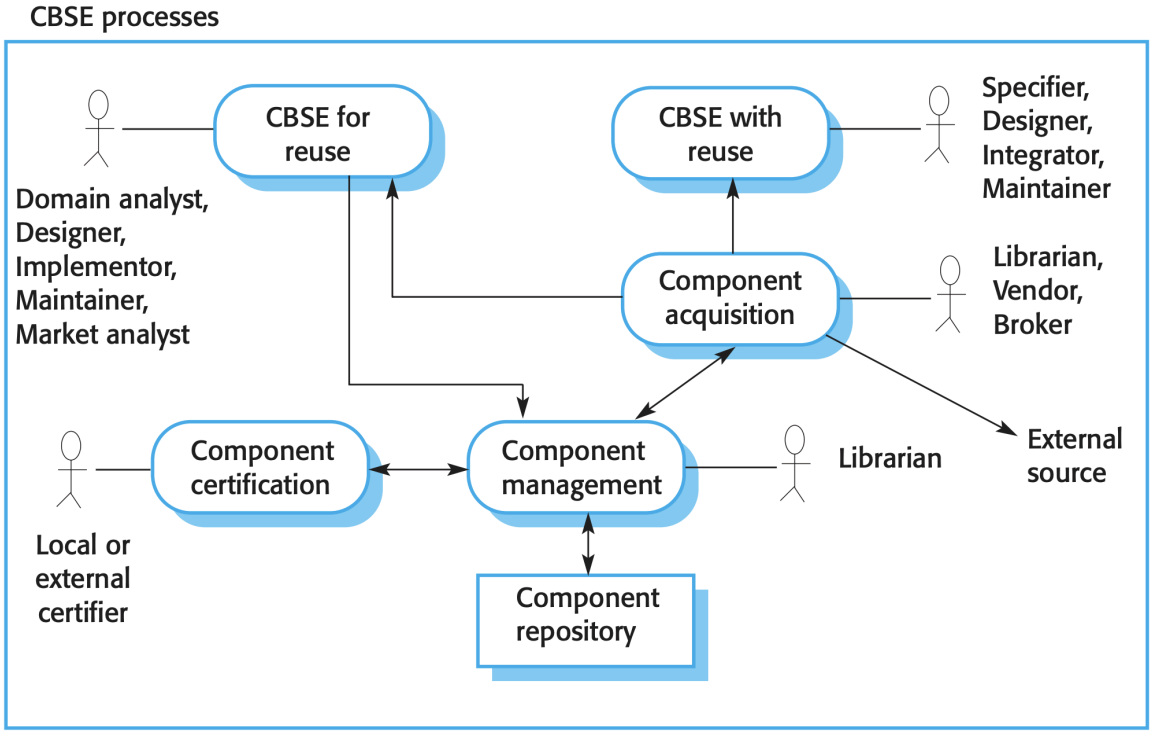


Рисунок 3.5 – Процеси компонентно-орієнтованої розробки програмного забезпечення (CBSE Processes).

Окрім двох основних процесів, компонентно-орієнтована програмна інженерія включає три допоміжні процеси:

- отримання компонентів (Component Acquisition) — процес пошуку, придбання або отримання компонентів, які можуть бути використані повторно. Компоненти можуть надходити як із внутрішніх корпоративних репозиторіїв, так і із зовнішніх джерел;
- керування компонентами (Component Management) — процес каталогізації, зберігання, супроводу та забезпечення доступності компонентів для повторного використання;
- сертифікація компонентів (Component Certification) — процес перевірки компонентів на відповідність встановленим вимогам і підтвердження того, що вони відповідають визначеним специфікаціям якості.

Для підтримки повторного використання організації, як правило, створюють репозиторії компонентів (Component Repository), у яких зберігаються не лише самі компоненти, але й документація, інформація про їх використання, версії та інші метадані. Наявність такого репозиторію значно спрощує пошук необхідних компонентів і підвищує ефективність повторного використання.

Під час розроблення програмних систем із використанням готових компонентів процес розроблення має низку особливостей. Спочатку вимоги до системи формуються лише на загальному рівні. Це дозволяє не обмежувати вибір компонентів занадто жорсткими вимогами. Після пошуку доступних компонентів вимоги можуть уточнюватися або змінюватися відповідно до функціональних можливостей знайдених програмних компонентів. Якщо використання готових компонентів дозволяє швидше або дешевше реалізувати необхідну функціональність, замовники нерідко погоджуються на певне коригування початкових вимог.

Після визначення архітектури програмної системи виконується повторний пошук компонентів і уточнення проєктних рішень. Це пов'язано з тим, що окремі компоненти можуть виявитися несумісними між собою або не забезпечувати необхідної функціональності. У такому випадку здійснюється пошук альтернативних компонентів або повторне уточнення вимог до програмної системи.

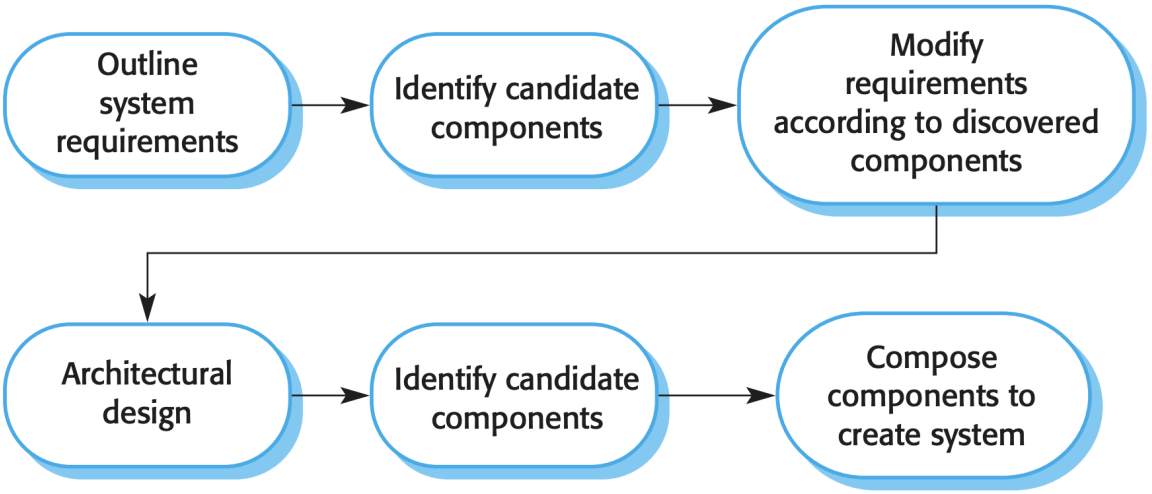


Рисунок 3.6 – Процес розроблення програмної системи з повторним використанням компонентів (CBSE with Reuse)

Таким чином, компонентно-орієнтований процес розроблення відрізняється від традиційних моделей програмної інженерії тим, що процес пошуку, оцінювання, вибору та інтеграції компонентів стає невід'ємною складовою всіх етапів розроблення програмного забезпечення. Такий підхід дозволяє максимально використовувати вже існуючі програмні компоненти, скорочувати терміни створення програмних систем та підвищувати їхню якість.

3.4.4 Композиція програмних компонентів

Після вибору необхідних компонентів наступним етапом компонентно-орієнтованої розробки є їх композиція, тобто об'єднання окремих компонентів у єдину програмну систему. Компоненти можуть розроблятися незалежно один від одного різними організаціями або виробниками, тому для забезпечення їх спільної роботи необхідно організувати правильну взаємодію між ними. Саме процес композиції компонентів забезпечує інтеграцію окремих програмних компонентів у цілісну систему.

Композиція компонентів може здійснюватися за різними схемами залежно від особливостей програмної системи та використовуваної компонентної моделі. Основними способами композиції є:

- послідовна композиція (Sequential Composition), за якої результат роботи одного компонента передається як вхідні дані для іншого компонента;
- ієрархічна композиція (Hierarchical Composition), коли один компонент координує роботу інших компонентів і керує їх взаємодією;
- додаткова композиція (Additive Composition), під час якої декілька незалежних компонентів виконують свої функції паралельно, забезпечуючи різні сервіси програмної системи.

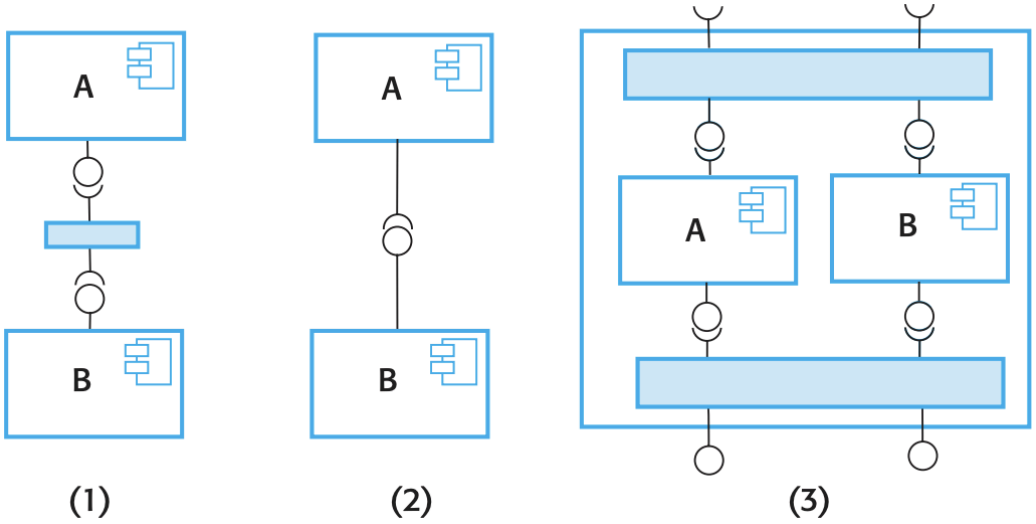


Рисунок 3.7 – Основні способи композиції компонентів (Types of Component Composition)

Однією з найпоширеніших проблем під час композиції компонентів є несумісність їх інтерфейсів. Незважаючи на однакове призначення компонентів, вони можуть використовувати різні формати даних, різні назви операцій, інший порядок передавання параметрів або різні механізми взаємодії. Унаслідок цього безпосередня інтеграція компонентів стає неможливою.

Основними причинами несумісності інтерфейсів є:

- відмінності у найменуванні операцій;
- різні типи або формати параметрів;
- різний порядок передавання параметрів;
- відмінності у форматах обміну даними;
- використання різних протоколів взаємодії між компонентами.

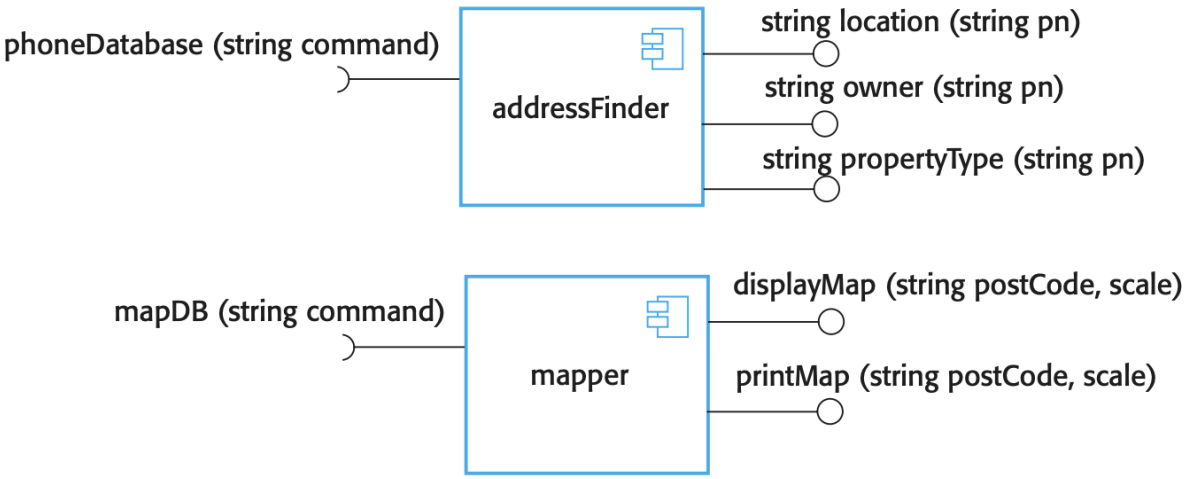


Рисунок 3.8 – Приклад несумісності інтерфейсів компонентів (Incompatible Interfaces)

Для усунення таких невідповідностей використовуються адаптери (Adaptors). Адаптер є спеціальним програмним компонентом, який забезпечує узгодження інтерфейсів двох або більше компонентів без зміни їх внутрішньої реалізації. Він виконує перетворення параметрів, змінює формат даних, узгоджує порядок викликів операцій або реалізує інші необхідні механізми взаємодії. Завдяки використанню адаптерів можна інтегрувати компоненти, розроблені незалежно один від одного, без внесення змін до їх програмного коду.

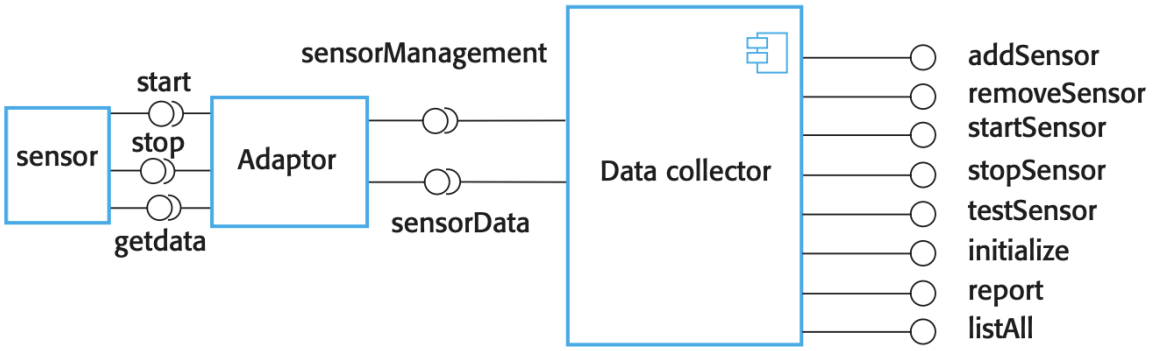


Рисунок 3.9 – Використання адаптера для узгодження інтерфейсів компонентів (Adaptor Component)

У багатьох випадках адаптер реалізується як окремий програмний модуль, який приймає виклики від одного компонента та перетворює їх у форму, зрозумілу іншому компоненту. Такий підхід дозволяє зберегти незалежність компонентів і забезпечити можливість їх повторного використання в інших програмних системах. Якщо виникає необхідність виконання додаткових операцій під час взаємодії компонентів, можуть використовуватися допоміжні програмні модулі (glue code), які реалізують необхідну логіку інтеграції.

Під час композиції компонентів важливо забезпечити збереження їх незалежності та мінімізувати взаємні залежності між ними. Чим менш зв'язаними є компоненти, тим простіше виконувати їх заміну, оновлення або повторне використання в інших програмних системах. Саме тому сучасні компонентні технології приділяють особливу увагу стандартизації інтерфейсів і використанню адаптерів для інтеграції програмних компонентів.

Таким чином, композиція програмних компонентів є одним із ключових етапів компонентно-орієнтованої розробки програмного забезпечення. Вона забезпечує інтеграцію незалежних компонентів у єдину програмну систему, дозволяє повторно використовувати вже існуючі програмні рішення та спрощує розвиток і супровід програмних продуктів.

3.4.5 Документування інтерфейсів компонентів

Однією з найважливіших умов ефективного повторного використання програмних компонентів є їх повне та зрозуміле документування. Розробники компонентів повинні надавати достатньо інформації про функціональні можливості компонентів, способи їх використання, вимоги до взаємодії з іншими компонентами та особливості реалізації інтерфейсів. Без якісної документації використання компонентів у нових програмних системах стає складним або навіть неможливим.

Документація компонента повинна містити специфікацію його інтерфейсів. Інтерфейс визначає набір сервісів, які надає компонент, параметри операцій, типи даних, виняткові ситуації, а також вимоги до правильного використання компонентів. Інтерфейс є єдиним засобом взаємодії між компонентом та іншими елементами програмної системи, тому він повинен бути повним, точним і однозначним.

У багатьох випадках звичайного опису інтерфейсу недостатньо. Для повного визначення поведінки компонента використовуються контракти (Contracts), які описують умови правильного використання компонентів. Контракт містить:

- передумови (Preconditions) — умови, які повинні бути виконані до виклику операції;
- післяумови (Postconditions) — умови, які гарантуються після успішного завершення операції;
- інваріанти (Invariants) — властивості компонента, які повинні залишатися істинними протягом усього часу його існування.

Використання контрактів дозволяє однозначно визначити поведінку компонентів, полегшує їх тестування, інтеграцію та повторне використання. Крім того, формальні специфікації інтерфейсів можуть застосовуватися для автоматизованої перевірки правильності взаємодії компонентів.

Для формального опису обмежень інтерфейсів часто використовується Object Constraint Language (OCL). Ця мова дозволяє описувати передумови, післяумови та інваріанти у формалізованому вигляді, не змінюючи реалізацію самого компонента. Використання OCL забезпечує більш точне документування поведінки компонентів і зменшує ймовірність неправильного використання їхніх сервісів.

Приклад специфікації інтерфейсу компонента демонструє, що документація повинна містити не лише перелік доступних операцій, але й опис їх призначення, параметрів, результатів виконання, можливих виняткових ситуацій і обмежень використання. Такий підхід дозволяє іншим розробникам швидко оцінити можливість використання компонента без необхідності аналізу його внутрішньої реалізації.

```
— The context keyword names the component to which the conditions apply

context addItem

— The preconditions specify what must be true before execution of addItem
pre:    PhotoLibrary.libSize() > 0
        PhotoLibrary.retrieve(pid) = null

— The postconditions specify what is true after execution
post:   libSize () = libSize()@pre + 1
        PhotoLibrary.retrieve(pid) = p
        PhotoLibrary.catEntry(pid) = photodesc

context delete

pre:    PhotoLibrary.retrieve(pid) <> null ;

post:   PhotoLibrary.retrieve(pid) = null
        PhotoLibrary.catEntry(pid) = PhotoLibrary.catEntry(pid)@pre
        PhotoLibrary.libSize() = libSize()@pre—1
```

Рисунок 3.10 – Приклад специфікації інтерфейсу компонента (Photo Library Interface Specification).

Повне документування компонентів є обов'язковою вимогою компонентно-орієнтованої розробки програмного забезпечення. Воно забезпечує можливість незалежного використання компонентів, спрощує їх інтеграцію в нові програмні системи, підвищує якість повторного використання та зменшує витрати на супровід програмного забезпечення.

3.5 Переваги та обмеження повторного використання програмного забезпечення

Повторне використання програмного забезпечення є одним із ключових принципів сучасної програмної інженерії, оскільки дозволяє скоротити час і вартість розроблення програмних систем, підвищити їх якість та ефективніше використовувати накопичений досвід створення програмного забезпечення. У сучасних інформаційних системах значна частина функціональності реалізується шляхом інтеграції вже існуючих програмних компонентів, бібліотек, сервісів або готових програмних продуктів, а не шляхом їх повторного розроблення. Такий підхід забезпечує швидше створення програмних систем та дозволяє розробникам зосередитися на реалізації специфічних функціональних вимог.

Основними перевагами повторного використання є скорочення термінів розроблення, зменшення вартості програмного забезпечення, підвищення його надійності завдяки використанню перевірених компонентів, ефективніше використання досвіду розробників, зниження ризиків реалізації проєкту та забезпечення відповідності програмних систем встановленим стандартам. Детальну характеристику цих переваг наведено в таблиці 3.1.

Разом із численними перевагами повторне використання має й певні обмеження. До основних проблем належать складність пошуку компонентів, необхідність їх оцінювання та адаптації, додаткові витрати на підтримку бібліотек компонентів, можливі труднощі супроводу, недостатня підтримка окремих інструментальних засобів розробки, а також небажання окремих розробників використовувати вже існуючі програмні рішення. Основні проблеми повторного використання наведено в таблиці 3.2.

Ефективне повторне використання програмного забезпечення потребує попереднього планування. Організація повинна визначити, які програмні компоненти доцільно створювати для повторного використання, які компоненти можуть бути використані з уже існуючих репозиторіїв, а також оцінити економічну доцільність їх використання. Планування повторного використання повинно здійснюватися ще на початкових етапах життєвого циклу програмного забезпечення, оскільки вибір архітектурних рішень безпосередньо впливає на можливість інтеграції готових компонентів.

Під час планування повторного використання необхідно враховувати не лише технічні характеристики компонентів, але й їх доступність, рівень документації, умови ліцензування, сумісність із програмною платформою, наявність технічної підтримки та перспективи подальшого розвитку. У багатьох випадках використання готового компонента є доцільнішим навіть тоді, коли його функціональність не повністю відповідає початковим вимогам, оскільки адаптація вимог може бути дешевшою та швидшою, ніж створення нового програмного рішення.

Для підтримки процесу повторного використання в організаціях створюються репозиторії програмних компонентів, у яких зберігаються програмні модулі, документація, приклади використання та інформація про сумісність компонентів. Наявність таких репозиторіїв значно спрощує пошук готових програмних рішень і підвищує ефективність їх повторного використання.

Таким чином, повторне використання програмного забезпечення є одним із найефективніших підходів до створення сучасних програмних систем. Його успішне застосування залежить не лише від наявності готових програмних компонентів, але й від правильного планування процесу повторного використання, ефективного керування репозиторіями компонентів та своєчасної оцінки переваг і можливих обмежень використання існуючих програмних рішень.

Висновки

Повторне використання програмного забезпечення є одним із ключових підходів сучасної програмної інженерії, що забезпечує підвищення ефективності процесу розроблення, скорочення витрат часу та ресурсів, а також покращення якості програмних систем. Використання програмних бібліотек, фреймворків, шаблонів

проєктування та компонентної архітектури дозволяє застосовувати перевірені програмні рішення, зменшувати дублювання програмного коду та спрощувати супровід і розвиток програмного забезпечення. Важливою складовою компонентно-орієнтованого підходу є використання стандартизованих інтерфейсів, компонентних моделей і засобів документування, що забезпечують незалежність програмних компонентів та можливість їх інтеграції в різні програмні системи. Разом із численними перевагами повторне використання має певні обмеження, пов'язані з пошуком, адаптацією, сумісністю та супроводом компонентів, тому його ефективне застосування потребує ретельного планування та обґрунтованого вибору програмних рішень. Комплексне використання сучасних технологій повторного використання є необхідною умовою створення гнучкого, масштабованого, надійного та економічно ефективного програмного забезпечення.

Лекція 4

Тема: Абстрактні типи даних

Мета: Ознайомити студентів із поняттям абстрактного типу даних, його роллю в об'єктно-орієнтованому проєктуванні програмного забезпечення, принципами опису інтерфейсів абстрактних типів даних, їх реалізації за допомогою класів, а також особливостями використання основних абстрактних структур даних під час розроблення програмних систем.

План лекції

1. Поняття абстрактного типу даних
2. Варіанти реалізації абстрактних типів даних
3. Формальна специфікація абстрактних типів даних
4. Абстрактний тип даних STACK як формальна модель обчислень
5. Перехід від абстрактних типів даних до класів
6. Висновки

4.1 Поняття абстрактного типу даних

Використання об’єктів як основи архітектури програмного забезпечення потребує їх належного опису. Знання теорії абстрактних типів даних допомагає краще зрозуміти практику об’єктно-орієнтованого аналізу, проєктування та програмування. Абстрактні типи даних утворюють теоретичну основу об’єктно-орієнтованого підходу, оскільки дають змогу описувати об’єкти незалежно від конкретних способів їх програмної реалізації.

Для правильного опису об’єктів необхідний метод, який задовольняє три основні умови:

- опис повинен бути точним та однозначним;
- опис повинен бути повним або настільки повним, наскільки це потрібно в конкретному випадку;
- опис не повинен містити надмірної специфікації.

Остання умова робить завдання опису об’єктів нетривіальним. Забезпечити точність, однозначність і повноту нескладно, якщо розкрити всі деталі внутрішнього представлення об’єкта. Проте для програмних модулів, які використовують цей об’єкт, така інформація зазвичай є надлишковою. Велика кількість деталей реалізації заважає розробникам зосередитися на власному завданні та ускладнює подальшу зміну програмного забезпечення. Небезпека виникає тоді, коли програмні модулі використовують структуру даних на основі відомостей про її фізичне представлення, а не про її основні властивості.

Необхідність абстрактного опису даних можна пояснити на прикладі стека. Стек використовується для зберігання та отримання елементів за принципом

«останнім надійшов — першим вийшов» (Last In, First Out, LIFO). Останній доданий до стека елемент є першим елементом, який із нього отримується. Стеки широко використовуються в програмних системах, зокрема в компіляторах, інтерпретаторах та інших засобах опрацювання даних.

Один і той самий стек може мати різні фізичні представлення. Він може бути реалізований за допомогою масиву, елементи якого заповнюються від початку до кінця; за допомогою масиву, що заповнюється у зворотному напрямку; або як зв'язана структура, у якій кожний елемент містить значення та посилання на попередній елемент. Усі ці варіанти реалізують одну структуру даних, хоча внутрішньо організовані по-різному. Вибір одного з них як єдиного визначення стека був би типовим випадком надмірної специфікації, оскільки масив, індекси, лічильники та посилання не є суттєвими для розуміння самого поняття стека.

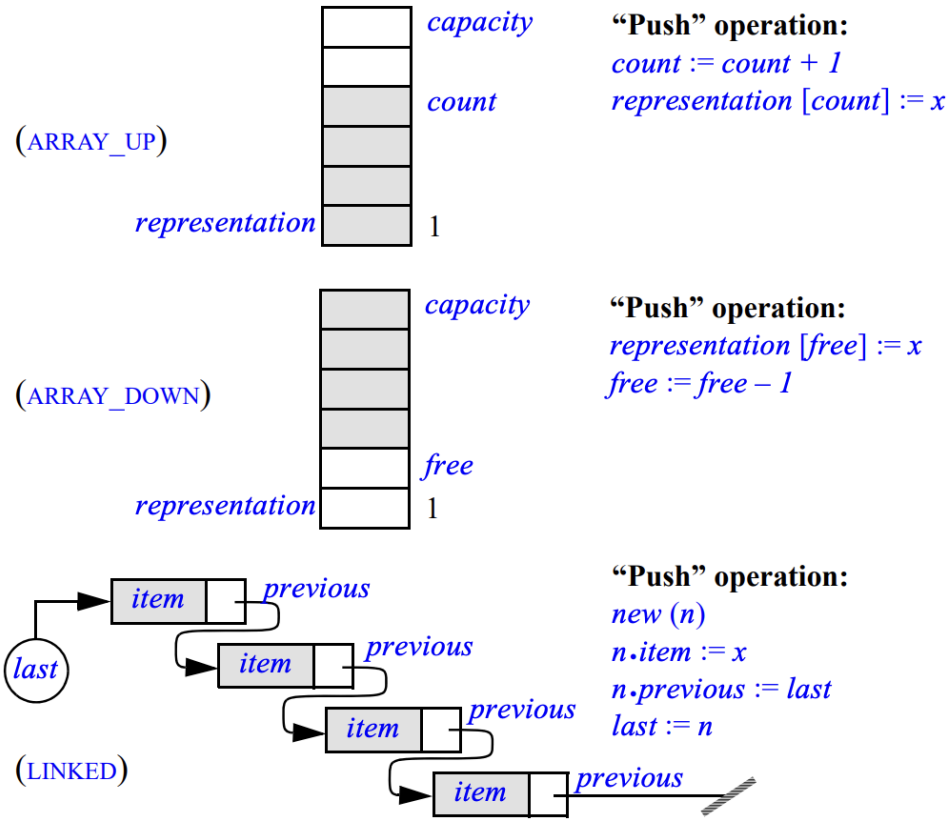


Рисунок 4.1 – Три можливі способи фізичного представлення стека: ARRAY_UP, ARRAY_DOWN і LINKED

Рисунок 4.1 демонструє, що однакові зовнішні властивості стека можуть бути забезпечені різними способами реалізації. У варіанті ARRAY_UP елементи зберігаються в масиві від першої позиції до позиції, визначеної лічильником count. У варіанті ARRAY_DOWN елементи зберігаються від кінця масиву, а змінна free визначає найвищу вільну позицію. У варіанті LINKED кожний елемент розміщується

в окремій комірці, яка містить саме значення та посилання на попередню комірку. Відмінності між цими способами не впливають на основне призначення структури: у кожному випадку вона забезпечує зберігання та отримання елементів відповідно до принципу LIFO.

Щоб зберегти точність, повноту та однозначність опису без надмірної специфікації, необхідно зосередитися не на конкретному представленні структури даних, а на операціях, які можуть над нею виконуватися, та на властивостях цих операцій. Саме операції об'єднують різні реалізації стека й дозволяють отримати його абстрактну, але водночас практично корисну характеристику.

Для стека зазвичай визначають такі операції:

- put — команда додавання елемента на вершину стека;
- remove — команда видалення верхнього елемента, якщо стек не порожній;
- item — запит для отримання верхнього елемента, якщо стек не порожній;
- empty — запит, який визначає, чи є стек порожнім;
- make — операція створення нового порожнього стека.

Операції над абстрактним типом даних поділяються на три категорії. Операції створення (creators) утворюють нові об'єкти. Запити (queries) повертають інформацію про об'єкти, не змінюючи їх. Команди (commands) можуть змінювати стан об'єктів. У прикладі зі стеком make є операцією створення; item та empty є запитами; put і remove є командами.

У традиційному підході поняття стека могло б задаватися декларацією конкретної структури даних, наприклад масивом і цілим числом, яке визначає кількість збережених елементів. Операції put, remove, item, empty і make у такому разі розглядалися б як підпрограми, що працюють із цією структурою. Ключовий крок до абстракції даних полягає у зміні цієї точки зору: необхідно тимчасово забути про представлення і розглядати самі операції як основу визначення структури даних. Отже, стеком є будь-яка структура, до якої клієнти можуть застосовувати визначений набір операцій і для якої виконуються встановлені властивості цих операцій.

Абстрактний тип даних описує не окремий об'єкт, а множину об'єктів із однаковими суттєвими властивостями. Наприклад, АТД STACK описує не один конкретний стек, а всі можливі стеки, що підтримують визначені операції та відповідають установленим правилам поведінки. Конкретний об'єкт, властивості якого відповідають опису абстрактного типу даних, є екземпляром цього типу.

Світ об'єктів і програмної архітектури можна розглядати як сукупність взаємодіючих агентів, які обмінюються запитами відповідно до точно визначених правил. Програмні модулі в такій взаємодії виступають постачальниками та клієнтами. Постачальник надає певні операції, а клієнт використовує їх, не

покладаючись на внутрішню реалізацію постачальника. Правила цієї взаємодії надалі формують основу програмних контрактів.

Важливим аспектом абстракції є узгодженість назв операцій. Традиційні назви операцій стека та назви, використані для уніфікованого опису контейнерів, наведено в таблиці 4.1.

Таблиця 4.1 – Назви основних операцій стека

Традиційна назва операції стека	**Уніфікована назва**
push	put
pop	remove
top	item
new	make

Стек є одним із різновидів контейнерів, а точніше — структурою типу «диспенсер». Така структура надає клієнтам механізми збереження, отримання та видалення об’єктів, але не дозволяє клієнтові довільно вибирати елемент, над яким виконується операція. Для стека політика LIFO означає, що отримати або видалити можна тільки останній збережений елемент. Іншим різновидом диспенсера є черга, що працює за принципом «першим надійшов — першим вийшов» (First In, First Out, FIFO): елемент додається з одного боку, а отримується та видаляється з іншого. Масив також є контейнером, але не належить до диспенсерів, оскільки індекси дозволяють клієнтові самостійно вибирати позиції для збереження та отримання елементів.

Спільні властивості різних контейнерів важливіші за відмінності між конкретними способами збереження, отримання та видалення елементів. Тому уніфікована термінологія повинна підкреслювати спільність структур даних і зменшувати кількість несумісних назв. Це особливо важливо для великих бібліотек повторно використовуваних програмних компонентів, які можуть містити десятки тисяч операцій. Без систематичної та зрозумілої номенклатури розробники й користувачі таких бібліотек зіткнулися б із великою кількістю специфічних і несумісних назв, що створювало б перешкоди для повторного використання програмного забезпечення.

Отже, назви не є лише косметичним елементом програмного забезпечення. Якісне повторно використовуване програмне забезпечення повинно не тільки надавати правильну функціональність, але й представляти її за допомогою правильних, послідовних і зрозумілих назв. Абстрактний тип даних дозволяє описувати структуру через доступні операції та їх суттєві властивості, приховуючи

конкретне фізичне представлення. Такий підхід забезпечує точність і повноту опису, запобігає надмірній специфікації та створює теоретичну основу для побудови гнучких об’єктно-орієнтованих програмних систем.

4.2 Варіанти реалізації абстрактних типів даних

Необхідність абстрактного опису об’єктів особливо чітко проявляється тоді, коли одна й та сама структура даних може мати декілька різних фізичних представлень. Якщо опис структури безпосередньо пов’язати з одним конкретним способом її реалізації, програмні модулі, що використовують цю структуру, також почнуть залежати від деталей її внутрішньої організації. Унаслідок цього будь-яка зміна представлення даних може вимагати внесення змін до значної кількості інших частин програмної системи. Тому опис об’єкта повинен ґрунтуватися не на його фізичній будові, а на суттєвих властивостях і доступних операціях.

Розглянемо можливі способи реалізації стека. У представленні ARRAY_UP стек описується за допомогою масиву representation і цілого числа count, значення якого змінюється від нуля для порожнього стека до значення capacity, що визначає місткість масиву. Елементи стека зберігаються в позиціях масиву від 1 до count. Під час додавання нового елемента значення count збільшується на одиницю, після чого новий елемент записується у відповідну позицію масиву:

count := count + 1

representation [count] := x

У представленні ARRAY_DOWN елементи також зберігаються в масиві, але його заповнення відбувається від кінця до початку. Замість лічильника count використовується змінна free, яка вказує на найвищу вільну позицію масиву. Для порожнього стека її значення дорівнює capacity, а в міру додавання елементів воно зменшується до нуля. Новий елемент спочатку записується в позицію free, після чого значення цієї змінної зменшується:

representation [free] := x

free := free – 1

У представленні LINKED кожний елемент стека зберігається в окремій комірці, яка містить два поля: item, де розташовується значення елемента, та previous, що містить посилання на комірку з попереднім елементом. Додатково використовується посилання last, яке вказує на вершину стека. Для додавання нового елемента необхідно створити нову комірку, записати до неї значення, зв’язати її з попередньою вершиною та оновити посилання last:

new (n)

n.item := x

n.previous := last

last := n

Для масивних реалізацій необхідно також перевіряти наявність вільного місця. У випадку ARRAY_UP операція додавання може бути виконана лише за умови count < capacity, а для ARRAY_DOWN — за умови free > 0. Ці перевірки відображають обмежену місткість реалізацій, заснованих на масивах. Зв'язане представлення не має наперед установленої місткості, однак залежить від обсягу доступної динамічної пам'яті.

Розглянуті способи є найбільш поширеними, проте існують й інші варіанти. Наприклад, якщо необхідно реалізувати два стеки з елементами одного типу за обмеженого обсягу пам'яті, можна використати один спільний масив із двома покажчиками вершини. Перший стек зростає від початку масиву до його кінця, як у представленні ARRAY_UP, а другий — від кінця масиву до початку, як у представленні ARRAY_DOWN. Перший стек використовує лічильник count, а другий — покажчик free. Спільне представлення вважається заповненим тоді й лише тоді, коли значення count і free збігаються.

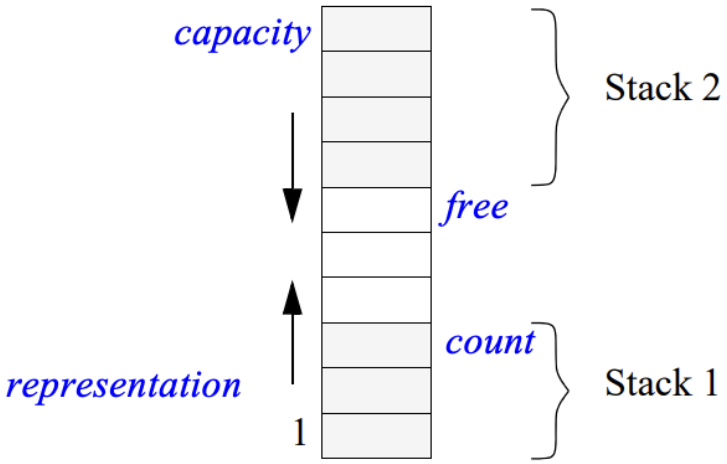


Рисунок 4.2 – Спільне представлення двох стеків в одному масиві

Перевагою такого представлення є зменшення ризику передчасного вичерпання пам'яті. Якщо для двох стеків використовувати два окремі масиви місткістю n, пам'ять одного з них буде вичерпана відразу після досягнення відповідним стеком n елементів, навіть якщо другий масив майже порожній. Якщо ж використовується один масив розміром 2n, пам'ять закінчиться тільки тоді, коли сумарна кількість елементів обох стеків досягне 2n. За умови незалежного зростання стеків така ситуація є менш імовірною.

Кожне з наведених представлень може бути корисним у певних умовах. Масивне представлення забезпечує простий і швидкий доступ до елементів, але обмежується наперед визначеною місткістю. Зв'язане представлення є гнучкішим щодо кількості елементів, однак потребує додаткової пам'яті для зберігання посилань і складнішого керування динамічною пам'яттю. Спільне представлення двох стеків

дозволяє ефективніше використовувати доступний обсяг пам’яті, але підходить лише для спеціальних ситуацій.

Вибір одного з цих варіантів як єдиного визначення стека був би типовим прикладом надмірної специфікації. Немає підстав вважати, що ARRAY_UP краще відображає сутність стека, ніж LINKED або будь-яка інша реалізація. Найпомітніші характеристики масивного представлення — сам масив, лічильник і верхня межа — не є важливими для розуміння абстрактної сутності стека. Клієнтові стека важливо знати, які операції він підтримує та якими властивостями характеризується їх поведінка, а не спосіб розміщення елементів у пам’яті.

Використання конкретного фізичного представлення як специфікації є небезпечним, оскільки створює тісну залежність програмного забезпечення від формату даних. Значна частина витрат на супровід програмних систем пов’язана саме з необхідністю враховувати зміни форматів даних. Якщо аналіз і проєктування системи ґрунтуються на фізичній структурі даних, отримане програмне забезпечення буде недостатньо гнучким і складним для подальшого розвитку.

Небезпека такої залежності стосується не лише традиційних структур даних. Вона проявляється і під час моделювання реальних об’єктів. Наприклад, інформаційна система може передбачати, що ім’я людини має лише один середній ініціал, поштовий індекс завжди складається з фіксованої кількості символів або рік можна зберігати лише двома цифрами. Такі рішення можуть здаватися прийнятними під час початкового розроблення, але стають джерелом серйозних проблем, коли фактичні дані не відповідають закладеним обмеженням або коли вимоги до системи змінюються.

У книзі Б. Майєра наведено приклад системи, розробники якої передбачили можливість зберігати лише один середній ініціал імені. Для особи з двома середніми ініціалами це призвело до появи декількох різних варіантів імені в інформаційних базах. Змінити систему було складно, оскільки обмеження на довжину поля було закладене в її внутрішню структуру. Цей приклад показує, що програмне забезпечення не повинно залежати від випадкових фізичних властивостей даних, які можуть виявитися неправильними або змінитися з часом.

Аналогічною проблемою було зберігання року двома цифрами, що стало причиною так званої проблеми 2000 року. У таких системах конкретне фізичне представлення часу помилково сприймалося як його суттєва властивість. Унаслідок цього зміна календарного періоду вимагала масштабного перегляду програмного забезпечення та значних фінансових витрат.

Отже, різні реалізації одного абстрактного типу даних можуть суттєво відрізнятися за внутрішньою структурою, використанням пам’яті та способом виконання операцій. Проте ці відмінності не повинні впливати на клієнтів структури даних. Специфікація абстрактного типу має описувати лише його суттєві властивості,

доступні операції та правила їх поведінки. Вибір представлення повинен залишатися прихованою частиною реалізації, яку можна змінювати без модифікації програмних модулів, що використовують відповідний тип даних.

4.3 Формальна специфікація абстрактних типів даних

Попередній розгляд абстракції даних показав, що структура даних повинна визначатися не через конкретне фізичне представлення, а через доступні операції та їхні властивості. Однак неформальних пояснень на зразок «операція put додає елемент на вершину стека», «remove видаляє останній доданий елемент», а «item повертає верхній елемент» недостатньо. Необхідно точно визначити, як клієнти можуть використовувати ці операції та які результати вони отримають. Таку інформацію надає формальна специфікація абстрактного типу даних.

Специфікація абстрактного типу даних складається з чотирьох основних частин:

- TYPES — типи;
- FUNCTIONS — функції;
- AXIOMS — аксіоми;
- PRECONDITIONS — передумови.

Для опису властивостей АТД використовується проста математична нотація. Вона не є мовою програмування, оскільки призначена не для реалізації програмних операцій, а для точного та однозначного визначення типів, функцій і правил їхньої поведінки. Така специфікація повинна описувати всі суттєві властивості типу, не прив’язуючись до конкретного способу його реалізації.

4.3.1 Визначення типів

Розділ TYPES містить перелік типів, які вводяться у специфікації. Для прикладу зі стеком визначається один абстрактний тип: ***TYPES - STACK [G]***

Абстрактний тип даних STACK [G] описує стеки, елементами яких можуть бути об’єкти довільного типу G. Абстрактний тип даних не є одним конкретним об’єктом. Наприклад, STACK — це не окремий стек, а множина всіх можливих стеків, які задовольняють визначені властивості. Конкретний стек, що відповідає специфікації STACK, називається екземпляром цього абстрактного типу даних.

Позначення G є формальним узагальненим параметром. Воно означає довільний, наперед не визначений тип елементів стека. Механізм, який дозволяє параметризувати специфікацію типом елементів, називається узагальненістю або genericity. Завдяки цьому немає потреби створювати окремі специфікації для стека цілих чисел, стека банківських рахунків, стека рядків або інших об’єктів. Усі ці структури мають однакові основні операції та відрізняються лише типом елементів.

Сам по собі STACK [G] є шаблоном типу. Щоб отримати конкретний тип, необхідно замінити формальний параметр G фактичним типом. Наприклад: STACK [INTEGER] визначає тип стека цілих чисел, а STACK [ACCOUNT] визначає тип стека банківських рахунків.

Узагальнені типи можуть вкладатися один в одного. Наприклад: STACK [STACK [ACCOUNT]] описує стек, елементами якого є інші стеки, що містять об’єкти типу ACCOUNT. Узагальненість забезпечує повторне використання не лише програмного коду, а й специфікацій типів.

4.3.2 Визначення функцій

Після розділу TYPES наводиться розділ FUNCTIONS, у якому перелічуються операції, що можуть застосовуватися до екземплярів абстрактного типу даних. Для типу STACK [G] визначаються такі функції (FUNCTIONS):

- *put: STACK [G] × G → STACK [G]*
- *remove: STACK [G] ↛ STACK [G]*
- *item: STACK [G] ↛ G*
- *empty: STACK [G] → BOOLEAN*
- *new: → STACK [G]*

Кожний рядок задає сигнатуру математичної функції, тобто визначає типи її аргументів і тип результату. Наприклад:

put: STACK [G] × G → STACK [G]

означає, що функція put приймає два аргументи: стек елементів типу G та один елемент типу G, після чого повертає новий стек типу STACK [G].

У математичній моделі операція put не змінює існуючий стек. Вона утворює новий стек, який містить усі попередні елементи та новий елемент на вершині. Такий підхід відрізняється від програмної реалізації, де відповідна команда зазвичай змінює стан уже існуючого об’єкта. Однак для формальної специфікації використання математичних функцій є необхідним, оскільки дозволяє уникнути понять стану, часу та побічних ефектів.

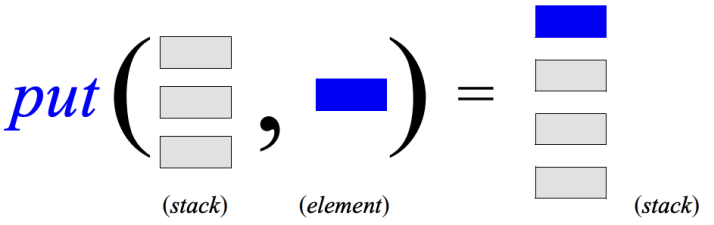


Рисунок 4.3 – Застосування функції put до стека та елемента

Роль функцій у специфікації стека визначається так:

- функція put повертає новий стек із доданим верхнім елементом;
- функція remove повертає стек без його верхнього елемента;

- функція item повертає верхній елемент стека;
- функція empty визначає, чи є стек порожнім;
- функція new повертає новий порожній стек.

Сигнатура функції містить типи аргументів ліворуч від стрілки та тип результату праворуч. Наприклад, у сигнатурі put множина аргументів є декартовим добутком STACK [G] × G, тобто множиною пар <s, x>, де s є стеком, а x — елементом типу G.

Функція new не потребує аргументів, оскільки завжди повертає порожній стек. Її повна сигнатура має вигляд:

new: → STACK [G]

Проте стрілку часто опускають і записують скорочено:

new: STACK [G]

4.3.3 Категорії функцій

Функції абстрактного типу даних поділяються на творці, запити та команди. Категорія визначається положенням типу, що специфікується, відносно стрілки в сигнатурі функції.

Функція-творець (creator) містить тип лише праворуч від стрілки. Вона створює новий екземпляр типу на основі інших значень або без аргументів. У специфікації стека такою функцією є:

new: → STACK [G]

Функція-запит (query) містить тип лише ліворуч від стрілки та повертає інформацію про екземпляр цього типу. До запитів належать:

item: STACK [G] ↛ G

empty: STACK [G] → BOOLEAN

Функція-команда (command) містить тип як ліворуч, так і праворуч від стрілки. Вона утворює новий екземпляр типу на основі вже існуючого екземпляра. До команд належать:

put: STACK [G] × G → STACK [G]

remove: STACK [G] ↛ STACK [G]

У літературі також використовуються назви «конструктор», «аксесор» і «модифікатор», однак терміни «творець», «запит» і «команда» краще відображають майбутню роль цих функцій як операцій над програмними об’єктами.

4.3.4 Аксіоми абстрактного типу даних

Самого переліку функцій недостатньо для повного визначення типу. Такий набір сигнатур міг би однаково описувати стек, чергу або іншу структуру типу «диспенсер». Щоб визначити саме стек, необхідно зафіксувати властивості функцій і зв’язки між ними.

Використання явного визначення функцій через конкретне представлення було б неправильним, оскільки змусило б обрати певну реалізацію. Наприклад, визначення

put через масив і лічильник count прив’язало б специфікацію до представлення ARRAY_UP. Щоб уникнути цього, функції визначаються не явно, а через їхні суттєві властивості. Ці властивості записуються в розділі AXIOMS.

Для типу STACK [G] аксіоми мають такий вигляд:

Для будь-яких x: G і s: STACK [G]:

- *A1: item (put (s, x)) = x*
- *A2: remove (put (s, x)) = s*
- *A3: empty (new)*
- *A4: not empty (put (s, x))*

Перші дві аксіоми визначають основну властивість стека — принцип LIFO. Аксіома A1 встановлює, що верхнім елементом стека після додавання x є саме x. Аксіома A2 визначає, що видалення щойно доданого елемента повертає початковий стек.

Аксіоми A3 і A4 визначають умови порожності стека. Новий стек є порожнім, а стек, до якого було додано хоча б один елемент, не є порожнім. Еквівалентно ці аксіоми можна записати так:

- *empty (new) = true*
- *empty (put (s, x)) = false*

Аксіоми не описують внутрішню реалізацію функцій, а лише визначають властивості, які повинні виконуватися для будь-якого можливого представлення стека. Завдяки цьому одна специфікація може бути реалізована масивом, зв’язаною структурою або іншим способом.

Неявний характер специфікації означає, що вона містить лише відомі та необхідні властивості типу, але не стверджує, що інших властивостей не існує. До АТД можна додавати нові операції та правила, не руйнуючи вже визначеної специфікації. Така відкритість надалі стає основою розширення класів за допомогою успадкування.

4.3.5 Часткові функції

Не всі операції можуть бути застосовані до кожного можливого об’єкта. Наприклад, неможливо отримати верхній елемент порожнього стека або видалити з нього елемент. Тому функції item і remove не визначені для всіх екземплярів STACK [G].

Функція, яка не визначена для всіх елементів множини аргументів, називається частковою функцією. Функція, визначена для всіх можливих аргументів, називається повною функцією. Для позначення часткових функцій використовується перекреслена стрілка:

remove: STACK [G] ↛ STACK [G]

item: STACK [G] ↛ G

Областю визначення часткової функції є та частина множини аргументів, для якої функція має значення. Для item і remove областю визначення є множина всіх непорожніх стеків.

Часткові функції можуть відображати як абстрактні властивості операцій, так і обмеження конкретної реалізації. Наприклад, item і remove є частковими незалежно від способу реалізації стека. Водночас put може стати частковою функцією лише для стека обмеженої місткості, реалізованого за допомогою масиву.

4.3.6 Передумови

Для кожної часткової функції необхідно точно визначити умови її застосування. Це виконується в розділі PRECONDITIONS.

Для типу STACK [G] передумови мають такий вигляд (PRECONDITIONS):

- *remove (s: STACK [G]) require not empty (s)*
- *item (s: STACK [G]) require not empty (s)*

Вираз після ключового слова require визначає умову, за якої аргумент належить до області визначення функції. Передумови remove та item вимагають, щоб стек s не був порожнім.

Передумова встановлює обов'язок клієнта: перед викликом операції він повинен переконатися, що встановлена умова виконується. У прикладі зі стеком клієнт може використати запит empty, щоб перевірити стан стека перед застосуванням item або remove.

4.3.7 Повна специфікація АТД STACK

Об'єднавши всі розглянуті частини, отримаємо повну специфікацію абстрактного типу даних STACK [G]. Абстрактний тип даних STACK [G]:

TYPES

- STACK [G]

FUNCTIONS

- put: STACK [G] × G → STACK [G]
- remove: STACK [G] ↛ STACK [G]
- item: STACK [G] ↛ G
- empty: STACK [G] → BOOLEAN
- new: → STACK [G]

AXIOMS

Для будь-яких x: G і s: STACK [G]:

- A1: item (put (s, x)) = x
- A2: remove (put (s, x)) = s
- A3: empty (new)
- A4: not empty (put (s, x))

PRECONDITIONS

- remove (s: STACK [G]) require not empty (s)

- item (s: STACK [G]) require not empty (s)

Ця специфікація охоплює всі суттєві властивості стека та не містить відомостей, притаманних лише окремим способам його реалізації. Вона не визначає, чи зберігаються елементи в масиві, зв'язаному списку або іншій структурі. Специфікація лише встановлює, які операції доступні клієнтам, які типи аргументів і результатів вони мають та яким правилам повинна відповідати їхня поведінка.

Отже, формальна специфікація абстрактного типу даних забезпечує точний, однозначний і незалежний від реалізації опис структури даних. Вона дозволяє визначати типи через операції, сигнатури, аксіоми та передумови, формує основу для перевірки правильності реалізацій і створює математичний фундамент об'єктно-орієнтованого проєктування.

4.4 Абстрактний тип даних STACK як формальна модель обчислень

Формальна специфікація абстрактного типу даних дає змогу не лише описати властивості структури даних незалежно від її реалізації, а й побудувати математичну модель виконання операцій над цією структурою. На прикладі АТД STACK [G] можна показати, що послідовність програмних дій подається у вигляді складного виразу, а результат обчислення такого виразу визначається шляхом послідовного застосування аксіом. Отже, виконання програми можна розглядати як алгебраїчне спрощення виразу.

Специфікація стека охоплює всі суттєві властивості цієї структури та не містить відомостей, які стосуються лише окремих варіантів її представлення. Вона описує всю необхідну інформацію про стек, але тільки цю інформацію. Саме в цьому полягає сила абстрактних типів даних: вони дозволяють точно визначити поведінку структури, не пов'язуючи її з масивом, зв'язаним списком, покажчиками, лічильниками або іншими деталями фізичної реалізації.

Функції АТД дозволяють будувати складні математичні вирази, а аксіоми — спрощувати їх до остаточного результату. Такий підхід подібний до звичайних перетворень в алгебрі. Наприклад, математичний вираз спрощується за допомогою відомих тотожностей незалежно від конкретних числових значень. Аналогічно вираз над стеком спрощується за допомогою аксіом A1–A4, що визначають поведінку операцій put, remove, item, empty і new.

Розглянемо складний вираз над стеком:

item (remove (put (remove (put (put (remove (put (put (put (new, x1), x2), x3)), item (remove (put (put (new, x4), x5)))), x6)), x7)))

Для полегшення аналізу цей вираз можна подати як послідовність допоміжних виразів:

s1 = new

s2 = put (put (put (s1, x1), x2), x3)

s3 = remove (s2)
s4 = new
s5 = put (put (s4, x4), x5)
s6 = remove (s5)
y1 = item (s6)
s7 = put (s3, y1)
s8 = put (s7, x6)
s9 = remove (s8)
s10 = put (s9, x7)
s11 = remove (s10)
stackexp = item (s11)

Ця послідовність описує такі дії. Спочатку створюється порожній стек s1. До нього послідовно додаються елементи x1, x2 і x3, унаслідок чого утворюється стек s2. Після видалення верхнього елемента x3 отримується стек s3, у якому залишаються x1 і x2. Далі створюється інший порожній стек s4, до якого додаються x4 і x5. Після видалення елемента x5 отримується стек s6, а запит item (s6) повертає x4. Значення x4 додається до першого стека, після чого виконуються подальші операції додавання і видалення елементів.

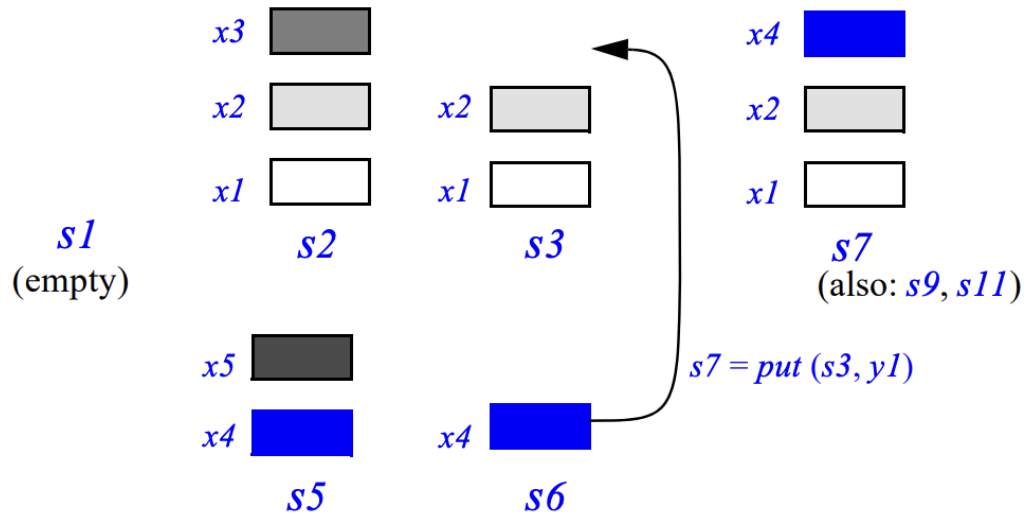


Рисунок 4.4 – Послідовність операцій над стеком

Значення виразу можна визначити графічно, послідовно відображаючи стан стека після кожної операції. Проте формальна специфікація дозволяє отримати той самий результат без використання рисунків — лише шляхом застосування аксіом.

Аксіома A2

remove (put (s, x)) = s

означає, що послідовне додавання елемента до стека та його видалення взаємно компенсуються. Тому вираз

s3 = remove (put (put (put (s1, x1), x2), x3))

можна спростити до

s3 = put (put (s1, x1), x2).

Аналогічно:

s6 = remove (put (put (s4, x4), x5))

спрощується до

s6 = put (s4, x4).

Після цього використовується аксіома A1

item (put (s, x)) = x,

відповідно до якої верхнім елементом стека після виконання put (s, x) є елемент x. Отже:

y1 = item (put (s4, x4)) = x4.

Тому стек s7 утворюється шляхом додавання елемента x4 до стека s3:

s7 = put (s3, x4).

Наступні операції також можна спростити за аксіомою A2:

s9 = remove (put (s7, x6)) = s7;

s11 = remove (put (s9, x7)) = s9.

Оскільки s9 = s7, отримуємо:

s11 = s7 = put (s3, x4).

Тоді остаточний вираз має вигляд:

stackexp = item (put (s3, x4)).

Застосування аксіоми A1 дає остаточний результат:

stackexp = x4.

Таким чином, значення складного виразу над стеком визначається без аналізу конкретного способу зберігання елементів. Не має значення, чи реалізовано стек за допомогою масиву, зв’язаної структури або іншого фізичного представлення. Результат залежить виключно від формальної специфікації АТД і визначених у ній аксіом.

Перед застосуванням аксіом необхідно також переконатися, що всі часткові функції використовуються відповідно до їхніх передумов. Операції item і remove можуть виконуватися лише для непорожнього стека. У наведеному прикладі кожному виклику цих функцій передує принаймні одна операція put, тому відповідні передумови виконуються. Якби операція item або remove застосовувалася безпосередньо до new, такий вираз не мав би визначеного значення.

Приклад зі стеком демонструє одну з основних теоретичних ролей абстрактних типів даних — формування математичної моделі програми та її виконання. Складний вираз над АТД є математичним аналогом програми, а його послідовне спрощення за допомогою аксіом відповідає процесу виконання цієї програми. У такій моделі відсутні змінні, значення яких змінюються з часом, стани пам’яті та послідовність

машинних команд. Вона ґрунтується виключно на стандартних математичних правилах обчислення виразів.

Абстрактний тип даних STACK [G] також демонструє незалежність специфікації від типу елементів. Завдяки формальному узагальненому параметру G ті самі операції та аксіоми можуть використовуватися для стеків цілих чисел, рядків, банківських рахунків або інших об’єктів. Значення x1–x7 у наведеному прикладі можуть бути екземплярами будь-якого типу, оскільки правила поведінки стека не залежать від внутрішніх властивостей його елементів.

Отже, АТД STACK [G] є цілісним прикладом формального опису структури даних. Його специфікація визначає доступні операції, типи їхніх аргументів і результатів, основні правила поведінки та умови допустимого використання. Аксіоми дозволяють формально аналізувати послідовності операцій і визначати результати обчислень без звернення до фізичної реалізації. Це підтверджує, що абстрактні типи даних є не лише засобом опису структур даних, а й математичною основою моделювання програм та їх виконання.

4.5 Перехід від абстрактних типів даних до класів

Абстрактні типи даних забезпечують математичний спосіб опису структур даних незалежно від конкретних деталей їх реалізації. Проте під час створення програмного забезпечення необхідно не лише визначити доступні операції та їх властивості, а й реалізувати ці операції засобами певної мови програмування. Перехід від абстрактного типу даних до програмної реалізації приводить до поняття класу як основної модульної конструкції об’єктно-орієнтованого програмування.

Клас можна визначити як абстрактний тип даних, доповнений повною або частковою реалізацією. Абстрактний тип даних визначає, які операції доступні для відповідних об’єктів, які типи аргументів і результатів вони мають, а також яким властивостям повинна відповідати їх поведінка. Клас доповнює цю специфікацію конкретним способом зберігання даних і програмними засобами виконання визначених операцій.

Для створення класу на основі абстрактного типу даних необхідно поєднати три складові:

- специфікацію абстрактного типу даних, яка містить функції, аксіоми та передумови;
- вибір внутрішнього представлення об’єктів;
- реалізацію функцій абстрактного типу даних відповідними програмними операціями.

Перша складова визначає зовнішні властивості типу, тоді як дві інші стосуються його внутрішньої реалізації. Наприклад, специфікація АТД STACK [G] визначає операції put, remove, item, empty і new, не вказуючи, як саме зберігаються елементи.

Під час створення класу стека розробник може вибрати масивне або зв'язане представлення та реалізувати операції відповідно до обраної структури.

Для масивного представлення ARRAY_UP стек може бути реалізований за допомогою масиву representation і лічильника count. Абстрактна функція

put: STACK [G] × G → STACK [G]

у програмній реалізації перетворюється на операцію, яка додає елемент до поточного об'єкта стека:

```
put (x: G)
   -- Додати x на вершину стека.
do
   count := count + 1
   representation [count] := x
end
```

У формальній специфікації функція put повертає новий стек, оскільки математична модель не передбачає зміни існуючих об'єктів. У програмному класі відповідна операція може змінювати стан поточного об'єкта. Така відмінність пов'язана з переходом від математичного опису до практичної реалізації програмного забезпечення.

Використання абстрактних типів даних як основи класів також визначає поділ компонентів класу на відкриті та приховані. Специфікація АТД становить відкриту частину класу, доступну його клієнтам. Вона повідомляє, які операції можна виконувати та які результати вони забезпечують. Вибір представлення і спосіб реалізації операцій утворюють приховану частину класу та не повинні бути доступними програмним модулям, які використовують цей клас.

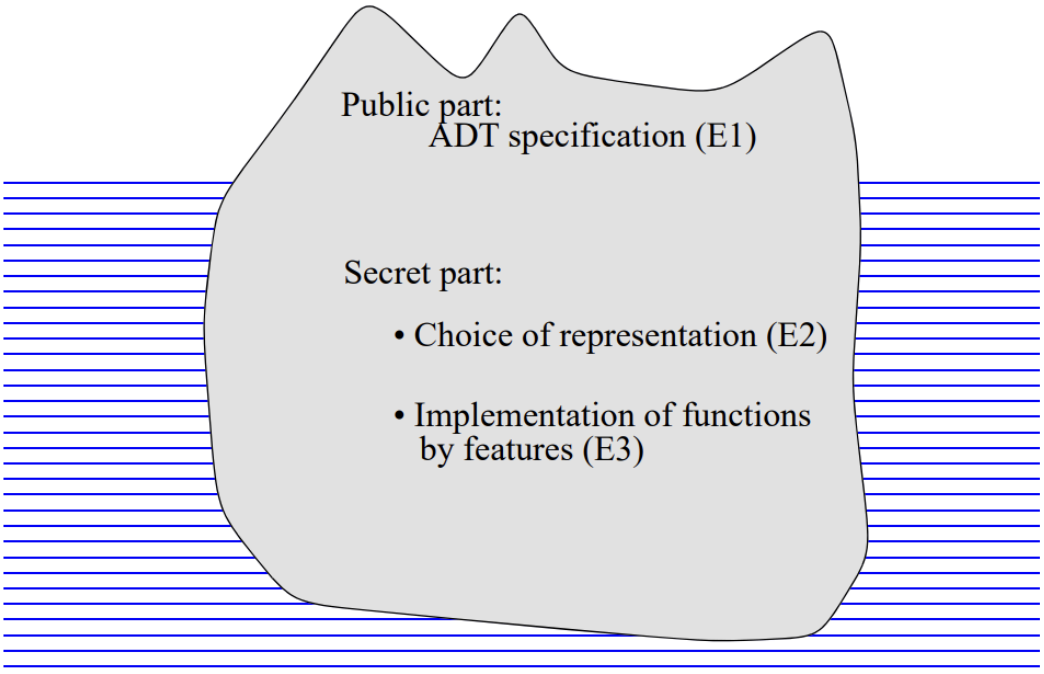


Рисунок 4.5 – Представлення класу з погляду абстрактного типу даних і приховування інформації

На рисунку 4.5 специфікація абстрактного типу даних утворює відкриту частину програмного модуля. Вибір внутрішнього представлення та реалізація функцій залишаються прихованими. Завдяки цьому клієнти класу залежать лише від його зовнішніх операцій, а не від способу зберігання та опрацювання даних.

Якщо інтерфейс класу залишається незмінним, внутрішнє представлення можна замінити без внесення змін до клієнтських програмних модулів. Наприклад, стек, спочатку реалізований за допомогою масиву, можна надалі реалізувати як зв’язану структуру. Для клієнтів класу ця зміна не повинна бути помітною, оскільки вони продовжують використовувати ті самі операції *put, remove, item і empty.*

Клас може містити повну або часткову реалізацію абстрактного типу даних. Повністю реалізований клас містить усі необхідні операції та конкретне представлення даних. Частково реалізований клас визначає спільну специфікацію та може залишати окремі операції без реалізації. У межах цієї лекції важливо лише зафіксувати, що класи можуть відрізнятися повнотою реалізації, тоді як детальні механізми їх розширення розглядатимуться окремо.

Отже, абстрактний тип даних визначає суттєві властивості певної множини об’єктів, а клас надає програмну реалізацію цієї специфікації. АТД відповідає на питання, які операції підтримує структура даних і якими властивостями вони характеризуються, тоді як клас додатково визначає, як ці операції реалізовано. Такий поділ дозволяє відокремити зовнішню поведінку програмного модуля від його внутрішньої будови, зменшити залежність між частинами системи та забезпечити можливість зміни реалізації без порушення роботи клієнтського коду.

Висновки

Абстрактні типи даних забезпечують точний і незалежний від реалізації спосіб опису структур даних через сукупність доступних операцій, їхні сигнатури, аксіоми та передумови. Такий підхід дозволяє зосередитися на суттєвих властивостях об’єктів, не пов’язуючи їх із конкретним способом зберігання даних у пам’яті. На прикладі стека показано, що одна структура може мати різні фізичні представлення, однак її зовнішня поведінка залишається незмінною та визначається встановленим набором операцій. Формальна специфікація АТД дає змогу однозначно описувати правила використання структури, перевіряти правильність послідовностей операцій і розглядати виконання програми як процес алгебраїчного спрощення виразів. Перехід від абстрактного типу даних до класу поєднує математичну специфікацію з конкретним або частковим програмним представленням, зберігаючи розділення між відкритим інтерфейсом і прихованою реалізацією. Отже, використання АТД створює

основу для побудови зрозумілих, модульних і придатних до супроводу програмних систем, у яких внутрішню реалізацію структур даних можна змінювати без порушення роботи клієнтського програмного коду.

Лекція 5

Тема: Управління пам'яттю

Мета: Ознайомити студентів із принципами розміщення та життєвого циклу програмних об’єктів у пам’яті, проблемами своєчасного вивільнення зайнятих ресурсів, ручними й автоматичними підходами до керування пам’яттю, а також механізмами підрахунку посилань і збирання сміття. Сформувати розуміння впливу способу керування пам’яттю на надійність, продуктивність і складність розроблення об’єктно-орієнтованого програмного забезпечення. Такий зміст відповідає темі, визначеній у робочій програмі дисципліни.

План лекції

1. Життєвий цикл об’єктів і способи розміщення даних у пам’яті
2. Проблема вивільнення пам’яті та ручне керування пам’яттю
3. Керування пам’яттю на рівні програмних компонентів
4. Автоматичне керування пам’яттю
5. Практичні аспекти й оптимізація автоматичного керування пам’яттю
6. Висновки

5.1 Життєвий цикл об’єктів і способи розміщення даних у пам’яті

Об’єктно-орієнтовані програми постійно створюють об’єкти. Динамічне створення дозволяє формувати гнучкі структури, які під час виконання програми пристосовуються до конкретних потреб системи. Проте пам’ять комп’ютера не є нескінченною, тому разом зі створенням об’єктів виникає необхідність визначати, де вони розміщуються, як довго існують і що відбувається з ними після завершення використання.

Життєвий цикл програмного об’єкта охоплює проміжок від моменту його створення до моменту, коли він перестає бути доступним програмі, а зайнята ним пам’ять може бути використана повторно. Тривалість цього циклу залежить від способу розміщення об’єкта, зв’язків з іншими об’єктами та механізмів керування пам’яттю, які підтримує мова програмування або середовище виконання.

Створення програмних об’єктів

Основна операція виділення пам’яті для нового об’єкта у використаній у книзі нотації має вигляд:

create x

Виконання цієї інструкції передбачає три дії:

- створення нового об’єкта;
- приєднання до нього посилання x;
- початкову ініціалізацію полів об’єкта.

Існують також варіанти інструкції створення, які викликають окрему процедуру ініціалізації. Нові об’єкти можуть утворюватися і шляхом копіювання вже наявних об’єктів. Однак усі такі операції зрештою ґрунтуються на базовому механізмі створення, тому для розгляду принципів керування пам’яттю достатньо зосередитися на простій інструкції create x.

У цьому контексті x є програмною сутністю, тобто ім’ям у тексті програми, яке під час виконання представляє певне значення або послідовність значень. До таких сутностей належать атрибути об’єктів, формальні аргументи процедур, локальні змінні та результат функції. Значенням сутності може бути безпосередньо об’єкт або посилання на об’єкт.

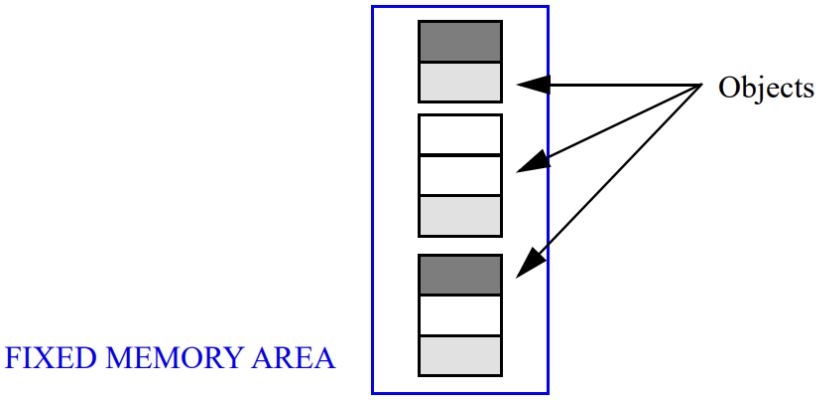
Сутність x вважається приєднаною до об’єкта O, якщо її значенням є сам об’єкт O або посилання на нього. Одна сутність посилального типу в конкретний момент може бути приєднана щонайбільше до одного об’єкта. Водночас до одного об’єкта можуть вести два або більше посилань. Така ситуація пов’язана з динамічним псевдонімуванням, коли один об’єкт доступний через декілька програмних сутностей.

Основні режими керування об’єктами

У програмних системах використовують три основні режими керування об’єктами:

- статичний режим;
- стековий режим;
- вільний, або динамічний, режим.

Вибір режиму визначає, коли виділяється пам’ять для об’єкта, як програмна сутність приєднується до нього та за яких умов зайнята ним пам’ять може бути вивільнена.



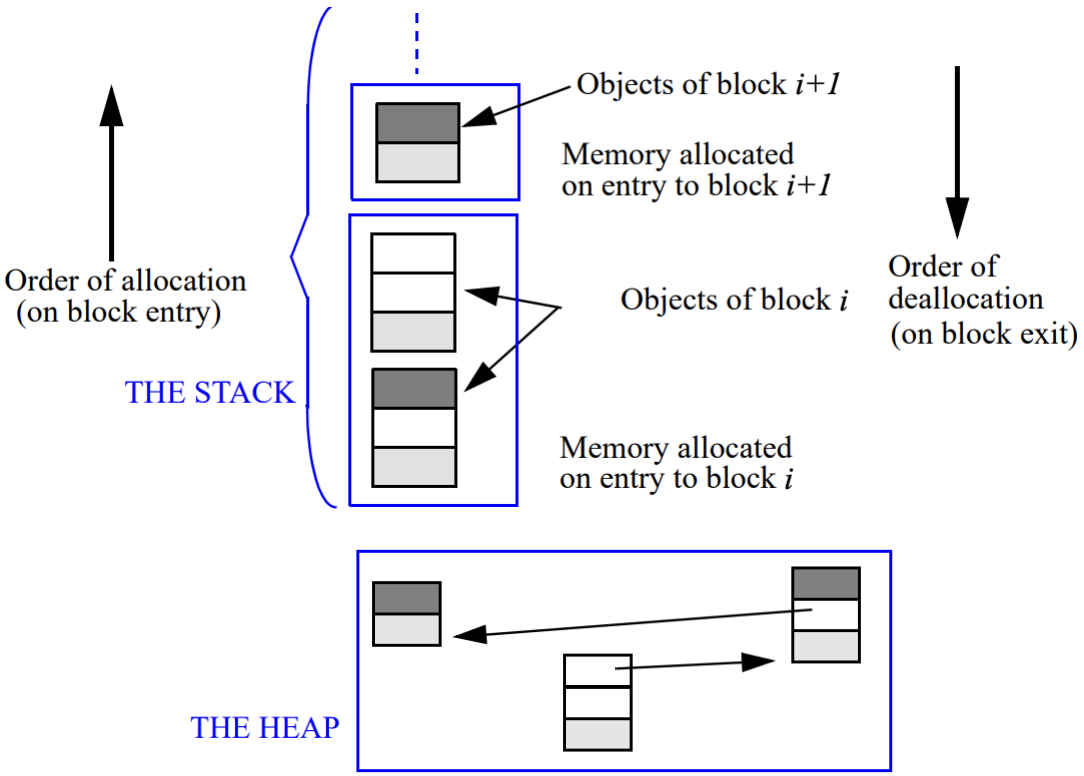


Рисунок 5.1 – Три режими керування об’єктами: статичний, стековий і вільний

У статичному режимі програмна сутність протягом усього виконання програми може бути приєднана щонайбільше до одного об’єкта. Пам’ять для всіх таких об’єктів виділяється один раз — під час завантаження програми або на початку її виконання. Об’єкти розміщуються у фіксованій області пам’яті та зберігаються там до завершення роботи програми.

Статичний режим є простим і забезпечує ефективну реалізацію на звичайних комп’ютерних архітектурах. Оскільки розмір і розташування всіх об’єктів відомі заздалегідь, середовище виконання не витрачає час на динамічне виділення та вивільнення пам’яті.

Водночас статичне розміщення має суттєві обмеження. Насамперед воно не підтримує рекурсію. Рекурсивна процедура може мати одночасно декілька активних викликів, кожен із власними локальними даними, що неможливо забезпечити за умови єдиного статичного набору об’єктів.

Статичний режим також не дозволяє повноцінно створювати структури даних, розмір яких визначається під час виконання програми. Компілятор повинен мати можливість визначити точний обсяг пам’яті для кожної структури лише на основі тексту програми. Наприклад, розмір масиву потрібно встановити до початку виконання.

Для структури, яка може збільшуватися або зменшуватися, доводиться резервувати максимально можливий обсяг пам’яті. Це призводить до неефективного використання ресурсів. Якщо ж максимально допустимий розмір було оцінено

неправильно, переповнення лише однієї структури може спричинити помилку виконання всієї програми.

У статичному режимі проблема вивільнення пам'яті фактично не виникає. Кожний об'єкт має відповідну програмну сутність і повинен залишатися в пам'яті, доки ця сутність існує. Пам'ять звільняється разом із завершенням роботи всієї програми.

Історично для зменшення використання пам'яті застосовувалося накладання областей пам'яті, коли двом змінним призначалася одна й та сама адреса за припущення, що вони не будуть потрібні одночасно. Проте ручне застосування цього підходу є небезпечним. Зміна програми може порушити початкове припущення та призвести до пошкодження даних.

Другим способом є стековий режим. У цьому випадку одна програмна сутність під час виконання може послідовно приєднуватися до декількох об'єктів. Об'єкти виділяються та вивільняються відповідно до принципу «останнім надійшов — першим вийшов» — Last In, First Out.

Під час входу до програмного блока або виклику процедури на вершині стека виділяється пам'ять для локальних даних. Коли виконання блока завершується, ця пам'ять вилучається зі стека. Після вивільнення об'єкта відповідна програмна сутність може знову представляти попереднє значення, якщо воно існувало.

Стекове розміщення підтримує рекурсію, оскільки кожний новий виклик процедури отримує власний набір локальних значень. Після завершення виклику відповідна область автоматично вилучається зі стека, а виконання повертається до попереднього виклику.

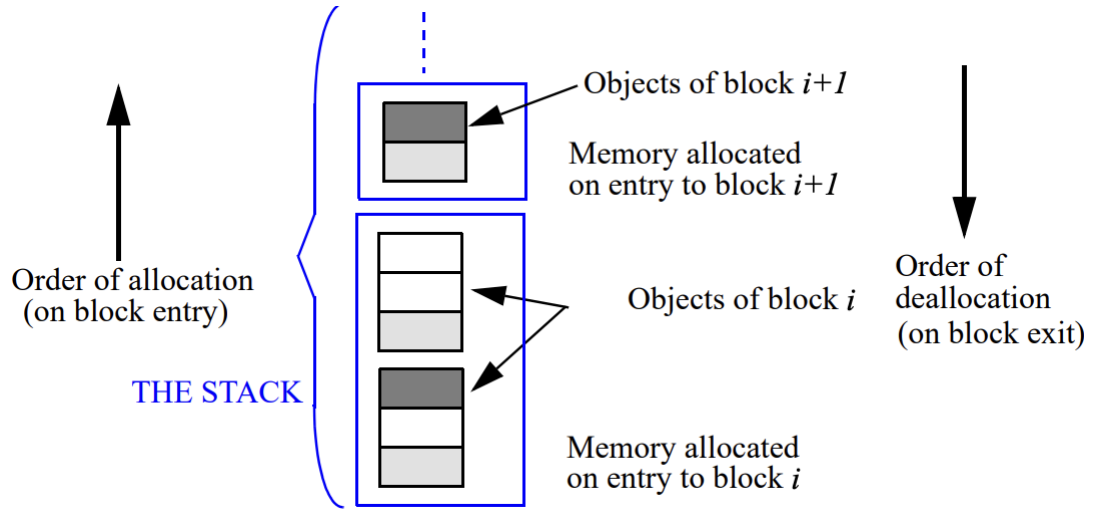


Рисунок 5.2 – Розміщення та вивільнення об'єктів у стеку

Для оформлення використати рисунок “The stack-based mode” з книги. На ньому показано порядок виділення пам’яті під час входу до блоків і зворотний порядок її вивільнення під час виходу.

У мовах із блоковою структурою стекове керування пам’яттю є особливо простим. Усі локальні значення, оголошені в одному блоці, можуть виділятися одночасно під час входу до блока та вивільнятися одночасно під час виходу:

Простота й ефективність цього механізму стали однією з причин поширення мов програмування з блоковою структурою. Середовище виконання не повинно окремо визначати момент завершення життєвого циклу кожного локального об’єкта, оскільки цей момент безпосередньо пов’язаний із завершенням відповідного блока або процедури.

Проте стековий режим не підтримує складні динамічні структури в повному обсязі. Порядок знищення об’єктів завжди повинен бути протилежним до порядку їх створення. Це зручно для локальних змінних і вкладених викликів процедур, але не підходить для об’єктів, життєві цикли яких не відповідають структурі викликів.

Для створення складних структур даних використовується вільний режим, який також називають динамічним або режимом розміщення в купі. У цьому режимі об’єкти створюються в довільні моменти виконання програми за допомогою явних запитів.

Одна програмна сутність може послідовно приєднуватися до будь-якої кількості об’єктів. Послідовність створення об’єктів зазвичай неможливо точно передбачити на етапі компіляції, оскільки вона залежить від умов виконання, вхідних даних і дій користувача.

Об’єкти, розміщені в купі, можуть містити посилання на інші об’єкти. Завдяки цьому можна створювати списки, дерева, графи та інші структури, форма і розмір яких змінюються під час роботи програми.

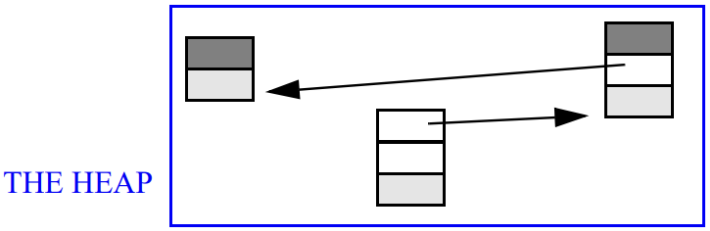


Рисунок 5.3 – Динамічне розміщення об’єктів у купі

Вільний режим є найзагальнішим і необхідним для повноцінних об’єктно-орієнтованих обчислень. Саме він дозволяє створювати об’єктні структури, які не обмежуються статично визначеним розміром або порядком, установленим стеком.

Різні мови можуть одночасно використовувати декілька режимів. Наприклад, частина даних розміщується статично, локальні значення процедур — у стеку, а динамічно створені об’єкти — у купі. Конкретне поєднання залежить від правил мови та реалізації середовища виконання.

Можливість динамічно створювати об’єкти приводить до питання про повторне використання зайнятої ними пам’яті. Коли об’єкт більше не потрібний програмі, його область пам’яті доцільно звільнити, щоб надалі використати для нових об’єктів.

У статичному режимі пам’ять зберігається протягом усього виконання програми, тому окреме вивільнення об’єктів не передбачається.

У стековому режимі момент завершення життєвого циклу об’єкта визначається структурою програми. Після виходу з блока або завершення процедури відповідна область вилучається зі стека. Отже, пам’ять вивільняється автоматично та в чітко визначеному порядку.

У вільному режимі ситуація є складнішою. Оскільки порядок створення об’єктів заздалегідь невідомий, неможливо лише на основі моменту створення визначити, коли об’єкт перестане бути потрібним. Об’єкт може залишатися доступним через інші посилання навіть після завершення процедури, у якій його було створено.

У вільному режимі об’єкти можуть ставати непотрібними в непередбачувані моменти виконання. Причиною цього є операції від’єднання, які є протилежними до операцій приєднання.

Від’єднання стосується програмних сутностей посилального типу. Якщо сутність містить об’єкт безпосередньо, відокремити її від цього об’єкта неможливо. Однак більший об’єкт, частиною якого є таке значення, може сам стати недосяжним.

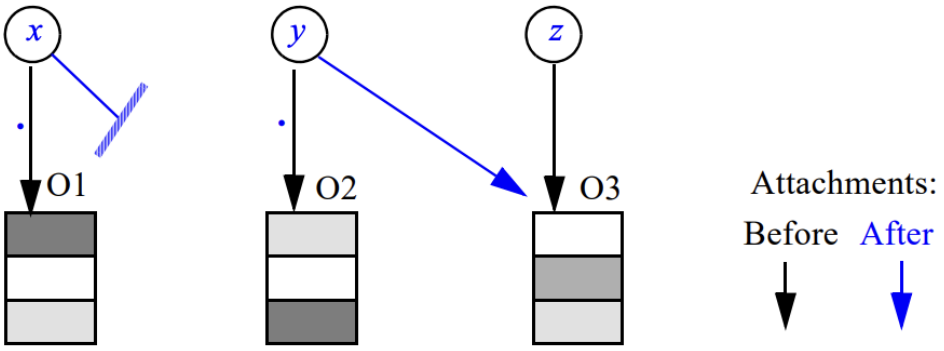


Рисунок 5.4 – Від’єднання програмних сутностей від об’єктів

Нехай посилальні сутності x і y спочатку приєднані відповідно до об’єктів O1 та O2. Основними причинами від’єднання є:

- присвоєння x := Void або присвоєння змінній x іншого порожнього посилання;

- присвоєння y := z, якщо z посилається на об’єкт, відмінний від O2;
- завершення виконання процедури, унаслідок чого її формальні аргументи та локальні сутності перестають існувати;
- виконання інструкції create x, яка приєднує x до нового об’єкта і водночас від’єднує його від попереднього.

Від’єднання посилання не обов’язково означає завершення життєвого циклу об’єкта. Через динамічне псевдонімування до цього об’єкта можуть залишатися приєднаними інші посилання. Тому втрата одного зв’язку не є достатньою підставою для звільнення пам’яті.

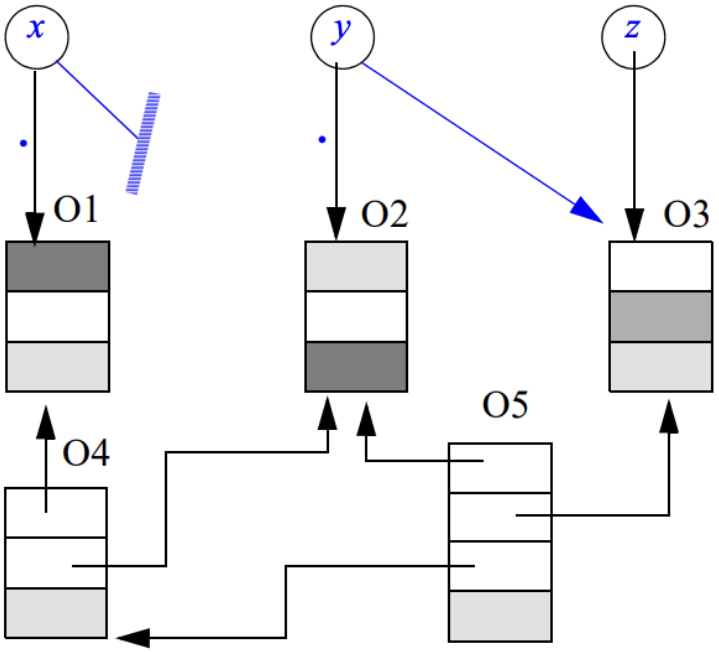


Рисунок 5.5 – Від’єднання не завжди означає недосяжність об’єкта

Для оформлення використати рисунок “Detachment is not always death” зі сторінки 284. Схема демонструє, що після від’єднання x та y об’єкти O1 і O2 можуть залишатися доступними через об’єкти O4 і O5.

На ширшій схемі може виявитися, що до від’єднаного об’єкта веде інше посилання. Проте й цього недостатньо для остаточного висновку: об’єкти, через які проходять такі посилання, також можуть бути недосяжними. Тому можливість вивільнення пам’яті визначається не локальним аналізом окремої змінної, а аналізом усієї структури об’єктів.

Для визначення об’єктів, пам’ять яких може бути звільнена, усі створені об’єкти поділяють на три категорії:

- об’єкти, безпосередньо приєднані до програмних сутностей, які в поточний момент потрібні для виконання;

- прямі й непрямі залежні об’єкти першої категорії, до яких можна перейти через посилання;
- об’єкти, що не належать до жодної з двох попередніх категорій.

Об’єкти першої категорії називаються об’єктами-джерелами, або початковими об’єктами. Разом зі своїми прямими та непрямими залежними об’єктами вони утворюють множину досяжних об’єктів.

Об’єкт є досяжним, якщо до нього можна перейти від одного з об’єктів-джерел через послідовність посилань. Такий об’єкт потенційно може бути використаний програмою, тому пам’ять, яку він займає, не можна звільняти.

Об’єкт є недосяжним, якщо він не є об’єктом-джерелом і до нього неможливо перейти від жодного джерела. Недосяжний об’єкт більше не може впливати на подальше виконання програми. Отже, зайнята ним пам’ять може бути вивільнена та використана для створення інших об’єктів.

Для досяжних і недосяжних об’єктів також використовують неформальні назви «живі» та «мертві» об’єкти. Проте можливість звільнення пам’яті визначається не тим, чи об’єкт активно використовується в певний момент, а тим, чи зберігається хоча б один шлях доступу до нього.

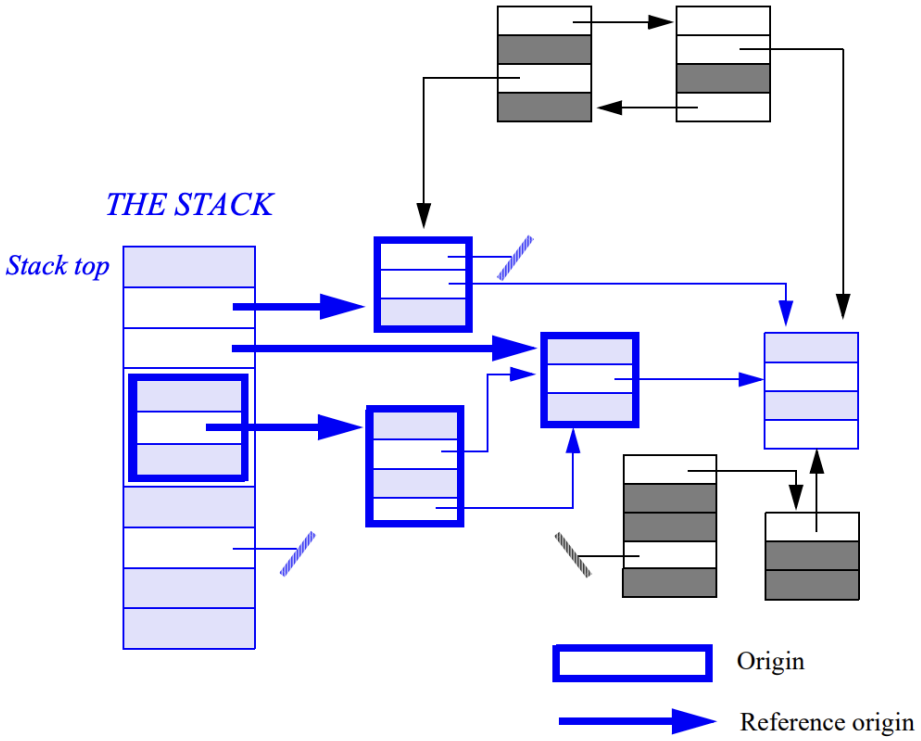


Рисунок 5.6 – Досяжні та недосяжні об’єкти в системі зі стековим і динамічним розміщенням

На рисунку досяжні об’єкти разом з об’єктами-джерелами виділені окремо, а недосяжні показані як ізольована частина структури. Для визначення досяжності

необхідно почати з посилань, розташованих у стеку, і послідовно переходити за всіма полями посилального типу.

У класичній моделі, яка поєднує стековий і вільний режими, джерелами доступу до об’єктів купи є посилання, розміщені у стеку. До них належать:

- значення локальної змінної або аргументу процедури посилального типу;
- поле посилального типу всередині складеного об’єкта, розміщеного у стеку.

Наприклад, під час виклику процедури у стеку можуть бути виділені ціле значення, складений об’єкт і посилальна змінна. Ціле значення не посилається на інші об’єкти та автоматично зникає після завершення процедури. Посилальна змінна може бути джерелом доступу до об’єкта в купі. Складений об’єкт також може містити поле-посилання, від якого починається шлях до інших об’єктів.

```
procedure p
    local
        n: INTEGER
        c: COMPOSITE
        s: reference COMPOSITE
    do
        …
    end
```

Every execution of *p* allocates three values on the stack:

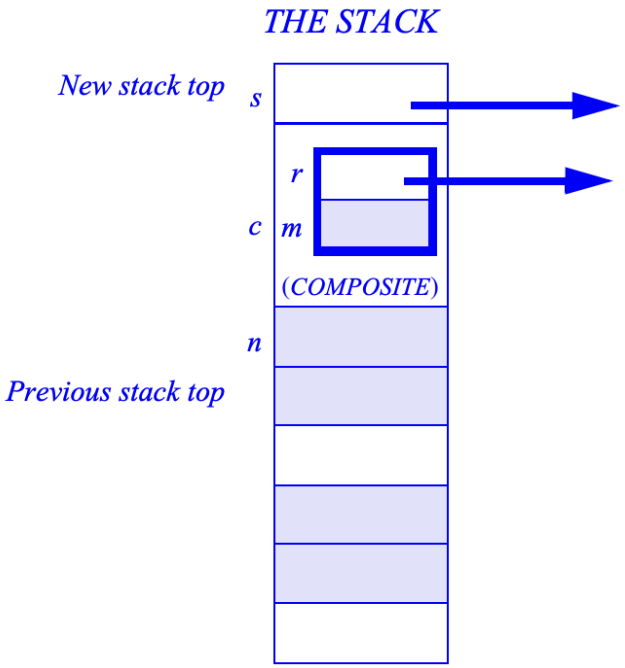


Рисунок 5.7 – Розміщення сутностей процедури у стеку

Отже, у моделі, що поєднує стек і купу, визначення досяжних об’єктів починається з посилань у стеку. Після цього середовище виконання послідовно переходить за посилальними полями всіх знайдених об’єктів.

Виконання об’єктно-орієнтованої системи починається зі створення кореневого об’єкта. Цей об’єкт є першим джерелом досяжності та забезпечує початкову точку виконання програми.

Додатковими джерелами є об’єкти, приєднані до локальних сутностей і формальних аргументів процедур, які виконуються в поточний момент. Поки процедура активна, її інструкції можуть звертатися до таких об’єктів, тому вони обов’язково належать до множини досяжних.

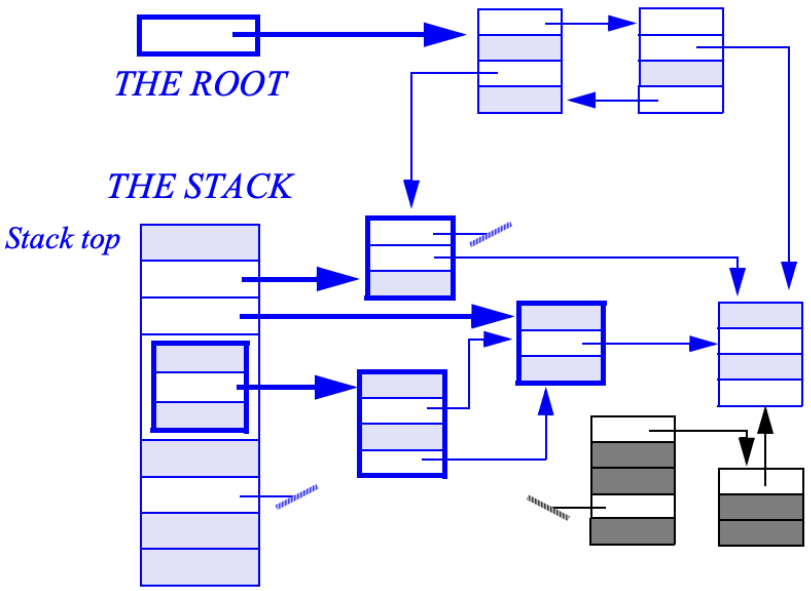


Рисунок 5.8 – Досяжність об’єктів в об’єктно-орієнтованій моделі

Після завершення процедури її локальні сутності та формальні аргументи перестають існувати. Унаслідок цього приєднані до них об’єкти перестають бути об’єктами-джерелами. Проте вони не обов’язково відразу стають недосяжними. Під час виконання процедури посилання на них могло бути записане в поле кореневого або іншого досяжного об’єкта.

Розглянемо процедуру, яка створює два об’єкти BOOK3. Посилання на перший із них записується в атрибут a поточного об’єкта, а другий залишається приєднаним лише до локальної змінної. Після завершення процедури перший об’єкт залишається досяжним через атрибут a. Другий об’єкт стає недосяжним, оскільки локальна змінна зникає і жодного іншого шляху доступу до нього немає.

Аналогічно об’єкт розгорнутого типу, який був локальним значенням процедури, перестає існувати після завершення виклику, якщо він не є частиною іншого досяжного об’єкта.

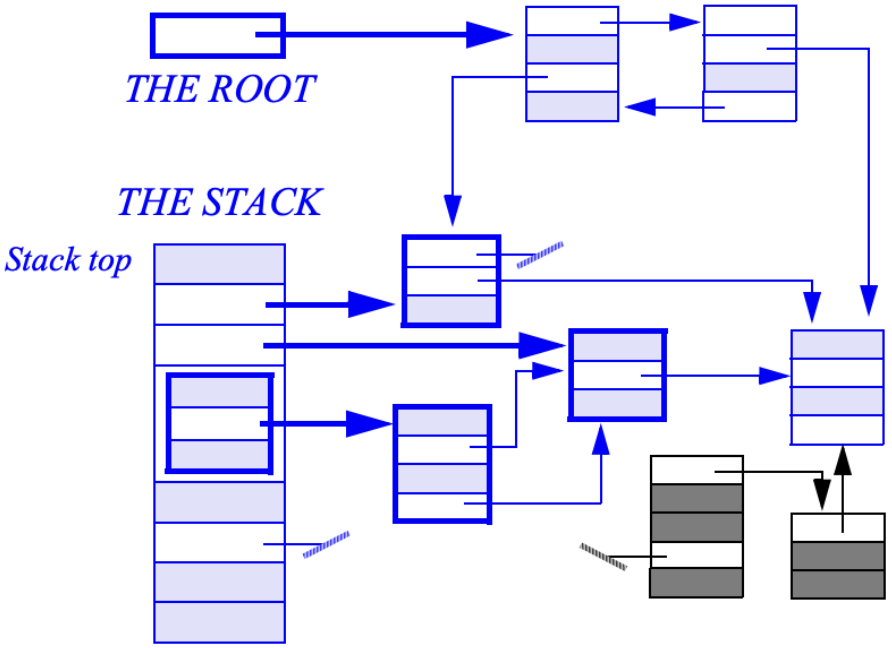


Рисунок 5.9 – Об’єкти, приєднані до локальних сутностей процедури

Узагальнюючи, у певний момент виконання об’єктно-орієнтованої системи множину об’єктів-джерел утворюють:

- кореневий об’єкт системи;
- об’єкти, приєднані до локальних сутностей і формальних аргументів процедур, які виконуються в цей момент;
- об’єкт, приєднаний до спеціальної локальної сутності Result під час виконання функції.

Будь-який прямий або непрямий залежний об’єкт цих джерел є досяжним. Усі інші об’єкти є недосяжними, і зайняту ними пам’ять можна звільнити без зміни правильної поведінки програми.

Проблема керування пам’яттю в об’єктно-орієнтованій системі виникає через непередбачуваність операцій створення та від’єднання об’єктів. Вони виконуються всередині умов, циклів, процедур і інших керувальних конструкцій, тому зазвичай неможливо лише за текстом програми точно встановити, скільки об’єктів буде створено, коли вони втратять усі шляхи доступу і в який момент зайнята ними пам’ять може бути використана повторно.

Отже, статичний, стековий і вільний режими по-різному визначають життєвий цикл об’єктів. Статичні об’єкти існують упродовж усього виконання програми, стекові — до завершення відповідного блока або процедури, а життєвий цикл об’єктів

у купі залежить від наявності шляхів доступу до них. Саме необхідність своєчасно виявляти недосяжні об’єкти в купі формує основну проблему керування пам’яттю, яка розглядатиметься в наступних підпунктах.

5.2 Проблема вивільнення пам’яті та ручне керування пам’яттю

Динамічне створення об’єктів забезпечує гнучкість програмних систем, однак пам’ять, зайнята об’єктами, які стали недосяжними, повинна з часом повертатися до множини вільних ресурсів. Якщо цього не робити, програма продовжуватиме накопичувати непотрібні об’єкти, хоча жодна її частина вже не може звернутися до них. У результаті доступна пам’ять поступово вичерпуватиметься.

Після того як об’єкт стає недосяжним, пам’ять, яку він займає, можна використати для розміщення нових об’єктів без впливу на правильну поведінку програми. Процес повернення такої пам’яті називають її вивільненням або повторним використанням. Основна складність полягає не лише у фізичному звільненні певної області пам’яті, а передусім у правильному визначенні моменту, коли об’єкт справді більше не може бути використаний.

Для недосяжних об’єктів можливі три загальні підходи:

- ігнорувати проблему та припускати, що пам’яті вистачить для всіх створених об’єктів;
- покласти пошук недосяжних об’єктів і вивільнення пам’яті на розробника;
- включити до середовища виконання автоматичні механізми виявлення та вивільнення недосяжних об’єктів.

Перший варіант у книзі названо недбалим підходом (casual approach). Він полягає в тому, щоб створювати об’єкти за потреби та не турбуватися про подальшу долю тих із них, які стали недосяжними. Такий підхід може бути допустимим для невеликих тестів і експериментів, у яких створюється незначна кількість об’єктів. Він також може застосовуватися в системах, для яких можна довести, що всі або майже всі створені об’єкти залишатимуться досяжними протягом усього виконання. У цьому разі фактично немає пам’яті, яку можна було б повернути.

Подібний підхід іноді використовують у системах жорсткого реального часу. Заради передбачуваності вони можуть створювати всі необхідні об’єкти статично або під час початкової ініціалізації, уникаючи непередбачуваного динамічного виділення пам’яті під час основної роботи. Однак це обмежує програмну систему та підходить лише для спеціальних сфер, у яких час виконання кожної операції повинен бути заздалегідь визначеним.

Збільшення обсягу фізичної та віртуальної пам’яті не усуває необхідності її вивільнення. Навіть великі обсяги пам’яті мають межі, а накопичення недосяжних об’єктів поступово зменшує кількість ресурсів, доступних для корисних даних.

Віртуальна пам’ять також не є повноцінною заміною оперативної пам’яті. Якщо в ній зберігається велика кількість об’єктів, серед яких лише невелика частина залишається досяжною, система змушена постійно переміщувати сторінки між оперативною пам’яттю та зовнішнім сховищем. Це явище називається трешингом (thrashing) і призводить до різкого погіршення швидкодії програми.

Апаратний розвиток не виправдовує марнотратного використання ресурсів. Збільшення обсягу пам’яті повинно використовуватися для розширення можливостей програмного забезпечення, а не для зберігання об’єктів, які більше ніколи не будуть потрібні. Тому після виходу за межі невеликих або строго контрольованих систем проблема вивільнення пам’яті стає неминучою.

Якщо програма створює об’єкти, які згодом стають недосяжними, але зайнята ними пам’ять не повертається для повторного використання, виникає витік пам’яті (memory leak). Окремий витік може бути дуже малим, однак за багаторазового виконання певної операції його наслідки накопичуються.

Причинами витоків пам’яті можуть бути:

- відсутність операції вивільнення після завершення використання об’єкта;
- втрата останнього посилання на об’єкт без повернення його пам’яті;
- неповне вивільнення складеної структури об’єктів;
- помилка в рідко виконуваній гілці програми;
- зміна програмного коду, після якої раніше правильна логіка керування пам’яттю стає неповною;
- складність відстеження всіх об’єктів, що створюються під час тривалої роботи системи.

Наслідки витоку можуть проявитися не відразу. Програма здатна успішно працювати під час коротких тестів, але поступово споживати дедалі більше пам’яті протягом годин, днів або тижнів. У певний момент вона може значно сповільнитися, перестати створювати нові об’єкти або аварійно завершити роботу.

У книзі наведено випадок інформаційної системи Лондонської служби швидкої допомоги. В офіційному звіті зазначалося, що в програмному коді залишилася помилка, через яку під час кожної операції мобілізації транспортного засобу використовувалася невелика частина пам’яті, яка надалі не звільнялася. Протягом приблизно трьох тижнів система поступово вичерпала всю доступну пам’ять і припинила роботу. У звіті також наголошувалося, що через характер цієї помилки її навряд чи вдалося б виявити звичайним тестуванням програмістами або користувачами. Цей приклад показує, що навіть незначний локальний витік може мати серйозні наслідки для довготривалої роботи критичної системи.

Проблема повернення пам’яті містить два окремі завдання:

- виявлення (detection) — визначення об’єктів, які стали недосяжними;

- вивільнення (reclamation) — фактичне повернення зайнятих ними областей пам’яті для подальшого використання.

Кожне з цих завдань може виконуватися на різних рівнях:

- на рівні реалізації мови програмування — компілятором і середовищем виконання;
- на рівні прикладної програми — безпосередньо її розробниками;
- на рівні повторно використовуваних програмних компонентів — авторами бібліотечних класів.

Ручне керування зазвичай поєднує автоматизоване фізичне вивільнення пам’яті з ручним визначенням моменту, коли його потрібно виконати. Середовище виконання надає спеціальну операцію для повернення області пам’яті, але саме програміст повинен вирішити, який об’єкт уже не потрібний і коли цю операцію можна безпечно застосувати.

Ручне керування пам’яттю приваблює розробників можливістю безпосередньо контролювати життєвий цикл об’єктів. Програміст сам визначає момент їх створення та знищення, а середовище виконання не витрачає додаткові ресурси на автоматичний пошук недосяжних об’єктів. Для реалізаторів мови такий підхід також є порівняно простим: достатньо надати примітивну операцію, яка повідомляє системі, що певний об’єкт більше не потрібний і зайняті ним комірки можна повторно використати.

Однак простота механізму не означає простоти його правильного застосування. Щоб безпечно звільнити об’єкт, необхідно встановити, що до нього не залишилося жодного прямого або непрямого шляху доступу. Для складних структур даних це може вимагати аналізу значної частини об’єктної структури програми.

Ручне керування створює дві протилежні небезпеки. Якщо програміст не звільнить недосяжний об’єкт, виникне витік пам’яті. Якщо ж він звільнить об’єкт занадто рано, у системі можуть залишитися посилання на область пам’яті, яка більше не містить цього об’єкта.

Помилкова операція вивільнення особливо небезпечна за наявності складних структур і декількох посилань на один об’єкт. Програміст може перевірити одне посилання, вирішити, що об’єкт більше не використовується, і повернути його пам’ять. Водночас інший компонент системи може зберігати ще одне посилання на той самий об’єкт.

Операція вивільнення, яка була правильною в початковій версії програми, може стати помилковою після її розвитку. Наприклад, нова функціональність може зберігати додаткове посилання на об’єкт, тоді як старий код керування пам’яттю продовжить звільняти його за попередніми правилами.

У великих програмах неможливо покладатися лише на уважність окремого розробника. Під час виконання можуть існувати тисячі або мільйони об’єктів,

пов’язаних великою кількістю посилань. Відстеження всіх цих зв’язків є завданням, яке краще виконує комп’ютер, а не людина.

Якщо об’єкт було звільнено, але в іншому об’єкті або змінній залишилося посилання на нього, виникає висяче посилання (dangling reference). Формально посилання продовжує містити певну адресу, однак об’єкта, для якого ця адреса була призначена, більше не існує.

Після вивільнення відповідна область пам’яті може бути використана для розміщення зовсім іншого об’єкта або інших даних. Тому спроба перейти за висячим посиланням може мати різні наслідки:

- негайне аварійне завершення програми;
- пошкодження даних;
- отримання випадкових або застарілих значень;
- зміна іншого об’єкта, який був розміщений у тій самій області пам’яті;
- нестабільна поведінка, що змінюється між різними запусками.

Помилки цього типу є особливо складними для пошуку. Причина помилки та її зовнішній прояв можуть бути значно віддалені в часі й у структурі програмного коду. Неправильне вивільнення може відбутися в одному модулі, а аварія — набагато пізніше в іншій частині системи, коли програма спробує використати збережене посилання.

Відтворення такої помилки також може залежати від способу, яким операційна система розміщує нові об’єкти в пам’яті. В одному запуску звільнена область ще може містити старі дані, через що помилка тимчасово не проявиться. В іншому запуску цю область буде відразу використано повторно, і поведінка програми зміниться.

Спроба використати об’єкт після вивільнення зайнятої ним пам’яті утворює помилку повторного доступу до звільненої пам’яті. Вона може виникати під час читання поля, виклику операції або зміни стану об’єкта через висяче посилання.

Особлива небезпека полягає в тому, що програма не завжди завершується негайно. Якщо звільнена область ще не була перезаписана, операція може випадково повернути очікуване значення. Це створює хибне враження правильності програми. Пізніше, після повторного використання тієї самої пам’яті, аналогічна операція може спричинити аварію або пошкодити дані.

Отже, ручне вивільнення не лише вимагає визначити момент, коли об’єкт більше не потрібний, а й гарантувати, що жодна частина програми надалі не скористається посиланням на нього.

Навіть якщо програміст правильно визначив, що певний об’єкт можна звільнити, повернення лише його власної області пам’яті часто є недостатнім. Об’єкт може містити посилання на інші об’єкти, які після його видалення також стануть недосяжними.

У такому разі необхідно розглянути:

- об’єкти, на які він посилається безпосередньо;
- їхні залежні об’єкти;
- усі наступні рівні об’єктної структури;
- можливі зовнішні посилання на кожний із цих об’єктів.

Пам’ять залежних об’єктів також потрібно звільнити, але лише за умови, що вони не залишаються доступними через інші частини програми. Це утворює проблему рекурсивного вивільнення (recursive dispose): операція повинна охоплювати не один об’єкт, а цілу структуру його прямих і непрямих залежностей.

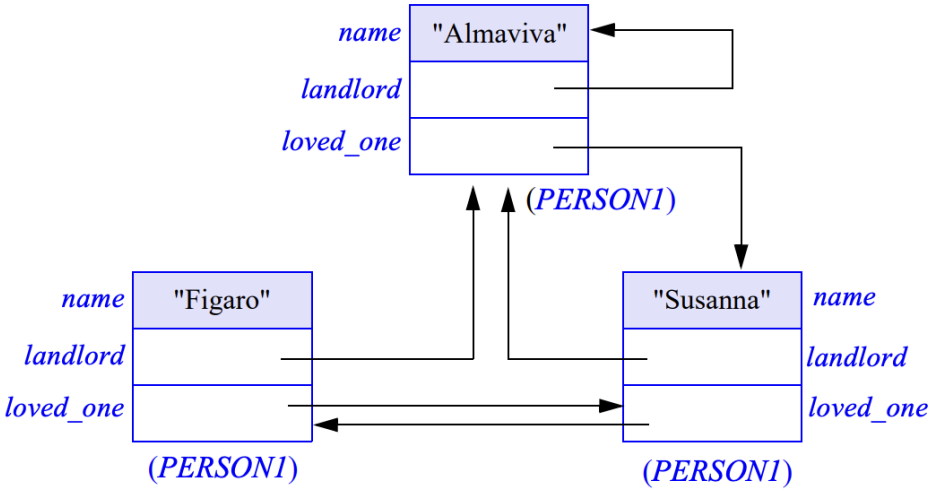


Рисунок 5.10 – Прямі та непрямі зв’язки між об’єктами під час вивільнення пам’яті

На рисунку об’єкти пов’язані прямими та непрямими посиланнями. Вивільнення верхнього об’єкта може зробити недосяжними два інші об’єкти, однак перед їх видаленням необхідно переконатися, що до них не ведуть зовнішні посилання. Саме наявність таких зв’язків робить ручне вивільнення складених структур трудомістким і схильним до помилок.

Для кожного типу, об’єкти якого можуть посилатися на інші об’єкти, розробникові довелося б створювати спеціальну процедуру вивільнення. Оскільки різні типи взаємопов’язані, такі процедури можуть стати взаємно рекурсивними та дуже складними. У результаті значна частина прикладної програми й зусиль розробників витрачатиметься не на розв’язання завдань предметної області, а на облік створених об’єктів і ручне повернення пам’яті.

Ручне керування пам’яттю збільшує обсяг і складність програмного коду. До основної логіки системи додаються операції вивільнення, перевірки стану посилань, обхід залежних об’єктів і допоміжні процедури керування ресурсами.

Це негативно впливає на:

- читабельність програмного коду;
- можливість виявлення помилок;
- простоту внесення змін;
- повторне використання компонентів;
- надійність системи;
- тривалість розроблення та тестування.

Підвищення складності додатково погіршує надійність: що складніша система, то вища ймовірність помилкового вивільнення або пропущеного недосяжного об’єкта. Особливо часто такі проблеми проявляються після переходу від тестування до промислової експлуатації, коли система починає створювати більші й складніші структури об’єктів.

Явне керування може бути прийнятним у вузьких і строго контрольованих ситуаціях, де правила створення та знищення об’єктів є простими й повністю відомими. Проте для загального прикладного програмного забезпечення покладання всієї відповідальності на розробника значно підвищує ризик витоків пам’яті, висячих посилань і повторного доступу до вже звільнених областей.

Отже, ручне керування пам’яттю надає програмісту безпосередній контроль над моментом її вивільнення, але водночас вимагає точного обліку всіх об’єктів та зв’язків між ними. Пропущене вивільнення спричиняє витік пам’яті, а передчасне — появу висячих посилань і небезпечний доступ до вже звільнених областей. Складність правильного керування великими об’єктними структурами робить ручний підхід ненадійним для більшості програмних систем і обґрунтовує необхідність компонентних або автоматичних механізмів вивільнення пам’яті.

5.3 Керування пам’яттю на рівні програмних компонентів

Перед переходом до автоматичного збирання сміття доцільно розглянути проміжний підхід, який дає змогу уникнути частини недоліків ручного керування пам’яттю. Його сутність полягає в тому, що відповідальність за створення, повторне використання та вивільнення об’єктів передається не прикладній програмі, а окремому повторно використовуваному класу. Такий підхід називають керуванням пам’яттю на рівні програмних компонентів.

Компонентний підхід застосовний передусім в об’єктно-орієнтованому проєктуванні, де структури даних не створюються безпосередньо всередині прикладної програми, а реалізуються у вигляді універсальних класів. Такі класи є програмними реалізаціями абстрактних типів даних і містять усі операції, необхідні для роботи з відповідними структурами.

У цьому разі між розробниками прикладної програми та розробниками компілятора або середовища виконання з’являється третя група — розробники повторно використовуваних компонентів. Вони відповідають за класи, що реалізують

основні структури даних. Оскільки розробник компонента контролює всі способи створення і видалення об’єктів певного класу, він може організувати керування пам’яттю для всіх екземплярів цього класу.

Якщо порядок створення та видалення об’єктів є достатньо простим, компонент може повторно використовувати пам’ять без спеціального механізму автоматичного збирання сміття. Для цього об’єкти, які більше не входять до активної структури, не знищуються відразу, а зберігаються у внутрішньому сховищі та надалі використовуються повторно.

Розглянемо клас LINKED_LIST, який описує зв’язаний список із заголовком і довільною кількістю зв’язаних комірок. Кожна комірка є об’єктом класу LINKABLE.

Для такого списку порядок виділення та вивільнення пам’яті є відносно простим. Нові комірки створюються лише операціями вставлення, а перестають використовуватися лише внаслідок операцій видалення. Отже, клас LINKED_LIST точно контролює обидва процеси.

Припустимо, що клас має дві основні операції вставлення:

- put_right — вставлення нового елемента праворуч від поточної позиції;
- put_left — вставлення нового елемента ліворуч від поточної позиції.

Кожна з цих операцій потребує однієї нової комірки LINKABLE. За звичайного підходу під час кожного вставлення створюється новий об’єкт:

create new.make (v)

active.put_linkable_right (new)

Перша інструкція виділяє пам’ять для нової комірки зі значенням v, а друга приєднує її до списку.

Клас також може мати операції видалення:

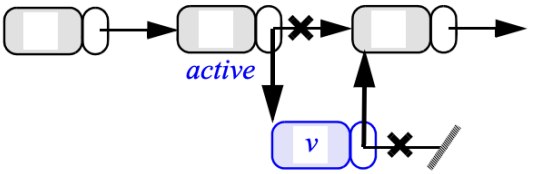
- remove;
- remove_right;
- remove_left.

Кожна така операція робить одну комірку недосяжною для активного списку. Інші операції, наприклад видалення всіх входжень певного значення або очищення списку, можуть бути побудовані на основі цих базових процедур.

Оскільки всі місця створення та видалення комірок відомі, компонент може зберігати вилучені об’єкти для подальшого повторного використання.

```
put_right (v: ELEMENT_TYPE)
        -- Insert an element of value v to the right of cursor position.
    require
        …
    local
        new: LINKABLE
    do
        create new.make (v)
        active. put_linkable_right (new)
        … Instructions to update other links …
    end

remove
        -- Delete element at cursor position.
    do
        …
        previous. put_linkable_right (next)
        … Instructions to update other links …
        active := next
    end
```



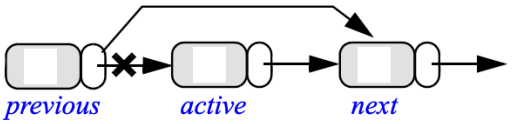


Рисунок 5.11 – Керування пам’яттю для зв’язаного списку

Вилучені зі списку комірки можна зберігати у внутрішній структурі available. Вона містить об’єкти LINKABLE, які більше не входять до активного списку, але можуть бути використані під час наступного вставлення.

Замість безпосереднього створення нового об’єкта операції put_right і put_left викликають внутрішню функцію:

new := fresh (v)

Функція fresh перевіряє сховище available. Якщо воно порожнє, створюється новий об’єкт. Якщо в ньому є раніше вилучена комірка, вона береться зі сховища, отримує нове значення та повторно включається до списку:

```
fresh (v: ELEMENT_TYPE): LINKABLE
    do
        if available.empty then
            create Result.make (v)
        else
            Result := available.item
            available.remove
            Result.put (v)
        end
    end
```

Операції видалення виконують протилежну дію. Перед вилученням комірки з активної структури вони передають її внутрішній процедурі recycle:

recycle (dead: LINKABLE)

```
require
   dead /= Void
do
   available.put (dead)
end
```

Процедури fresh і recycle є прихованими компонентами класу. Вони не повинні бути доступними клієнтам, оскільки використовуються лише для внутрішнього керування пам'яттю.

Сховище available можна реалізувати як стек. Процедура recycle додає вилучену комірку на вершину стека, а функція fresh забирає верхню комірку для повторного використання.

Для зв'язування вільних комірок можна використати вже наявне поле right класу LINKABLE. Завдяки цьому окремі поля або додаткові об'єкти для організації сховища не потрібні. Комірки, які не входять до активного списку, утворюють власний зв'язаний ланцюг:

available → LINKABLE → LINKABLE → LINKABLE

Під час вставлення перша комірка вилучається зі сховища і повертається до активного списку. Під час видалення комірка забирається з активного списку та додається до available.

Якщо available є звичайним атрибутом класу, кожний зв'язаний список матиме власне сховище вільних комірок. За наявності в системі кількох списків ефективніше використовувати спільний пул об'єктів, доступний усім екземплярам відповідного класу. Це дозволяє комірці, вилученій з одного списку, надалі використовуватися в іншому.

Головна перевага полягає в локалізації відповідальності за пам'ять. Прикладна програма працює лише зі звичайними операціями списку і не повинна окремо створювати, звільняти або повторно використовувати його комірки.

Основними перевагами є:

- зосередження операцій керування пам'яттю в одному класі;
- приховування цих операцій від клієнтів;
- зменшення кількості фактичних виділень пам'яті;
- повторне використання раніше створених об'єктів;
- зменшення ризику витоків пам'яті та висячих посилань у клієнтському коді;
- можливість застосовувати компонент без змін у різних програмах.

Виявлення об'єктів, які більше не використовуються, у цьому випадку також є локальним. Клас точно знає, що комірка стає вільною під час виконання операції видалення. Тому немає потреби аналізувати всю структуру об'єктів програми.

Компонентний підхід не є загальним розв’язанням проблеми керування пам’яттю. Він працює лише тоді, коли клас повністю контролює створення та припинення використання своїх внутрішніх об’єктів.

Підхід стає непридатним, якщо:

- об’єкти створюються за межами компонента;
- клієнти можуть зберігати посилання на внутрішні об’єкти;
- один об’єкт одночасно належить кільком структурам;
- неможливо точно визначити момент, коли об’єкт більше не використовується;
- структура зв’язків між об’єктами є складною та змінюється непередбачувано.

Наприклад, вилучена зі списку комірка може бути повторно використана лише за умови, що жодний клієнт не зберігає посилання на неї. Якщо такі посилання дозволені, передавання комірки до available може спричинити помилковий доступ до об’єкта, який уже використовується в іншому місці.

Отже, керування пам’яттю на рівні компонентів дозволяє перенести відповідальність із прикладних програм до повторно використовуваних класів. Для структур із простим і повністю контрольованим життєвим циклом, таких як внутрішні комірки зв’язаного списку, вилучені об’єкти можна зберігати у списку вільних елементів і повторно використовувати під час наступних операцій вставлення. Це зменшує кількість виділень пам’яті та спрощує клієнтський код, однак не замінює автоматичного керування пам’яттю в системах зі складними й непередбачуваними зв’язками між об’єктами.

5.4 Автоматичне керування пам’яттю

Розглянуті раніше підходи не забезпечують повного розв’язання проблеми керування пам’яттю. Ручне вивільнення покладає надмірну відповідальність на програміста, а керування на рівні компонентів застосовне лише до структур, у яких порядок створення та припинення використання об’єктів є простим і наперед відомим. Загальне розв’язання потребує підтримки на рівні реалізації мови програмування та середовища виконання.

Середовище об’єктно-орієнтованого програмування повинно автоматично виявляти недосяжні об’єкти та повертати зайняту ними пам’ять. Завдяки цьому розробник може зосередитися на реалізації прикладної логіки, а не на постійному відстеженні життєвого циклу кожного створеного об’єкта.

Автоматичне керування пам’яттю повинно виконувати два основні завдання:

- визначати об’єкти, які більше не можуть бути використані програмою;
- повертати зайняту ними пам’ять для розміщення нових об’єктів.

Повернення пам’яті може здійснюватися двома способами. У першому випадку звільнені області додаються до внутрішнього списку вільних комірок середовища виконання. Під час наступного створення об’єкта система спочатку шукає відповідну

область у цьому списку і лише за її відсутності запитує нову пам’ять в операційної системи. У другому випадку пам’ять фактично повертається операційній системі. На практиці обидва способи часто поєднуються: звільнені області тимчасово накопичуються у внутрішньому списку, а після досягнення певного обсягу повертаються операційній системі.

Для довготривалих програм недостатньо лише внутрішнього повторного використання комірок. Якщо програма постійно створює об’єкти різних розмірів, вона може продовжувати запитувати нову пам’ять, хоча значна кількість старих об’єктів уже стала недосяжною. Тому повноцінне середовище виконання повинно підтримувати фактичне повернення невикористаної пам’яті.

Основними підходами до автоматичного керування пам’яттю є:

- підрахунок посилань;
- автоматичне збирання сміття.

Ідея підрахунку посилань полягає в тому, що для кожного об’єкта зберігається кількість посилань, які в поточний момент ведуть до нього. Коли значення лічильника стає нульовим, об’єкт вважається недосяжним, а його пам’ять може бути повторно використана.

Під час створення нового об’єкта його лічильник посилань установлюється в одиницю, оскільки новостворений об’єкт одразу приєднується до певної програмної сутності:

create a

Після виконання цієї операції змінна a посилається на новий об’єкт, тому:

reference_count = 1

Кожне приєднання нового посилання до об’єкта збільшує його лічильник на одиницю. Це відбувається, зокрема, у таких випадках:

- присвоєння одного посилання іншій змінній;
- передавання посилання як аргументу процедури або функції.

Наприклад, після присвоєння

b := a

змінні a і b посилаються на один об’єкт, тому його лічильник збільшується на одиницю.

Кожне від’єднання посилання відповідно зменшує лічильник. Це відбувається:

- коли змінній присвоюється інше посилання;
- коли змінна отримує порожнє значення;
- після завершення виклику процедури, якщо припиняють існувати її аргументи або локальні змінні.

Після кожного зменшення середовище виконання перевіряє значення лічильника. Якщо воно дорівнює нулю, об’єкт можна звільнити:

if reference_count = 0 then reclaim object

Підрахунок посилань є відносно простим для реалізації, однак потребує додаткових витрат. Кожний об’єкт повинен містити поле для лічильника, а кожна операція приєднання або від’єднання посилання супроводжується арифметичною операцією. Під час від’єднання також необхідно перевіряти, чи не став лічильник нульовим.

Основним недоліком підрахунку посилань є неможливість правильно опрацьовувати циклічні структури. Цикл виникає тоді, коли декілька об’єктів прямо або опосередковано посилаються один на одного.

Розглянемо три об’єкти O1, O2 і O3, пов’язані взаємними посиланнями. Спочатку до O1 також веде зовнішнє посилання з об’єкта O. Якщо це зовнішнє посилання видалити, усі три об’єкти стануть недосяжними для програми. Проте їхні лічильники не дорівнюватимуть нулю, оскільки вони продовжують посилатися один на одного.

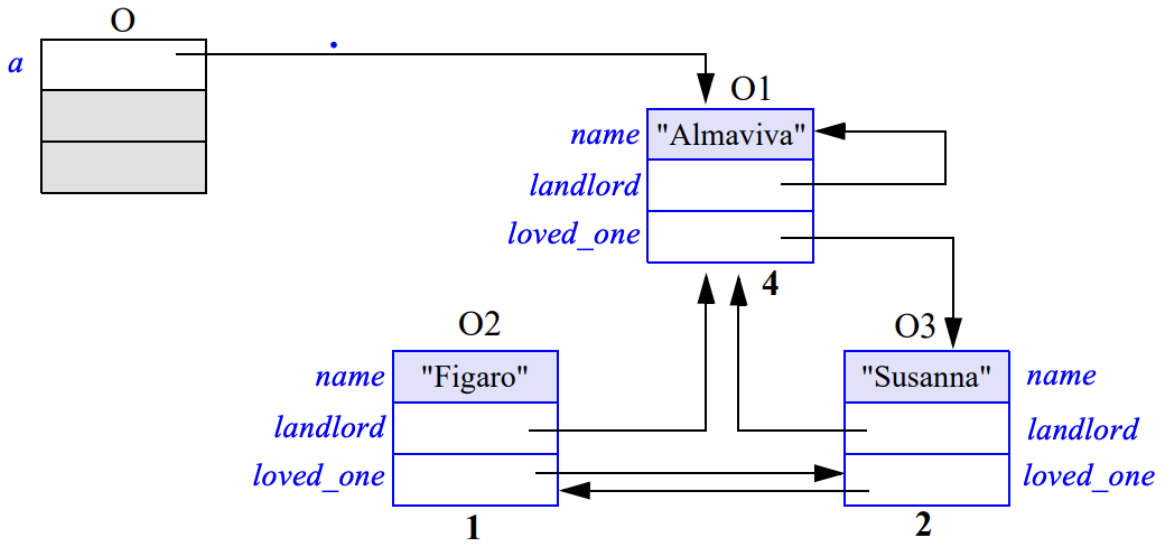


Рисунок 5.12 – Циклічна структура, яку неможливо звільнити за допомогою підрахунку посилань

Після видалення зовнішнього посилання об’єкти циклу більше не можуть бути використані програмою, але механізм підрахунку посилань не виявляє цього. Їхні лічильники залишаються додатними, тому пам’ять не звільняється.

Через цю проблему підрахунок посилань придатний лише для структур, у яких гарантовано не виникають цикли. Оскільки в довільній об’єктно-орієнтованій системі таку гарантію забезпечити неможливо, цей підхід не є універсальним механізмом керування пам’яттю.

Найбільш загальним підходом є автоматичне збирання сміття. Збирач сміття (garbage collector) — це складова середовища виконання мови програмування, яка самостійно виявляє недосяжні об’єкти та звільняє зайняту ними пам’ять.

Збирач сміття виконує обидві частини процесу:

- визначає, які об’єкти залишаються досяжними;
- повертає пам’ять усіх виявлених недосяжних об’єктів.

Прикладна програма не повинна явно видаляти кожний об’єкт. Водночас середовище може надавати засоби керування моментом запуску збирача, однак правильність визначення недосяжних об’єктів залишається відповідальністю самого середовища виконання.

Коректний збирач сміття повинен задовольняти дві основні властивості.

Безпечність (soundness) означає, що кожний звільнений збирачем об’єкт справді є недосяжним:

кожний звільнений об’єкт є недосяжним.

Збирач не повинен видаляти об’єкт, до якого програма ще може звернутися. Безпечність є абсолютною вимогою, оскільки передчасне звільнення досяжного об’єкта спричиняє втрату даних, висячі посилання та непередбачувану поведінку програми.

Повнота (completeness) означає, що кожний недосяжний об’єкт буде звільнений:

кожний недосяжний об’єкт буде з часом звільнений.

Якщо ця властивість не виконується, частина непотрібних об’єктів залишається в пам’яті, що фактично призводить до витоків. Повноту доцільно розуміти не як негайне видалення об’єкта після втрати останнього шляху доступу, а як гарантію того, що він буде виявлений і звільнений під час одного з наступних циклів збирання.

Простий механізм може легко забезпечити лише одну із цих властивостей. Наприклад, збирач, який не видаляє жодного об’єкта, є безпечним, але не повним. Механізм, який звільняє всю пам’ять, є повним щодо недосяжних об’єктів, але небезпечним, оскільки разом із ними видаляє досяжні. Складність полягає в одночасному забезпеченні безпечності та повноти.

Окрім цих двох властивостей, важливим є час між моментом, коли об’єкт стає недосяжним, і моментом фактичного звільнення його пам’яті. Збирач може бути коректним, але звільняти деякі об’єкти із затримкою. Така затримка є прийнятною, якщо пам’ять зрештою повертається і програма не вичерпує доступні ресурси.

Щоб визначити недосяжні об’єкти, збирач починає з множини початкових об’єктів, або коренів. До них належать об’єкти, безпосередньо приєднані до програмних сутностей, доступних у поточний момент:

- кореневого об’єкта системи;
- локальних змінних активних процедур;
- формальних аргументів активних викликів;
- інших посилань, які середовище виконання визначає як початкові.

Від цих об’єктів збирач переходить за всіма посилальними полями. Кожний знайдений об’єкт також перевіряється, а його посилання використовуються для

подальшого обходу. Усі об’єкти, до яких можна дістатися таким способом, вважаються досяжними. Решта об’єктів є недосяжними і можуть бути звільнені.

На відміну від підрахунку посилань, цей підхід правильно опрацьовує цикли. Якщо циклічна структура не має шляху від жодного кореня, її об’єкти не будуть позначені як досяжні й можуть бути видалені разом.

Базовий алгоритм збирання сміття складається з двох концептуальних фаз:

1. Позначення (mark).
2. Очищення (sweep).

Під час фази позначення збирач починає з кореневих об’єктів і рекурсивно переходить за посиланнями. Кожний знайдений об’єкт позначається як досяжний. Якщо об’єкт уже позначено, повторно обходити його не потрібно. Це також запобігає нескінченному обходу циклічних структур.

У спрощеній формі процес має такий вигляд:

- позначити всі кореневі об’єкти;
- для кожного позначеного об’єкта позначити всі об’єкти, на які він посилається;
- повторювати, доки не залишиться нових об’єктів для обходу.

Під час фази очищення збирач переглядає всю область пам’яті. Об’єкти, які не отримали позначки, вважаються недосяжними, тому їхня пам’ять звільняється. Для позначених об’єктів позначка скидається, щоб підготувати їх до наступного циклу:

- якщо об’єкт не позначений — звільнити;
- якщо об’єкт позначений — зняти позначку.

Для збереження позначки достатньо одного додаткового біта на об’єкт, тому витрати пам’яті на цей механізм є незначними.

Алгоритм позначення та очищення безпосередньо відповідає вимогам коректності. Фаза позначення повинна знайти всі об’єкти, доступні від коренів, що забезпечує безпечність. Фаза очищення повинна звільнити всі об’єкти, які залишилися непозначеними, що забезпечує повноту.

Отже, автоматичне керування пам’яттю переносить відповідальність за пошук і звільнення недосяжних об’єктів із прикладного коду до середовища виконання. Підрахунок посилань є простим, але не може звільняти циклічні структури. Збирач сміття аналізує досяжність об’єктів від коренів і тому є загальнішим механізмом. Його правильність визначається безпечністю, за якої не звільняються досяжні об’єкти, і повнотою, за якої кожний недосяжний об’єкт з часом буде виявлений та звільнений.

5.5 Практичні аспекти й оптимізація автоматичного керування пам’яттю

Ефективне автоматичне керування пам’яттю потребує не лише правильного алгоритму пошуку недосяжних об’єктів. Середовище виконання повинно визначати

момент запуску збирача сміття, обмежувати тривалість пауз, повертати невикористану пам’ять операційній системі, підтримувати взаємодію із зовнішнім програмним кодом і надавати розробникові засоби налаштування механізму.

Збирач сміття зазвичай активується тоді, коли програма запитує пам’ять для нового об’єкта. Такий запит може виникати під час виконання інструкції створення або операції клонування:

- *create x*
- *clone (x)*

Збирання сміття не обов’язково починається лише після повного вичерпання пам’яті. Доцільніше діяти за принципом попередження: середовище виконання аналізує стан пам’яті й запускає відповідний алгоритм до того, як нестача ресурсу спричинить помилку.

Якщо основна область розміщення об’єктів заповнюється, система спочатку виконує збирання молодих об’єктів. У більшості випадків цього достатньо для отримання потрібного обсягу вільної пам’яті. Якщо результат є недостатнім, виконується повний цикл позначення й очищення, після якого за потреби здійснюється ущільнення пам’яті. Лише коли ці операції не забезпечили необхідного простору, середовище виконання звертається до операційної системи із запитом на виділення додаткової пам’яті.

Засоби керування збирачем доцільно об’єднати в окремому класі MEMORY. Серед його операцій можуть бути:

- collection_off — тимчасове вимкнення автоматичного збирання;
- collection_on — відновлення роботи збирача;
- collect_now — примусовий запуск повного збирання.

За звичайних умов збирач повинен працювати автоматично. Ручне керування його запуском потрібне лише для спеціальних ділянок програми, у яких розробник має конкретні вимоги до часу виконання.

Повне збирання сміття передбачає перевірку всієї доступної області об’єктної пам’яті. Спочатку збирач позначає всі об’єкти, досяжні від коренів системи. Потім він переглядає пам’ять і звільняє непозначені об’єкти.

Такий підхід є загальним і дозволяє виявити всі недосяжні структури, зокрема циклічні. Однак перевірка всієї пам’яті може займати значний час. Якщо на період збирання виконання прикладної програми повністю зупиняється, користувач помічає паузу.

Тривалі паузи є особливо небажаними для:

- інтерактивних програм;
- серверних систем;
- систем, що працюють упродовж тривалого часу;
- програм із вимогами до часу реакції.

Тому повне збирання не повинно виконуватися після кожного запиту пам’яті. Воно запускається періодично, коли швидші способи не дали достатнього результату або коли внутрішні показники середовища свідчать про накопичення значної кількості недосяжних об’єктів.

Поступове, або інкрементне, збирання поділяє роботу збирача на невеликі частини, які виконуються між операціями прикладної програми. Замість однієї тривалої зупинки система проводить багато коротких етапів перевірки пам’яті.

Одним зі способів наблизитися до поступового збирання є поділ об’єктів на покоління. Основою цього підходу є спостереження: значна частина об’єктів стає недосяжною незабаром після створення, тоді як об’єкти, що пережили декілька циклів збирання, з великою ймовірністю використовуватимуться довше.

Нові об’єкти належать до молодого покоління і перевіряються частіше. Якщо об’єкт залишається досяжним після кількох циклів, він переходить до старшого покоління. Старі об’єкти перевіряються рідше, що зменшує обсяг роботи збирача.

Практичні реалізації можуть використовувати не два, а декілька поколінь із різними правилами збирання. Після кожного успішного циклу об’єкти, які залишилися досяжними, стають старшими. Потрібно правильно встановити момент їх переведення до наступного покоління:

- надто раннє переведення збільшує кількість старих об’єктів;
- надто пізнє переведення спричиняє надмірно часту перевірку довгоживучих об’єктів.

Збирання за поколіннями істотно скорочує середню тривалість роботи збирача, оскільки більшість циклів охоплює лише частину пам’яті. Проте воно не усуває потреби періодично проводити повне збирання.

Іншим підходом є паралельне збирання, за якого прикладна програма і збирач працюють як окремі потоки керування. Прикладна програма створює об’єкти, а збирач одночасно шукає недосяжні об’єкти й повертає зайняту ними пам’ять.

У такій моделі:

- лише прикладний потік виділяє пам’ять;
- лише збирач звільняє пам’ять;
- збирач безперервно повторює фази позначення й очищення.

Паралельне виконання зменшує помітність пауз, але потребує узгодження дій потоків. Поки збирач аналізує структуру посилань, прикладна програма може створювати нові об’єкти, змінювати поля та вилучати зв’язки. Тому алгоритм повинен гарантувати, що такі зміни не призведуть до помилкового звільнення досяжного об’єкта.

Робота збирача в окремому програмному потоці також споживає процесорний час, потрібний прикладній програмі. Ефективнішим може бути використання

окремого процесора або ядра, призначеного для керування пам’яттю. Однак такий підхід залежить від можливостей апаратного забезпечення та середовища виконання.

Після багаторазового створення та видалення об’єктів у пам’яті можуть утворюватися численні вільні ділянки різного розміру. Загальний обсяг вільної пам’яті може бути достатнім, але її поділ на невеликі несуміжні області ускладнює розміщення великих об’єктів. Це явище називається фрагментацією пам’яті.

Для зменшення фрагментації застосовується ущільнення. Досяжні об’єкти переміщуються ближче один до одного, а вільні ділянки об’єднуються у більші безперервні блоки. У результаті частину повністю звільнених блоків можна повернути операційній системі.

Переміщення об’єктів є складною частиною керування пам’яттю, оскільки після зміни адреси об’єкта необхідно оновити всі посилання на нього. Усередині повністю об’єктно-орієнтованого середовища це виконується автоматично й не повинно впливати на прикладну програму.

Ущільнення не обов’язково виконується за один цикл. Пам’ять можна поділити на n блоків і поступово ущільнити їх протягом n – 1 циклів. Це зменшує тривалість окремих пауз і наближає механізм до поступової роботи.

Недостатньо лише повторно використовувати вивільнені області всередині поточного процесу. Для програм, які працюють постійно або тривалий час, важливо фактично повертати зайві блоки операційній системі, щоб ними могли скористатися інші програми.

Водночас системні операції виділення та повернення пам’яті є відносно дорогими. Тому середовище не повинно негайно повертати кожний звільнений блок. Доцільно зберігати невеликий резерв пам’яті для наступних операцій створення.

Це особливо важливо для програм із періодичними змінами навантаження. Така програма може протягом певного часу інтенсивно створювати об’єкти, потім звільняти більшість із них, а згодом знову переходити до активного створення. Негайне повернення всієї пам’яті змусило б середовище постійно запитувати її в операційної системи, повертати, а потім запитувати знову.

Отже, механізм ущільнення має бути поміркованим: повертати операційній системі значні надлишки, але зберігати резерв для найближчих запитів прикладної програми.

Автоматичне звільнення пам’яті не завжди є достатнім. Об’єкт може бути пов’язаний із зовнішнім ресурсом, який не належить до об’єктної пам’яті:

- відкритим файлом;
- мережевим з’єднанням;
- системним дескриптором;
- пристроєм;
- ресурсом зовнішньої бібліотеки.

Перед остаточним видаленням такого об’єкта необхідно виконати спеціальну завершальну операцію. Цей механізм називають фіналізацією.

У класі MEMORY може бути визначена процедура dispose, яка за замовчуванням не виконує жодних дій:

```
dispose
   -- Дія перед вивільненням об’єкта збирачем сміття.
   -- За замовчуванням нічого не виконується.
do
end
```

Клас, об’єкти якого використовують зовнішні ресурси, може перевизначити цю операцію. Наприклад, клас FILE повинен закрити пов’язаний фізичний файл перед вивільненням об’єкта:

```
dispose
   -- Закрити файл перед вивільненням об’єкта.
do
   if opened then
      close
   end
end
```

Збирач автоматично викликає dispose перед поверненням пам’яті. Для більшості класів використовується порожня реалізація, а спеціальна поведінка визначається лише там, де вона справді необхідна.

Об’єктно-орієнтована програма може викликати процедури, написані іншими мовами, та передавати їм посилання на свої об’єкти. Це створює дві небезпеки.

По-перше, зовнішня процедура може зберегти адресу об’єкта. Якщо збирач перемістить цей об’єкт під час ущільнення, збережена адреса стане неправильною.

По-друге, середовище виконання може вважати об’єкт недосяжним, оскільки воно не бачить посилання, яке зберігається у зовнішньому коді. У такому випадку об’єкт може бути звільнений, хоча зовнішня процедура ще планує його використати.

Для розв’язання цієї проблеми середовище має надавати спеціальні операції:

- тимчасове виключення об’єкта з механізму переміщення і збирання;
- відновлення звичайного керування після завершення роботи зовнішньої процедури;
- доступ до об’єкта через спеціальний механізм, який знаходить його навіть після переміщення.

У книзі ці дії описуються як «усиновлення» об’єкта зовнішнім кодом і подальше його «відлучення». Поки об’єкт усиновлений, збирач не повинен його видаляти. Після відлучення він знову може бути звільнений, якщо стане недосяжним.

Такі низькорівневі операції не повинні використовуватися безпосередньо у звичайному прикладному коді. Їх необхідно зосереджувати в спеціальних допоміжних класах, які приховують деталі взаємодії з іншими мовами.

Практичне середовище керування пам’яттю не обмежується одним алгоритмом. Залежно від стану системи воно може поєднувати:

- збирання молодого покоління;
- повне позначення й очищення;
- ущільнення пам’яті;
- інші спеціалізовані механізми.

Під час кожного запуску середовище вибирає алгоритм або їх комбінацію, враховуючи терміновість запиту пам’яті та внутрішню статистику. Збирання молодих об’єктів використовується як швидкий основний варіант. Повне позначення й очищення виконується рідше, щоб виявити об’єкти, пропущені поколіннєвим алгоритмом. Ущільнення застосовується для усунення фрагментації та повернення пам’яті операційній системі.

Неможливо одночасно мінімізувати використання пам’яті, повністю усунути паузи та не витрачати процесорний час на роботу збирача. Тому практична реалізація завжди повинна знаходити компроміс між:

- швидкістю прикладної програми;
- частотою запуску збирача;
- тривалістю пауз;
- обсягом зайнятої пам’яті;
- частотою звернень до операційної системи;
- повнотою ущільнення.

Наприклад, режим оптимізації швидкості може змусити систему раніше запитувати додаткову пам’ять в операційної системи й витрачати менше часу на повне ущільнення. У цьому разі програма працює швидше, але використовує більше пам’яті.

Режим оптимізації пам’яті, навпаки, частіше запускає збирання та ущільнення. Це зменшує обсяг зайнятих ресурсів, але збільшує витрати процесорного часу й може спричиняти помітніші паузи.

Середовище виконання повинно збирати статистику про кількість створених об’єктів, обсяг зайнятої пам’яті та результати попередніх циклів. На основі цих даних воно визначає, який алгоритм запустити. Розробникові можуть надаватися параметри налаштування, але стандартні значення повинні забезпечувати прийнятну поведінку без ручного втручання.

Добре реалізований збирач сміття працює автоматично та майже непомітно для прикладної програми. Він увімкнений за замовчуванням, своєчасно звільняє недосяжні об’єкти й повертає надлишкову пам’ять операційній системі.

Автоматичне керування пам’яттю дозволяє розробникам:

- не розміщувати операції явного вивільнення в прикладному коді;
- уникати значної частини витоків пам’яті та висячих посилань;
- вільніше створювати складні динамічні структури;
- зменшити обсяг допоміжного коду;
- зосередитися на логіці предметної області.

Водночас автоматичне керування не звільняє розробника від потреби раціонально використовувати ресурси. Якщо програма зберігає непотрібні посилання, відповідні об’єкти залишаються формально досяжними, тому збирач не може їх видалити. Також окремої уваги потребують зовнішні ресурси й взаємодія з кодом, який зберігає фізичні адреси об’єктів.

Отже, практичне автоматичне керування пам’яттю ґрунтується на поєднанні декількох алгоритмів, поступовому виконанні операцій, ущільненні пам’яті та її фактичному поверненні операційній системі. Середовище повинно підтримувати фіналізацію об’єктів, безпечну взаємодію із зовнішнім кодом і засоби налаштування збирача. Правильний вибір параметрів дозволяє підтримувати баланс між швидкістю виконання, обсягом використаної пам’яті та тривалістю пауз, не перекладаючи низькорівневі завдання керування пам’яттю на розробників прикладних програм.

Висновки

Управління пам’яттю є важливою складовою надійності та продуктивності програмних систем. Статичне, стекове й динамічне розміщення даних по-різному визначають життєвий цикл об’єктів і момент вивільнення зайнятих ними ресурсів. Найскладнішою є робота з об’єктами в купі, оскільки їхній час існування залежить не від структури програми, а від наявності шляхів доступу до них. Ручне вивільнення пам’яті забезпечує безпосередній контроль, але підвищує ризик витоків, висячих посилань і повторного доступу до вже звільнених областей. Керування на рівні програмних компонентів дозволяє локалізувати операції створення та повторного використання об’єктів, однак придатне лише для структур із контрольованим життєвим циклом. Автоматичне керування пам’яттю переносить відповідальність за виявлення недосяжних об’єктів і повернення їхньої пам’яті до середовища виконання. Підрахунок посилань є простим, але не розв’язує проблему циклічних структур, тоді як збирання сміття забезпечує загальніший аналіз досяжності. Практичні реалізації поєднують поступове, поколіннєве, паралельне й повне збирання, а також ущільнення пам’яті та її повернення операційній системі. Ефективна система керування пам’яттю повинна підтримувати баланс між швидкістю виконання програми, обсягом використаних ресурсів і тривалістю пауз, зменшуючи складність прикладного коду та кількість помилок, пов’язаних із життєвим циклом об’єктів.

Лекція 6

Тема: Універсалізація

Мета: Універсалізація в програмуванні означає створення програмного коду, який може бути використаний у різних контекстах та для різних завдань. На цій лекції ми розглянемо методики та підходи до створення універсального програмного коду, який забезпечує його використання в різних сценаріях.

План лекції

1. Поняття універсалізації та способи узагальнення типів
2. Необхідність параметризації типів
3. Узагальнені класи та їх параметри
4. Масиви як приклад узагальненого класу
5. Переваги, вартість та обмеження універсалізації

6.1 Поняття універсалізації та способи узагальнення типів

Поняття класу об’єднує модуль і тип та є основою об’єктно-орієнтованого методу. Проте для забезпечення розширюваності, повторного використання та надійності програмного забезпечення класи повинні бути достатньо гнучкими. Така гнучкість розвивається у двох напрямах. Перший напрям пов’язаний з абстрагуванням і спеціалізацією класів, а другий — із параметризацією класів типами. У книзі ці напрями названо відповідно вертикальним і горизонтальним узагальненням типів.

Універсалізація, або узагальнення, полягає у створенні програмного компонента, який описує не один окремий випадок, а цілу групу подібних випадків. Універсальний компонент містить спільні властивості та операції, а відмінні характеристики задаються окремо. Завдяки цьому одна реалізація може застосовуватися в різних контекстах без копіювання та ручного редагування програмного коду.

Розглянемо клас LIST_OF_BOOKS, екземпляри якого представляють списки книг. Такий клас може містити операції put для додавання елемента, remove для його видалення, count для визначення кількості елементів та інші операції списку. Поняття списку книг можна узагальнити двома способами:

- перейти від списку книг до більш загальних або спеціалізованих структур;
- перейти від списку книг до списків об’єктів інших типів.

Ці способи відповідають вертикальному та горизонтальному узагальненню.

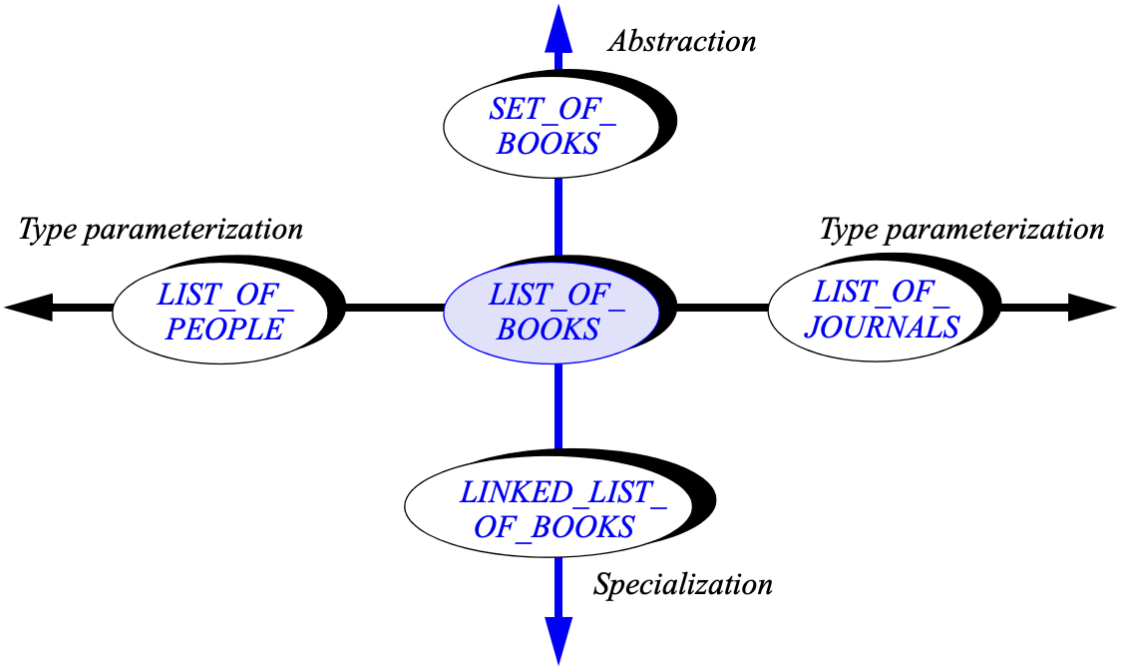


Рисунок 6.1 – Горизонтальний і вертикальний напрями узагальнення типів

Горизонтальне узагальнення типів

Список книг є окремим випадком списку об’єктів певного типу. Аналогічними структурами можуть бути:

- список журналів;
- список людей;
- список цілих чисел;
- список банківських рахунків;
- список графічних об’єктів.

Основні операції над такими списками однакові. Незалежно від типу елементів до списку потрібно додавати елементи, вилучати їх, визначати кількість, перевіряти порожність та отримувати поточний елемент. Відмінність полягає переважно в типі об’єктів, які зберігаються у списку.

Написання окремих класів LIST_OF_BOOKS, LIST_OF_PEOPLE, LIST_OF_JOURNALS та LIST_OF_INTEGERS призвело б до створення декількох майже однакових програмних компонентів. Їхні операції та алгоритми повторювалися б, а змінювалися б лише декларації типу елементів. Такий підхід суперечить вимогам повторного використання програмного забезпечення.

Горизонтальне узагальнення усуває це дублювання за допомогою параметризації типом. Замість окремих класів створюється один універсальний клас:

LIST [G]

Позначення G представляє довільний тип елементів списку. Під час використання класу замість G указується конкретний тип:

- *LIST [BOOK]*

- *LIST [PERSON]*
- *LIST [JOURNAL]*
- *LIST [INTEGER]*

Отже, горизонтальне узагальнення полягає у визначенні класу, параметризованого типом об'єктів, з якими він працює. Один клас охоплює кілька паралельних варіантів тієї самої програмної структури. Саме цей напрям узагальнення називають універсалізацією, або genericity.

Горизонтальне узагальнення особливо важливе для контейнерних класів. Контейнерами є структури, призначені для зберігання інших об'єктів, зокрема:

- списки;
- стеки;
- черги;
- множини;
- дерева;
- масиви;
- матриці.

Алгоритм роботи контейнера зазвичай не залежить від конкретного типу його елементів. Наприклад, правила додавання і видалення елементів зі стека залишаються однаковими для стека цілих чисел, книг або банківських рахунків. Тому тип елемента доцільно перетворити на параметр класу:

STACK [G]

З такого універсального класу можна отримати конкретні типи:

- *STACK [INTEGER]*
- *STACK [BOOK]*
- *STACK [ACCOUNT]*

При цьому зберігається статична перевірка типів. До стека STACK [BOOK] можна додавати лише об'єкти типу BOOK, а спроба додати об'єкт типу ACCOUNT повинна бути відхилена компілятором. Отже, універсалізація поєднує повторне використання однієї реалізації з надійністю, яку забезпечують явні декларації типів.

Вертикальне узагальнення типів

Другий напрям узагальнення стосується співвідношення між загальними та спеціалізованими поняттями. Наприклад, список є одним із різновидів контейнера. Водночас зв'язаний список є конкретним способом реалізації списку.

Для класу LIST_OF_BOOKS можна визначити:

- більш абстрактний клас SET_OF_BOOKS, який описує загальні властивості структури для зберігання книг;
- більш спеціалізований клас LINKED_LIST_OF_BOOKS, який реалізує список книг за допомогою зв'язаних елементів.

Перехід від спеціалізованого класу до загальнішого поняття є абстрагуванням. Перехід від загального класу до конкретнішого варіанта є спеціалізацією. Ці переходи утворюють вертикальний напрям узагальнення.

Вертикальне узагальнення реалізується за допомогою успадкування. Воно дозволяє визначити новий клас на основі вже наявного, зберігаючи його властивості та додаючи або уточнюючи окремі характеристики. Детально механізми успадкування розглядатимуться в наступних лекціях. У межах цієї теми важливо зафіксувати, що вертикальне узагальнення формує відношення між більш загальними та більш спеціалізованими класами.

Горизонтальне і вертикальне узагальнення розв'язують різні завдання наведено в таблиці 6.1.

Таблиця 6.1 – Порівняння напрямів узагальнення типів

Напрям узагальнення	**Призначення**	**Спосіб реалізації**	**Приклад**
Горизонтальне	Використання одного програмного компонента для роботи з різними типами даних	Параметризація типом	*LIST [BOOK], LIST [PERSON]*
Вертикальне	Побудова ієрархії загальних і спеціалізованих класів	Успадкування	*SET_OF_BOOKS, LIST_OF_BOOKS, LINKED_LIST_OF_BOOKS*

Горизонтальний напрям дозволяє використовувати одну програмну структуру для різних типів даних. Вертикальний напрям дозволяє будувати сімейства класів, переходячи від загальних понять до спеціалізованих реалізацій. Обидва механізми підвищують гнучкість програмних компонентів, але не замінюють один одного.

Відмінність універсалізації від окремих реалізацій

Без універсалізації для кожного типу елементів довелося б створювати окремий клас:

- *INTEGER_STACK*
- *REAL_STACK*
- *POINT_STACK*
- *BOOK_STACK*

Такі класи були б майже однаковими. Тип INTEGER, REAL, POINT або BOOK повторювався б у деклараціях аргументів, результатів функцій, атрибутів і локальних змінних, але тіла основних операцій залишалися б незмінними.

Наприклад, у класі *INTEGER_STACK* операції могли б мати такі декларації:

put (element: INTEGER)

item: INTEGER

Для стека книг довелося б створити інший клас і замінити тип *INTEGER* на *BOOK*:

put (element: BOOK)

item: BOOK

Алгоритми додавання, видалення та перевірки порожності при цьому не змінюються. Виникає необґрунтоване дублювання програмного коду.

Цей підхід має кілька недоліків:

- збільшується кількість класів;
- однаковий код потрібно підтримувати в декількох місцях;
- виправлення помилки необхідно повторювати в усіх реалізаціях;
- зміни можуть бути внесені непослідовно;
- ускладнюється створення бібліотек повторно використовуваних компонентів.

Універсалізація замінює всі ці класи одним типопараметризованим класом:

STACK [G]

Усередині цього класу G використовується як умовне ім’я типу елементів:

put (element: G)

item: G

Під час застосування класу користувач указує фактичний тип:

- *STACK [INTEGER]*
- *STACK [BOOK]*
- *STACK [POINT]*

Таким чином, універсальний клас є не одним конкретним типом, а шаблоном типу. Конкретний тип утворюється після підстановки фактичного параметра. Одна реалізація використовується для багатьох типів даних, але кожне її конкретне застосування залишається типобезпечним.

Отже, універсалізація є механізмом горизонтального узагальнення, який дозволяє параметризувати програмні компоненти типами. Вона усуває необхідність створювати багато майже однакових класів і водночас зберігає явні декларації та статичний контроль типів. Вертикальне узагальнення відображає відношення абстрагування і спеціалізації між класами та реалізується через успадкування. Поєднання обох напрямів створює основу для гнучких, надійних і повторно використовуваних програмних компонентів.

6.2 Необхідність параметризації типів

Універсалізація вже використовувалася під час формального опису абстрактних типів даних. Наприклад, стек було визначено як STACK [G], де G позначає довільний тип елементів. Завдяки цьому одна специфікація описує стеки цілих чисел, дійсних

чисел, книг, банківських рахунків та інших об’єктів. Без параметризації для кожного типу елементів довелося б створювати окрему специфікацію або окремий програмний клас.

Розглянемо стек уже не як математичний абстрактний тип даних, а як програмний клас. Можна створити клас INTEGER_STACK, призначений для зберігання цілих чисел. Він міститиме такі основні операції:

- count — кількість елементів;
- put — додавання нового елемента;
- item — отримання верхнього елемента;
- remove — видалення верхнього елемента;
- empty — перевірка стека на порожність.

Тип INTEGER у такому класі використовуватиметься в деклараціях усіх сутностей, що представляють елементи стека. Наприклад:

put (element: INTEGER)

item: INTEGER

Явне зазначення типу потрібне для того, щоб компілятор і розробник точно знали, які об’єкти можуть зберігатися в стеку та який тип значення повертає операція item.

Однак для створення стека дійсних чисел довелося б визначити інший клас:

REAL_STACK

У ньому декларації мали б вигляд:

put (element: REAL)

item: REAL

Для точок, книг і рядків аналогічно знадобилися б класи:

POINT_STACK

BOOK_STACK

STRING_STACK

Усі ці класи були б майже однаковими. Відрізнялися б лише типи аргументів, результатів та окремих атрибутів. Алгоритми додавання, видалення, перевірки порожності й визначення кількості елементів залишалися б незмінними, оскільки основні операції стека не залежать від типу його елементів.

Створення окремого класу для кожного типу призводить до необґрунтованого дублювання програмного коду. Однакові операції потрібно повторно реалізовувати, тестувати та супроводжувати. Після виявлення помилки її доведеться виправляти в усіх класах. Якщо зміни будуть внесені не до кожної реалізації, поведінка подібних компонентів стане неузгодженою.

Проблема стосується не лише стеків. Більшість контейнерних структур призначена для зберігання об’єктів різних типів. До таких структур належать:

- списки;

- черги;
- множини;
- дерева;
- масиви;
- матриці;
- таблиці.

Наприклад, алгоритм додавання елемента до зв’язаного списку не залежить від того, чи містить список цілі числа, рядки, книги або графічні об’єкти. У кожному випадку потрібно створити новий елемент, встановити зв’язки із сусідніми елементами та оновити службову інформацію списку.

Без параметризації довелося б створити окремі класи:

- *INTEGER_LIST*
- *STRING_LIST*
- *BOOK_LIST*
- *PERSON_LIST*

Такі класи повторювали б однакову структуру та однакові операції. Змінювався б лише тип значення, яке зберігається в кожному елементі списку.

Параметризація дозволяє винести цей змінний тип за межі реалізації класу. Замість багатьох окремих класів визначається один універсальний компонент:

LIST [G]

У його операціях формальний параметр G використовується як тип елементів:

put (element: G)

item: G

Під час застосування класу вказується конкретний фактичний параметр:

- *LIST [INTEGER]*
- *LIST [STRING]*
- *LIST [BOOK]*
- *LIST [PERSON]*

Отже, клас *LIST [G]* описує спільну структуру та поведінку всіх списків, а фактичний параметр визначає тип об’єктів, які зберігатимуться в конкретному списку.

Розглянемо узагальнений клас *CONTAINER [G]*, призначений для зберігання елементів довільного типу. Він може надавати такі операції:

put (value: G)

item: G

remove

count: INTEGER

empty: BOOLEAN

Позначення *G* є формальним параметром типу. Воно не вказує на конкретний тип, але використовується в усіх місцях, де клас працює зі своїми елементами.

Для контейнера цілих чисел створюється тип:

CONTAINER [INTEGER]

У такому контейнері операція put приймає ціле число, а item повертає значення типу *INTEGER*.

Для рядків використовується:

CONTAINER [STRING]

Тепер операція put приймає рядок, а item повертає результат типу *STRING*.

Для об’єктів класу BOOK визначається:

CONTAINER [BOOK]

Одна реалізація контейнера використовується в усіх трьох випадках. Водночас отримані типи залишаються різними. *CONTAINER [INTEGER]* не є контейнером довільних значень: він призначений саме для цілих чисел. Аналогічно *CONTAINER [BOOK]* повинен зберігати об’єкти типу *BOOK*.

Параметризація типів не означає відмови від типізації. Навпаки, вона дозволяє створювати універсальні компоненти, зберігаючи точну інформацію про типи об’єктів у кожному конкретному застосуванні.

Необхідність параметризації виникає через суперечність між двома важливими цілями якості програмного забезпечення:

- надійністю, яка вимагає явного зазначення типів і перевірки правильності операцій;
- повторним використанням, яке вимагає створення одного програмного компонента для кількох варіантів певного поняття.

Для забезпечення лише повторного використання можна було б створити контейнер без точного типу елементів. Такий контейнер дозволяв би додавати будь-які об’єкти. Однак після отримання елемента програма не мала б надійної інформації про його тип.

Для забезпечення лише суворої типізації можна створювати окремі класи для кожного типу елементів. Це підвищило б надійність, але призвело б до дублювання коду та втрати повторного використання.

Параметризація типів поєднує обидві вимоги. Клас створюється один раз, але під час його використання конкретний тип елементів зазначається явно:

- *STACK [INTEGER]*
- *STACK [STRING]*
- *STACK [BOOK]*

Компілятор розглядає кожну таку конструкцію як точно визначений тип і перевіряє допустимість усіх операцій.

Явні декларації типів потрібні з двох основних причин. Перша причина пов’язана з читабельністю. Декларація типу чітко повідомляє, для яких об’єктів призначена програмна сутність. Це допомагає зрозуміти код його авторові, іншим

розробникам і фахівцям, які надалі виконуватимуть супровід або розширення програми.

Наприклад, декларація

books: STACK [BOOK]

відразу показує, що змінна books представляє стек книг. Для розуміння цього не потрібно аналізувати всі операції, які над нею виконуються.

Друга причина полягає в надійності. Завдяки явним типам компілятор може виявити помилкові операції до запуску програми. Він перевіряє:

- чи існує викликана операція в класі об’єкта;
- чи доступна ця операція клієнтові;
- чи відповідає кількість аргументів її декларації;
- чи сумісні типи переданих аргументів;
- чи правильно використовується результат операції.

Наприклад, до стека банківських рахунків не можна додати графічний об’єкт:

accounts: STACK [ACCOUNT]

circle: CIRCLE

accounts.put (circle)

Компілятор повинен відхилити таку операцію, оскільки *STACK [ACCOUNT]* приймає елементи типу *ACCOUNT*, а не *CIRCLE*.

Без статичної перевірки програма могла б зберегти коло в контейнері, а пізніше отримати його та спробувати використати як банківський рахунок:

- *current_account := accounts.item*
- *current_account.withdraw (5000)*

У такій ситуації програма фактично намагалася б зняти кошти з графічного кола. Статична типізація запобігає подібним помилкам ще на етапі компіляції.

Поєднання гнучкості зі статичним контролем

Для універсального компонента потрібно одночасно виконати дві вимоги:

- кожна сутність у тексті класу повинна мати явно визначений тип;
- клас не повинен бути прив’язаний до одного конкретного типу елементів.

На перший погляд ці вимоги суперечать одна одній. Розв’язання полягає у введенні формального параметра типу. Замість конкретної назви *INTEGER, STRING* або *BOOK* у класі використовується умовне ім’я G.

Наприклад:

STACK [G]

put (element: G)

item: G

Усі сутності мають явно вказаний тип G, тому механізм статичної перевірки отримує необхідну інформацію. Водночас клас не визначає, яким саме буде G. Конкретний тип задається лише під час використання класу.

Якщо оголошено:

numbers: STACK [INTEGER]

то для *numbers* параметр *G* замінюється типом *INTEGER*.

Якщо оголошено:

texts: STACK [STRING]

то параметр *G* відповідає типу *STRING*.

Завдяки цьому одна програмна реалізація використовується для різних типів даних, але компілятор окремо перевіряє кожне її конкретне застосування.

Отже, параметризація типів потрібна для усунення дублювання класів, що відрізняються лише типом оброблюваних елементів. Вона особливо важлива для контейнерів, оскільки їхні основні алгоритми не залежать від типу вмісту. Формальний параметр дозволяє створити одну універсальну реалізацію, а фактичні параметри утворюють конкретні типи для цілих чисел, рядків та інших об’єктів. Такий механізм поєднує гнучкість і повторне використання програмного коду з читабельністю, надійністю та статичним контролем типів.

6.3 Узагальнені класи та їх параметри

Узагальнений клас — це клас, опис якого містить один або декілька параметрів, що представляють типи. Такий клас визначає спільну структуру та операції для цілої групи типів, не прив’язуючись до конкретного типу об’єктів. Наприклад, клас стека повинен мати явно типізовані елементи, але водночас залишатися придатним для побудови стеків довільних об’єктів. Для поєднання цих вимог класу надають ім’я умовного типу — формальний узагальнений параметр.

Узагальнений клас записують із параметрами у квадратних дужках після його назви:

STACK [G]

За прийнятою в книзі домовленістю перший узагальнений параметр позначають літерою *G* від слова Generic. Якщо клас потребує декількох параметрів, використовують наступні літери:

C [G, H, I]

Це правило є рекомендацією щодо стилю, а не обов’язковим синтаксичним обмеженням.

Формальний узагальнений параметр — це умовне ім’я типу, яке використовується всередині опису узагальненого класу. Під час написання класу ще невідомо, який конкретний тип він представлятиме.

Наприклад, спрощений опис стека має такий вигляд:

```
class STACK [G]
feature
    count: INTEGER
```

item: G
put (x: G)
remove
empty: BOOLEAN
end

У цьому класі *G* використовується як звичайна назва типу. Він може зазначатися:

- як тип аргументу процедури;
- як тип результату функції;
- як тип атрибута;
- як тип локальної сутності.

У наведеному прикладі функція item повертає значення типу G, а процедура put приймає аргумент такого самого типу:

item: G
put (x: G)

Отже, усі сутності класу мають явно визначені типи, що задовольняє вимоги статичної типізації. Водночас клас не містить відомостей про те, яким конкретно типом буде *G*, тому одна реалізація може використовуватися для різних видів об'єктів.

Узагальнений клас сам по собі не є конкретним типом, придатним для оголошення об'єктів. Він є шаблоном, або моделлю сімейства типів. Наприклад, *STACK [G]* описує загальну структуру стека, але не визначає, які саме елементи в ньому зберігатимуться.

Щоб використати узагальнений клас, клієнт повинен замінити кожний формальний параметр конкретним типом. Такий тип називається фактичним узагальненим параметром.

Наприклад:

- *sp: STACK [POINT]*
- У цьому оголошенні:
- *STACK* — узагальнений клас;
- *G* — формальний параметр у тексті класу;
- *POINT* — фактичний параметр;
- *STACK [POINT]* — конкретний тип;
- *sp* — сутність цього типу.

Після підстановки типу *POINT* усі входження *G* у класі *STACK* розглядаються як *POINT*. Отже, для об'єкта sp операція put приймає точку, а функція item повертає точку:

sp.put (p)
p := sp.item
де p має тип *POINT.*

Кількість фактичних параметрів повинна відповідати кількості формальних параметрів класу. Якщо клас оголошено як:

C [G, H]

то під час його використання необхідно вказати два фактичні типи:

C [TYPE_1, TYPE_2]

Перший фактичний параметр замінює *G*, а другий — *H*.

Надання фактичних параметрів узагальненому класу з метою отримання типу називається узагальненим виведенням, або generic derivation. Тип, отриманий у такий спосіб, називається узагальнено виведеним типом.

Наприклад:

STACK [POINT]

є типом, узагальнено виведеним із класу *STACK* із використанням типу *POINT* як фактичного параметра.

Узагальнене виведення одночасно потребує типу та створює тип:

- для виконання виведення потрібен уже визначений тип POINT;
- результатом виведення є новий тип STACK [POINT].

У книзі не рекомендується називати цей процес узагальненим створенням екземпляра. Створення екземпляра є подією часу виконання, під час якої з класу створюється конкретний об’єкт. Узагальнене виведення є статичним механізмом, який формує тип у тексті програми ще до її виконання.

Отже, потрібно розрізняти:

- узагальнене виведення — отримання типу STACK [POINT] із класу STACK [G];
- створення екземпляра — створення конкретного об’єкта типу STACK [POINT] під час виконання програми.

Узагальнений клас може мати не лише один, а декілька формальних параметрів:

C [G, H, I]

Кожний параметр представляє окремий тип і може використовуватися незалежно в атрибутах, аргументах та результатах операцій. Під час узагальненого виведення клієнт повинен надати фактичний тип для кожного формального параметра у відповідній позиції.

Наприклад, для класу:

C [G, H]

тип:

C [POINT, INTEGER]

означає, що в межах цього конкретного виведення:

- *G* представляє *POINT*;
- *H* представляє *INTEGER*.

Механізм із декількома параметрами потрібний для компонентів, які одночасно працюють із різними категоріями даних. При цьому кожна категорія зберігає власний тип, а компілятор може окремо перевіряти правильність її використання.

Фактичним параметром може бути не лише звичайний тип, а й інший узагальнено виведений тип. Це дозволяє створювати вкладені структури.

Якщо визначено класи:

STACK [G]

LIST [G]

то можна оголосити стек списків точок:

slp: STACK [LIST [POINT]]

Зовнішній клас STACK отримує як фактичний параметр тип *LIST [POINT]*. Отже, кожний елемент такого стека є списком точок.

Можна також створити стек стеків точок:

ssp: STACK [STACK [POINT]]

У цьому випадку елементами зовнішнього стека є об’єкти типу *STACK [POINT]*.

Вкладеність може продовжуватися й далі:

STACK [LIST [STACK [POINT]]]

Формального обмеження на глибину такого вкладення немає. Проте надмірно складні конструкції погіршують читабельність програмного тексту, тому їх слід використовувати лише тоді, коли вони відповідають структурі розв’язуваної задачі.

Універсалізація гарантує, що конкретна структура даних міститиме елементи визначеного типу. Розглянемо оголошення:

sc: STACK [CIRCLE]

sa: STACK [ACCOUNT]

c: CIRCLE

a: ACCOUNT

Правильними є такі операції:

sc.put (c)

sa.put (a)

c := sc.item

До стека кіл додається об’єкт типу *CIRCLE*, до стека рахунків — об’єкт типу *ACCOUNT*, а функція item стека кіл повертає результат типу *CIRCLE*.

Неправильними є операції:

sc.put (a)

sa.put (c)

c := sa.item

Перша операція намагається додати банківський рахунок до стека кіл. Друга додає коло до стека рахунків. У третій результат типу *ACCOUNT* намагаються

присвоїти сутності типу *CIRCLE*. Усі ці випадки повинні бути відхилені під час статичної перевірки типів.

Загальне правило можна пояснити так. Нехай узагальнений клас *C [G]* містить операцію:

h (a: G): G

а клієнт має сутність:

y: C [V]

У виклику:

y.h (e)

формальний тип *G* замінюється фактичним типом *V*. Тому аргумент e повинен мати тип *V*, а результат операції також матиме тип *V*.

Для стека це означає:

- якщо *s* має тип *STACK [POINT]*, то *s.put (z)* потребує аргументу типу *POINT*, а *s.item* повертає *POINT*;
- якщо *s* має тип *STACK [INTEGER]*, то *s.put (z)* потребує аргументу типу *INTEGER*, а *s.item* повертає *INTEGER*.

Після введення успадкування правило сумісності буде розширене: у певних випадках замість точного типу можна буде використовувати сумісний тип на основі класу-нащадка. У межах базової універсалізації головним є правило відповідності формального параметра фактичному типу, вказаному під час узагальненого виведення.

Оскільки формальний параметр *G* може представляти довільний тип, усередині узагальненого класу до сутностей типу *G* можна застосовувати лише операції, допустимі для будь-якого типу. Такі сутності можна:

- присвоювати іншим сумісним сутностям;
- передавати як аргументи операціям, що очікують тип *G*;
- порівнювати на рівність або нерівність;
- копіювати за допомогою загальних операцій, доступних усім об’єктам.

Натомість усередині класу *STACK [G]* не можна припускати, що *G* має арифметичні, текстові або інші спеціалізовані операції. Наприклад, для двох значень типу *G* не можна без додаткових обмежень виконувати додавання, оскільки фактичним параметром може стати тип, для якого така операція не визначена.

Отже, узагальнений клас є параметризованим шаблоном типів. Формальні параметри використовуються в його описі як умовні назви типів, а фактичні параметри задаються клієнтом під час отримання конкретного типу. Узагальнені класи можуть мати декілька параметрів і підтримують вкладені типи. Статична перевірка гарантує, що операції над кожним узагальнено виведеним типом виконуються лише з сумісними об’єктами, поєднуючи повторне використання однієї реалізації з надійністю типізованого програмного коду.

6.4 Масиви як приклад узагальненого класу

Масив є одним із найпоширеніших прикладів контейнерної структури. У традиційних мовах програмування масив часто розглядається як спеціальна конструкція, безпосередньо вбудована в синтаксис мови. Об'єктно-орієнтований підхід дозволяє не вводити для нього окремого механізму: масив можна представити як звичайний об'єкт — екземпляр класу ARRAY. Клас ARRAY описує послідовність значень одного типу або сумісних із ним типів, доступ до яких здійснюється за цілими індексами з певного неперервного діапазону.

Масиви можуть містити цілі числа, рядки, точки, банківські рахунки та інші об'єкти. Структура масиву й основні операції над ним при цьому залишаються однаковими. Змінюється лише тип елементів. Тому клас масиву доцільно визначити як узагальнений:

```
class ARRAY [G]
creation
   make
feature
   make (minindex, maxindex: INTEGER)
      -- Створити масив із нижньою межею minindex
      -- і верхньою межею maxindex.
   do
      ...
   end
   lower, upper, count: INTEGER
      -- Мінімальний і максимальний допустимі індекси
      -- та кількість елементів.
   put (v: G; i: INTEGER)
      -- Записати значення v в елемент з індексом i.
   do
      ...
   end
   item (i: INTEGER): G
      -- Повернути значення елемента з індексом i.
   do
      ...
   end
end
```

Параметр *G* визначає тип елементів масиву. Завдяки цьому одна реалізація класу *ARRAY* використовується для масивів різного вмісту. У межах класу *G*

застосовується як звичайна назва типу: процедура put приймає значення типу *G*, а функція item повертає значення того самого типу.

Сам клас *ARRAY [G]* є шаблоном типів. Для отримання конкретного типу необхідно замінити формальний параметр *G* фактичним типом. Наприклад:

ARRAY [INTEGER]

визначає масив цілих чисел;

ARRAY [STRING]

визначає масив рядків;

ARRAY [POINT]

визначає масив об'єктів типу *POINT;*

ARRAY [BOOK]

визначає масив книг.

У кожному випадку використовується однакова структура масиву й однакові операції. Водночас компілятор точно знає тип його елементів. Для *ARRAY [POINT]* процедура *put* приймає значення типу *POINT*, а функція item повертає об'єкт типу *POINT*.

Тип елементів визначає допустимі операції з конкретним масивом. До масиву *ARRAY [INTEGER]* не можна записати рядок, а до масиву *ARRAY [BOOK]* — об'єкт несумісного класу. Отже, узагальненість не послаблює типізацію, а забезпечує повторне використання класу зі збереженням статичного контролю типів.

Для сутності a, оголошеної як *ARRAY [T]*, масив із межами *m* і *n* створюється за допомогою операції:

create a.make (m, n)

Процедура make отримує два цілі аргументи:

- *minindex* — нижню межу індексів;
- *maxindex* — верхню межу індексів.

На відміну від мов, у яких індексація масиву обов'язково починається з нуля або одиниці, клас *ARRAY* дозволяє визначити довільний неперервний діапазон індексів. Наприклад:

create pa.make (-32, 101)

створює масив, допустимі індекси якого перебувають у межах від -32 до 101.

Розглянемо повний приклад створення масиву точок:

```
pa: ARRAY [POINT]
p1: POINT
i, j: INTEGER
...
create pa.make (-32, 101)
pa.put (p1, i)
...
```

p1 := pa.item (j)

У цьому прикладі:

- *pa* є масивом об’єктів типу *POINT*;
- процедура *make* створює масив із визначеними межами;
- виклик put записує точку *p1* у позицію *i*;
- функція item повертає точку з позиції *j*.

Аналогічно можна створити масиви інших типів:

numbers: ARRAY [INTEGER]
names: ARRAY [STRING]
books: ARRAY [BOOK]
create numbers.make (1, 100)
create names.make (0, 49)
create books.make (1, 20)

Кожна сутність використовує той самий узагальнений клас *ARRAY*, але представляє окремий конкретний тип.

Клас *ARRAY [G]* містить операцію створення, запити про межі та розмір, операцію зміни елемента і функцію доступу до елемента.

Процедура make створює масив із заданими межами:

make (minindex, maxindex: INTEGER)

Після створення значення *lower і upper* визначають допустимий діапазон індексів.

Запит *lower* повертає найменший допустимий індекс:

lower: INTEGER

Запит upper повертає найбільший допустимий індекс:

upper: INTEGER

Запит count повертає кількість елементів масиву:

count: INTEGER

Для непорожнього масиву кількість елементів визначається співвідношенням:

count = upper − lower + 1

Запит count може бути реалізований як атрибут або як функція, що обчислює значення на основі меж масиву. З погляду клієнта спосіб реалізації не має значення.

Процедура *put* змінює значення елемента:

put (v: G; i: INTEGER)

Виклик:

a.put (x, i)

записує значення x у позицію з індексом i.

Функція item забезпечує доступ до елемента:

item (i: INTEGER): G

Виклик:

x := a.item (i)

повертає значення елемента з індексом i.

У традиційному записі ці дві операції могли б мати вигляд:

a [i] := x

x := a [i]

В об'єктно-орієнтованій моделі доступ і зміна елементів подаються як звичайні операції об'єкта. Масив не потребує окремої спеціальної конструкції мови, оскільки є контейнером із власним способом доступу до елементів.

Коректне звернення до елемента потребує, щоб індекс належав допустимому діапазону:

$lower \leq i \leq upper$

У подальшій лекції про проєктування за контрактом це обмеження буде виражено передумовами операцій put і item. У межах цієї лекції достатньо зафіксувати, що операції доступу й зміни можуть застосовуватися лише до чинних індексів масиву.

Клас *ARRAY* описує одновимірні масиви. Для структур із більшою кількістю вимірів можуть використовуватися аналогічні класи:

- *ARRAY2 [G]*
- *ARRAY3 [G]*

Вони також параметризуються типом елементів, але для доступу до значення потребують відповідно двох або трьох індексів. Принцип універсалізації залишається незмінним: один клас використовується для багатовимірних масивів будь-якого допустимого типу.

Без універсалізації для кожного типу даних довелося б створити окремий клас:

- *INTEGER_ARRAY*
- *STRING_ARRAY*
- *POINT_ARRAY*
- *BOOK_ARRAY*

Такі класи містили б однакові алгоритми створення, індексованого доступу, зміни елементів і визначення меж. Відрізнявся б лише тип значень, що зберігаються.

Узагальнений клас:

ARRAY [G]

усуває це дублювання. Клас реалізується, тестується і супроводжується один раз, а потім використовується з різними фактичними параметрами:

Таблиця 6.2 – Приклади використання узагальненого класу *ARRAY*

Конкретний тип	**Тип елементів**	**Приклад значення**
ARRAY [INTEGER]	цілі числа	*25*

ARRAY [STRING]	рядки	*"Text"*
ARRAY [POINT]	точки	*p1*
ARRAY [BOOK]	книги	*book1*

Зміни або виправлення в реалізації класу *ARRAY* автоматично стосуються всіх його узагальнено виведених типів. Не потрібно окремо оновлювати класи масивів цілих чисел, рядків або інших об’єктів.

Представлення масиву як звичайного класу також забезпечує його участь у загальних механізмах об’єктно-орієнтованого програмування. Наприклад, інші класи можуть використовувати ARRAY як основу для реалізації складніших контейнерів. У книзі наведено клас *ARRAYED_LIST*, який реалізує абстрактне поняття списку за допомогою масиву.

Використання операцій item і put не обов’язково погіршує ефективність порівняно з традиційним індексованим доступом. Компілятор може розпізнавати клас *ARRAY* і перетворювати виклики цих операцій на безпосередній доступ до елементів. Для розробника *ARRAY* залишається звичайним класом, тоді як його низькорівнева реалізація може бути оптимізована середовищем виконання.

Отже, масив є наочним прикладом узагальненого контейнерного класу. Клас *ARRAY [G]* визначає спільну структуру та операції для масивів будь-якого типу, а формальний параметр G задає тип їхніх елементів. Конкретні типи утворюються підстановкою фактичних параметрів, наприклад *ARRAY [INTEGER], ARRAY [STRING]* або *ARRAY [POINT]*. Одна реалізація забезпечує створення масивів, доступ до елементів, зміну значень і визначення меж для різних типів даних, поєднуючи повторне використання програмного коду зі статичною перевіркою типів.

6.5 Переваги, вартість та обмеження універсалізації

Універсалізація призначена для підвищення повторного використання, розширюваності та надійності програмного забезпечення. Водночас її застосування не повинно спричиняти суттєвих витрат часу виконання або пам’яті. Тому під час оцінювання узагальнених класів необхідно враховувати не лише переваги для розроблення, а й можливі витрати на компіляцію та виконання програми.

Основна перевага універсалізації полягає в тому, що один клас може використовуватися для роботи з різними типами даних. Без параметризації довелося б створювати окремі класи:

- *INTEGER_LIST*
- *STRING_LIST*
- *BOOK_LIST*

- *POINT_LIST*

Їхні операції були б майже однаковими, а відмінність полягала б лише в типі елементів. Узагальнений клас:

LIST [G]

замінює всі ці окремі реалізації. Алгоритми додавання, видалення, пошуку та доступу до елементів описуються один раз, а конкретний тип задається під час використання класу:

- *LIST [INTEGER]*
- *LIST [STRING]*
- *LIST [BOOK]*
- *LIST [POINT]*

Унаслідок цього зменшується обсяг вихідного коду, а зміни й виправлення потрібно вносити лише в одну реалізацію.

Підвищення повторного використання класів

Узагальнений клас описує спільну структуру та поведінку цілої групи типів. Наприклад, стек може зберігати будь-які об’єкти, оскільки правила додавання й видалення елементів не залежать від їхнього конкретного типу:

- *STACK [INTEGER]*
- *STACK [ACCOUNT]*
- *STACK [POINT]*

Один клас можна повторно використовувати в багатьох програмних системах і для різних предметних областей. Це особливо важливо під час створення бібліотек програмних компонентів, де кожний клас повинен охоплювати не один окремий випадок, а широку групу подібних задач.

Універсалізація не означає використання контейнера для довільних і невідомих об’єктів. Під час отримання конкретного типу клієнт указує фактичний параметр: accounts: *STACK [ACCOUNT]*

Після цього компілятор знає, що процедура put повинна приймати об’єкт типу *ACCOUNT*, а функція *item* — повертати значення цього типу.

Спроба додати до такого стека об’єкт несумісного типу повинна бути виявлена під час компіляції. Отже, універсалізація поєднує дві властивості:

- повторне використання однієї реалізації;
- статичну перевірку типів для кожного конкретного застосування.

У книзі підкреслено, що універсалізація потрібна саме в типізованій мові, оскільки дозволяє зберігати типобезпечність, яку можна перевірити до виконання програми.

Узагальнені класи особливо добре підходять для опису контейнерних структур, реалізація яких не залежить від типу елементів. До них належать:

- масиви;

- списки;
- стеки;
- черги;
- множини;
- дерева.

Замість окремого класу для кожної комбінації структури й типу даних бібліотека може містити універсальні класи:

- *ARRAY [G]*
- *LIST [G]*
- *STACK [G]*
- *QUEUE [G]*

Користувач бібліотеки сам визначає фактичний тип елементів. Це зменшує кількість класів у бібліотеці, забезпечує узгоджені назви операцій і спрощує її розвиток та тестування.

Питання вартості універсалізації особливо пов’язане зі способом її реалізації в компіляторі. Деякі компілятори можуть буквально трактувати параметризацію та створювати окрему копію програмного коду для кожного фактичного параметра.

Наприклад, якщо програма використовує:

- *LIST [INTEGER]*
- *LIST [STRING]*
- *LIST [WIDGET]*
- *LIST [BLIDGET]*

компілятор може згенерувати окремі версії операцій head, tail та insert для кожного з чотирьох типів. Якщо узагальнений клас широко використовується з багатьма параметрами, це може спричинити створення значної кількості повторюваного машинного коду.

Можливими наслідками такого підходу є:

- збільшення часу компіляції;
- збільшення розміру згенерованого коду;
- збільшення обсягу пам’яті, потрібної для виконання;
- додаткові витрати на опрацювання великої кількості реалізацій.

Проте ці витрати не є обов’язковою властивістю універсалізації. Вони залежать від архітектури мови та якості компілятора.

За належного проєктування мови та компілятора універсалізація не повинна призводити до дублювання машинного коду. Для одного узагальненого класу можна створити спільну реалізацію, яка використовуватиметься для різних фактичних параметрів.

У книзі зазначено, що за якісної реалізації вплив універсалізації повинен бути малим або відсутнім для таких показників:

- час компіляції;
- розмір згенерованого коду;
- час виконання;
- обсяг пам’яті під час виконання.

Тоді розробник може використовувати універсалізацію без побоювання, що вона суттєво погіршить продуктивність програми.

Робота компілятора при цьому не є простою. Він повинен:

- перевірити правильність підстановки фактичних типів;
- забезпечити статичну сумісність усіх операцій;
- визначити спосіб спільного використання згенерованого коду;
- зберегти ефективність доступу до даних;
- узгодити універсалізацію з іншими механізмами об’єктно-орієнтованого програмування.

Таким чином, складність переноситься з прикладного програмного коду до реалізації мови та компілятора. Саме середовище повинно захищати розробника від необхідності вручну оптимізувати кожне використання узагальненого класу.

Універсалізація, розглянута в цій лекції, має певні обмеження. Формальний параметр G представляє довільний тип, тому в межах класу до об’єктів типу G можна застосовувати лише операції, доступні для всіх типів.

Наприклад, клас: *VECTOR [G]* не може автоматично виконувати додавання елементів типу G, оскільки не кожний фактичний тип підтримує операцію додавання. Для такого класу необхідно встановити вимогу, що фактичний параметр має певну операцію. Цей підхід називають обмеженою універсалізацією.

Інше обмеження стосується поліморфних структур. Наприклад, у графічній програмі може знадобитися стек, який містить точки, кола та прямокутники. Базовий механізм вимагає точного типу фактичного параметра й сам по собі не пояснює, як безпечно зберігати в одному контейнері об’єкти різних, але пов’язаних типів.

Розв’язання цих питань потребує поєднання універсалізації з успадкуванням. Оскільки успадкування є темою наступних лекцій, у цьому розділі достатньо зафіксувати наявність таких обмежень.

Узагальнений клас доцільно створювати тоді, коли:

- декілька класів мають однакову структуру й алгоритми, але працюють із різними типами;
- тип елементів не впливає на основну логіку компонента;
- компонент планується використовувати в різних частинах системи або різних проєктах;
- необхідно зберегти статичний контроль типів;
- потрібно створити універсальну контейнерну структуру.

Не варто вводити універсалізацію лише для можливого використання в невизначеному майбутньому. Якщо компонент працює тільки з одним конкретним типом і його логіка безпосередньо залежить від властивостей цього типу, звичайний клас може бути простішим і зрозумілішим.

Отже, вибір універсалізації має ґрунтуватися на реальній спільності структури та поведінки різних типів, а не лише на бажанні зробити клас максимально загальним.

Узагальнення переваг і можливих витрат наведено в таблиці 6.3.

Таблиця 6.3 – Переваги та можливі витрати універсалізації

Аспект	**Характеристика**
Усунення дублювання програмного коду	Одна реалізація використовується замість багатьох однотипних класів.
Повторне використання класів	Один узагальнений клас застосовується для роботи з різними типами даних.
Типобезпечність	Статична перевірка правильності використання фактичних параметрів і операцій над ними.
Побудова бібліотек компонентів	Спрощується створення універсальних контейнерів і повторно використовуваних програмних компонентів.
Час компіляції	Може збільшуватися, якщо компілятор генерує окремі реалізації для різних параметрів типу.
Розмір машинного коду	Може зростати через дублювання згенерованого коду для різних конкретних типів.
Реалізація компілятора	Потребує складніших механізмів аналізу, перевірки типів та оптимізації.
Обмеження	Для підтримки спеціалізованих операцій і поліморфних структур універсалізацію необхідно поєднувати з механізмом успадкування.

Отже, універсалізація усуває дублювання програмного коду, підвищує повторне використання класів і зберігає статичну типобезпечність. Вона є основою для побудови універсальних бібліотек контейнерів. Можливі витрати у вигляді збільшення часу компіляції або розміру машинного коду виникають переважно через недосконалу реалізацію компілятора, а не через сам принцип параметризації. За належної підтримки універсалізація може застосовуватися без суттєвого впливу на час виконання й використання пам’яті.

Висновки

Універсалізація є одним із основних механізмів повторного використання програмних компонентів в об’єктно-орієнтованому програмуванні. Вона дає змогу створювати класи, структура та поведінка яких не залежать від конкретного типу даних, що дозволяє використовувати одну реалізацію для різних типів об’єктів.

Основою універсалізації є формальні родові параметри, які задаються під час оголошення класу та замінюються конкретними типами при його використанні. Завдяки цьому класи на зразок *LIST [G], STACK [G]* або *ARRAY [G]* можуть працювати з різними типами елементів без дублювання програмного коду.

Необмежена універсалізація застосовується тоді, коли для родового параметра не висуваються спеціальні вимоги. Якщо ж алгоритм повинен використовувати певні операції або властивості об’єктів, застосовується обмежена універсалізація, яка встановлює вимоги до допустимих фактичних типів.

Поєднання універсалізації з успадкуванням і поліморфізмом забезпечує побудову гнучких та типобезпечних програмних структур. Універсалізація відповідає за варіювання типів, з якими працює компонент, тоді як успадкування дає змогу спеціалізувати або розширювати його поведінку.

Правильне використання універсалізації зменшує дублювання коду, підвищує повторне використання класів, забезпечує контроль типів на етапі компіляції та спрощує подальше розширення програмних систем. Тому універсальні класи є важливою складовою побудови бібліотек і повторно використовуваних компонентів об’єктно-орієнтованого програмного забезпечення.

Лекція 7

Тема: Проєктування за контрактом

Мета: сформувати у студентів розуміння концепції проєктування за контрактом як засобу підвищення надійності об’єктно-орієнтованого програмного забезпечення; ознайомити з призначенням передумов, постумов та інваріантів класу, принципами розподілу відповідальності між клієнтом і постачальником та застосуванням контрактів для специфікації, перевірки й документування поведінки програмних компонентів.

План лекції

1. Поняття проєктування за контрактом
2. Передумови та постумови
3. Інваріанти класу
4. Контракти в описі класів та операцій
5. Перевірка контрактів і оброблення порушень
6. Контракти, успадкування та повторне визначення операцій
7. Переваги та практичне застосування проєктування за контрактом
8. Висновки

Проєктування за контрактом — це підхід до розроблення програмного забезпечення, за якого поведінка програмних компонентів і правила їхньої взаємодії визначаються за допомогою явних формальних умов. Контракт установлює взаємні зобов’язання між програмним компонентом та його клієнтом: клієнт повинен виконати визначені умови перед викликом операції, а компонент гарантує певний результат після її завершення. У межах лекції буде розглянуто основні поняття проєктування за контрактом, зокрема передумови, постумови та інваріанти класів, а також їхню роль у забезпеченні правильності, надійності й зрозумілості програмного забезпечення. Окрему увагу буде приділено перевагам контрактного підходу та практичним прикладам його застосування в об’єктно-орієнтованому програмуванні.

7.1 Поняття проєктування за контрактом

Розглянуті раніше поняття класу, об’єкта та універсалізації дають змогу створювати модульні, розширювані й повторно використовувані програмні компоненти. Однак гнучкість і повторне використання не повинні досягатися за рахунок надійності. Програмна система має бути не лише зручною для розширення, а й правильною та стійкою до помилок.

Клас є реалізацією певного абстрактного типу даних. Тому він повинен описувати не лише набір доступних операцій, а й семантичні властивості цих операцій: за яких умов їх можна викликати та який результат вони повинні

забезпечити. Без такого опису інтерфейс класу показує лише назви й типи операцій, але не визначає повністю їхню поведінку.

Правильність програмного компонента не можна оцінювати окремо від його специфікації. Програма, операція або окрема інструкція не є правильною чи неправильною сама по собі. Вона є правильною лише щодо конкретного опису того, що повинна виконувати.

Наприклад, інструкція: *x := y + 1* може бути правильною щодо вимоги «значення x і y повинні відрізнятися», але неправильною щодо вимоги «значення x повинно бути від'ємним». Отже, питання правильності стосується не окремого програмного елемента, а пари, що складається з програмного елемента та його специфікації.

Проєктування за контрактом — це підхід, за якого відносини між програмним компонентом і його клієнтами розглядаються як формальна угода. Така угода визначає права та обов'язки кожної сторони.

У контрактній взаємодії розрізняють:

- **клієнта** — програмний модуль, який викликає певну операцію;
- **постачальника** — клас або операцію, що надає клієнтові певну послугу.

Клієнт зобов'язується виконати умови, необхідні для правильного виклику операції. Постачальник, зі свого боку, зобов'язується виконати операцію та забезпечити визначений результат. Тільки точне визначення вимог і відповідальності кожного програмного модуля дає змогу підвищити довіру до великих програмних систем.

Подібно до контракту між людьми або організаціями, програмний контракт визначає не лише обов'язки, а й переваги для обох сторін. Обов'язок однієї сторони зазвичай є перевагою для іншої:

- клієнт зобов'язаний викликати операцію лише за допустимих умов;
- постачальник отримує право припускати, що ці умови виконані;
- постачальник зобов'язаний забезпечити обіцяний результат;
- клієнт отримує право покладатися на цей результат.

Загальний зміст контракту можна сформулювати так:

> Якщо клієнт викликає операцію за умов, визначених контрактом, постачальник гарантує результат, також визначений контрактом.

Проєктування за контрактом спирається на твердження. Твердження — це логічний вираз, який описує властивість програмних сутностей у певний момент виконання. Наприклад:

n > 0

x /= Void

Перше твердження означає, що значення *n* повинно бути додатним, а друге — що посилання *x* не повинно бути порожнім. У програмному тексті твердження подаються як булеві вирази, які можуть набувати значення *True* або *False*.

Твердження використовуються не лише для перевірки програми під час виконання. Їхнє основне призначення полягає в тому, щоб:

- точно визначати поведінку класів та операцій;
- допомагати проєктувати правильне програмне забезпечення;
- полегшувати розуміння програмного коду;
- створювати основу для документації;
- підтримувати систематичне тестування та налагодження.

Запис специфікації одночасно з програмним кодом або ще до його реалізації допомагає краще зрозуміти задачу й побудувати програму, правильність якої закладена в її проєкт.

У межах контрактного підходу надійність програмного забезпечення охоплює дві основні властивості:

- правильність — здатність програмного забезпечення працювати відповідно до своєї специфікації;
- робастність — здатність належним чином реагувати на ситуації, які не передбачені специфікацією.

Твердження та контракти насамперед стосуються правильності. Реагування на порушення контракту й непередбачені ситуації пов’язане з механізмом оброблення винятків.

Отже, проєктування за контрактом доповнює програмний код його точною семантичною специфікацією. Клас визначає не тільки те, які операції він надає, а й те, за яких умов вони можуть виконуватися та які результати гарантують. Це дає змогу чітко розподілити відповідальність між клієнтом і постачальником та зробити правильність частиною самого проєкту програмної системи.

7.2 Передумови та постумови

Першим застосуванням тверджень у проєктуванні за контрактом є специфікація окремих операцій. Операція є не просто фрагментом програмного коду. Вона повинна виконувати певне корисне завдання, і це завдання необхідно описати точно. Неможливо переконатися в правильності операції, якщо заздалегідь не визначено, що саме вона повинна виконувати.

Завдання операції описується двома твердженнями:

- передумовою;
- постумовою.

Передумова визначає властивості, які повинні виконуватися перед викликом операції. Вона встановлює умови, за яких операція може працювати правильно.

Постумова визначає властивості, які операція гарантує після свого завершення, якщо під час виклику було виконано передумову.

У загальному вигляді специфікацію операції можна подати формулою правильності:

$$\{P\}\ A\ \{Q\}$$

де:

- A — операція;
- P — передумова;
- Q — постумова.

Така формула означає: будь-яке виконання операції A, розпочате у стані, в якому виконується P, повинно завершитися у стані, в якому виконується Q.

Наприклад: *{x >= 9} x := x + 5 {x >= 13}*

Передумова стверджує, що до виконання операції значення *x* не менше ніж *9*. Після додавання числа *5* значення x гарантовано буде не менше ніж *13*. Насправді можна сформулювати сильнішу постумову *x >= 14*, але й наведена формула залишається правильною.

У тексті класу передумови й постумови записуються як частини оголошення операції:

- передумова вводиться ключовим словом require;
- постумова вводиться ключовим словом ensure.

Загальна структура операції має такий вигляд:

```
operation
   require
      передумова
   do
      реалізація операції
   ensure
      постумова
   end
```

Обидві частини є необов’язковими. Операція може не мати передумови, постумови або обох цих частин, якщо для неї не потрібно задавати додаткові умови.

Розглянемо узагальнений клас стека *STACK [G]*. Для його операцій можна визначити такі властивості:

- операції remove і item можна застосовувати лише до непорожнього стека;
- операцію put можна застосовувати лише тоді, коли стек не заповнений;
- після виконання put кількість елементів збільшується на один;
- після виконання remove кількість елементів зменшується на один.

Ці властивості не залежать від конкретної реалізації стека. Вони описують його семантику і тому повинні бути явно включені до програмного тексту.

Операцію додавання елемента до стека можна описати так:

```
put (x: G)
   -- Додати x на вершину стека.
   require
      not full
   do
      ...
   ensure
      not empty
      item = x
      count = old count + 1
   end
```

Передумова: *not full* означає, що перед викликом put стек не повинен бути заповнений.

Постумови гарантують, що після завершення операції:

- стек не буде порожнім;
- його верхнім елементом буде доданий об’єкт x;
- кількість елементів збільшиться на один.

Операцію видалення можна описати так:

```
remove
   -- Видалити верхній елемент.
   require
      not empty
   do
      ...
   ensure
      not full
      count = old count - 1
   end
```

Передумова забороняє викликати операцію для порожнього стека. Постумови гарантують, що після видалення стек не буде заповненим, а кількість елементів зменшиться на один.

Передумова поширюється на всі виклики операції: як із клієнтських класів, так і з інших операцій того самого класу. У правильній програмній системі операція ніколи не повинна викликатися у стані, який не задовольняє її передумову.

Постумова описує стан після завершення операції. Вона є гарантією з боку розробника класу, але ця гарантія діє лише тоді, коли клієнт виконав передумову.

Для порівняння значень до і після виконання операції у постумовах використовується конструкція old. Вираз: *old e* позначає значення виразу e, яке він мав

перед початком виконання операції. Той самий вираз без *old* позначає значення після її завершення.

Наприклад: *count = old count + 1* означає, що нове значення count повинно бути на одиницю більшим за значення, яке цей атрибут мав перед викликом операції.

Передумови й постумови утворюють контракт між клієнтом і постачальником:

Таблиця 7.1 – Розподіл обов'язків і переваг між клієнтом та постачальником

Сторона контракту	**Обов'язок**	**Перевага**
Клієнт	Забезпечити виконання передумови перед викликом операції	Отримати результат, гарантований постумовою
Постачальник	Забезпечити виконання постумови після завершення операції	Припускати, що під час виклику операції її передумова виконана

Як видно з таблиці 7.1, передумова є обов'язком клієнта й одночасно перевагою постачальника, оскільки постачальник може не опрацьовувати випадки, заборонені передумовою. Постумова, навпаки, є обов'язком постачальника та перевагою клієнта, який має право розраховувати на гарантований результат.

Розглянутий розподіл відповідальності можна конкретизувати на прикладі операції put, призначеної для додавання елемента до стека. Контракт цієї операції наведено в таблиці 7.2.

Таблиця 7.2 – Обов'язки та переваги сторін контракту операції put

Сторона контракту	**Обов'язки**	**Переваги**
Клієнт	Викликати операцію put (x) лише для незаповненого стека	Отримати стек, у якому елемент x розміщений на вершині, стек не є порожнім, а кількість елементів збільшена на один
Постачальник	Розмістити елемент x на вершині стека, збільшити значення count на один і забезпечити непорожній стан стека	Виконувати операцію, припускаючи, що перед її викликом стек не був заповнений

Якщо клієнт не виконав передумову, постачальник більше не зобов'язаний виконувати постумову. Відповідальність за такий некоректний виклик лежить на клієнті.

Одним із важливих правил проєктування за контрактом є відсутність дублювання перевірок. Якщо певну вимогу вже визначено як передумову, тіло операції не повинно повторно перевіряти цю саму умову.

Наприклад:

```
square_root (x: REAL): REAL
   require
      x >= 0
   do
      ...
   end
```

Розробник операції обчислення квадратного кореня може реалізувати алгоритм, припускаючи, що *x* не є від'ємним. Перевірка цієї умови належить до відповідальності клієнта й уже подана в передумові. Додаткова перевірка $x < 0$ у тілі операції дублювала б контракт і змішувала б відповідальність клієнта та постачальника.

Передумови можуть бути сильнішими або слабшими. Сильна передумова допускає меншу кількість станів, у яких можна викликати операцію, тому полегшує роботу постачальника, але накладає більше обмежень на клієнта. Слабша передумова допускає більше можливих викликів і тому є вигіднішою для клієнта.

Для постумов ситуація протилежна. Сильна постумова гарантує більше властивостей і є вигіднішою для клієнта, але створює більше обов'язків для постачальника. Слабка постумова гарантує менше та спрощує реалізацію операції.

Отже, передумова визначає умови допустимого виклику операції, а постумова — результат, який операція повинна гарантувати. Разом вони точно розподіляють відповідальність між клієнтом і постачальником, перетворюють неявні припущення на частину програмного тексту та створюють основу для проєктування, документування, тестування й перевірки правильності програмних компонентів.

7.3 Інваріанти класу

Передумови та постумови описують властивості окремих операцій. Передумова визначає стан, у якому операцію дозволено викликати, а постумова — стан, який ця операція повинна забезпечити після завершення. Проте для повного опису класу цього недостатньо. Необхідно також визначити загальні властивості, які характеризують усі об'єкти класу й повинні зберігатися під час їхнього використання.

Такі властивості утворюють інваріант класу. Інваріант виражає загальні умови узгодженості та цілісності, що стосуються кожного екземпляра класу як цілого. На відміну від передумов і постумов, які пов'язані з конкретними операціями, інваріант описує фундаментальні властивості самого класу.

Розглянемо клас стека, реалізованого за допомогою масиву:

```
class STACK [G]
feature
   capacity: INTEGER
```

```
    count: INTEGER
feature {NONE} -- Реалізація
    representation: ARRAY [G]
end
```

Стан об’єкта цього класу визначається трьома атрибутами:

- representation — масив, у якому зберігаються елементи стека;
- capacity — максимальна місткість стека;
- count — поточна кількість елементів.

Передумови та постумови операцій put, remove та item описують правила їхнього виклику й результати виконання. Однак вони не виражають усіх зв’язків між атрибутами класу.

Наприклад, кількість елементів стека не може бути від’ємною:

0 <= count

Кількість елементів не може перевищувати місткість:

count <= capacity

Значення capacity повинно відповідати розміру масиву, який використовується як внутрішнє представлення:

capacity = representation.capacity

Ці умови стосуються не окремої операції, а кожного допустимого стану будь-якого об’єкта класу. Саме тому вони повинні бути виражені в інваріанті.

Інваріант може описувати не лише числові обмеження атрибутів, а й семантичні зв’язки між атрибутами та функціями. Наприклад, ознака порожнього стека повинна відповідати нульовій кількості елементів:

empty = (count = 0)

Для класу банківського рахунку можна визначити таку умову:

deposits_list.total - withdrawals_list.total = balance

Вона встановлює, що поточний баланс рахунку повинен дорівнювати загальній сумі внесень за вирахуванням загальної суми зняттів. Це є основною умовою узгодженості між операціями, збереженими в історії рахунку, та значенням його балансу.

Інваріант записується в секції invariant, розміщеній після всіх секцій feature і перед завершальним end класу:

```
class STACK [G]
feature
    ...
invariant
    count_non_negative:
        0 <= count
    count_bounded:
```

```
        count <= capacity
    consistent_with_array_size:
        capacity = representation.capacity
    empty_if_no_elements:
        empty = (count = 0)
    item_at_top:
        (count > 0) implies
        (representation.item (count) = item)
end
```

Кожній частині інваріанта може бути надано мітку, яка коротко пояснює її призначення:

- *count_non_negative* — кількість елементів не є від'ємною;
- *count_bounded* — кількість елементів не перевищує місткість;
- *consistent_with_array_size* — місткість узгоджена з розміром масиву;
- *empty_if_no_elements* — стек порожній тоді, коли кількість елементів дорівнює нулю;
- *item_at_top* — функція item повертає елемент, що міститься на вершині стека.

Інваріант є кон'юнкцією всіх записаних у ньому умов. Це означає, що коректний стан об'єкта повинен одночасно задовольняти кожну з них.

Інваріант не обов'язково повинен виконуватися в кожний момент роботи операції. Він повинен виконуватися в усіх стабільних станах об'єкта, тобто в моменти, коли його стан може спостерігатися клієнтом.

До стабільних станів належать:

- стан одразу після створення об'єкта;
- стан перед викликом експортованої операції;
- стан після завершення експортованої операції.

Отже, після виконання команди створення: *create a.make (...)* об'єкт a повинен задовольняти інваріант свого класу. Інваріант також повинен виконуватися до і після кожного кваліфікованого виклику: a.f (...) де f є доступною клієнтові операцією класу.

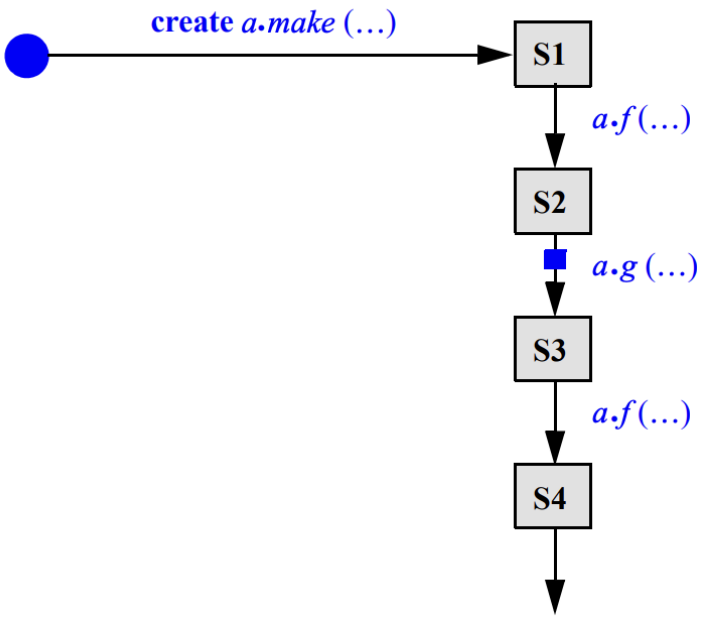


Рисунок 7.1 – Життєвий цикл об’єкта та його стабільні стани

На рисунку 7.1 стани *S1, S2, S3 і S4* є стабільними. Стан *S1* виникає після створення об’єкта. Наступні стабільні стани виникають після викликів операцій *f і g*. У кожному з цих станів інваріант повинен бути істинним.

Незважаючи на назву, інваріант може тимчасово порушуватися під час виконання операції. Процедура може спочатку змінити частину стану об’єкта так, що одна або декілька умов інваріанта перестануть виконуватися, а потім відновити узгодженість перед завершенням.

Наприклад, процедура банківського рахунку може спочатку додати нову операцію до списку зняттів, унаслідок чого значення balance тимчасово перестане відповідати спискам операцій. Після цього процедура оновлює баланс і відновлює інваріант:

deposits_list.total - withdrawals_list.total = balance

Тимчасове порушення інваріанта допустиме, якщо об’єкт у цей момент не доступний клієнтові, а перед завершенням експортованої операції інваріант буде відновлено.

Це означає, що інваріант описує не кожний проміжний стан виконання, а лише зовнішньо спостережувані стани об’єкта. Клієнт має право припускати, що перед викликом доступної операції об’єкт перебуває в узгодженому стані. Операція, зі свого боку, повинна залишити об’єкт у такому самому узгодженому стані після завершення.

Обов’язок зберігати інваріант поширюється на експортовані операції класу, тобто операції, доступні клієнтам. Кваліфікований виклик: *a.f (...)* повинен починатися і завершуватися у стані, що задовольняє інваріант.

Внутрішні некваліфіковані виклики: *f (...)* можуть використовуватися як допоміжні етапи виконання іншої операції. Вони не обов'язково повинні починатися та завершуватися зі справдженим інваріантом, оскільки клієнт не може безпосередньо спостерігати ці проміжні стани.

Отже, вимога відновлення інваріанта безпосередньо стосується:

- процедур створення;
- загальнодоступних операцій;
- операцій із вибірковим експортуванням, доступних певним клієнтам.

Секретні допоміжні операції можуть тимчасово працювати зі станом, який не задовольняє інваріант, але експортована операція, що їх викликає, повинна відновити його до повернення керування клієнтові.

Твердження I є правильним інваріантом класу C, якщо виконуються дві умови.

Перша умова: кожна процедура створення класу, викликана з аргументами, що задовольняють її передумову, повинна сформувати стан, у якому виконується інваріант.

Друга умова: кожна експортована операція, викликана у стані, що задовольняє інваріант і передумову операції, повинна завершитися у стані, який знову задовольняє інваріант.

У скороченому вигляді:

1. Створення об'єкта встановлює інваріант.
2. Кожна доступна клієнтам операція зберігає або відновлює інваріант.

Це правило гарантує, що всі об'єкти класу перебуватимуть у коректному стані в кожний момент, коли до них може звернутися клієнт.

Інваріант можна розглядати як умову, неявно додану до передумови та постумови кожної експортованої операції. Нехай:

- INV — інваріант класу;
- pre — передумова операції;
- body — тіло операції;
- post — її постумова.

Тоді вимога правильності має вигляд:

{INV and pre} body {INV and post}

Це означає, що операція починає виконання у стані, який задовольняє інваріант і передумову, та завершується у стані, який задовольняє інваріант і постумову.

Для розробника операції інваріант одночасно є перевагою та обов'язком:

- перевага полягає в тому, що на початку операції можна припускати виконання інваріанта;
- обов'язок полягає в необхідності відновити інваріант перед завершенням операції.

Наприклад, розробник операції класу *BANK_ACCOUNT* може припускати, що на початку її виконання значення списків внесень, зняттів і балансу є узгодженими. Проте після внесення змін операція повинна знову забезпечити цю узгодженість, щоб наступна операція могла покладатися на ту саму властивість.

Теоретично умови інваріанта можна було б додати до передумов і постумов кожної операції. Однак це призвело б до багаторазового дублювання однакових тверджень і ускладнило б описи операцій.

Крім того, інваріант має глибше значення. Він характеризує не окрему операцію, а клас у цілому. Інваріант поширюється не лише на операції, які вже існують, а й на ті, що можуть бути додані в майбутньому. Будь-яка нова експортована операція повинна зберігати фундаментальні властивості класу.

Операції та внутрішнє представлення класу можуть змінюватися під час розвитку програмної системи. Інваріант зазвичай відображає стабільніші семантичні властивості. Саме тому він є важливим засобом контролю подальших змін класу.

Інваріант допомагає відповісти на питання: що саме характеризує коректний об’єкт цього класу? Розуміння інваріанта часто дає глибше уявлення про призначення класу, ніж простий перелік його операцій.

Отже, інваріант класу є набором загальних умов цілісності, які повинні виконуватися для кожного об’єкта в усіх стабільних станах. Процедури створення встановлюють інваріант, а експортовані операції повинні його зберігати або відновлювати. Інваріант доповнює передумови й постумови та формує загальну семантичну специфікацію класу.

7.4 Контракти в описі класів та операцій

Контрактний опис класу поєднує програмну реалізацію з її семантичною специфікацією. Клас містить не лише атрибути, функції та процедури, а й твердження, які визначають допустимі умови використання його операцій, гарантовані результати та загальні правила узгодженості об’єктів.

Основними складниками контракту є:

- передумови операцій;
- постумови операцій;
- інваріант класу.

У програмному тексті вони записуються за допомогою конструкцій:

- *require* — для передумов;
- *ensure* — для постумов;
- *invariant* — для інваріанта класу.

Контрактний опис класу можна подати в такому вигляді:

class CLASS_NAME
creation

```
    make
feature -- Створення
    make (...)
        require
            умови_створення
        do
            реалізація
        ensure
            результат_створення
        end
feature -- Доступ
    query (...): TYPE
        require
            умови_виклику
        do
            реалізація
        ensure
            властивості_результату
        end
feature -- Зміна стану
    command (...)
        require
            умови_виклику
        do
            реалізація
        ensure
            результат_зміни
        end
invariant
    загальні_умови_цілісності
end
```

Передумови й постумови належать до конкретних операцій. Інваріант розміщується наприкінці класу та стосується всіх його об’єктів і всіх експортованих операцій.

Процедура створення має особливе значення в контрактному підході. До створення об’єкта його атрибути мають початкові значення за замовчуванням, які не завжди задовольняють інваріант. Завдання процедури створення полягає в тому, щоб перевести новий об’єкт у перший правильний стан.

Передумова процедури створення визначає допустимі аргументи ініціалізації. Її постумова описує властивості створеного об’єкта. Після завершення процедури повинен виконуватися не лише її власний ensure, а й інваріант класу.

Тому створення можна подати формулою:

{Default and pre} creation_body {post and INV}

де:

- Default — початкові значення атрибутів;
- pre — передумова процедури створення;
- post — її постумова;
- INV — інваріант класу.

Процедура створення повинна забезпечити, щоб кожен новий об’єкт починав своє існування у стані, який відповідає фундаментальним правилам класу.

Для кожної експортованої операції r правильність визначається так:

{pre_r and INV} Body_r {post_r and INV}

Операція може припускати, що:

- клієнт виконав її передумову;
- об’єкт перед викликом задовольняє інваріант.

Після завершення операція повинна гарантувати, що:

- виконується її постумова;
- інваріант класу відновлено.

Таким чином, контракт операції не обмежується явно записаними секціями require та ensure. Інваріант неявно входить до початкових і кінцевих умов кожної експортованої операції.

Клас є правильним щодо своїх тверджень тоді й лише тоді, коли реалізації всіх його операцій узгоджені з передумовами, постумовами та інваріантом.

Для процедур створення повинна виконуватися умова:

{Default_C and pre_p} Body_p {post_p and INV}

Для кожної експортованої операції:

{pre_r and INV} Body_r {post_r and INV}

Перша умова гарантує правильність початкового стану нових об’єктів. Друга гарантує, що кожний правильний об’єкт залишатиметься правильним після застосування доступної клієнтові операції.

Це можна пояснити як послідовність:

1. Процедура створення формує перший стан об’єкта, що задовольняє інваріант.
2. Клієнт викликає операцію лише за умови виконання її передумови.
3. Операція забезпечує свою постумову та відновлює інваріант.
4. Наступний клієнт отримує об’єкт у правильному стані.

Отже, коректність усіх можливих станів об’єкта встановлюється на основі правильності його створення та збереження інваріанта всіма експортованими операціями.

Клас *ARRAY [G]* є наочним прикладом того, як передумови, постумови та інваріант утворюють цілісний контракт.

```
class ARRAY [G]
creation
   make
feature -- Ініціалізація
   make (minindex, maxindex: INTEGER)
      -- Створити масив із заданими межами.
      require
         meaningful_bounds:
            maxindex >= minindex - 1
      do
         ...
      ensure
         exact_bounds_if_non_empty:
            (maxindex >= minindex) implies
            ((lower = minindex) and
             (upper = maxindex))
         conventions_if_empty:
            (maxindex < minindex) implies
            ((lower = 1) and
             (upper = 0))
      end
feature -- Доступ
   lower: INTEGER
      -- Найменший допустимий індекс.
   upper: INTEGER
      -- Найбільший допустимий індекс.
   count: INTEGER
      -- Кількість елементів.
   item (i: INTEGER): G
      -- Значення елемента з індексом i.
      require
         index_not_too_small:
            lower <= i
         index_not_too_large:
```

```
            i <= upper
        do
            ...
        end
    feature -- Зміна елементів
        put (v: G; i: INTEGER)
            -- Записати v у позицію i.
        require
            index_not_too_small:
                lower <= i
            index_not_too_large:
                i <= upper
        do
            ...
        ensure
            element_replaced:
                item (i) = v
        end
    invariant
        consistent_count:
            count = upper - lower + 1
        non_negative_count:
            count >= 0
    end
```

Передумова процедури *make*: *maxindex >= minindex - 1* дозволяє створювати непорожній масив або порожній масив із погодженими межами.

Постумови make визначають результат створення:

- якщо maxindex >= minindex, межі нового масиву повинні дорівнювати переданим значенням;
- якщо maxindex < minindex, створюється порожній масив із прийнятими умовними межами lower = 1 та upper = 0.

Передумови операцій item і put вимагають, щоб індекс перебував у допустимому діапазоні:

lower <= i
i <= upper

Клієнт відповідає за виконання цих умов. Тому реалізація операцій може припускати, що значення індексу є правильним.

Постумова процедури put: *item (i) = v* гарантує, що після завершення операції елемент у позиції і дорівнюватиме переданому значенню *v*.

Інваріант класу встановлює зв'язок між межами та кількістю елементів: *count = upper - lower + 1*, а також гарантує, що розмір масиву не може бути від'ємним:

count >= 0

Ці властивості повинні виконуватися після створення масиву та після завершення кожної його експортованої операції.

Окремим умовам контракту доцільно надавати змістовні мітки:

```
require
   index_not_too_small:
      lower <= i
   index_not_too_large:
      i <= upper
```

Мітка не змінює логічного значення твердження, але:

- пояснює його призначення;
- покращує читабельність класу;
- полегшує документацію;
- допомагає визначити порушену умову під час перевірки контракту.

Контракт при цьому залишається точним формальним описом, але водночас стає зрозумілішим для людини.

Постумова може порівнювати кінцевий стан об'єкта з його станом перед викликом операції. Для цього використовується конструкція *old*.

Наприклад: *count = old count + 1* означає, що після завершення операції нове значення count повинно бути на одиницю більшим за його початкове значення.

Для процедури видалення: *count = old count - 1* означає зменшення кількості елементів на один.

Без *old* вираз count у постумові позначає значення після виконання операції. Вираз old count позначає значення, яке атрибут мав під час входу в операцію.

Конструкція *old* використовується лише в постумовах, оскільки саме там потрібно описувати зміни між початковим і кінцевим станами.

Для опису значення, яке повертає функція, використовується попередньо визначена сутність Result.

Наприклад, функцію перевірки порожнього стека можна описати так:

```
empty: BOOLEAN
   do
      ...
   ensure
      definition:
         Result = (count = 0)
   end
```

У постумові *Result* позначає значення, повернене функцією. Такий запис установлює точний зв’язок між результатом операції та станом об’єкта.

Постумова функції може посилатися:

- на її результат через Result;
- на кінцевий стан об’єкта;
- на початкові значення через old;
- на аргументи функції.

Інваріант, на відміну від постумови, описує лише стан об’єкта і не пов’язаний з окремим викликом.

Операції класу можна поділити на запити та команди.

Запит повертає інформацію про стан об’єкта, не змінюючи його. Контракт запиту визначає:

- умови, за яких інформацію можна отримати;
- властивості поверненого результату.

Наприклад:

```
item (i: INTEGER): G
   require
      lower <= i
      i <= upper
   do
      ...
   ensure
      ...
   end
```

Команда змінює стан об’єкта. Її контракт визначає:

- допустимий початковий стан;
- зміну, яку повинна виконати команда;
- властивості кінцевого стану.

Наприклад:

```
put (v: G; i: INTEGER)
   require
      lower <= i
      i <= upper
   do
      ...
   ensure
      item (i) = v
   end
```

Усі команди повинні не лише виконувати власні постумови, а й залишати об’єкт у стані, що задовольняє інваріант.

Клас є реалізацією абстрактного типу даних. Абстрактний тип містить:

- назву типу;
- перелік функцій та їхні сигнатури;
- аксіоми, що визначають властивості результатів;
- передумови, що обмежують застосування функцій.

Якщо в описі класу залишити лише перелік операцій, його семантика буде визначена недостатньо. Наприклад, самі назви put, remove та item не гарантують, що вони реалізують поведінку стека. Саме твердження повертають до класу семантичні властивості відповідного абстрактного типу.

Передумови абстрактних функцій перетворюються на секції require. Аксіоми, що описують наслідки команд, перетворюються на ensure. Властивості, які пов’язують декілька запитів або атрибутів і стосуються класу в цілому, виражаються в invariant.

Отже, контрактний опис класу складається з передумов, постумов та інваріанта. Передумови визначають законні виклики операцій, постумови — гарантовані результати, а інваріант — загальні правила цілісності всіх об’єктів класу. Процедури створення повинні встановлювати інваріант, а кожна експортована операція — зберігати або відновлювати його. Завдяки цьому клас містить не лише реалізацію операцій, а й точну специфікацію їхньої поведінки.

7.5 Перевірка контрактів і оброблення порушень

Твердження контракту можуть контролюватися автоматично під час виконання програми. Рівень перевірки встановлюється параметрами компіляції, тому для зміни режиму контролю не потрібно редагувати текст класів. Якщо всі перевірені твердження мають значення True, виконання програми продовжується звичайним чином. Якщо певне твердження набуває значення False, виникає виняткова ситуація, яка за відсутності спеціального обробника зазвичай завершує виконання програми. Середовище може сформувати трасування викликів із зазначенням класу, операції, виду порушення та мітки невиконаної умови.

Розрізняють такі основні рівні контролю тверджень:

- no — перевірки вимкнено, а твердження виконують лише роль документації;
- require — перевіряються передумови під час входу в операцію;
- ensure — додатково перевіряються постумови після завершення операції;
- invariant — перевіряються інваріанти класу перед і після кваліфікованих викликів;
- loop — додатково контролюються інваріанти та варіанти циклів;

- check, або all, — перевіряються всі підтримувані твердження.

Кожний наступний рівень охоплює попередні. Наприклад, перевірка постумови має сенс лише тоді, коли перед викликом було підтверджено виконання передумови.

Порушення контракту вказує на наявність програмної помилки, але джерело помилки залежить від виду невиконаного твердження. Порушення передумови означає, що клієнт викликав операцію в недопустимому стані, тому помилка міститься в клієнтському коді. Порушення постумови означає, що операція була викликана правильно, але постачальник не забезпечив обіцяного результату. Порушення інваріанта також зазвичай указує на помилку в реалізації класу, оскільки процедура створення або експортована операція залишила об’єкт у неузгодженому стані.

Під час розроблення та тестування доцільно використовувати найповніший рівень контролю. Це дає змогу швидко виявляти порушення та визначати місце їх виникнення. У виробничій версії рівень перевірки обирають з урахуванням довіри до програмного забезпечення, вимог до швидкодії та можливих наслідків невиявленої помилки. Одним із практичних варіантів є збереження контролю передумов, оскільки він допомагає виявляти неправильне використання компонентів із меншими витратами, ніж перевірка всіх постумов та інваріантів.

Контракти не слід ототожнювати зі звичайною перевіркою вхідних даних. Передумови регулюють взаємодію між програмними модулями, але не можуть гарантувати правильність даних, отриманих від користувача, датчика або мережі. Зовнішні дані необхідно перевіряти за допомогою умовних конструкцій, спеціальних модулів валідації або механізму винятків. Після перевірки модуль введення може передати дані основній частині системи, гарантуючи виконання її передумов.

Твердження також не є керуючими конструкціями. Якщо операція повинна передбачати декілька нормальних варіантів поведінки, для цього використовують if, inspect або інші засоби керування. Порушення твердження описує не альтернативний сценарій, а невідповідність програми її специфікації.

7.6 Контракти, успадкування та повторне визначення операцій

Успадкування не скасовує контрактів батьківського класу. Клас-нащадок повинен зберігати всі зобов’язання, які були визначені для його предків. Це необхідно через поліморфізм і динамічне зв’язування: клієнт може працювати із сутністю батьківського типу, не знаючи, що під час виконання вона посилається на об’єкт класу-нащадка.

Нехай клас A містить операцію r із передумовою α і постумовою β, а клас A' повторно визначає цю операцію. Клієнт, який працює з типом A, викликає операцію, виконуючи передумову α, і після завершення розраховує на постумову β. Тому версія операції у класі A' не може вимагати від клієнта більше або гарантувати йому менше.

Інакше об’єкт класу-нащадка не можна було б безпечно використовувати замість об’єкта батьківського класу.

Звідси випливає правило повторного визначення тверджень:

- передумова в класі-нащадку може залишатися незмінною або бути послаблена;
- постумова може залишатися незмінною або бути посилена.

Послаблення передумови означає, що нова операція приймає щонайменше всі виклики, допустимі для батьківської версії. Посилення постумови означає, що вона забезпечує щонайменше всі раніше гарантовані результати, а за потреби — додаткові властивості.

У мові Eiffel це правило підтримується спеціальними конструкціями:

require else

додаткова_умова

...

ensure then

додаткова_гарантія

Умова після require else логічно додається до успадкованої передумови за допомогою операції «або», тому загальна передумова стає слабшою або залишається незмінною. Умова після ensure then додається до успадкованої постумови за допомогою операції «і», тому загальна постумова стає сильнішою. Якщо ці конструкції відсутні, операція зберігає початковий контракт.

Інваріанти батьківських класів автоматично поширюються на нащадків. Власний інваріант класу-нащадка додається до успадкованих умов, а не замінює їх. Отже, об’єкт нащадка повинен задовольняти інваріанти всіх прямих і непрямих предків, а також власні додаткові обмеження.

Таке правило відповідає принципу підстановки: об’єкт класу-нащадка повинен безпечно використовуватися скрізь, де очікується об’єкт батьківського класу. Він повинен приймати всі законні запити до батьківського типу та надавати не слабші гарантії. У термінах проєктування за контрактом успадкування розглядається як передавання роботи субпідряднику: нащадок може виконати операцію ефективніше або забезпечити кращий результат, але не має права порушувати початкову угоду з клієнтом.

7.7 Переваги та практичне застосування проєктування за контрактом

Основна перевага контрактного підходу полягає в тому, що вимоги до правильності стають явною частиною програмного забезпечення. Передумови визначають допустиме використання операцій, постумови — гарантовані результати, а інваріанти — загальні умови цілісності об’єктів. Це зменшує кількість неявних припущень між компонентами та полегшує пошук джерела помилки.

Контракти сприяють розробленню правильного програмного забезпечення від самого початку. Запис тверджень до або одночасно з реалізацією допомагає точніше зрозуміти задачу, визначити відповідальність компонентів і перевірити відповідність реалізації її призначенню. Твердження також створюють основу для систематичного тестування та налагодження.

Передумови, постумови й інваріанти покращують документацію класів. Вони показують не лише сигнатури операцій, а й умови їх використання та результати. Документація може автоматично формуватися з тексту класу, тому зменшується ризик розходження між реалізацією та окремим описом. Такий контрактний інтерфейс особливо важливий для повторно використовуваних бібліотек, оскільки клієнт може зрозуміти правила використання компонента без аналізу його внутрішньої реалізації.

Висновки

Контрактний підхід також підтримує повторне використання та розширення класів. Чітка специфікація дозволяє замінювати реалізації, повторно визначати операції та створювати класи-нащадки, не змінюючи очікувань наявних клієнтів. Перевірка тверджень під час виконання допомагає швидко виявляти помилки, які залишилися після проєктування та тестування.

Контракти мають і певні обмеження. Складні семантичні властивості не завжди можна повністю виразити звичайними булевими умовами. Перевірка багатих постумов та інваріантів може також збільшувати час виконання. Крім того, контракти не замінюють валідацію зовнішніх даних, керування нормальними альтернативними сценаріями та оброблення виняткових ситуацій.

Отже, проєктування за контрактом доцільно застосовувати для компонентів із чітко визначеними правилами взаємодії, особливо в бібліотеках, складних об’єктно-орієнтованих системах і програмному забезпеченні з високими вимогами до надійності. Воно не усуває потреби в тестуванні або обробленні помилок, але робить специфікацію поведінки явною та допомагає виявляти порушення ближче до місця їх виникнення.

Лекція 8

Тема: Коли контракт порушується

Мета: сформувати у студентів розуміння понять винятку та відмови програмної операції, ознайомити з основними причинами порушення програмних контрактів, принципами дисциплінованого оброблення винятків, механізмами rescue і retry, правилами відновлення інваріанта класу та засобами запобігання повторним помилкам під час виконання програмного забезпечення.

План лекції

1. Основні поняття винятку та відмови.
2. Причини виникнення винятків і порушень контракту.
3. Принципи дисциплінованого оброблення винятків.
4. Механізм оброблення винятків rescue і retry.
5. Приклади оброблення винятків.
6. Завдання секції rescue та відновлення інваріанта.
7. Розширене оброблення винятків і запобігання повторним порушенням.
8. Висновки.

Незважаючи на застосування статичної типізації, контрактів, перевірки тверджень та інших засобів забезпечення правильності, під час виконання програмної системи можуть виникати непередбачені й небажані події. Така подія може перервати нормальне виконання операції та поставити під загрозу виконання її контракту.

Виняткові ситуації можуть бути спричинені програмними дефектами, порушенням передумов і постумов, помилками доступу до пам’яті, арифметичним переповненням, нестачею ресурсів, відмовами зовнішніх пристроїв або невдалим виконанням викликаної операції.

Механізм оброблення винятків повинен не приховувати такі проблеми, а забезпечувати контрольоване реагування на них. Операція повинна або відновити умови, необхідні для повторного виконання, або завершитися з повідомленням про відмову, залишивши об’єкт у коректному стані.

8.1 Основні поняття винятку та відмови

Операція є не довільною послідовністю програмних інструкцій, а реалізацією певної специфікації. Її контракт визначає початкові умови виконання та властивості, які повинні бути забезпечені після завершення.

Виклик операції вважається успішним, якщо він завершується у стані, що задовольняє її постумову та інваріант класу. Крім того, під час виконання не повинно виникати неконтрольованих сигналів операційної системи, помилок доступу до пам’яті або інших подій, які переривають нормальний потік керування.

Успішне виконання операції — це виконання, яке завершується відповідно до контракту.

Відмова операції — це ситуація, за якої конкретний виклик операції не може завершитися відповідно до контракту.

Потрібно зазначити, що відмовляє не операція як частина тексту програми, а конкретний її виклик під час виконання системи. Та сама операція може успішно виконуватися багато разів, але за певного стану системи або зовнішнього середовища її окремий виклик може завершитися відмовою.

Причиною відмови є певна подія, яка перериває нормальне виконання. Таку подію називають винятком.

Виняток — це подія під час виконання програми, яка може призвести до відмови викликаної операції.

Виняток і відмова не є тотожними поняттями. Виняток може виникнути, але операція може перехопити його, відновити допустимий стан і повторити виконання. У такому разі виняток не призведе до відмови.

Отже:

- кожна відмова є наслідком винятку;
- не кожний виняток обов’язково завершується відмовою;
- відмова виникає тоді, коли операція не змогла відновитися після винятку.
- У програмній інженерії також розрізняють поняття помилки, дефекту та збою:
- помилка — неправильне рішення, прийняте під час розроблення;
- дефект — неправильна властивість програмного забезпечення, створена внаслідок помилки;
- збій — відхилення поведінки системи від очікуваної під час виконання;
- відмова операції — неможливість конкретного виклику виконати встановлений контракт.

Виняток може бути проявом збою, але якщо його виникнення було передбачено і система успішно відновилася, виконання операції може залишитися правильним.

8.2 Причини виникнення винятків і порушень контракту

Винятки можуть виникати через помилки в програмному коді, порушення тверджень контракту, відмови викликаних операцій або ненормальні події, виявлені апаратним забезпеченням та операційною системою.

Основні категорії джерел винятків узагальнено в таблиці 8.1.

Таблиця 8.1 – Основні джерела винятків під час виконання програмної системи

Категорія	**Причина виникнення**	**Приклад**
Помилки роботи з посиланнями	Виклик операції для порожнього посилання або спроба присвоїти	Виклик a.f, коли a = Void

	порожнє значення розширеній сутності	
Апаратні та системні події	Ненормальна умова, виявлена процесором або операційною системою	Арифметичне переповнення, нестача пам'яті, помилка введення-виведення
Відмова викликаної операції	Операція, викликана поточною операцією, не змогла виконати власний контракт	Відмова операції читання файла
Порушення передумови	Клієнт викликав операцію в недопустимому стані	Звернення до елемента за межами масиву
Порушення постумови	Постачальник не забезпечив гарантованого результату	Кількість елементів не змінилася після додавання
Порушення інваріанта класу	Об'єкт залишено в неузгодженому стані	Кількість елементів перевищує місткість контейнера
Порушення тверджень циклу	Не виконується інваріант або не зменшується варіант циклу	Цикл не наближається до завершення
Порушення check	Не виконується твердження, яке програміст вважав істинним	Помилкове внутрішнє припущення
Явне створення винятку	Програма навмисно ініціює виняткову ситуацію	Виклик спеціальної операції trigger

Як видно з таблиці 8.1, частина винятків виникає через порушення контрактів, а частина — через зовнішні або системні обставини, які неможливо повністю передбачити за допомогою передумов.

Наприклад, перед виконанням арифметичної операції теоретично можна було б перевіряти, чи не спричинить вона переповнення. Проте така перевірка може бути складною, залежати від апаратної платформи або коштувати стільки ж, скільки сама операція. У подібних випадках доцільніше виконати операцію та перехопити виняток у разі ненормального результату.

Особливе значення має відмова викликаної операції. Якщо операція r викликає операцію *q*, а *q* завершується відмовою, для *r* це стає новим винятком. Якщо *r* також не може відновитися, її відмова передається наступній операції в ланцюгу викликів.

Таким чином, виняток може поширюватися від нижчого рівня системи до вищого, доки не буде знайдено операцію, здатну виконати відновлення, або доки не завершиться виконання всієї програми.

8.3 Принципи дисциплінованого оброблення винятків

Механізм винятків не повинен використовуватися як заміна звичайним умовним операторам або як спосіб переходу між довільними частинами програми. Винятки призначені для ненормальних ситуацій, а не для реалізації звичайних варіантів алгоритму.

Неправильним підходом є продовження виконання точно з того місця, де виник виняток, без усунення його причини. Наприклад, після арифметичного переповнення, нестачі пам'яті або порушення контракту стан програми може вже не дозволяти безпечне продовження.

Ще небезпечнішим є приховування відмови. Операція не повинна повідомляти клієнтові про успіх, якщо її основне завдання не було виконано. Просте виведення повідомлення про помилку та звичайне повернення до клієнта порушує контракт: клієнт продовжує роботу, помилково вважаючи, що отримав обіцяний результат.

Дисципліноване оброблення винятків передбачає лише дві основні реакції.

Повторне виконання (*Retry*). Операція намагається змінити стан або умови, які спричинили виняток, після чого починає виконання заново.

Контрольована відмова (*Organized Panic*). Операція відновлює узгоджений стан об'єкта, припиняє виконання та повідомляє про відмову своєму клієнтові.

У рідкісних випадках системний сигнал може виявитися хибною тривогою. Наприклад, графічне середовище може повідомити про зміну розміру вікна операції, яка взагалі не використовує це вікно. Однак більшість винятків не можна просто ігнорувати. Порушення передумови, постумови або арифметичне переповнення свідчить про реальну проблему.

Під час повторного виконання можна:

- застосувати інший алгоритм;
- змінити значення атрибутів або локальних сутностей;
- звільнити або повторно отримати певний ресурс;
- повторити операцію після завершення тимчасової зовнішньої проблеми;
- використати резервну реалізацію.

Якщо відновлення неможливе, операція повинна виконати контрольовану відмову. Перед передаванням винятку клієнтові вона повинна:

- відновити інваріант класу;
- звільнити тимчасово захоплені ресурси;
- залишити об'єкт у стабільному стані;
- повідомити клієнтові, що контракт не виконано.

Виняток спочатку опрацьовує операція, виконання якої було перервано. Якщо вона не може відновитися, виникає виняток у її клієнта. Процес продовжується вгору ланцюгом викликів до операції, яка виконає успішне відновлення, або до кореневої операції системи.

8.4 Механізм оброблення винятків rescue і retry

Для реалізації дисциплінованого оброблення винятків використовується секція rescue. Вона визначає дії, які повинні виконуватися, якщо нормальне тіло операції було перерване винятком.

Загальний запис операції має такий вигляд:

routine
  require
    передумова
  local
    локальні сутності
  do
    нормальне тіло операції
  ensure
    постумова
  rescue
    дії з відновлення
  end

Секція rescue розташовується після нормального тіла й постумови. Вона виконується лише у випадку виникнення винятку.

В одній операції може бути лише одна секція rescue, але всередині неї можна визначати різні реакції залежно від виду винятку.

Для повторного запуску операції використовується інструкція: *retry*

Інструкція *retry* може міститися лише в секції rescue. Вона запускає нормальне тіло операції повторно від самого початку. Значення атрибутів і локальних сутностей, змінені перед повторним запуском, зберігаються, що дозволяє наступній спробі виконуватися за інших умов.

Виконання операції з секцією *rescue* відбувається за таким алгоритмом:

1. Виконується нормальне тіло *do*.
2. Якщо виняток не виник, перевіряється постумова й операція успішно завершується.
3. Якщо виник виняток, нормальне тіло припиняється.
4. Керування передається до секції rescue.
5. Якщо секція виконує retry, тіло операції запускається повторно.
6. Якщо retry не виконується, операція завершується відмовою.

Виконання секції *rescue* до кінця без інструкції *retry* означає, що поточний виклик не зміг виконати контракт. Виняток передається клієнтові.

Якщо операція не містить явної секції rescue, за замовчуванням вона не виконує відновлення. Виняток призводить до відмови цієї операції та поширюється до її клієнта.

Схему виконання операції з обробленням винятку наведено на рисунку 8.1.

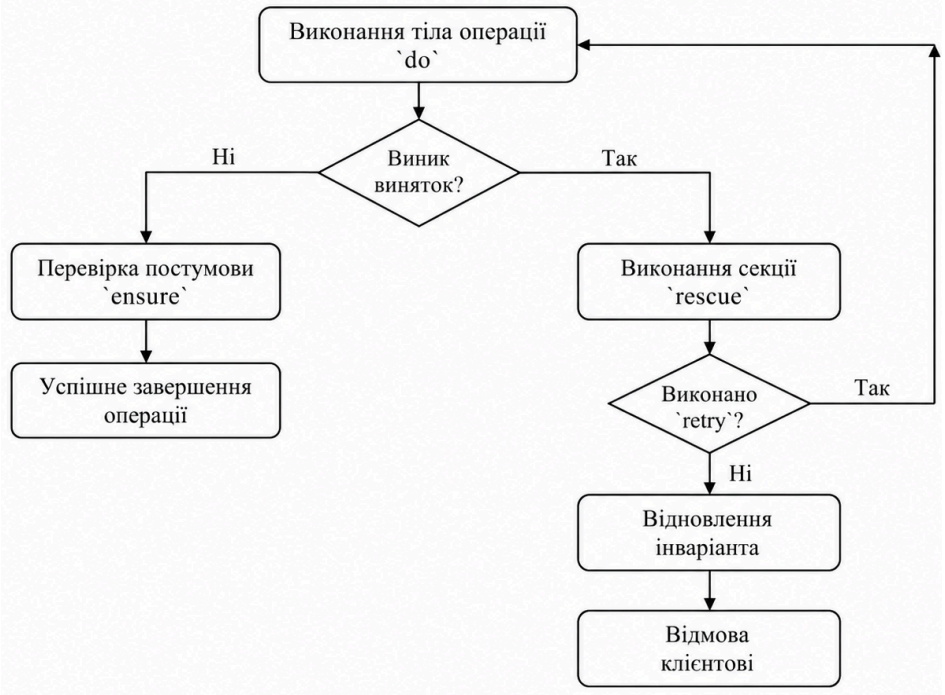


Рисунок 8.1 – Варіанти завершення операції після виникнення винятку

Як видно з рисунка 8.1, секція rescue не повинна самостійно завершувати основне завдання операції. Вона або готує умови для повторного запуску, або відновлює стабільний стан перед повідомленням про відмову.

Якщо жодна операція в ланцюгу викликів не змогла відновитися, середовище виконання формує історію винятків. Вона може містити:

- об’єкт, для якого викликалася операція;
- назву класу;
- назву перерваної операції;
- тип винятку;
- мітку порушеного твердження;
- результат опрацювання — *Retry* або *Fail*.

Таке трасування значно спрощує визначення первинної причини помилки, навіть якщо відмова поширилася через декілька рівнів викликів.

8.5 Приклади оброблення винятків

Нехай операція read_one_integer намагається зчитати ціле число, але завершується винятком, якщо користувач увів неправильне значення.

Найпростіший варіант відновлення має такий вигляд:

```
get_integer
   -- Отримати ціле число від користувача.
   do
      print ("Введіть ціле число: ")
      read_one_integer
   rescue
      retry
   end
```

Якщо користувач ввів неправильне значення, секція rescue запускає операцію повторно. Проте необмежена кількість повторів може призвести до нескінченного виконання. Тому доцільно встановити максимальну кількість спроб:

```
Maximum_attempts: INTEGER = 5
get_integer
   -- Виконати не більше Maximum_attempts спроб.
   local
      attempts: INTEGER
   do
      if attempts < Maximum_attempts then
         print ("Введіть ціле число: ")
         read_one_integer
         integer_was_read := True
      else
         integer_was_read := False
      end
   rescue
      attempts := attempts + 1
      retry
   end
```

Після завершення клієнт перевіряє значення *integer_was_read*. Якщо число було успішно зчитане, використовується отриманий результат. Інакше клієнт виконує окремий сценарій реагування.

У сучасних системах перевірка користувацького введення зазвичай виконується звичайними конструкціями валідації, а не винятками. Наведений приклад показує ситуацію, коли розробник змушений використовувати ненадійну зовнішню операцію, яка повідомляє про неправильні дані лише через виняток.

Розглянемо функцію, яка повинна повернути обернене значення числа або нуль, якщо обчислення неможливе:

```
quasi_inverse (x: REAL): REAL
   -- Повернути 1/x, якщо це можливо, інакше 0.
```

```
local
   division_tried: BOOLEAN
do
   if not division_tried then
      Result := 1 / x
   end
rescue
   division_tried := True
   retry
end
```

Під час першої спроби виконується ділення. Якщо виникає арифметичне переповнення, секція rescue установлює division_tried := True і повторно запускає операцію. Під час другої спроби ділення вже не виконується, а результат залишається рівним початковому значенню 0.

У цьому прикладі виняток використовується тому, що попереднє визначення можливості обчислення може бути складним, непереносимим або неефективним.

У програмному редакторі одна помилкова команда не повинна обов'язково завершувати роботу всієї системи й призводити до втрати даних. Операція виконання однієї команди може бути організована так:

```
execute_one_command
   -- Отримати й виконати команду користувача.
   do
      "Розпізнати запит користувача"
      "Виконати відповідну команду"
   rescue
      message ("Не вдалося виконати команду")
      message ("Спробуйте іншу команду")
      "Відновити стабільний стан редактора"
      retry
   end
```

Секція rescue не намагається довільно продовжити незавершену команду. Вона відновлює стан редактора й повертає керування до початку циклу, щоб користувач міг вибрати іншу дію.

У критично важливій системі одна задача може мати декілька незалежних реалізацій. Якщо перша реалізація завершується винятком, операція може спробувати використати іншу:

```
do_task
   -- Виконати задачу однією з доступних реалізацій.
   local
```

```
    attempts: INTEGER
  do
    if attempts = 0 then
      implementation_1
    elseif attempts = 1 then
      implementation_2
    end
  rescue
    attempts := attempts + 1
    if attempts < 2 then
      "Відновити стабільний стан"
      retry
    end
  end
```

Після вичерпання доступних реалізацій секція rescue завершується без retry, а операція повідомляє про відмову. Такий підхід може застосовуватися для програмної відмовостійкості, але не замінює необхідності створення правильної та ретельно перевіреної основної реалізації.

8.6 Завдання секції rescue та відновлення інваріанта

Після виникнення винятку виконання нормального тіла операції припиняється. Подальша поведінка визначається секцією rescue.

Можливі два випадки:

- Секція виконує retry. Нормальне тіло запускається повторно. Якщо нова спроба успішна, операція виконує контракт, а клієнт може навіть не знати про виняток.
- Секція завершується без retry. Операція визнає, що не може виконати контракт, і повідомляє клієнтові про відмову.
- Секція rescue не повинна завершувати основну роботу замість нормального тіла. Наприклад, якщо операція повинна обчислити результат, секція відновлення не повинна повертати довільне запасне значення та повідомляти клієнтові про успіх. Досягнення постумови є завданням нормального тіла.

Між нормальним тілом і секцією відновлення існує чіткий розподіл відповідальності.

Для нормального тіла операції правильність можна подати формулою:

{pre and INV} Body {post and INV}

Нормальне тіло може припускати виконання передумови та інваріанта. Воно повинно забезпечити постумову і зберегти інваріант.

Для секції rescue, яка завершується відмовою:

{True} Rescue {INV}

Секція не може припускати нічого про початковий стан, оскільки виняток може виникнути в будь-який момент виконання. Її обов'язком є лише відновлення інваріанта перед передаванням відмови клієнтові.

Для гілки, яка виконує повторний запуск:

{True} Retry_part {INV and pre}

Перед виконанням retry необхідно відновити не тільки інваріант класу, а й передумову операції, щоб повторний запуск відбувався в допустимому початковому стані.

Отже, нормальне тіло та секція rescue мають різні завдання:

- нормальне тіло реалізує основне призначення операції;
- секція rescue усуває наслідки винятку;
- нормальне тіло забезпечує постумову;
- секція rescue відновлює інваріант;
- секція rescue може підготувати умови для повторного запуску;
- секція rescue не повинна імітувати успішне завершення.

Оброблення винятку має бути простим. Чим складніша секція rescue, тим вища ймовірність виникнення нового винятку безпосередньо під час відновлення. Тому вона повинна містити лише дії, необхідні для повернення об'єкта до стабільного стану, звільнення ресурсів і прийняття рішення щодо повторного виконання.

8.7 Розширене оброблення винятків і запобігання повторним порушенням

Базового механізму rescue і retry достатньо для більшості випадків. Проте іноді необхідно визначити точний вид винятку, отримати відомості про місце його виникнення або виконати різні дії для різних причин.

Для цього може використовуватися спеціалізований клас *EXCEPTIONS*. Він надає запити для отримання інформації про останній виняток:

```
exception: INTEGER
   -- Код останнього винятку.
original_exception: INTEGER
   -- Код первинного винятку.
tag_name: STRING
   -- Мітка порушеного твердження.
recipient_name: STRING
   -- Назва перерваної операції.
class_name: STRING
   -- Назва класу.
is_assertion_violation: BOOLEAN
   -- Чи пов'язаний виняток із порушенням твердження?
```

is_system_exception: BOOLEAN

-- Чи спричинено виняток системною подією?

Різниця між exception та original_exception важлива під час поширення відмови. Поточний виняток може означати лише «відмова викликаної операції», тоді як первинною причиною було, наприклад, переповнення або порушення передумови на декілька рівнів нижче.

На основі цих запитів можна виконувати вибіркове реагування:

```
rescue
   if is_assertion_violation then
      "Опрацювати порушення твердження"
   elseif is_system_exception then
      "Опрацювати системний виняток"
   end
```

Однак надмірно детальний аналіз винятків може зробити секцію відновлення складною та ненадійною. Рекомендується опрацьовувати лише ті види винятків, для яких операція має чітко визначену стратегію відновлення. Невідомі або неочікувані винятки краще передавати клієнтові, ніж приховувати.

Програма також може навмисно ініціювати виняток. Такий виняток називають **винятком розробника**. Він може містити числовий код, текстове повідомлення та додатковий контекст.

Винятки розробника слід використовувати обережно. Вони не повинні перетворюватися на звичайний механізм переходів між операціями або замінювати умовні конструкції. Їх застосування виправдане лише тоді, коли операція справді не може продовжувати нормальне виконання.

Для запобігання повторним порушенням контрактів необхідно дотримуватися таких рекомендацій:

1. Формулювати точні контракти. Передумови, постумови та інваріанти повинні відображати реальні властивості компонента.
2. Увімкнути перевірку тверджень під час розроблення. Це дозволяє виявляти порушення ближче до місця їх виникнення.
3. Не використовувати винятки для звичайних сценаріїв. Очікувані ситуації потрібно опрацьовувати умовними конструкціями та валідацією.
4. Перехоплювати лише ті винятки, після яких можливе відновлення. Безпідставне перехоплення приховує помилки.
5. Обмежувати кількість повторних спроб. Нескінченне виконання retry може призвести до зависання системи.
6. Перед повторним запуском усувати причину винятку. Просте повторення тієї самої дії за тих самих умов зазвичай спричиняє той самий результат.

7. Відновлювати інваріант. Об’єкт не повинен залишатися доступним клієнтам у частково зміненому стані.

8. Звільняти ресурси. Перед відмовою необхідно закривати файли, мережеві з’єднання, блокування та інші тимчасові ресурси.

9. Зберігати діагностичну інформацію. Історія викликів, тип винятку та мітка порушеного твердження допомагають визначити першопричину.

10. Після усунення дефекту додавати тест. Тест повинен відтворювати ситуацію, яка раніше спричинила порушення, і запобігати її повторній появі.

Механізм винятків є одним із засобів забезпечення робастності, але не замінює правильного проєктування, контрактів, тестування, валідації зовнішніх даних і контролю ресурсів.

Висновки

Виняток є подією під час виконання програми, яка може перервати нормальне виконання операції. Відмова виникає тоді, коли операція не змогла відновитися після винятку й виконати свій контракт.

Основними причинами винятків є порушення контрактів, неправильна робота з посиланнями, відмови викликаних операцій, апаратні й системні події та явне ініціювання винятку програмним кодом.

Дисципліноване оброблення винятків допускає дві основні реакції: повторне виконання після відновлення допустимих умов або контрольовану відмову з повідомленням клієнта. Операція не повинна приховувати відмову та вдавати, що її контракт виконано.

Секція rescue призначена для відновлення стабільного стану об’єкта, а інструкція *retry* — для повторного запуску нормального тіла. Перед повторним запуском повинні бути відновлені інваріант класу та передумова операції.

Розширені засоби дозволяють визначати тип винятку, його первинну причину, місце виникнення та мітку порушеного твердження. Проте оброблення винятків повинно залишатися простим і застосовуватися лише для справді ненормальних ситуацій.

Правильне поєднання контрактів, перевірки тверджень, контрольованого відновлення, тестування та діагностування допомагає локалізувати наслідки помилок і підвищити надійність програмного забезпечення.

Лекція 9

Тема:Принципи успадкування

Мета: сформувати у студентів розуміння сутності успадкування в об’єктно-орієнтованому програмуванні, ознайомити з механізмами створення класів-нащадків, повторного використання та перевизначення операцій, а також розглянути переваги, обмеження й правила правильного застосування успадкування.

План лекції

1. Поняття та призначення успадкування.
2. Батьківські класи, класи-нащадки та успадковані компоненти.
3. Поліморфізм і сумісність типів.
4. Повторне визначення компонентів і динамічне зв’язування.
5. Відкладені компоненти та класи.
6. Значення, переваги й обмеження успадкування.
7. Правила правильного застосування успадкування.
8. Висновки.

Програмні системи рідко створюються повністю з нуля. Зазвичай нове програмне забезпечення розвиває раніше створені рішення, уточнює їх, поєднує або пристосовує до нових умов. Класи вже забезпечують модульність, приховування інформації, універсалізацію та можливість визначення контрактів. Однак для повноцінного повторного використання й розширення програмного забезпечення потрібен механізм, який дозволяє явно подати спільність і відмінності між пов’язаними класами.

Таким механізмом є успадкування. Воно дозволяє визначити новий клас як розширення, спеціалізацію або конкретну реалізацію іншого класу. Новий клас отримує компоненти наявного класу, може додавати власні компоненти та змінювати реалізацію окремих успадкованих операцій. Завдяки цьому спільні властивості групи класів описуються лише один раз, а специфічні властивості визначаються в окремих класах-нащадках.

9.1 Поняття та призначення успадкування

Успадкування — це механізм об’єктно-орієнтованого програмування, за допомогою якого новий клас створюється на основі одного або декількох вже наявних класів. Новий клас отримує їхні компоненти та може розширювати або уточнювати успадковану поведінку.

Клас, від якого успадковуються компоненти, називають батьківським класом, або предком. Клас, який успадковує компоненти, називають класом-нащадком.

Успадкування призначене для подання двох взаємопов’язаних аспектів:

- розширення програмного модуля;
- спеціалізації програмного типу.

З погляду модульності клас-нащадок є розширенням батьківського класу. Він отримує вже реалізовані операції та може додати до них нові.

З погляду типів клас-нащадок описує більш спеціалізований тип. Об'єкт класу-нащадка можна розглядати як об'єкт батьківського типу. Наприклад:

- кожний прямокутник є багатокутником;
- кожний квадрат є прямокутником;
- кожне коло є геометричною фігурою.

Водночас зворотні твердження не є правильними: не кожний багатокутник є прямокутником і не кожний прямокутник є квадратом.

Розглянемо графічну бібліотеку, яка містить класи для подання точок, відрізків, багатокутників, прямокутників, квадратів, еліпсів і кіл. Усі багатокутники мають вершини, периметр і підтримують операції переміщення, обертання та відображення.

Загальний клас багатокутника може мати такий вигляд:

```
class POLYGON
feature -- Доступ
   count: INTEGER
      -- Кількість вершин.
   perimeter: REAL
      -- Периметр багатокутника.
   vertices: LINKED_LIST [POINT]
      -- Послідовність вершин.
feature -- Перетворення
   display
      -- Відобразити багатокутник.
   do
      ...
   end
   rotate (center: POINT; angle: REAL)
      -- Повернути навколо точки center на кут angle.
   do
      ...
   end
   translate (a, b: REAL)
      -- Перемістити на a по горизонталі та b по вертикалі.
   do
      ...
   end
```

```
invariant
   same_count_as_implementation:
      count = vertices.count
   at_least_three:
      count >= 3
end
```

Клас *POLYGON* містить спільні властивості всіх багатокутників. Для обчислення периметра загального багатокутника потрібно послідовно обчислити довжини його сторін та знайти їх суму.

Прямокутник є окремим видом багатокутника. Він має всі загальні операції багатокутника, але додатково характеризується двома довжинами сторін, діагоналлю, чотирма вершинами та прямими кутами. Крім того, його периметр можна обчислити простіше:

perimeter = 2 × (side1 + side2)

Замість повторного опису всіх властивостей багатокутника клас *RECTANGLE* можна оголосити нащадком класу *POLYGON*:

```
class RECTANGLE
inherit
   POLYGON
feature
   side1, side2: REAL
      -- Довжини сторін.
   diagonal: REAL
      -- Довжина діагоналі.
end
```

Після такого оголошення клас *RECTANGLE* автоматично отримує компоненти *count, vertices, display, rotate, translate і perimeter.* У його тексті потрібно описувати лише ті компоненти, які є специфічними для прямокутників або потребують іншої реалізації.

Основні поняття, пов’язані з успадкуванням, узагальнено в таблиці 9.1.

Таблиця 9.1 – Основні поняття механізму успадкування

Поняття	**Характеристика**
Батьківський клас	Клас, компоненти якого успадковуються іншим класом
Безпосередній нащадок	Клас, у секції inherit якого безпосередньо зазначено батьківський клас
Предок	Сам клас або будь-який клас, від якого він успадковує прямо чи опосередковано

Нащадок	Сам клас або будь-який клас, який прямо чи опосередковано успадковує від нього
Успадкований компонент	Атрибут або операція, отримані від одного з предків
Власний компонент	Компонент, уперше оголошений безпосередньо в поточному класі
Повторно визначений компонент	Успадкований компонент, для якого клас-нащадок надає нову реалізацію

Як видно з таблиці 9.1, успадкування є транзитивним. Якщо SQUARE успадковує від RECTANGLE, а RECTANGLE — від POLYGON, то SQUARE також отримує компоненти класу POLYGON.

Клас може мати:

- одного безпосереднього предка — одинарне успадкування;
- декількох безпосередніх предків — множинне успадкування.

У межах цієї лекції основну увагу приділено одинарному успадкуванню. Множинне успадкування потребує додаткових правил для усунення конфліктів і неоднозначностей.

Успадкування також поширює інваріанти. Інваріант класу-нащадка утворюється логічним поєднанням:

- інваріанта самого класу;
- інваріантів усіх його предків.

Наприклад, інваріант класу POLYGON: *count >= 3* автоматично застосовується до класу RECTANGLE. Клас RECTANGLE може додати сильнішу умову: *count = 4.*

Вона не замінює батьківський інваріант, а доповнює його. Об’єкт прямокутника повинен одночасно задовольняти обидві умови.

9.2 Батьківські класи, класи-нащадки та успадковані компоненти

Під час створення класу-нащадка розробник повинен визначити, які властивості батьківського класу можна використати без змін, які потрібно повторно визначити та які нові компоненти необхідно додати.

Успадкований компонент за замовчуванням залишається доступним у класі-нащадку. Його не потрібно повторно оголошувати. Наприклад, клас *RECTANGLE* успадковує від *POLYGON* операцію translate. Якщо алгоритм переміщення всіх вершин однаково підходить для довільного багатокутника і прямокутника, операція використовується без змін.

Клас-нащадок може додавати нові компоненти:

feature

```
    side1, side2: REAL
    diagonal: REAL
```

Ці компоненти доступні об'єктам типу *RECTANGLE*, але не належать загальному типу *POLYGON*.

Клас-нащадок може також змінити реалізацію успадкованого компонента. Такий процес називають повторним визначенням, або перевизначенням.

Наприклад, загальний алгоритм обчислення периметра багатокутника перебирає всі вершини. Для прямокутника доступна простіша реалізація:

```
class RECTANGLE
inherit
    POLYGON
        redefine
            perimeter
        end
feature -- Доступ
    perimeter: REAL
        -- Периметр прямокутника.
    do
        Result := 2 * (side1 + side2)
    end
end
```

Секція *redefine* явно повідомляє, що клас-нащадок надає нову реалізацію успадкованої операції. Без такого оголошення повторне оголошення *perimeter* було б помилкою, оскільки клас уже отримав однойменний компонент від предка.

Після повторного визначення:

- для об'єкта *POLYGON* використовується загальна реалізація;
- для об'єкта *RECTANGLE* використовується реалізація прямокутника;
- для об'єкта *SQUARE* може бути визначено ще одну реалізацію, яка повертає чотири довжини сторони.

Повторне визначення повинно зберігати призначення операції. Якщо perimeter у батьківському класі обчислює периметр, версія в класі-нащадку не може обчислювати площу або виконувати інше завдання. Змінювати можна реалізацію, але не загальну семантику операції.

Контракти обмежують повторне визначення:

- передумову можна зберегти або послабити;
- постумову можна зберегти або посилити;
- інваріанти предків залишаються чинними.

Завдяки цьому клієнт батьківського класу може безпечно працювати з об'єктом класу-нащадка.

Процедури створення мають особливий статус. Процедура, яка створює об'єкт батьківського класу, не обов'язково підходить для створення об'єкта класу-нащадка.

Наприклад, загальний багатокутник можна створювати на основі списку з трьох або більше точок:

```
make_polygon (vl: LINKED_LIST [POINT])
  require
    vl.count >= 3
  do
    ...
  end
```

Для створення прямокутника зручніше передати:

- координати центра;
- дві довжини сторін;
- кут орієнтації.

Тому клас *RECTANGLE* визначає власну процедуру:

```
make (center: POINT; s1, s2, angle: REAL)
```

Батьківська процедура створення повинна встановити інваріант батьківського класу. Однак клас-нащадок зазвичай має сильніший інваріант і додаткові атрибути. Отже, успадкована процедура не гарантує правильного початкового стану нового класу.

З цієї причини статус процедури створення автоматично не успадковується. Успадкована операція залишається звичайним компонентом класу-нащадка, але вважається його процедурою створення лише тоді, коли її явно зазначено у власній секції створення класу.

Потрібно розрізняти:

- відношення «є різновидом»;
- відношення «має у своєму складі».

Успадкування подає відношення:

- *B* є різновидом *A*

Клієнтське відношення подає залежність:

- *B* має об'єкт типу *A* або використовує його послуги

Наприклад:

- прямокутник є багатокутником — доцільне успадкування;
- власник автомобіля має автомобіль — потрібне клієнтське відношення;
- літак має систему вентиляції — потрібне клієнтське відношення;
- коло є еліпсом зі спільними фокусами — можливе успадкування.

Неправильний опис власника автомобіля мав би вигляд:

```
class CAR_OWNER
inherit
```

```
    PERSON
    CAR
end
```

Він означав би, що кожний власник автомобіля одночасно є людиною й автомобілем. Правильна модель має інший вигляд:

```
class CAR_OWNER
inherit
    PERSON
feature
    my_car: CAR
end
```

Клас *CAR_OWNER* успадковує від *PERSON*, оскільки власник автомобіля є людиною, але виступає клієнтом *CAR*, оскільки він має автомобіль.

9.3 Поліморфізм і сумісність типів

Успадкування дозволяє розглядати об’єкт спеціалізованого класу як об’єкт більш загального типу. На цій властивості ґрунтується поліморфізм.

Поліморфізм — це здатність сутності, оголошеної з певним статичним типом, під час виконання приєднуватися до об’єктів різних сумісних типів.

Нехай оголошено:

```
p: POLYGON
r: RECTANGLE
t: TRIANGLE
```

Правильними є присвоєння:

```
p := r
p := t
```

Сутність *p* оголошена як багатокутник, тому вона може посилатися на прямокутник або трикутник, оскільки обидва класи є нащадками *POLYGON*.

Зворотне присвоєння: *r := p* не є правильним. Сутність *p* може посилатися на довільний багатокутник, наприклад на трикутник або п’ятикутник, які не є прямокутниками.

Такі присвоєння називають поліморфними приєднаннями. Поліморфізм також виникає під час передавання аргументів:

```
display_polygon (p: POLYGON)
    do
       p.display
    end
```

Операцію можна викликати так:

```
display_polygon (r)
```

display_polygon (t)

Фактичний аргумент може мати тип *RECTANGLE, TRIANGLE* або інший тип, сумісний із *POLYGON*.

Поліморфізм не змінює тип самого об'єкта. Після створення об'єкт прямокутника завжди залишається об'єктом типу *RECTANGLE*. Змінюється лише посилання: сутність загального типу може в різні моменти посилатися на об'єкти різних спеціалізованих типів.

Статичний тип сутності визначається в її оголошенні та не змінюється: *p: POLYGON*. Статичним типом p є *POLYGON*.

Динамічний тип визначається типом об'єкта, до якого сутність приєднана в певний момент виконання.

Після: *create p* динамічним типом *p* є *POLYGON*.

Після: *p := r* динамічним типом *p* є *RECTANGLE*, хоча статичним типом залишається *POLYGON*.

Динамічний тип завжди повинен бути сумісним зі статичним. Об'єкт класу-нащадка можна приєднати до сутності батьківського типу, але не навпаки.

Тип *U* вважають сумісним із типом *T*, якщо базовий клас *U* є нащадком базового класу *T*.

Наприклад:

SQUARE сумісний із *RECTANGLE*;

RECTANGLE сумісний із *POLYGON*;

TRIANGLE сумісний із *POLYGON*;

CIRCLE сумісний із *FIGURE*;

CIRCLE не сумісний із *POLYGON*.

Загальне правило сумісності можна сформулювати так:

Джерело присвоєння або фактичний аргумент повинні мати тип, сумісний із типом цільової сутності або формального аргументу.

Таким чином, присвоєння виконується від спеціалізованого типу до загального:

нащадок → предок

але не від загального до спеціалізованого:

предок → нащадок

Це правило забезпечує статичну типобезпечність. Компілятор відхиляє операції, які можуть призвести до застосування компонента до об'єкта, що його не підтримує.

Набір операцій, які можна застосувати до сутності, визначається її статичним типом.

Нехай:

p: POLYGON

r: RECTANGLE

Правильними є виклики:

p.perimeter
p.rotate (...)
r.perimeter
r.rotate (...)
r.diagonal
Неправильним є:
p.diagonal

Операція *diagonal* визначена для прямокутників або чотирикутників, але не для всіх багатокутників. Навіть якщо в певний момент *p* посилається на прямокутник, компілятор враховує статичний тип *POLYGON*.

Це обмеження є необхідним. Сутність *p* може пізніше посилатися на трикутник, для якого операція diagonal у такому розумінні не визначена. Тому через сутність загального типу дозволено викликати лише операції, гарантовані цим загальним типом.

Поліморфними можуть бути не лише окремі сутності, а й елементи контейнерів:
figures: ARRAY [FIGURE]
До такого масиву можна додати об’єкти різних класів-нащадків:
figures.put (rectangle, 1)
figures.put (circle, 2)
figures.put (triangle, 3)
figures.put (square, 4)

Масив містить об’єкти різних динамічних типів, але всі вони сумісні з типом *FIGURE*.

Поліморфна структура дозволяє обмежити набір допустимих елементів вибором узагальненого параметра:

- *LIST [RECTANGLE]* може містити прямокутники та квадрати, але не трикутники;
- *LIST [POLYGON]* може містити прямокутники, квадрати й трикутники, але не кола;
- *LIST [FIGURE]* може містити всі геометричні фігури;
- *LIST [ANY]* може містити об’єкти довільних типів.

Отже, поєднання універсалізації та успадкування забезпечує одночасно гнучкість і типобезпечність контейнерів.

Ієрархію геометричних типів, використану для пояснення поліморфізму, доцільно подати на рисунку 9.1.

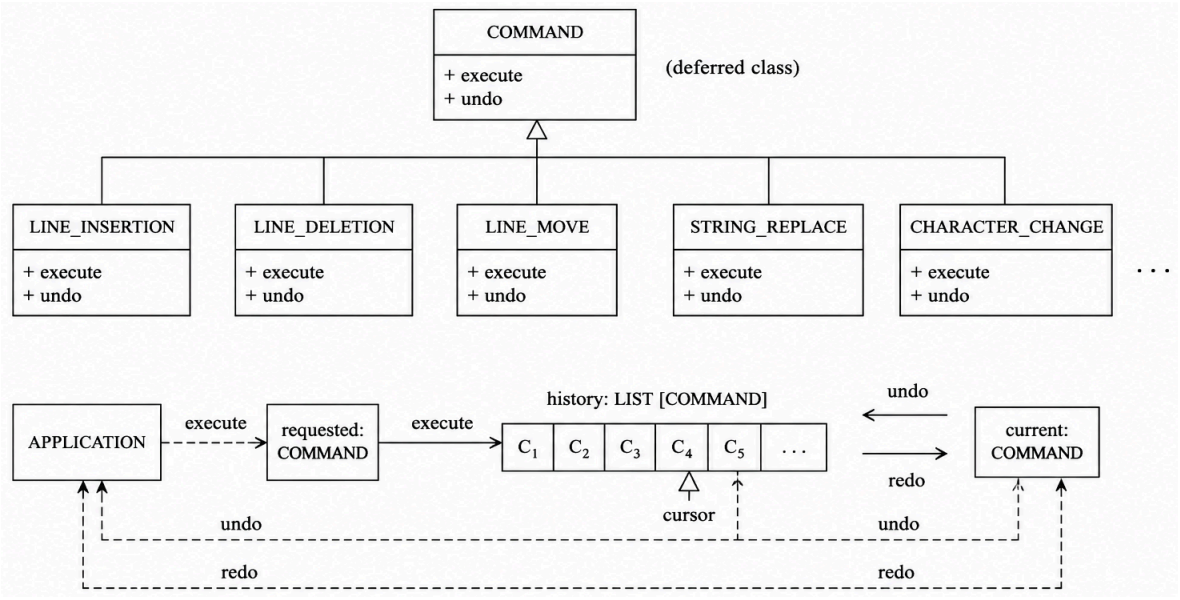


Рисунок 9.1 – Ієрархія типів геометричних фігур

На рисунку 9.1 клас *FIGURE* є найбільш загальним. Від нього успадковують класи відкритих і замкнених фігур. Класи *POLYGON* та *ELLIPSE* спеціалізують замкнені фігури, а *RECTANGLE, SQUARE і CIRCLE* утворюють наступні рівні спеціалізації.

9.4 Повторне визначення компонентів і динамічне зв'язування

Поліморфізм дозволяє сутності загального типу посилатися на об'єкт спеціалізованого класу. Повторне визначення дозволяє цьому спеціалізованому класу мати власну реалізацію успадкованої операції. Динамічне зв'язування поєднує ці два механізми.

Динамічне зв'язування — це правило, відповідно до якого конкретна версія операції вибирається за динамічним типом об'єкта, а не лише за статичним типом сутності.

Розглянемо:

p: POLYGON
r: RECTANGLE
x: REAL
create r.make (...)
p := r
x := p.perimeter

Статичним типом *p* є *POLYGON*, але динамічним типом після присвоєння є *RECTANGLE*. Тому буде викликана повторно визначена версія perimeter із класу *RECTANGLE*.

Без динамічного зв’язування виклик через *p* завжди застосовував би загальний алгоритм класу *POLYGON*. Це зменшило б користь поліморфізму, оскільки клієнт був би змушений сам визначати точний тип фігури.

Завдяки динамічному зв’язуванню клієнт може написати:

```
print_perimeter (p: POLYGON)
   do
      print (p.perimeter)
   end
```

і передавати до операції:

```
print_perimeter (rectangle)
print_perimeter (triangle)
print_peri
```
meter (square)

У кожному випадку автоматично викликається реалізація, що відповідає фактичному типу об’єкта.

Динамічне зв’язування особливо важливе тоді, коли точний тип об’єкта неможливо визначити з тексту програми. Наприклад, тип фігури може залежати від:

- вибору користувача;
- даних, отриманих із файла;
- результату обчислення;
- елемента поліморфного контейнера;
- фактичного аргументу операції.

Клієнт працює через загальний інтерфейс FIGURE або POLYGON, а середовище виконання самостійно вибирає відповідну реалізацію операції.

Наприклад:

```
from
   figures.start
until
   figures.after
loop
   figures.item.display
   figures.item.rotate (center, angle)
   figures.forth
end
```

У контейнері можуть знаходитися кола, прямокутники, трикутники та інші фігури. Для кожного елемента викликається його власна реалізація display і rotate.

Це усуває потребу у великій кількості умов:

```
if figure_type = rectangle then
   ...
elseif figure_type = circle then
```

```
    ...
elseif figure_type = triangle then
    ...
end
```

Додавання нового класу фігур не вимагає змінювати всі операції, які працюють із фігурами. Новий клас сам надає потрібні реалізації, а динамічне зв'язування включає їх у роботу системи.

Динамічне зв'язування було б небезпечним, якби клас-нащадок міг довільно змінювати значення операції. Клієнт класу POLYGON очікує, що perimeter повертає периметр. Якщо клас-нащадок використав би цю операцію для обчислення площі, клієнт отримав би неправильний результат.

Тому повторне визначення повинно зберігати контракт:

- успадкована передумова не може бути посилена;
- успадкована постумова не може бути послаблена;
- інваріант батьківського класу залишається обов'язковим;
- загальне призначення операції не змінюється.

Контракти визначають допустимий діапазон реалізацій. Клас-нащадок може запропонувати ефективніший алгоритм або сильніші гарантії, але не може порушити очікування клієнтів батьківського класу.

9.5 Відкладені компоненти та класи

У верхній частині ієрархії часто розташовуються загальні класи, для яких можна визначити призначення операцій, але неможливо надати єдину правильну реалізацію.

Наприклад, кожну геометричну фігуру можна:

- відобразити;
- перемістити;
- повернути;
- визначити її просторові межі.

Однак не існує універсального алгоритму обертання довільної фігури без відомостей про її внутрішню структуру. Багатокутник обертається шляхом зміни координат вершин, а коло — шляхом зміни положення центра.

Неправильним рішенням було б визначити в загальному класі порожню операцію:

```
rotate (...)
    do
        -- Нічого не виконувати.
    end
```

Операція має чітке призначення, тому бездіяльність не є правильною реалізацією.

Для таких випадків використовується відкладений компонент.

```
rotate (center: POINT; angle: REAL)
   -- Повернути фігуру навколо center на angle.
   deferred
   end
```

Ключове слово *deferred* означає, що:

- операція належить інтерфейсу класу;
- її призначення та контракт можуть бути визначені;
- реалізацію повинні надати класи-нащадки.

Компонент, який має реалізацію, називають ефективним. Компонент без реалізації називають відкладеним.

Надання реалізації успадкованому відкладеному компонентові називають ефективацією:

```
class POLYGON
inherit
   CLOSED_FIGURE
feature
   rotate (center: POINT; angle: REAL)
      -- Повернути всі вершини багатокутника.
   do
      from
         vertices.start
      until
         vertices.after
      loop
         vertices.item.rotate (center, angle)
         vertices.forth
      end
   end
end
```

Потрібно розрізняти:

- повторне визначення — заміну однієї ефективної реалізації іншою;
- ефективацію — надання першої реалізації компонентові, який у предку був відкладеним.

Клас, який містить хоча б один відкладений компонент, є відкладеним класом:

```
deferred class FIGURE
feature
```

```
    display
      deferred
      end
    rotate (center: POINT; angle: REAL)
      deferred
      end
    translate (a, b: REAL)
      deferred
      end
  end
```

Відкладений клас описує загальну абстракцію, але не повну реалізацію. Він може:

- визначати спільний інтерфейс;
- містити ефективні операції, спільні для всіх нащадків;
- визначати передумови, постумови та інваріант;
- залишати частину операцій без реалізації.

Ефективний клас не містить невизначених відкладених компонентів. Якщо клас успадковує відкладену операцію та не надає її реалізації, він також залишається відкладеним.

Наприклад:

FIGURE є відкладеним;

CLOSED_FIGURE може залишатися відкладеним;

POLYGON стає ефективним після реалізації всіх потрібних операцій;

RECTANGLE є ефективним спеціалізованим класом.

Безпосередньо створити об’єкт відкладеного класу не можна:

```
f: FIGURE
create f    -- Неправильно
```

Водночас сутність відкладеного типу може посилатися на об’єкт ефективного класу-нащадка:

```
f: FIGURE
create {RECTANGLE} f.make (...)
f.display
f.rotate (center, angle)
```

Статичний тип *FIGURE* гарантує наявність загальних операцій, а динамічний тип *RECTANGLE* визначає їхню конкретну реалізацію.

Відкладені класи мають важливе архітектурне значення. Вони дозволяють програмувати на рівні абстракцій, не прив’язуючи клієнтський код до конкретної реалізації.

Наприклад:

```
transform (f: FIGURE)
  do
    f.rotate (center, angle)
    f.translate (10, 20)
    f.display
  end
```

Операція transform працює з довільною фігурою. Вона не знає, чи буде передано прямокутник, коло або багатокутник, і не повинна цього знати.

9.6 Значення, переваги й обмеження успадкування

Клас одночасно є програмним модулем і типом. Тому успадкування також має подвійне значення:

- як механізм розширення модулів;
- як механізм спеціалізації типів.

З модульного погляду клас-нащадок отримує набір послуг батьківського класу, додає власні операції та за потреби повторно визначає успадковані.

З погляду типів клас-нащадок описує спеціалізовану множину об’єктів. Кожний об’єкт класу-нащадка можна використовувати там, де очікується об’єкт батьківського класу.

Без успадкування класи **POLYGON, RECTANGLE і SQUARE** могли б окремо реалізовувати:

- зберігання вершин;
- відображення;
- переміщення;
- обертання;
- перевірку загальних властивостей.

Це призвело б до повторення однакових фрагментів коду. Зміна або виправлення спільної операції потребували б редагування декількох класів.

Успадкування дозволяє розмістити спільну реалізацію на найвищому обґрунтованому рівні ієрархії. Вона автоматично стає доступною всім нащадкам.

Новий клас можна створити як розширення вже перевіреного компонента. Розробник використовує наявні операції та зосереджується лише на відмінностях нового типу.

Це скорочує:

- обсяг нового програмного коду;
- час реалізації;
- кількість повторних тестів;
- ризик внесення несумісних змін.

Успадкування підтримує принцип відкритості-закритості. Батьківський клас може бути закритий для прямої модифікації, але відкритий для розширення через класи-нащадки. Нові можливості додаються без зміни класу та без порушення роботи його наявних клієнтів.

Успадкування дозволяє організувати пов’язані поняття від загальних до спеціалізованих:

FIGURE
  CLOSED_FIGURE
    POLYGON
      RECTANGLE
        SQUARE
    ELLIPSE
      CIRCLE

Кожний наступний рівень:

- зберігає властивості попереднього;
- вводить додаткові обмеження;
- додає нові операції;
- надає спеціалізовані реалізації.

Така структура полегшує розуміння предметної області та програмної архітектури.

Найважливішою перевагою успадкування є можливість працювати з різними типами через спільний інтерфейс.

Операція: *display_all (figures: LIST [FIGURE])* може опрацьовувати довільні фігури. Додавання нового класу STAR не потребує змінювати операцію display_all, якщо STAR успадковує від FIGURE та реалізує display.

Це забезпечує розширюваність системи.

У традиційному підході одна велика операція часто містить перелік усіх можливих типів:

```
if figure_type = rectangle then
  ...
elseif figure_type = circle then
  ...
elseif figure_type = triangle then
  ...
end
```

Така операція має надмірні знання про всю систему. Додавання нового типу вимагає знайти й змінити всі подібні умовні конструкції.

В об’єктно-орієнтованому підході кожний клас сам містить власну реалізацію. Додавання нового типу переважно обмежується створенням нового класу. Наявні класи не повинні знати повний перелік нащадків.

Децентралізація локалізує зміни та знижує ризик поширення помилок.

Клієнт викликає операцію через загальний інтерфейс, не знаючи конкретного алгоритму:

present := table.has (x)

Якщо динамічним типом *table* є хеш-таблиця, використовується пошук за хешем. Якщо це бінарне дерево пошуку, застосовується алгоритм дерева.

Клієнт формулює завдання, а динамічне зв’язування вибирає реалізацію.

Успадкування є потужним, але складним механізмом. Його неправильне застосування може створити більше проблем, ніж переваг.

До основних обмежень належать:

- сильний постійний зв’язок між класом-нащадком і предком;
- залежність нащадка від інтерфейсу та частково від структури предка;
- ризик успадкування непотрібних компонентів;
- ускладнення глибоких ієрархій;
- конфлікти імен у множинному успадкуванні;
- складність прогнозування взаємодії повторних визначень;
- можливість порушення контрактів через неправильну спеціалізацію;
- надмірне використання ієрархій там, де достатньо композиції.

Зміни в батьківському класі можуть вплинути на всіх нащадків. Тому його інтерфейс і контракт мають бути стабільними та достатньо загальними.

Глибока ієрархія також ускладнює розуміння класу. Щоб установити повний набір його компонентів, може знадобитися аналіз декількох рівнів предків.

9.7 Правила правильного застосування успадкування

Знання синтаксису успадкування не гарантує правильного проєктування. Перед створенням зв’язку між двома класами потрібно визначити, чи справді він відповідає їхньому змісту.

Клас *B* повинен успадковувати від класу *A* лише тоді, коли кожний об’єкт *B* можна обґрунтовано розглядати як об’єкт *A*.

Правильні приклади:

- кожний квадрат є прямокутником;
- кожний прямокутник є багатокутником;
- кожний текстовий файл є файлом;
- кожний ощадний рахунок є банківським рахунком.

Неправильні приклади:

- власник автомобіля є автомобілем;

- літак є системою вентиляції;
- яблучний пиріг є яблуком;
- трояндовий кущ є трояндою.

В останніх випадках між поняттями існує відношення володіння, складу або використання, а не успадкування.

Правило «є різновидом» є необхідним, але не завжди достатнім. Іноді ту саму модель можна подати як через успадкування, так і через клієнтське відношення. Тому потрібно враховувати додаткові критерії.

Успадкування описує постійне відношення між класами. Об’єкт не може під час виконання перестати бути екземпляром типу, визначеного його класом.

Наприклад, об’єкт *SOFTWARE_ENGINEER*, який успадковує від *ENGINEER*, завжди залишається інженером.

Клієнтське відношення дозволяє змінювати складові об’єкта:

class SOFTWARE_ENGINEER
feature
  vocation: VOCATION
end

Посилання *vocation* може приєднуватися до різних сумісних об’єктів. Отже, якщо відповідний компонент повинен змінюватися під час виконання, доцільно застосувати композицію, а не успадкування.

Загальне правило:

Не використовуйте успадкування для відношення, яке повинно динамічно змінюватися під час існування об’єкта.

Успадкування є доцільним, якщо сутності загального типу повинні посилатися на об’єкти спеціалізованого типу.

Наприклад:

employee: EMPLOYEE

може посилатися на:

- *MANAGER;*
- *ENGINEER;*
- *ACCOUNTANT;*
- *SOFTWARE_ENGINEER.*

Так само: *employees: LIST [EMPLOYEE]* може містити працівників різних спеціалізованих класів.

Якщо така поліморфна підстановка потрібна, між класами має існувати відношення успадкування.

Композиція є гнучкішою для змінних зв’язків між об’єктами, тоді як успадкування необхідне для поліморфної підстановки.

Однією з типових помилок є створення великої кількості класів, які не додають нової поведінки, властивостей або обмежень.

Наприклад, поділ *PERSON* на *MALE* і *FEMALE* не завжди потребує успадкування. Якщо система лише зберігає значення статі й не має специфічних операцій, достатньо атрибута:

gender: GENDER

Окремі класи виправдані тоді, коли варіанти мають:

- власні операції;
- різні реалізації успадкованих операцій;
- додаткові інваріанти;
- суттєво різні контракти.

Загальне правило:

Кожний клас-нащадок повинен додавати компонент, повторно визначати успадкований компонент або вводити додаткову умову інваріанта.

Ієрархія повинна зменшувати складність системи, а не створювати зайві рівні класифікації.

Не слід успадковувати від класу лише тому, що в ньому є зручна операція. Наприклад, якщо клас *REPORT* потребує операції сортування, це не означає, що він повинен успадковувати від класу *SORTER*. Зв’язок: *REPORT* є *SORTER* не має змісту.

Доцільніше:

- використати об’єкт *SORTER* як компонент;
- викликати його операції через клієнтське відношення;
- виділити спільну абстракцію в окремий клас, якщо це обґрунтовано.

Повторне використання реалізації є перевагою успадкування, але воно не повинно бути єдиною причиною встановлення зв’язку між непов’язаними абстракціями.

Клас-нащадок повинен дотримуватися специфікації батьківського класу:

- не вимагати від клієнта більше;
- гарантувати не менше;
- зберігати інваріанти предків;
- підтримувати значення успадкованих операцій.

Якщо ці вимоги неможливо виконати, обрана ієрархія, ймовірно, є неправильною.

Компонент потрібно розміщувати на найвищому рівні ієрархії, де він має правильний зміст для всіх нащадків.

Наприклад:

- *display* доцільно оголосити в *FIGURE*;
- *perimeter* — у *CLOSED_FIGURE*;
- *diagonal* — у *QUADRANGLE*;

- *side1 і side2* — у *RECTANGLE*;
- умову *side1 = side2* — в інваріанті *SQUARE*.

Занадто низьке розміщення спричиняє дублювання. Занадто високе призводить до появи операцій, які не мають змісту для частини нащадків.

Не існує універсальної максимальної кількості рівнів успадкування. Проте кожний рівень повинен мати чітке семантичне призначення.

Надмірно глибокі ієрархії:

- ускладнюють пошук реалізації операцій;
- збільшують кількість неявно успадкованих властивостей;
- ускладнюють тестування;
- підвищують ризик небажаного впливу змін у предках.

Доцільно надавати перевагу коротким, змістовним ієрархіям та поєднувати успадкування з композицією.

Загальні поняття, для яких неможливо надати повну реалізацію, доцільно описувати відкладеними класами.

Відкладений клас повинен:

- визначати спільний інтерфейс;
- формулювати контракти;
- містити спільну реалізацію там, де вона справді є загальною;
- залишати спеціалізовану поведінку класам-нащадкам.

Не слід створювати фіктивні реалізації, які нічого не виконують або повертають довільні значення.

Композиція часто забезпечує більшу гнучкість:

```
class WINDOW
feature
   toolkit: TOOLKIT
end
```

Сутність *toolkit* може посилатися на різні реалізації:

WINDOWS_TOOLKIT;
LINUX_TOOLKIT;
WEB_TOOLKIT.

Об’єкт вікна залишається вікном, але використовуваний інструментарій можна замінити. Успадкування при цьому застосовується для ієрархії *TOOLKIT*, а композиція — для зв’язку між *WINDOW* і *TOOLKIT*.

Отже, успадкування й композиція не є взаємовиключними. У складних системах вони часто доповнюють одне одного.

Висновки

Успадкування є механізмом створення нового класу на основі вже наявного класу. Клас-нащадок автоматично отримує компоненти предка, може додавати власні компоненти та надавати нові реалізації успадкованих операцій.

Успадкування має подвійне значення. З модульного погляду воно забезпечує розширення та повторне використання програмних компонентів. З погляду типів воно описує спеціалізацію й дозволяє використовувати об’єкти класів-нащадків замість об’єктів батьківського типу.

Поліморфізм дає змогу сутності загального типу посилатися на об’єкти різних сумісних типів. Статичний тип визначає доступний інтерфейс, а динамічний тип — конкретний вид об’єкта під час виконання. Правила сумісності дозволяють виконувати приєднання від спеціалізованого типу до загального, зберігаючи статичну типобезпечність.

Динамічне зв’язування автоматично вибирає реалізацію операції відповідно до динамічного типу об’єкта. Завдяки цьому клієнтський код працює через загальні абстракції та не містить повного переліку конкретних типів.

Відкладені компоненти визначають операції без конкретної реалізації, а відкладені класи описують загальні, частково реалізовані абстракції. Безпосереднє створення об’єктів відкладених класів заборонене, але їхні типи використовуються для поліморфних сутностей.

До основних переваг успадкування належать усунення дублювання, повторне використання класів, розширюваність, підтримка поліморфізму, децентралізація архітектури та незалежність клієнтів від конкретних реалізацій.

Успадкування слід застосовувати лише за наявності обґрунтованого відношення «*є різновидом*». Для зв’язків «має», змінних компонентів і простого використання послуг доцільніше застосовувати клієнтське відношення або композицію. Кожний клас-нащадок повинен мати власне змістове призначення, зберігати контракти предків і не створювати зайвих рівнів у структурі системи.

Лекція 10

Тема:Наслідування: відкат в інтерактивних системах

Мета: сформувати у студентів розуміння принципів побудови механізмів скасування та повторного виконання команд в інтерактивних системах, показати роль успадкування, поліморфізму та динамічного зв'язування у створенні універсального механізму Undo/Redo, а також розглянути використання історії команд для підтримки стабільності та зручності програмного забезпечення.

План лекції

1. Поняття відкату в інтерактивних системах.
2. Команда як об'єкт і базовий клас COMMAND.
3. Успадкування, поліморфізм і динамічне зв'язування в механізмі Undo.
4. Багаторівневий механізм Undo/Redo та історія команд.
5. Практичні аспекти реалізації та інтерфейс користувача.
6. Переваги, обмеження та загальні принципи побудови механізму відкату.
7. Висновки.

У попередній лекції було розглянуто успадкування як механізм створення нових класів на основі вже існуючих. Однак значення успадкування полягає не лише в повторному використанні програмного коду. У поєднанні з поліморфізмом і динамічним зв'язуванням воно дозволяє будувати загальні архітектурні рішення, які можна повторно застосовувати в різних програмних системах.

Одним із таких прикладів є механізм скасування команд в інтерактивній системі. У підручнику Б. Мейєра це питання розглядається як окремий приклад застосування успадкування: «**Inheritance case study: "undo" in an interactive system**». Основна ідея полягає в тому, щоб не реалізовувати Undo окремо для кожної команди, а представити саму команду як об'єкт зі стандартними операціями виконання та скасування.

10.1 Поняття відкату в інтерактивних системах

В інтерактивних системах користувач постійно виконує певні дії: вводить текст, видаляє об'єкти, змінює параметри, переміщує елементи, підтверджує операції тощо. Оскільки помилки неминучі, система повинна надавати можливість скасувати результат останньої дії.

Undo — це операція, яка скасовує ефект останньої виконаної команди, що ще не була скасована.

Головне призначення *Undo* полягає у відновленні стану системи після помилкової дії користувача. Однак така можливість має ширше значення: користувач може експериментувати з різними діями, знаючи, що небажаний результат можна

легко скасувати. Це робить взаємодію із системою менш ризикованою та більш зручною.

Однорівневий механізм дозволяє скасувати лише останню команду. У складніших системах необхідний **багаторівневий Undo**, який дозволяє послідовно повертатися на декілька кроків назад.

Для багаторівневого скасування потрібна також операція **Redo**.

Redo — це повторне виконання останньої скасованої команди, яка ще не була повторно виконана.

Наприклад, користувач виконав послідовність:

$A \rightarrow B \rightarrow C \rightarrow D$

Після двох операцій *Undo* система повертається до стану після команди *B*:

$A \rightarrow B \leftarrow C \leftarrow D$

Після одного Redo знову застосовується команда *C*:

$A \rightarrow B \rightarrow C$

При багаторівневому механізмі *Undo і Redo* повинні розглядатися як окремі операції.

До механізму відкату висуваються чотири основні вимоги:

Таблиця 10.1 – Основні вимоги до механізму *Undo/Redo*

Позначення	**Вимога**
U1	Механізм повинен бути придатним для широкого класу інтерактивних систем незалежно від предметної області
U2	Додавання нової команди не повинно вимагати перепроєктування механізму Undo/Redo
U3	Для підтримки відкату необхідно зберігати лише мінімально необхідний обсяг інформації
U4	Рішення повинно підтримувати як однорівневий, так і багаторівневий Undo

Як видно з таблиці 10.1, механізм повинен бути загальним і розширюваним. Неправильним рішенням було б створення великої операції:

```
if last_command = INSERT_LINE then
   "Скасувати вставлення рядка"
elseif last_command = DELETE_LINE then
   "Скасувати видалення рядка"
elseif last_command = REPLACE then
   "Скасувати заміну"
...
end
```

За такого підходу кожне додавання нової команди потребувало б зміни операції *Undo*. Крім того, код *Undo* мав би знати внутрішні деталі всіх команд. Це суперечить розширюваності та принципу єдиного вибору.

Інше просте рішення — перед кожною командою зберігати повну копію стану всієї системи. Тоді *Undo* просто відновлював би попередню копію. Такий підхід працездатний, але потребує надмірного обсягу пам'яті. Тому необхідно зберігати не весь стан, а тільки інформацію, потрібну для скасування конкретної команди.

10.2 Команда як об'єкт і базовий клас COMMAND

Ключовим кроком об'єктно-орієнтованого проєктування є пошук правильної абстракції. У цьому випадку такою абстракцією є команда.

Команда розглядається не просто як виклик певної операції, а як повноцінний об'єкт, над яким можна виконувати декілька дій:

- створити;
- виконати;
- зберегти;
- скасувати;
- повторно виконати.

Тому вводиться загальний відкладений клас *COMMAND*:

```
deferred class COMMAND
feature
   execute
      -- Виконати команду.
      deferred
      end
   undo
      -- Скасувати результат команди.
      deferred
      end
end
```

Клас *COMMAND* є відкладеним, оскільки неможливо визначити єдину реалізацію execute і undo для всіх можливих команд. Конкретні команди подаються ефективними класами-нащадками.

Наприклад, команда видалення рядка може бути представлена класом *LINE_DELETION:*

```
class LINE_DELETION
inherit
   COMMAND
feature
```

```
    deleted_line_index: INTEGER
    deleted_line: STRING
    execute
        -- Видалити рядок.
    do
        "Зберегти текст рядка в deleted_line"
        "Зберегти позицію в deleted_line_index"
        "Видалити рядок"
    end
    undo
        -- Відновити видалений рядок.
    do
        "Вставити deleted_line у позицію deleted_line_index"
    end
end
```

Об’єкт *LINE_DELETION* зберігає тільки ту інформацію, яка необхідна для відновлення попереднього стану:

- текст видаленого рядка;
- його позицію.

Зберігати повну копію документа не потрібно.

Аналогічно можуть бути створені:

- *LINE_INSERTION*
- *LINE_DELETION*
- *LINE_MOVE*
- *STRING_REPLACE*
- *CHARACTER_CHANGE*

Усі вони є нащадками *COMMAND* і мають спільний інтерфейс:

- *execute*
- *undo*

Їхні реалізації при цьому відрізняються.

Таким чином, один об’єкт команди фактично описує різницю між станом системи до виконання команди та станом після її виконання. Саме ця інформація потрібна для повернення назад.

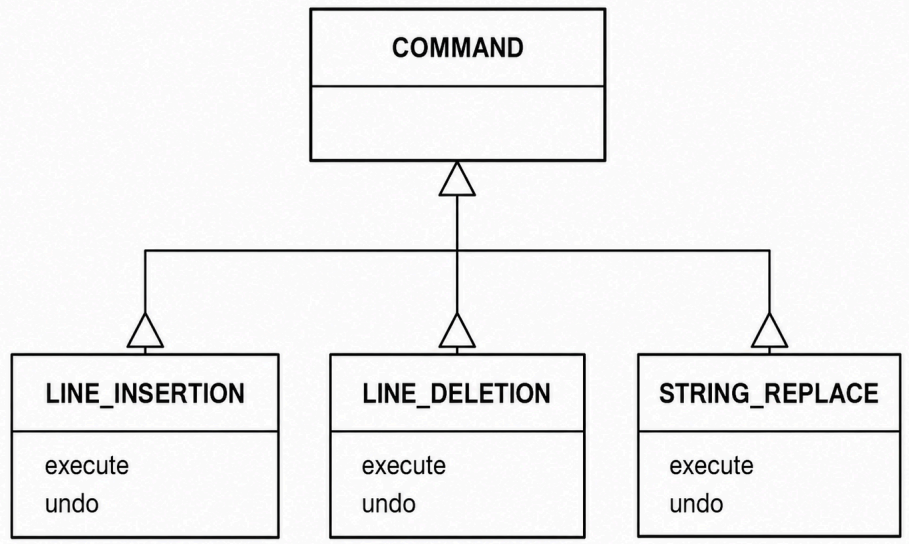


Рисунок 9.1 - Ієрархія класу *COMMAND*

Тут *COMMAND* задає спільну абстракцію, а нащадки визначають специфічну поведінку кожної команди.

10.3 Успадкування, поліморфізм і динамічне зв'язування в механізмі Undo

Для збереження останньої команди можна використати сутність:

requested: COMMAND

Статичним типом requested є *COMMAND*, але під час виконання вона може посилатися на об'єкти різних класів-нащадків:

- *LINE_INSERTION*
- *LINE_DELETION*
- *STRING_REPLACE* та інших.

Отже, requested є поліморфною сутністю.

Наприклад:

create {LINE_DELETION} requested.make (...)

requested.execute

Якщо користувач запросив іншу операцію:

create {LINE_INSERTION} requested.make (...)

requested.execute

Той самий виклик: *requested.execute* призводить до виконання різних реалізацій залежно від динамічного типу об'єкта.

Те саме стосується скасування: *requested.undo.*

Якщо requested посилається на *LINE_DELETION*, автоматично викликається *LINE_DELETION.undo*. Якщо він посилається на *LINE_INSERTION*, виконується *LINE_INSERTION.undo*.

Саме динамічне зв'язування усуває необхідність перевіряти тип останньої команди:

if command_is_deletion then ...

elseif command_is_insertion then ...

Система просто викликає: *requested.undo*, а правильна реалізація вибирається автоматично.

Це є центральною роллю успадкування в механізмі відкату. Всі команди мають однаковий загальний інтерфейс, але реалізують його відповідно до власної семантики.

При цьому загальна частина системи не залежить від предметної області. Механізм керування командами однаково може використовуватися:

- у текстовому редакторі;
- графічному редакторі;
- CAD-системі;
- навчальній програмі;
- іншій інтерактивній системі.

Предметно-залежними залишаються лише конкретні класи команд та їхні операції *execute і undo.*

10.4 Багаторівневий механізм Undo/Redo та історія команд

Для однорівневого Undo достатньо зберігати лише останній об’єкт *COMMAND*. Для багаторівневого механізму потрібно зберігати послідовність виконаних команд.

Вводиться структура:

history: *SOME_LIST [COMMAND]*

Це поліморфний список, оскільки всі його елементи мають загальний статичний тип *COMMAND*, але фактично можуть бути об’єктами різних класів-нащадків.

Наприклад, список може містити:

- *LINE_INSERTION*
- *LINE_DELETION*
- *LINE_MOVE*
- *STRING_REPLACE*
- *CHARACTER_CHANGE*

Для роботи зі списком використовується курсор. У звичайному режимі курсор знаходиться на останній виконаній команді.

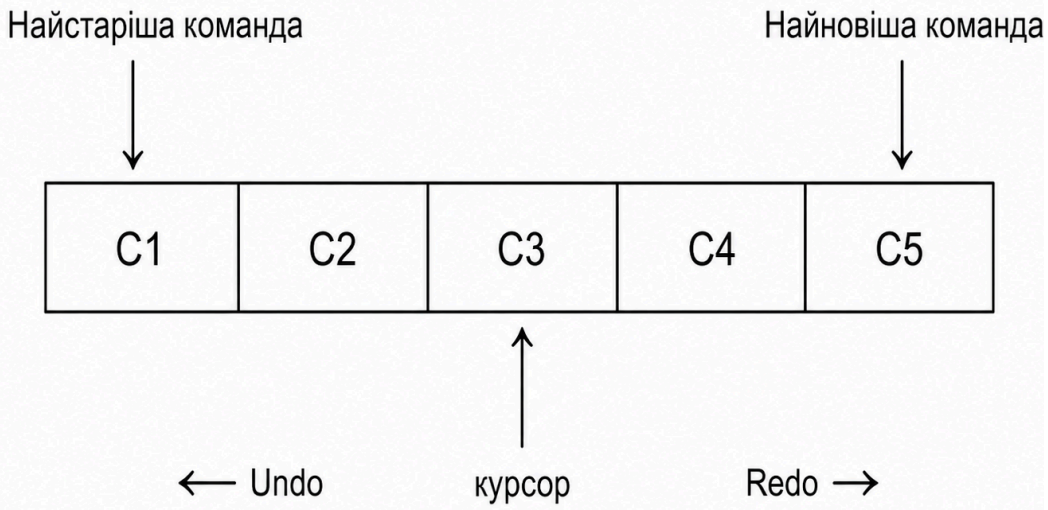


Рисунок 9.2 - Спрощене представлення історії команд

Під час Undo курсор переміщується назад. Під час *Redo* — уперед.

Алгоритм *Undo* має дуже простий вигляд:

```
if on_item then
   history.item.undo
   history.back
else
   message ("Немає команд для скасування")
end
```

Незалежно від того, який тип об’єкта міститься в поточному елементі історії, динамічне зв’язування автоматично викликає правильну реалізацію *undo*.

Для Redo використовується аналогічний алгоритм:

```
if not_last then
   history.forth
   history.item.redo
else
   message ("Немає команд для повторного виконання")
end
```

До класу *COMMAND* додається операція:

```
redo
   -- Повторно виконати скасовану команду.
do
   execute
end
```

У більшості випадків повторне виконання команди не відрізняється від початкового, тому redo просто викликає execute. Якщо для певної команди потрібна інша поведінка, відповідний клас-нащадок може повторно визначити redo.

Отже, клас *COMMAND* набуває такого загального вигляду:

```
deferred class COMMAND
feature
   execute
      deferred
      end
   undo
      deferred
      end
   redo
      do
         execute
      end
end
```

Важливе правило виникає, якщо користувач виконав декілька *Undo*, а потім замість Redo виконує нову команду.

Наприклад:

$A \rightarrow B \rightarrow C \rightarrow D$

Після двох *Undo*:

$A \rightarrow B \quad C \quad D$

$\uparrow$

поточний стан

Якщо тепер виконується команда *E*, попередні *C і D* вже не можуть бути повторно застосовані:

$A \rightarrow B \rightarrow E$

Тому всі елементи історії праворуч від курсора видаляються. Для цього використовується операція:

remove_all_right

Після цього нова команда додається в історію та виконується:

```
if not is_last then
   remove_all_right
end
history.put (requested)
requested.execute
```

10.5 Практичні аспекти реалізації та інтерфейс користувача

Деяким командам потрібні додаткові дані. Наприклад, команда вставлення рядка повинна знати:

- текст нового рядка;

- позицію вставлення.

Найпростіший варіант — зберігати аргументи безпосередньо в об’єкті команди.

Однак у великих системах може бути вигідніше відокремити тип команди від конкретного її виконання. Для цього вводиться клас, подібний до:

COMMAND_INSTANCE

command_type: COMMAND

argument: ANY

Тоді можна створити лише один об’єкт для кожного типу команди, а історія зберігатиме окремі екземпляри виконання з їхніми аргументами. Це зменшує витрати пам’яті та кількість створюваних об’єктів.

Ще одна оптимізація полягає у створенні масиву шаблонів команд:

commands: ARRAY [COMMAND]

Кожний елемент містить об’єкт певного класу-нащадка:

commands [1] → LINE_INSERTION

commands [2] → LINE_DELETION

commands [3] → STRING_REPLACE

...

Після розпізнавання коду команди система може отримати потрібний об’єкт без великої конструкції *if* або *inspect*:

requested := commands @ code

За потреби об’єкт може бути клонований для збереження окремого екземпляра в історії.

Теоретично історія може бути необмеженою. На практиці іноді встановлюється максимальна кількість команд, що запам’ятовуються.

Для цього список можна реалізувати циклічним масивом. Після заповнення всієї доступної пам’яті найстаріші команди поступово замінюються новими.

Мейєр рекомендує не встановлювати занадто мале обмеження та, за можливості, дозволити користувачеві самостійно задавати максимальну кількість команд у параметрах системи.

Внутрішній список *history* може мати пряме відображення в інтерфейсі користувача у вигляді вікна історії.

Таке вікно показує послідовність останніх команд. Поточна команда виділяється. Виконання *Undo* переміщує виділення вгору, а *Redo* — вниз.

Якщо після декількох *Undo* користувач виконує нову команду, всі скасовані, але не повторно виконані команди зникають як із внутрішньої історії, так і з видимого списку.

Таким чином, структура інтерфейсу безпосередньо відповідає внутрішній структурі механізму *Undo/Redo*.

10.6 Переваги, обмеження та загальні принципи побудови механізму відкату

Розглянутий приклад демонструє, що успадкування використовується не лише для повторного використання вже написаного коду. Воно дозволяє створити спільну абстракцію *COMMAND*, яка об'єднує всі дії інтерактивної системи.

Основні переваги такого підходу наведено в таблиці 10.2.

Таблиця 10.2 – Переваги об'єктно-орієнтованого механізму *Undo/Redo*

Аспект	Результат
Успадкування	Усі конкретні команди мають спільний інтерфейс *COMMAND*
Поліморфізм	Історія може зберігати різні типи команд в одній структурі
Динамічне зв'язування	Автоматично вибирається правильна реалізація execute, undo або *redo*
Розширюваність	Нова команда додається створенням нового класу-нащадка
Економія пам'яті	Зберігається лише інформація, потрібна для відновлення команди
Повторне використання	Загальний механізм історії не залежить від предметної області

Важливо, що кожний новий тип команди оформлюється окремим, зазвичай невеликим класом. Велика кількість таких класів не повинна розглядатися як недолік. Вони представляють окремі абстракції, мають просту ієрархію та локалізують поведінку конкретних команд.

Згодом клас *COMMAND* може бути розширений додатковими компонентами:

- argument — аргумент команди;
- *help* — довідка;
- інформація для журналювання;
- статистика використання.

Тобто клас, який спочатку вводився для підтримки *Undo*, поступово стає повноцінною абстракцією взаємодії користувача із системою.

Не кожна команда може мати операцію *Undo*. Наприклад:

- фізична дія після її виконання може бути незворотною;
- друк документа вже не можна «віддрукувати назад»;
- збереження даних у зовнішній системі може бути занадто складно або дорого скасовувати.

Такі випадки повинні бути обмеженими й зрозумілими користувачеві. Інтерфейс має чітко показувати, що команда не підлягає скасуванню.

Такий варіант прямо запропоновано в завданнях до розділу як подальше розширення механізму.

Основний результат полягає в тому, що механізм історії перестає бути спеціальним рішенням конкретного текстового або графічного редактора. Його можна оформити як загальну програмну компоненту та повторно використовувати в різних інтерактивних системах.

Мейєр окремо зазначає, що механізм історії команд є корисним незалежно від предметної області і може бути оформлений як загальний клас, від якого відповідні інтерактивні засоби отримуватимуть необхідну функціональність через успадкування.

Висновки

Механізм *Undo/Redo* є важливою складовою інтерактивних систем, оскільки дозволяє користувачеві скасовувати помилкові або небажані дії та за потреби повторно їх виконувати.

Об’єктно-орієнтований підхід полягає у представленні команди як окремого об’єкта. Відкладений клас *COMMAND* визначає спільні операції execute і undo, а конкретні класи-нащадки реалізують їх відповідно до власного призначення. Для Redo базовий клас може надавати реалізацію за замовчуванням через повторний виклик execute.

Поліморфізм дозволяє зберігати різні команди в одній структурі history, а динамічне зв’язування автоматично вибирає правильну реалізацію операції під час виконання.

Багаторівневий механізм *Undo/Redo* реалізується за допомогою списку історії та курсора, який переміщується назад під час *Undo* і вперед під час Redo. Якщо після скасування користувач виконує нову команду, скасовані команди праворуч від поточного стану видаляються з історії.

Такий підхід дозволяє зберігати лише інформацію, необхідну для відновлення конкретної команди, не копіюючи повний стан системи. Механізм залишається незалежним від конкретної предметної області та легко розширюється додаванням нових класів команд.

Приклад *Undo/Redo* показує практичну силу поєднання успадкування, поліморфізму, динамічного зв’язування та відкладених класів для побудови гнучких і повторно використовуваних інтерактивних систем.

Лекція 11

Тема: ООП методології

Мета: сформувати у студентів розуміння сучасних гнучких підходів до організації розроблення програмного забезпечення, ознайомити з основними положеннями Agile, структурою Scrum та принципами Kanban, а також показати їх застосування під час командного розроблення об’єктно-орієнтованих програмних систем.

План лекції

1. Поняття методології розроблення програмного забезпечення.
2. Agile та основні принципи гнучкого розроблення.
3. Scrum: структура, команда, події та артефакти.
4. Kanban та керування потоком робіт.
5. Порівняння Agile, Scrum і Kanban.
6. Застосування гнучких підходів під час об’єктно-орієнтованого розроблення.
7. Висновки.

Поняття Agile, Scrum і Kanban часто використовуються разом, однак вони не є тотожними. З академічної точки зору також важливо уточнити, що Agile, Scrum і Kanban не визначають механізми інкапсуляції, успадкування, поліморфізму чи побудови класів. Вони належать насамперед до організації процесу створення програмного забезпечення. У контексті цієї дисципліни їх доцільно розглядати як підходи, за допомогою яких організовується командна розробка об’єктно-орієнтованих систем.

11.1 Поняття методології розроблення програмного забезпечення

Створення програмного забезпечення включає не лише написання програмного коду. Команда повинна визначати вимоги, планувати роботу, розподіляти завдання, проєктувати архітектуру, реалізовувати компоненти, перевіряти їх, отримувати зворотний зв’язок і вносити зміни. Метод або процес розроблення визначає, як ці види діяльності організовані та як вони пов’язані між собою.

Традиційні процеси часто передбачали послідовне проходження великих етапів: спочатку повне визначення вимог, потім проєктування, реалізація і лише після цього тестування та передавання результату замовникові. Такий процес може бути прийнятним, якщо вимоги достатньо стабільні, але стає складнішим, коли вони змінюються вже під час створення системи.

Гнучкі підходи виходять з іншого припущення: під час роботи над програмним продуктом команда поступово отримує нову інформацію, тому вимоги, технічні рішення та пріоритети можуть змінюватися. Замість одного великого результату

наприкінці проєкту робота виконується невеликими частинами з регулярним аналізом отриманих результатів.

Для об'єктно-орієнтованого розроблення такий підхід є природним. Початкова модель класів не обов'язково залишається незмінною протягом усього проєкту. У процесі розроблення можуть уточнюватися відповідальності класів, з'являтися нові абстракції, змінюватися зв'язки між об'єктами або виконуватися рефакторинг уже реалізованої структури.

Основне завдання методології полягає не у визначенні структури конкретного класу, а в організації процесу, в межах якого така структура створюється, перевіряється та вдосконалюється.

11.2 Agile та основні принципи гнучкого розроблення

Основою сучасного гнучкого розроблення є **Agile Manifesto**, сформульований у 2001 році групою розробників програмного забезпечення. Agile не є конкретним процесом із визначеними ролями, етапами або обов'язковими подіями. Це передусім система цінностей і принципів, на основі яких можуть будуватися різні методи розроблення.

Agile Manifesto визначає чотири основні цінності:

Люди та співпраця важливіші за процеси та інструменти. Процеси й засоби автоматизації необхідні, але результат розроблення значною мірою залежить від взаємодії людей, їхньої компетентності та здатності спільно приймати рішення.

Працюючий програмний продукт важливіший за вичерпну документацію. Документація зберігає свою цінність, однак головним результатом розроблення є програмне забезпечення, яке реально виконує необхідні функції.

Співпраця із замовником важливіша за формальне обговорення умов контракту. Регулярна взаємодія із замовником дозволяє швидше уточнювати потреби та коригувати напрям розроблення.

Готовність до змін важливіша за дотримання початкового плану. План є необхідним засобом організації роботи, але він не повинен перешкоджати внесенню обґрунтованих змін.

Agile Manifesto при цьому не стверджує, що процеси, документація, контракти або плани не потрібні. Їхня цінність визнається, але за наявності конфлікту більшу увагу рекомендується приділяти елементам, розташованим у лівій частині кожної пари.

Чотири цінності доповнено дванадцятьма принципами. Для програмної інженерії особливо важливими є раннє та регулярне постачання корисного програмного забезпечення, готовність приймати зміни вимог, регулярна співпраця бізнесу й розробників, підтримання сталого темпу роботи, увага до технічної досконалості, простота та регулярний аналіз командою власної ефективності.

Agile-підхід фактично формує короткий цикл: **планування → реалізація → отримання результату → зворотний зв'язок → адаптація.**

Після отримання результату команда не просто переходить до наступної частини початкового плану, а перевіряє, чи залишається план правильним. Якщо змінилися вимоги або команда отримала нові знання про систему, наступні рішення можуть бути скориговані.

Це особливо важливо під час проєктування складних об'єктно-орієнтованих систем. На ранньому етапі розробники не завжди можуть правильно визначити всі класи та зв'язки між ними. Реалізація невеликої частини системи дозволяє перевірити початкову модель і за потреби змінити її без повного перепроєктування всього продукту.

Отже, Agile створює загальні принципи гнучкої розробки, але не встановлює єдиного алгоритму організації роботи. Одним із конкретних фреймворків, що реалізує ітеративний та інкрементальний підхід, є Scrum.

11.3 Scrum: структура, команда, події та артефакти

Scrum визначається як легкий фреймворк, що допомагає людям, командам та організаціям створювати цінність за допомогою адаптивних рішень для складних проблем. Scrum ґрунтується на емпіризмі та Lean-мисленні й використовує ітеративний та інкрементальний підхід.

Основою Scrum є три емпіричні складові: **прозорість, інспекція** та **адаптація**. Стан роботи має бути достатньо видимим для учасників; результати та прогрес регулярно перевіряються; якщо виявлено проблему або відхилення, процес чи продукт коригуються.

Основною організаційною одиницею є **Scrum Team**. Відповідно до Scrum Guide вона складається з одного Product Owner, одного Scrum Master та Developers. Усередині Scrum Team не передбачено окремих підкоманд або внутрішньої ієрархії; команда є кросфункціональною та самокерованою.

Product Owner відповідає за максимізацію цінності продукту та ефективне керування Product Backlog. До його відповідальності належать формування й пояснення Product Goal, визначення елементів Product Backlog, їх упорядкування та забезпечення прозорості списку робіт.

Developers створюють придатний до використання інкремент продукту. Вони формують план роботи на спринт, забезпечують відповідність результату Definition of Done, щоденно коригують план відповідно до Sprint Goal і спільно відповідають за професійне виконання роботи.

Scrum Master відповідає за правильне застосування Scrum і допомагає команді підвищувати ефективність. Його робота включає підтримку самокерованості команди,

допомогу в усуненні перешкод і забезпечення продуктивного проведення подій Scrum.

Робота Scrum організовується навколо **Sprint**. Спринт є фіксованим за тривалістю періодом не більше одного місяця, протягом якого створюється корисний інкремент продукту. Після завершення одного спринту одразу починається наступний.

У межах спринту передбачено основні події Scrum.

Sprint Planning відкриває спринт. Команда визначає цінність майбутнього спринту, вибирає частину роботи з Product Backlog та формує план її виконання. Результатом є Sprint Backlog, що містить Sprint Goal, вибрані елементи Product Backlog та план створення інкременту.

Daily Scrum — коротка щоденна подія для Developers. Її мета полягає у перевірці прогресу щодо Sprint Goal і коригуванні плану майбутньої роботи. Scrum Guide визначає для Daily Scrum тривалість 15 хвилин.

Sprint Review виконується наприкінці спринту для перевірки отриманого результату. Scrum Team і ключові зацікавлені сторони аналізують виконану роботу, зміни в середовищі та визначають можливі наступні кроки. За результатами може бути змінено Product Backlog.

Sprint Retrospective спрямована не стільки на продукт, скільки на сам процес роботи. Команда аналізує людей, взаємодію, процеси, інструменти та проблеми попереднього спринту і визначає зміни, які можуть підвищити якість та ефективність наступної роботи.

Scrum також визначає три основні артефакти.

Product Backlog — упорядкований і такий, що розвивається, список того, що необхідно для вдосконалення продукту. Він є єдиним джерелом роботи Scrum Team.

Sprint Backlog включає мету спринту, вибрані елементи Product Backlog та конкретний план їх реалізації.

Increment — конкретний крок у напрямі Product Goal. Щоб робота вважалася частиною інкременту, вона повинна відповідати Definition of Done.

Для об’єктно-орієнтованого проєкту елементами Product Backlog можуть бути, наприклад, реалізація нового сценарію використання, створення або зміна групи класів, рефакторинг ієрархії, реалізація інтерфейсу або усунення дефекту. Scrum при цьому не визначає, які саме класи необхідно створити чи які шаблони проєктування використати. Ці технічні рішення залишаються відповідальністю розробників.

11.4 Kanban та керування потоком робіт

На відміну від Scrum, Kanban концентрується насамперед на потоці роботи. У чинному The Kanban Guide Kanban визначається як стратегія оптимізації потоку цінності через процес. Вона базується на трьох взаємопов’язаних практиках:

визначенні та візуалізації робочого процесу, активному керуванні елементами цього процесу та його постійному вдосконаленні.

Ключовим поняттям є **flow** — рух потенційної цінності через систему. Мета полягає не просто в тому, щоб усі учасники були постійно зайняті, а в тому, щоб елементи роботи проходили процес ефективно та передбачувано.

Для цього команда визначає **Definition of Workflow (DoW)** — спільне розуміння того, як саме робота рухається через систему. Воно включає типи робочих елементів, початкові та кінцеві точки процесу, стани, через які проходить робота, правила контролю незавершеної роботи та явні політики переходу між станами.

Робочий процес зазвичай візуалізується за допомогою **Kanban-дошки**. Найпростіший варіант може містити стани To Do, In Progress, Review та Done. Кожне завдання подається окремим елементом і переміщується між станами відповідно до фактичного виконання роботи. Kanban Guide не встановлює обов'язкової форми дошки; її структура залежить від конкретного процесу.

Особливе значення має поняття **Work in Progress (WIP)** — робота, яку вже розпочато, але ще не завершено. Kanban вимагає явно контролювати кількість такої роботи. Новий елемент повинен розпочинатися тільки тоді, коли в системі існує достатня пропускна здатність для його виконання. Такий підхід сприяє формуванню системи витягування роботи — pull system.

Наприклад, якщо в колонці In Progress встановлено WIP-обмеження 3, команда не повинна брати четверте завдання, поки хоча б одне з трьох поточних не перейде до наступного стану. Таким чином увага спрямовується не на початок максимальної кількості завдань, а на завершення вже розпочатих.

Kanban також передбачає активне керування елементами роботи: контроль WIP, виявлення заблокованих завдань, недопущення надмірного «старіння» елементів і регулярний аналіз стану потоку.

На відміну від Scrum, Kanban не вимагає поділяти роботу на спринти. Зміни можуть надходити до системи безперервно, а нове завдання починається тоді, коли для нього з'являється вільна місткість у відповідній частині робочого процесу.

Це робить Kanban зручним, наприклад, для команд супроводу програмного забезпечення, де нові дефекти, запити користувачів та технічні задачі надходять постійно й не завжди доцільно чекати початку наступної ітерації.

11.5 Порівняння Agile, Scrum і Kanban

Об'єктно-орієнтована методологія визначає не лише спосіб написання програмного коду, а й підхід до аналізу предметної області, виділення об'єктів і класів, розподілу відповідальності між ними, встановлення зв'язків та організації їхньої взаємодії. Різні методології об'єктно-орієнтованого проєктування можуть використовувати різні способи пошуку класів: аналіз понять предметної області,

визначення відповідальностей об’єктів або моделювання реальних сутностей і перенесення їх у програмну модель. Універсального способу такої декомпозиції не існує, тому модель системи уточнюється відповідно до особливостей конкретного проєкту.

У цьому контексті Agile, Scrum і Kanban не замінюють об’єктно-орієнтованого аналізу та проєктування. Вони визначають організацію процесу, в межах якого застосовуються об’єктні технології. Це важливе розмежування: ООП відповідає на питання, як побудувати програмну систему з класів та об’єктів, тоді як Agile, Scrum і Kanban допомагають визначити, як організувати поступове створення та вдосконалення цієї системи командою.

Для об’єктно-орієнтованої розробки особливо важливо, що початкова модель класів рідко залишається незмінною протягом усього життєвого циклу системи. Після реалізації окремих сценаріїв використання може виявитися, що:

- певний клас має надто багато відповідальностей;
- декілька класів містять однакові компоненти;
- обрана ієрархія успадкування є невдалою;
- замість успадкування доцільніше застосувати композицію;
- необхідно виділити новий абстрактний клас або інтерфейс;
- зв’язки між об’єктами потрібно спростити;
- контракт класу або окремої операції потребує уточнення.

Саме тому гнучкі методології добре поєднуються з об’єктними технологіями: програмна модель може розвиватися поступово разом із функціональністю системи.

Agile підтримує ітеративне вдосконалення об’єктної моделі. Замість спроби одразу побудувати остаточну ієрархію класів команда може реалізувати необхідний мінімум, перевірити його в працездатному програмному забезпеченні та після отримання нової інформації виконати рефакторинг. При цьому зміна структури класів не повинна змінювати зовнішню поведінку вже правильно працюючої частини системи.

У межах Agile особливого значення набувають такі практики об’єктно-орієнтованого розроблення:

- поступове уточнення класів і їхніх відповідальностей;
- рефакторинг;
- підтримання слабкого зв’язування між класами;
- інкапсуляція змін;
- використання поліморфізму замість великих умовних конструкцій;
- повторне використання компонентів;
- постійне тестування після зміни структури програми.

Scrum надає часову й організаційну структуру для такого поступового розвитку об’єктної системи. У межах одного спринту команда може реалізувати завершений

сценарій використання, що включає декілька взаємодіючих класів. Після перевірки інкременту на Sprint Review можуть з'явитися нові вимоги, а під час наступного спринту модель класів може бути розширена або уточнена.

Наприклад, реалізація нового сценарію може потребувати:

1. визначення нових об'єктів предметної області;
2. створення відповідних класів;
3. установлення асоціацій між ними;
4. визначення операцій та відповідальності;
5. використання успадкування або композиції;
6. реалізації та тестування;
7. рефакторингу після перевірки отриманого рішення.

Таким чином, кожний спринт може містити повний невеликий цикл об'єктно-орієнтованого аналізу, проєктування, реалізації та перевірки.

Kanban не задає структури об'єктної моделі, але дозволяє зробити видимим процес роботи над її компонентами. Наприклад, завдання, пов'язані з класами та об'єктами, можуть проходити через стани:

Analysis → Design → Implementation → Code Review → Testing → Done.

На Kanban-дошці окремими робочими елементами можуть бути:

- «виділити клас ORDER»;
- «розділити відповідальність класу CUSTOMER»;
- «реалізувати інтерфейс PAYMENT»;
- «замінити успадкування композицією»;
- «виконати рефакторинг ієрархії»;
- «додати тестування контракту класу».

Це дозволяє контролювати не лише реалізацію функціональних вимог, а й архітектурні та рефакторингові роботи.

Роль розглянутих підходів у процесі об'єктно-орієнтованої розробки узагальнено в таблиці 11.1.

Таблиця 11.1 – Роль Agile, Scrum і Kanban в об'єктно-орієнтованій розробці

Підхід	**Роль у процесі розроблення**	**Застосування в ООП та об'єктних технологіях**
Agile	Визначає загальні принципи адаптивного та ітеративного розроблення	Підтримує поступове уточнення класів, рефакторинг, зміну архітектури та постійне вдосконалення об'єктної моделі
Scrum	Організовує роботу короткими спринтами з отриманням завершеного інкременту	Дозволяє реалізовувати об'єктну систему частинами, поступово додаючи класи, зв'язки та поведінку

Kanban	Організовує безперервний потік робіт і обмежує незавершені задачі	Візуалізує виконання задач аналізу, проєктування класів, реалізації, рефакторингу, перевірки коду та тестування

Отже, Agile, Scrum і Kanban слід розглядати не як альтернативу об’єктно-орієнтованій методології, а як організаційні засоби її практичного застосування. Об’єктно-орієнтований підхід визначає структуру програмної системи через класи, об’єкти, інкапсуляцію, успадкування, універсалізацію, поліморфізм та композицію, тоді як гнучкі методи визначають спосіб організації роботи над цією структурою.

Таке поєднання дає змогу не намагатися сформувати остаточну модель системи на початку проєкту, а поступово уточнювати її в процесі реалізації. Саме тому об’єктні технології добре підтримують ітеративний розвиток: класи можуть розширюватися, реалізації операцій — замінюватися, нові типи — вводитися через успадкування або композицію, а клієнтський код при правильному проєктуванні залишається максимально незалежним від цих змін.

11.6 Застосування гнучких підходів під час об’єктно-орієнтованого розроблення

Гнучкі методи не замінюють принципів об’єктно-орієнтованого проєктування. Agile не визначає структуру ієрархії класів, Scrum не встановлює правила успадкування, а Kanban не визначає механізм поліморфізму. Вони відповідають на інше питання — **як організувати роботу команди над програмною системою.**

Під час створення об’єктно-орієнтованої системи команда може спочатку реалізувати лише невелику частину предметної області. Після отримання працездатного результату перевіряється правильність обраної декомпозиції. Якщо відповідальність певного класу виявилася занадто широкою, клас може бути розділений. Якщо декілька класів містять однакову поведінку, вона може бути винесена до спільної абстракції. Якщо успадкування створює надмірну залежність, його можна замінити композицією.

Отже, архітектура не повинна розглядатися як повністю завершена ще до написання першого рядка програми. Водночас це не означає відмови від проєктування. Agile Manifesto окремо підкреслює значення технічної досконалості та доброго дизайну, а також принцип простоти.

У Scrum технічне вдосконалення може бути частиною роботи спринту. Definition of Done дає команді спільне розуміння того, за яких умов робота справді вважається завершеною. Якщо програмний компонент не відповідає встановленим вимогам до якості, він не повинен вважатися частиною готового інкременту.

Kanban, своєю чергою, допомагає зробити видимим технічний процес. Наприклад, окремими станами можуть бути Design, Implementation, Code Review, Testing та Done. Якщо велика кількість задач накопичується на етапі Code Review, команда отримує явний сигнал про вузьке місце процесу і може спрямувати зусилля на його усунення.

Для командного об’єктно-орієнтованого розроблення особливе значення мають короткі цикли зворотного зв’язку. Вони дозволяють швидко перевіряти не лише функціональні можливості, але й якість архітектурних рішень. Помилкова абстракція, невдала ієрархія успадкування або надмірна залежність класів можуть бути виявлені раніше, ніж вони поширяться на всю систему.

Водночас гнучкість не означає відсутності дисципліни. Команда повинна підтримувати узгоджені правила програмування, виконувати тестування, проводити перевірку коду, контролювати якість та систематично виконувати рефакторинг. У Scrum це підтримується, зокрема, через Definition of Done, а Agile-принципи прямо пов’язують гнучкість із технічною досконалістю та добрим дизайном.

Висновки

Agile, Scrum і Kanban є засобами організації сучасного процесу розроблення програмного забезпечення, але мають різне призначення.

Agile визначає систему цінностей і принципів, відповідно до яких перевага надається взаємодії людей, працездатному програмному забезпеченню, співпраці із замовником та готовності адаптуватися до змін.

Scrum конкретизує ітеративний та інкрементальний спосіб роботи через Scrum Team, спринти, регулярні події та три основні артефакти. Його основою є прозорість, інспекція й адаптація, а результатом кожного спринту має бути придатний до використання інкремент.

Kanban орієнтується на оптимізацію потоку цінності. Основними його механізмами є визначення й візуалізація робочого процесу, активне керування роботою, контроль WIP та постійне вдосконалення потоку.

Під час об’єктно-орієнтованого розроблення ці підходи не замінюють принципів створення класів, контрактів, успадкування, поліморфізму чи інших технічних механізмів. Вони створюють організаційне середовище, в якому об’єктно-орієнтована система може розроблятися невеликими частинами, регулярно перевірятися та вдосконалюватися відповідно до отриманого зворотного зв’язку.

Лекція 12

Тема: Використовуйте наслідування правильно

Мета: сформувати у студентів розуміння методологічних правил правильного застосування успадкування, навчити відрізняти успадкування від клієнтського зв'язку між класами, розглянути типові помилки побудови ієрархій та основні допустимі форми успадкування, а також показати підходи до поступового формування гнучких і стабільних структур класів.

План лекції

1. Необхідність правильного застосування успадкування.
2. Вибір між успадкуванням і клієнтським зв'язком.
3. Типові помилки використання успадкування.
4. Основні допустимі форми успадкування.
5. Успадкування для розширення та повторного використання.
6. Побудова та розвиток ієрархій класів.
7. Практичні правила правильного використання успадкування.
8. Висновки.

Технічне знання синтаксису успадкування ще не означає правильного його застосування. Б. Мейєр підкреслює, що питання про те, коли і як використовувати успадкування, є одним із найбільш дискусійних у об'єктній технології. Методологічна проблема полягає не в тому, як записати зв'язок між класами, а в тому, чи відповідає цей зв'язок реальним абстракціям програмної системи та чи сприятиме він її подальшому розвитку. Саме цьому присвячено розділ 24 Using inheritance well книги Object-Oriented Software Construction.

12.1 Необхідність правильного застосування успадкування

Успадкування є одним із найпотужніших механізмів об'єктно-орієнтованого програмування. Воно дозволяє визначати нові класи на основі вже існуючих, повторно використовувати компоненти, спеціалізувати типи, підтримувати поліморфізм і поступово розширювати програмні системи.

Проте саме універсальність цього механізму створює небезпеку його надмірного використання. Не кожна подібність між двома поняттями повинна приводити до створення зв'язку успадкування. Не кожен випадок повторного використання коду також є достатньою підставою для створення класу-нащадка.

Основне правило Мейєра можна сформулювати через відношення **«is-a» — «є різновидом»**: Клас *B* не повинен успадковувати від класу *A*, якщо неможливо обґрунтовано стверджувати, що кожний об'єкт *B* можна розглядати як об'єкт *A*.

Наприклад:

- *SQUARE* може бути різновидом *RECTANGLE*;
- *RECTANGLE* може бути різновидом *POLYGON*;
- *SOFTWARE_ENGINEER* може бути різновидом *ENGINEER*.

Натомість:

- *CAR_OWNER* не є різновидом *CAR*;
- *AIRPLANE* не є різновидом *VENTILATION_SYSTEM*;
- *APPLE_PIE* не є різновидом *APPLE*.

У таких випадках між об'єктами існує відношення володіння, використання або включення, але не успадкування.

Класичний приклад неправильного моделювання:

```
class CAR_OWNER
inherit
   PERSON
   CAR
end
```

означав би, що власник автомобіля одночасно є людиною та автомобілем.

Правильніше представити зв'язок так:

```
class CAR_OWNER
inherit
   PERSON
feature
   my_car: CAR
end
```

Тут успадкування від *PERSON* відображає відношення «власник автомобіля є людиною», а атрибут *my_car* — відношення «власник має автомобіль». Саме такий поділ наведено Мейєром як базовий приклад правильного розмежування двох типів зв'язків.

Водночас правило *is-a* є насамперед засобом відхилення очевидно неправильних рішень. Самого твердження *«B є A»* недостатньо, щоб автоматично зробити *B* нащадком *A*. Вибір структури залежить також від того, як об'єкти повинні змінюватися та використовуватися в програмі.

12.2 Вибір між успадкуванням і клієнтським зв'язком

В об'єктно-орієнтованій системі два класи можуть бути пов'язані насамперед двома способами:

- **успадкуванням** — один клас є спеціалізованим різновидом іншого;
- **клієнтським зв'язком** — один клас використовує об'єкт іншого класу як компонент або постачальника послуг.

У спрощеній формі:

- успадкування → *is-a*
- клієнтський зв’язок → *has-a*

Проте вибір між ними не завжди очевидний. Навіть відношення, яке можна описати через «є», інколи можна реалізувати через «має». Тому Мейєр пропонує два додаткові критерії: правило **зміни** та **правило поліморфізму**.

Успадкування встановлює постійне відношення між класами. Якщо *B* успадковує *A*, кожний об’єкт *B* залишається об’єктом типу *A* протягом усього свого існування. Такий зв’язок не можна змінити під час виконання програми без зміни самого визначення класу.

Клієнтський зв’язок є гнучкішим. Об’єкт може містити посилання на інший об’єкт, а значення цього посилання може змінюватися під час виконання.

Отже:

> Якщо відповідний компонент об’єкта повинен змінюватися під час виконання програми, слід використовувати клієнтський зв’язок, а не успадкування.
>
> Наприклад, якщо клас *WINDOW* використовує певний графічний інструментарій, доцільно мати:
>
> toolkit: *TOOLKIT*
>
> Тоді toolkit може посилатися на різні реалізації. Якщо ж *WINDOW* безпосередньо успадковує конкретний платформний клас, зв’язок стає статичним.

Протилежний критерій виникає тоді, коли необхідно використовувати об’єкти спеціалізованого класу через сутності загального типу.

Нехай: *employee: EMPLOYEE* повинен під час виконання посилатися на об’єкти:

- *ENGINEER*
- *MANAGER*
- *ACCOUNTANT*

Якщо ці класи є нащадками *EMPLOYEE*, поліморфне приєднання є природним.

Так само: *staff: LIST [EMPLOYEE]* може містити об’єкти різних класів-нащадків.

Мейєр формулює це так: якщо сутність або компонент структури загального типу повинен мати можливість посилатися на об’єкти спеціалізованого типу, необхідне успадкування.

Критерії вибору узагальнено в таблиці 12.1.

Таблиця 12.1 – Критерії вибору між успадкуванням і клієнтським зв’язком

Ситуація	**Доцільний зв’язок**
В має компонент типу А, який може змінюватися під час виконання	Клієнтський зв’язок

Об’єкт B лише використовує послуги A	Клієнтський зв’язок
Кожний B обґрунтовано можна розглядати як A	Можливе успадкування
Сутність типу A повинна мати можливість посилатися на об’єкт B	Успадкування
Поліморфна структура типу A повинна містити об’єкти B	Успадкування
Відношення між A і B повинно змінюватися між A і B повинно змінюватися динамічно	Клієнтський зв’язок

Як видно з таблиці 12.1, вибір визначається не лише подібністю понять, а насамперед необхідною поведінкою програмних об’єктів.

12.3 Типові помилки використання успадкування

Мейєр виділяє декілька характерних способів неправильного використання успадкування.

Найпростіша помилка виникає тоді, коли успадкування використовується для моделювання складової частини об’єкта.

Приклади неправильних тверджень:

- *CAR_OWNER is-a CAR*
- *AIRPLANE is-a VENTILATION_SYSTEM*
- *ROSE_TREE is-a ROSE*

У кожному з цих випадків насправді потрібно описувати наявність компонентаа.

Інша типова помилка — convenience inheritance, коли клас успадковує від іншого тільки тому, що розробникові потрібні декілька готових операцій.

Наприклад, якщо *REPORT* потребує операцій форматування з класу *TEXT_PROCESSOR*, це ще не означає: *REPORT is-a TEXT_PROCESSOR*

Повторне використання окремої реалізації саме по собі не повинно створювати неправильну модель класів. Мейєр не забороняє успадкування реалізації як таке; помилкою є використання випадкового класу-предка без зміс.

Таксономанія (taxomania) — це надмірне прагнення створювати класи-нащадки для кожного можливого різновиду об’єктів.

Наприклад, наявність атрибута статі в *PERSON* ще не означає, що необхідні:

- *MALE*
- *FEMALE*

Якщо ці класи не мають власних операцій, не перевизначають успадковані операції й не вводять додаткових інваріантів, вони не додають нової програмної абстракції.

Звідси випливає **правило таксономанії:**

Кожний клас-нащадок повинен ввести новий компонент, повторно визначити успадкований компо.

Якщо система лише повинна зберігати значення певної властивості, іноді достатньо атрибута: *gender: GENDER.*

Створення окремих класів виправдане тоді, коли відповідні різновиди справді мають різну поведінку або різні семантичні обмеження.

Отже, ієрархія класів не повинна копіювати всі можливі класифікації реального світу. Її призначення — підтримувати потреби конкретного програмного забезпечення.

12.4 Основні допустимі форми успадкування

У книзі Object-Oriented Software Construction успадкування розглядається значно ширше, ніж проста спеціалізація типу. Мейєр виділяє декілька допустимих форм, серед яких підтипізація, обмеження, розширення, варіація, реіфікація, структурне, реаліня.

Основні форми узагальнено в таблиці 12.2.

Таблиця 12.2 – Основні форми правильного застосування успадкування

Форма	**Призначення**
Успадкування підтипу	Представлення B як спеціалізованого типу A
Успадкування з обмеженням	Спеціалізація шляхом введення додаткових умов та інваріантів
Успадкування з розширенням	Додавання до успадкованої абстракції нових компонентів
Функціональна варіація	Зміна реалізації успадкованих операцій
Варіація типів	Уточнення сигнатур успадкованих компонентів
Реіфікація	Перехід від загальної відкладеної абстракції до конкретнішої реалізації
Структурне успадкування	Надання класу загальної структурної властивості
Успадкування реалізації	Використання реалізації іншого класу як основи власної
Сервісне успадкування	Отримання набору спеціально підготовлених допоміжних засобів

Це один із найбільш природних випадків:

```
class EMPLOYEE
    ...
end

class MANAGER
inherit
```

EMPLOYEE

feature

managed_team: TEAM

...

end

MANAGER зберігає властивості працівника та додає нові компоненти.

Клас-нащадок може залишатися близьким до батьківського, але використовувати іншу реалізацію певної операції.

Наприклад, *RECTANGLE* може повторно визначити операцію perimeter, успадковану від *POLYGON*, оскільки для прямокутника існує простіший алгоритм.

Функціональна варіація є одним зі способів реалізації принципу відкритості-закритості: новий варіант створюється без зміни вже.

У цьому випадку батьківський клас описує загальну абстракцію, часто у відкладеному вигляді, а нащадок визначає конкретну реалізацію.

Наприклад: *LIST [G]* може задавати загальне поняття списку, тоді як конкретний клас реалізує його через зв’язану або масивну структуру.

Батьківський клас задає певну властивість, що стосується всього об’єкта. Наприклад:

COMPARABLE

STORABLE

Клас, який успадковує *COMPARABLE*, оголошує, що його об’єкти можуть порівнюватися. Важливо, щоб ця властивість стосувалася саме об’єкта в цілому. Саме тому *AIRPLANE* може бути *COMPARABLE*, але не повинен успадковувати *VENTILATION_SYSTEM*: вентиляційна система є складовою літака, а.

Іноді клас створюється спеціально як постачальник набору пов’язаних засобів для нащадків. Мейєр називає це facility inheritance.

Наприклад, клас *ASCII* містить константи для *ASCII*-кодів і призначений для успадкування класами, яким ці константи потрібні. Аналогічно допоміжний клас може надавати операції математичної бібліотеки.

Таким чином, успадкування не слід зводити лише до простої біологічної моделі «рід — вид». Воно є інструментом побудови програмних абстракцій, але кожен зв’язок повинен мати чітке методологічне призначення.

12.5 Успадкування для розширення та повторного використання

Успадкування виконує одночасно дві важливі функції:

- спеціалізацію типів;
- розширення програмних модулів.

Саме поєднання цих властивостей забезпечує можливість повторно використовувати не лише програмний код, а й архітектурні рішення.

Водночас успадкування реалізації є сильнішим зв’язком, ніж звичайне використання класу як постачальника. Клас-нащадок може залежати від деталей реалізації предка, тому зміни в батьківському класі потенційно впливають на його нащадків. Мейєр зазначає, що використання реалізації є більш зобов’язувальним рішенням.

Звідси випливає практичний принцип: не потрібно застосовувати успадкування тільки тому, що воно дозволяє швидко отримати готовий код.

Перед встановленням такого зв’язку необхідно перевірити:

- чи існує змістовна абстракція батьківського класу;
- чи є нащадок дійсним різновидом або допустимою програмною спеціалізацією;
- чи потрібен поліморфізм;
- чи є зв’язок постійним;
- чи не буде простіше використати об’єкт іншого класу як компонент.

При повторному визначенні операцій також необхідно зберігати їхню семантику. Відповідно до розглянутого раніше проєктування за контрактом клас-нащадок не повинен вимагати від клієнта більше, ніж предок, і не повинен гарантувати менше. Успадковані інваріанти. Отже, повторне використання повинно здійснюватися не ціною руйнування абстракцій, а через правильно побудовану структуру класів.

12.6 Побудова та розвиток ієрархій класів

Поширена помилка полягає в припущенні, що ієрархію завжди необхідно будувати строго зверху вниз: спочатку визначити найбільш загальний клас, потім його спеціалізації, а потім конкретні класи.

У реальній розробці це відбувається не завжди.

Мейєр зазначає, що структура, яку зручно пояснювати від загального до конкретного, не обов’язково була створена саме таким способом. Часто спочатку виникає конкретний клас, а загальна абстракція стає очевидно.

Для побудови ієрархій використовуються два взаємодоповнювальні напрями.

Розробник спочатку визначає загальне поняття, а потім створює його різновиди:

FIGURE → POLYGON → RECTANGLE → SQUARE

Це рух від абстрактного до конкретного.

Іноді спочатку існує клас *B*, а згодом стає зрозуміло, що він є лише окремим випадком більш загальної абстракції *A*. Тоді вводиться новий батьківський клас, а наявний клас перетворюється на його нащадка.

Інший типовий випадок виникає тоді, коли вже існують два класи *E і F*, у яких виявляється спільна сутність. Спільні властивості переносяться до нового батьківського класу *D*.

Таке перепроєктування не слід розглядати як помилку. Воно є нормальною частиною розвитку об’єктно-орієнтованої системи. Мейєр описує практичне формування ієрархій як поєднання дедуктивного та індуктивного підходів — руху від.

Особливо важливо, що абстрагування та факторизація часто не потребують змін клієнтського коду. Якщо клієнт просто використовує експортовані операції класу, зміна його предків може залишитися для клієнта непомітною. Це є практичним проявом.

Отже, не потрібно прагнути отримати «ідеальну» ієрархію з першої спроби. Важливіше забезпечити можливість її послідовного вдосконалення без руйнування наявних клієнтів.

12.7 Практичні правила правильного використання успадкування

Основні рекомендації щодо побудови ієрархій можна сформулювати таким чином:

1. Перевіряйте відношення is-a.

Якщо не можна обґрунтовано стверджувати, що кожний об’єкт класу-нащадка є об’єктом батьківського типу, успадкування, найімовірніше, вибране неправильно.

2. Не використовуйте успадкування для відношення has-a.

Об’єкти, які є частинами інших об’єктів або використовуються ними, слід представляти через клієнтські зв’язки.

3. Враховуйте можливість зміни під час виконання.

Якщо компонент потрібно замінювати динамічно, клієнтський зв’язок забезпечує більшу гнучкість.

4. Використовуйте успадкування, коли потрібен поліморфізм.

Якщо сутності загального типу повинні посилатися на об’єкти спеціалізованих типів, необхідна ієрархія успадкування.

5. Не створюйте порожні рівні класифікації.

Клас-нащадок повинен вводити нову властивість, нову операцію, повторно визначати успадковану поведінку або посилювати інваріант.

6. Не успадковуйте лише заради випадкової зручності.

Наявність корисної операції в іншому класі ще не робить цей клас правильним предком.

7. Кожний зв’язок успадкування повинен мати зрозуміле призначення.

Мейєр формулює загальне правило: кожне використання успадкування повинно належати до однієї з методологічно виправданих категорій. Для простоти бажано, щоб один зв’ль.

8. Зберігайте контракти предків.

Перевизначення операції не повинно руйнувати очікування клієнтів батьківського класу.

9. Не ускладнюйте ієрархію без потреби.

Успадкування повинно допомагати керувати складністю, а не створювати її.

10. Дозволяйте структурі розвиватися.

Абстрагування та факторизація після появи конкретних класів є нормальним способом удосконалення об’єктної моделі.

Узагальнюючи методологію глави, Мейєр підкреслює, що найскладнішою частиною побудови структури класів є не саме успадкування, а пошук правильних абстракцій. Якщо абстракції вибрано правильно, відповідна структура успадкува.

Висновки

Успадкування є потужним засобом об’єктно-орієнтованого конструювання, але його використання повинно бути методологічно обґрунтованим.

Основним початковим критерієм є правило is-a: кожний об’єкт класу-нащадка повинен мати можливість розглядатися як об’єкт батьківського типу. Відношення володіння або включення повинні моделюватися клієнтським зв’язком.

Під час вибору між успадкуванням і клієнтським зв’язком важливі два додаткові критерії. Якщо відношення повинно змінюватися під час виконання, доцільнішим є клієнтський зв’язок. Якщо необхідна поліморфна підстановка об’єктів спеціалізованого типу замість заг.

Типовими помилками є використання успадкування для відношення has-a, створення надмірних класифікацій — таксономанія — та успадкування лише заради доступу до окремих готових операцій.

Правильне успадкування може використовуватися для спеціалізації, розширення, варіації поведінки, реалізації абстракцій, структурних властивостей і повторного використання реалізації. Кожний зв’язок повинен мати зрозуміле призначення.

Ієрархії класів не обов’язково створюються остаточно на початку проєкту. Вони можуть поступово вдосконалюватися через спеціалізацію, абстрагування та факторизацію. Головна мета полягає не в побудові формально красивої класифікації, а в отриманні системи класів, яка надає клієнтам прості, зрозумілі, стабільні й розширювані інтерфейси. Мейєр прямо наголошує, що найкраща ієрархія — це не абсолютна філософська класифікація, а структура, яка найкраще відповідає потреба.

Лекція 13

Тема: Як знайти класи

Мета: сформувати у студентів розуміння процесу ідентифікації класів під час об’єктно-орієнтованого аналізу та проєктування, навчити визначати потенційні класи за вимогами й моделлю предметної області, оцінювати доцільність їх виокремлення, виявляти помилкові абстракції та використовувати повторне використання і аналіз наявних рішень як джерела класів.

План лекції

1. Поняття ідентифікації класів.
2. Аналіз вимог як джерело потенційних класів.
3. Критерії виокремлення самостійного класу та класів в об’єктно-орієнтованій системі.
4. Ознаки помилкового визначення класів.
5. Додаткові джерела класів та повторне використання.
6. Послідовність формування системи класів.
7. Висновки.

Визначення класів є одним із центральних завдань об’єктно-орієнтованого проєктування. Від того, наскільки правильно виділено основні абстракції, залежить структура майбутньої системи, розподіл відповідальності між її компонентами, можливість повторного використання коду та подальшого розширення програмного забезпечення.

Пошук класів не зводиться до механічного перетворення понять предметної області на програмні сутності. Частина класів безпосередньо відповідає поняттям зовнішньої предметної області, інші виникають у процесі архітектурного проєктування, а ще частина потрібна для реалізації алгоритмів і структур даних. Тому знайти класи означає визначити **суттєві абстракції як у предметній області, так і в просторі програмного рішення.**

13.1 Поняття ідентифікації класів

Під час об’єктно-орієнтованого аналізу розробник намагається визначити не послідовність функцій, які повинна виконати система, а типи об’єктів, з якими система працюватиме. Для кожного такого типу необхідно встановити його стан, доступні операції, властивості та зв’язки з іншими об’єктами.

Клас повинен представляти **чітко визначену абстракцію**. Це означає, що його компоненти мають стосуватися одного логічного поняття, а об’єкти класу повинні мати власні характеристики та поведінку. Клас не є просто контейнером для

довільного набору даних або способом об’єднати декілька функцій під спільною назвою.

Пошук класів складається з двох взаємопов’язаних процесів:

- формування кандидатів на класи;
- перевірки та відхилення невдалих кандидатів.

Ці процеси виконуються не послідовно, а паралельно. У міру аналізу предметної області виникають нові кандидати, частина попередніх відкидається, декілька понять можуть об’єднуватися в одну абстракцію, а одна надто велика абстракція —ілька понять можуть об’єднуватися велика абстракція —ілька понять можуть об’єднуватися в одну абстракцію, а одна надто велика абстракція — розділятися на декілька класів.

Таким чином, завдання полягає не в тому, щоб отримати максимальну кількість класів. Необхідно побудувати систему **достатньої кількості змістовних і взаємопов’язаних абстракцій.**

Наприклад, під час проєктування інформаційної системи університету очевидними кандидатами можуть бути:

- *STUDENT;*
- *TEACHER;*
- *COURSE;*
- *GROUP;*
- *ASSESSMENT.*

Однак такі поняття, як «перевірити оцінку», «сформувати звіт» або «відобразити список», самі по собі ще не визначають класи. Спочатку необхідно з’ясувати, над якою абстракцією виконуються ці операції і чи існує для них окремий тип об’єктів.

Тому пошук класів є передусім **пошуком правильних абстракцій**, а не пошуком назв для програмних модулів.

13.2 Аналіз вимог як джерело потенційних класів

Документ із вимогами є природним початковим джерелом інформації про майбутню систему. У ньому описуються об’єкти предметної області, їхні властивості, дії користувачів та правила функціонування системи.

Відомий простий підхід пропонує знаходити в тексті вимог іменники й розглядати їх як потенційні класи. Наприклад, речення:

«Ліфт повинен закрити двері перед переміщенням на інший поверх»

містить поняття *ELEVATOR, DOOR і FLOOR.*

Проте граматичний аналіз може використовуватися лише як початкове джерело ідей. Структура програмної системи не повинна залежати від того, яким стилем

написанот можна сформулювати різними реченнями, а отже, ті самі поняття можуть виступати іменниками, дієсловами або взагалі не бути явно названими.

Наприклад, наявність слова «двері» у вимогах ще не означає необхідність класу *DOOR*. Якщо для системи керування ліфтом важливий лише стан дверей, достатньо компонента:

door_open: BOOLEAN

та операцій:

open_door

close_door

у класоцільний тоді, коли двері мають власний суттєвий стан і набір операцій: тип приводу, датчики, блокування, несправності, режими роботи тощо.

Схожа ситуація виникає з поняттям поверху. Якщо вся необхідна інформація про поверх зводиться до його номера, окремий *FLOOR* може бути зайвим — доверх має власні правила доступу, назву, призначення, зони безпеки або інші специфічні властивості, виникають підстави для окремого класу.

Отже, під час аналізу вимог доцільно звертати увагу не на граматичну форму слова, а на:

- поняття, які часто повторюються;
- поняття, для яких у вимогах наведено окремі визначення;
- спеціальні терміни предметної області;
- поняття, для яких описано власні властивості;
- об’єкти, над якими виконується декілька пов’язаних операцій;
- поняття, що беруть участь у багатьох взаємодіях системи.

Водночас деякі важливі класи можуть взагалі не бути прямо названими у вимогах. Наприклад, опис «після кожного переміщення ліфта створюється запис у базі даних», хоча слово «переміщення» у конкретному формулюванні могло бути використано як дієслово. Отже, зміст важливіший за граматичну форму.

13.3 Критерії виокремлення самостійного класу та класів в об’єктно-орієнтованій системі

Основним критерієм доцільності створення класу є наявність самостійної абстракції даних. Для потенційного поняття потрібно визначити, чи має воно достатньо власних операцій і властивостей, які є важливими саме для цієї програмної системи.

Під час перевірки кандидата корисно поставити такі запитання:

- Чи можна чітко пояснити, що представляє цей клас?
- Чи існуватимуть під час виконання об’єкти цього класу?
- Які властивості можна отримати від такого об’єкта?
- Які операції можна виконувати над ним?

- Чи має він стан, що змінюється?
- Чи існують правила, які повинні завжди виконуватися для його об’єктів?
- Чи не реал повинні завжди виконуватися для його об’єктів?
- Чи не реалізує вже інший клас усі необхідні властивості цього поняття?
- Чи є ця абстракція важливою для завдань конкретної системи?

Типовий добре сформований клас має декілька характеристик: він представляє зрозумілу абстракцію, має змістовну назву, визначає можливу множину об’єктів, надає запити для отримання їхніх властивостя його операцій зазвичай можна сформулювати обмеження, передумови, постумови та інваріанти. Ці ознаки не є абсолютною формальною перевіркою, але створюють практичну основу для оцінювання кандидата.

Важливо також враховувати контекст системи. Один і той самий об’єкт предметної області в одній системі може потребувати власного класу, а в іншій — ні.

Наприклад, поняття *FLOOR*:

- у простій системі керування ліфтом може представлятися числом;
- у системі контролю доступу може мати власний набір дозволів;
- у CAD-системі для проєктування будівель може мати геометрію, приміщення, високі характеристики.

Тобто не існує правила «кожному реальному об’єкту — окремий клас». Клас створюється лише тоді, коли відповідна абстракція має самостійну роль у конкретній програмній системі. Для узагальнення критеріїв доцільності створення класу використаємо таблицю 13.1.

Таблиця 13.1 – Основні критерії виокремлення самостійного класу

Критерій	**Характеристика**
Самостійна абстракція	Клас представляє одне чітко визначене поняття
Власний стан	Об’єкти мають значущі властивості, характерні саме для цієї абстракції
Власна поведінка	Над об’єктами визначено декілька пов’язаних операцій
Ідентичність об’єктів	Під час виконання існує потреба розрізняти окремі екземпляри
Семантичні правила	Для об’єктів можна визначити обмеження, передумови, постумови або інваріант
Самостійна відповідальність	Функціональність не належить природно до іншого вже визначеного класу
Значущість для системи	Поняття має практичне значення саме для задач розроблюваного програмного забезпечення

Таким чином, окрема функція стає підставою для нового класу не тому, що вона є складною або великою, а тоді, коли разом із пов'язаними даними та операціями вона вказує на наявність раніше невиокремлен класів в об'єктно-орієнтованій системі

Не всі класи походять безпосередньо з предметної області. У загальному випадку доцільно розрізняти **класи аналізу, класи проєктування та класи реалізації.**

Класи аналізу відображають суттєві поняття зовнішньої області, яку моделює програмне забезпечення. Наприклад:

- *EMPLOYEE* у кадровій системі;
- *AIRCRAFT* у системі керування повітряним рухом;
- *PARAGRAPH* у системі оброблення документів;
- *PART* у системі обліку деталей.

Важливо пам'ятати, що такі поняття не обов'язково мають бути матеріальними. Класом може бути й ласті, якщо воно має суттєві властивості та операції. Наприклад, *MARKET_TENDENCY, AUTHORIZATION, PRIORITY* або *SENIORITY_RULE* можуть бути не менш важливими абстракціями, ніж фізичні об'єкти.

Класи проєктування виникають у просторі програмного рішення та описують архітектурні поняття, які можуть не мати прямого аналога в зовнішньому світі. Прикладами є:

- *COMMAND;*
- *STATE;*
- *HISTORY;*
- *CONTROLLER;*

класи ітекладнішими для виявлення, оскільки вони не випливають безпосередньо з назв у вимогах. Вони виникають у результаті аналізу структури системи та пошуку способу правильно розподілити відповідальність.

Класи реалізації представляють структури й механізми, необхідні для ефективної реалізації алгоритмів:

ARRAY;

HASH_TABLE;

TREE.

Вони можуть бути ефективними або відкладеними. Наприклад, загальний клас QUEUE може задавати абстрактний інтерфейс черги, а його нащадки — конкретні способи реалізації.

Відмінності цих категорій наведено в таблиці 13.2.

Таблиця 13.2 – Основні категорії класів програмної системи

Категорія	**Джерело**	**Приклади**	**Основне призначення**

Класи аналізу	Предметна область	*STUDENT, DOCUMENT, ACCOUNT*	Моделювання суттєвих понять зовнішньої області
Класи проєктування	Архітектура програмного рішення	*COMMAND, STATE, CONTROLLER*	Організація взаємодії та відповідальності компонентів
Класи реалізації	Алгоритми та структури даних	*ARRAY, LIST, TREE, HASH_TABLE*	Забезпечення внутрішніх механізмів реалізації

Як видно з таблиці 13.2, обмеження пошуку лише предметною областю неминуче призведе до втрати значної кількості важливих класів. Особливо це стосується проєктних абстракцій, які часто визначають гнучкість усієї архітектури.

13.4 Ознаки помилкового визначення класів

Не кожний потенційний клас потрібно залишати в остаточній структурі системи. Існує ряд ознак, які повинні змусити розробника додатково перевірити правильність абстракції.

Першою небезпечною ознакою є ситуація, коли клас описується словами «цей клас виконує...». Наприклад: «клас друкує результат», «клас обробляє введення», «клас перевіряє дані». Це мої декомпозиції використовується функціональна, а так званий клас насправді є звичайною процедурою. Справжній клас зазвичай представляє певний тип об’єктів і надає набір пов’язаних послуг над ними.

Підозрілою є й назва класу у формі дієслова, наприклад:

- *PRINT*
- *PARSE*
- *VALIDATE*

Краще, щоб назва характеризувала саму абстракцTOR, *VALIDATION_RESULT*— залежно від того, що саме моделюється. Для класів, які представляють операції як об’єкти, використовують іменникові форми, наприклад *LINE_DELETION*, а не *DELETE_LINE*.

Іншим сигналом є ефективний клас з однією експортованою операцією. Він може виявитися лише процедурою, оформленою у вигляді класу. Винятком є ситбстракцією — наприклад, об’єкт-команда. Навіть у цьому випадку зазвичай з часом з’являється кілька операцій, наприклад виконання, скасування, повторне виконання або отримання характеристик команди.

Протилежна проблема — клас, який містить тільки дані та не має поведінки. Такий клас іноді є впросто структурою даних, атрибути якої логічніше включити до

іншого класу. Необхідно перевірити, чи існують специфічні операції над цією інформацією і чи справді вона утворює самостійну абстракцію.

Ще однією важливою ознстракцій в одному класі. Якщо частина компонентів описує одне поняття, а інша — зовсім інше, клас необхідно розділити. Усі компоненти класу повинні належати до однієї чітко визначеної абстракції.

Основні сигнали для повторної перевірки кандидата узагальнено в таблиці 13.3.

Таблиця 13.3 – Ознаки потенційно неправильно визначеного класу

Ознака	**Можлива проблема**
Назва класу є дієсловом	Замість класу виділено окрему функцію
Клас описується як «той, що виконує одну дію»	Модуль може бути процедурою, а не абстракцією
Ефективний клас має лише одну операцію	Можлива функціональна декомпозиція
Клас містить лише пасивні дані	Дані можуть бути атрибутами іншого класу
Компоненти класу належать до різних понять	В одному класі змішано декілька абстракцій
Клас майже нічого не додає до предка	Можлива зайва класифікація
Ієрархія спадкування визначається до самих класів	Класифікацію розпочато передчасно

Особливо небажано починати проєктування з побудови великої ієрархії успазуміти самі абстракції, і лише після цього визначати зв’язки між ними. Класифікація та уточнення класів можуть виконуватися ітеративно, але успадкування не повинно заміняти пошук змістовних класів.

13.5 Додаткові джерела класів та повторне використання

Вимоги є лише одним із джерел потенційних класів. Корисні абстракції можна отримати з попередніх програмних систем, наявної документації, розмов із фахівцями предметної області, структур даних, моделей баз даних, бібліотек і відомих архітектурних рішень.

Під час аналізу вже існуючої системи особливу увагу потрібно звертати на потоки даних між модулями. Якщо певна структура або набір даних передається через вможе означати, що в системі пропущено самостійну абстракцію. Замість постійного передавання однакового набору параметрів доцільно розглянути можливість представлення цієї інформації окремим класом.

Корисними джерелами є також:

- сутності ER-моделей;

- значущі структури та записи у старих прог
- структури даних, над якими працює декілька модулів;
- поняття, які постійно використовують експерти предметної області;
- перевірені архітектурні шаблони;
- класи з попередніх подібних проєктів.

Сценарії використання можуть допомогти перевірити, чи підтримує отримана модель необхідну поведінку системи. Однак визначати структуру класів лише за послідовностями дій користувача недоцільно: це створює рїана модель повинна передусім описувати стійкі абстракції, їхні операції та обмеження, а сценарії можуть використовуватися для перевірки повноти отриманої структури.

Особливо важливим джерелом класів є повторне використання. Перед створенням нового класу потрібно перевірити, чи не існує вжтній, корпоративній або сторонній бібліотеці. Використання готової абстракції зменшує обсяг нового коду та дозволяє спиратися на компонент, який уже використовувався і перевірявся в інших системах.

Наявний клас не обов’язково повинен повністю відповідати новій рібну абстракцію, але потребує спеціалізації, його можна адаптувати через успадкування, композицію або інші механізми повторного використання, не змінюючи без необхідності вже стабільний компонент.

Основні джерела ідей для класів наведено в таблиці 13.4.

Таблиця 13.4 – Основні джерела потенційних класів

Джерело	**Що необхідно шукати**
Вимоги до системи	Часто використовувані поняття, визначення, специфічні терміни
Спілкування з користувачами та експертами	Основні поняття й професійну термінологію предметної області
Існуючі програмні системи	Структури даних, важливі інформаційні об’єкти, потоки даних між модулями
ER-моделі та бази даних	Сутності, які можуть стати основою класів
Алгоритми та структури даних	Відомі абстракції для ефективної реалізації
Попередні проєкти	Перевірені проєктні рішення та архітектурні абстракції
Бібліотеки компонентів	Готові класи, які можна використати або адаптувати
Шаблони проєктування	Перевірені проєктні абстракції для типових задач

Перелік у таблиці 13.4 по обмежуватися одним документом із вимогами. Якісна об’єктна модель формується з урахуванням предметної області, наявного програмного досвіду та вже створених повторно використовуваних компонентів.

13.6 Послідовність формування системи класів

Практичний процес визначення класів доцільно виконувати ітеративно. Не потрібно очікувати, що всі правильні класи будуть знайдені під час першого аналізу. Частина абстракцій стає очевидною лише після побудови початкової структури або реалізації окремих компонентів.

На першому етапі збирають потенційні абстракції з різних джерел: вимог, предметної області, наявних систем, бібліотек, моделей даних та попереднього досвіду. На цьому етапі краще не відкидати кандидата лише тому, що його роль ще не повністю зрозуміла.

Далі кожний кандидат перевіряється з точки зору абстракції. Необхідно визначити:

- що саме представляє клас;
- які об’єкти є його екземплярами;
- який стан вони мають;
- які запити можна виконувати;
- які команди змінюють їхній стан;
- які правила та обмеження діють для цих об’єктів;
- чи не належать ці властивості вже іншій абстракції.

Після цього аналізуються зв’язки між кандидатами. Дубльовані поняття об’єднуються, надто великі класи розділяються, а пасивні набори даних за потреби переносяться до тих класів, відповідальності яких вони стосуються.

Наступним кроком є перевірка потоків інформації. Якщо один і той самий набір параметрів регулярно проходить через декілька класів, це є підставою перевірити, чи не приховується за ним самостійна абстракція. Якщо клас повинен знстан багатьох інших класів, це також може вказувати на неправильний розподіл відповідальності. Аналіз зв’язності модулів і передачі даних часто дозволяє знайти класи, які були пропущені на початку.

Лише після отримання достатньо зрозумілого набору класів доцільно остаточно визначати:

- клієнтські зв’язки;
- агрегацію та композицію;
- успадкування;
- узагальнені типи;
- видимість компонентів;
- контракти класів.

Завершальним етапом є повторна перевірка отриманої структури з погляду вимог і типових сценаріїв. Модель повинна підтримувати необхідну поведінку, але при цьому залишатися побудованою навколо стабільних абстракцій, а не навколо однієї конкретної послідовності дій.

Висновки

Отже, процес можна подати як послідовність: *пошук кандидатів → визначення абстракцій → відхилення слабких кандидатів → розподіл відповідальності → аналіз взаємодії → повторне використання → уточнення структури класів.овою*. Під час проєктування окремі кроки повторюються, а система класів поступово уточнюється. Саме тому ідентифікація класів розглядається як одночасний процес пропонування та відбор розглядається як одночасний процес пропонування та відбору абстракцій. Це є процесом визначення суттєвих абстракцій предметної області та програмного рішення. Класи не можна отримати простим механічним виділенням іменників із тексту вимог: такий підхід може одночасно створити зайві класи та пропустити важливі проєктні абстракції.

Основним критерієм створення класу є наявність самостійної абстракції з власними властивостями, поведінкою та семантичними правилами, які мають значення для конкретної системи. Один і той самий об’єкт зовнішнього світу залежно від задачі може бути окремим класом, простим значенням або часій системі доцільно розрізняти класи аналізу, що моделюють поняття предметної області, класи проєктування, що відображають архітектурні рішення, та класи реалізації, пов’язані зі структурами даних і алгоритмами. Найменш очевидними часто є саме проєктні класи.

Під час оцінювання кандидатів потрібно звертати увагу на небезпечні ознаки: дієслівні назви, клас з однією операцією, пасивний набір даних без власної поведінки, змішування кількох абстракцій в одному класі та передчасне створення ієрархії успадкування.

Джерелами класів можуть бути не лише вимоги, а й предметні експерти, попередні системи, ER-моделі, структури даних, потоки інформації, архітектурні шаблони та бібліотеки. Повторне використання вже наявних класів часто є ефективнішим за створення нової абстракції з нуля.

У підсумку правильна ідентифікація класів полягає не в максимальному поділі системи на об’єкти, а в знаходженні невеликого набору чітких, змістовних і альності яких добре визначені та які можуть незалежно розвиватися разом із програмною системою.

Лекція 14

Тема: Глобальні об’єкти і константи

Мета: сформувати у студентів розуміння призначення глобальних об’єктів і констант в об’єктно-орієнтованих програмних системах, розглянути способи безпечного спільного використання даних різними компонентами, принципи застосування символічних констант і одноразової ініціалізації, а також визначити ризики надмірного використання глобального стану.

План лекції

1. Глобальна інформація в об’єктно-орієнтованій системі.
2. Символічні та явні константи.
3. Константи класових типів і одноразові функції.
4. Спільні об’єкти та одноразова ініціалізація.
5. Рядкові константи та фіксовані набори значень.
6. Проблеми надмірного використання глобальних даних.
7. Практичні правила використання глобальних об’єктів і констант.
8. Висновки.

Об’єктно-орієнтоване програмне забезпечення будується з автономних класів і об’єктів, кожен з яких має власну відповідальність та приховує деталі своєї реалізації. Проте повністю ізольованими компоненти системи бути не можуть. У багатьох програмах існує інформація або об’єкти, до яких повинні звертатися різні частини системи: параметри середовища, спільний журнал, засоби введення-виведення, вікно повідомлень, доступ до бази даних або певні системні параметри.

Основне завдання полягає в тому, щоб надати спільний доступ до такої інформації, не руйнуючи модульність системи. Просте введення великої кількості глобальних змінних створює приховані залежності між компонентами та суперечить ідеї автономних програмних модулів. Тому глобальну інформацію доцільно представляти через чітко визначені об’єктні механізми — константи, спільні об’єкти та операції одноразової ініціалізації.

14.1 Глобальна інформація в об’єктно-орієнтованій системі

Глобальною можна вважати інформацію, яка потрібна не одному окремому об’єкту, а декільком незалежним компонентам програмної системи. Наприклад, різні класи можуть потребувати доступу до одного вікна системних повідомлень, спільного мережевого з’єднання або параметра, що визначає обсяг доступної пам’яті.

У традиційній програмі таку проблему часто вирішують за допомогою глобальної змінної, створеної на рівні головної програми. В об’єктно-орієнтованій архітектурі цей підхід є менш природним. Система складається з класів, які повинні

залишатися максимально автономними, а поняття єдиного центрального модуля, що володіє всіма глобальними ресурсами, суперечить децентралізованій структурі.

Не завжди правильним рішенням буде й передавання спільного об’єкта параметром у кожний метод, який його потребує. Якщо десятки компонентів використовують один ресурс, постійне передавання посилання призводить до ускладнення інтерфейсів. Крім того, для справді спільного ресурсу часто немає одного природного об’єкта-власника.

Важливо відрізняти **глобальний доступ** від **глобального змінного стану**. Сам факт того, що певне значення доступне багатьом класам, не є помилкою. Небезпека виникає тоді, коли багато незалежних компонентів можуть довільно змінювати цей стан і залежать від неявного порядку таких змін.

До типових видів глобальної інформації можна віднести:

- незмінні числові та символьні параметри;
- рядкові повідомлення;
- параметри, які визначаються один раз під час запуску системи;
- спільні об’єкти-сервіси;
- системні ресурси, ініціалізація яких повинна виконуватися лише один раз;
- фіксовані коди станів або результатів операцій.

Ці випадки потребують різних механізмів. Для простого значення достатньо символічної константи, для складного об’єкта може використовуватися одноразово створений спільний екземпляр, а для системної ініціалізації — процедура, яка гарантовано виконується лише один раз.

14.2 Символічні та явні константи

Константа — це значення, яке не повинно змінюватися в межах визначеного контексту. У програмному коді можна використовувати безпосередньо саме значення або надати йому змістовне ім’я.

Наприклад:

```
if attempts > 3 then
   ...
end
```

Число 3 у такому фрагменті не пояснює власного призначення. Читачеві необхідно додатково з’ясовувати, чому використано саме три спроби.

Краще визначити:

```
Max_attempts: INTEGER = 3
```

і використовувати:

```
if attempts > Max_attempts then
   ...
end
```

У першому випадку маємо **явну, або manifest, константу** — значення безпосередньо записане в алгоритмі. У другому — **символічну константу**, тобто значення, якому надано змістовне ім’я. Основними перевагами символічних констант є читабельність та можливість змінити значення в одному місці без пошуку всіх його використань у програмі.

Наприклад:

Max_connections: INTEGER = 100

Default_port: INTEGER = 8080

Pi: REAL = 3.1415926535

Backslash: CHARACTER = '\'

Такі оголошення одразу пояснюють семантику значень.

Основні відмінності наведено в таблиці 14.1.

Таблиця 14.1 – Порівняння явних і символічних констант

Вид	**Приклад**	**Особливість**
Явна числова константа	*100*	Значення безпосередньо записане в алгоритмі
Явна рядкова константа	*"File not found"*	Текст міститься безпосередньо в програмному коді
Символічна числова константа	*Max_connections*	Значення має змістовне ім’я
Символічна рядкова константа	*File_not_found*	Повідомлення централізовано визначене в одному місці
Динамічно обчислювана константа	*Available_memory*	Значення визначається під час виконання, але лише один раз

Для більшості значень, що мають предметне або системне значення, варто використовувати символічні константи. Це стосується максимальних розмірів, портів, кодів, часових інтервалів, коефіцієнтів, повідомлень і параметрів алгоритмів.

Водночас не потрібно перетворювати будь-яке число на окрему константу. Значення 0, 1, порожній рядок та інші природні елементи базових операцій часто є зрозумілими без додаткового іменування. Наприклад, перетворення кожної одиниці на One лише погіршило б читабельність.

Якщо клас потребує значної кількості взаємопов’язаних констант, це може бути ознакою окремої абстракції. У такій ситуації значення доцільно згрупувати в спеціальному класі. Наприклад, клас *EDITOR_CONSTANTS* може містити коди клавіш, а клас *ASCII* — характеристики набору символів.

14.3 Константи класових типів і одноразові функції

Для базових типів константу створити просто:

Pi: REAL = 3.1415926535

Але ситуація ускладнюється, коли постійне значення повинно бути об’єктом певного класу.

Нехай існує клас *COMPLEX*, що описує комплексні числа, і система часто використовує число і, тобто *(0, 1)*. Спроба описувати такий об’єкт безпосередньо через його поля порушує інформаційне приховування. Клієнт класу не повинен знати, чи зберігається комплексне число у декартовій, полярній або іншій внутрішній формі.

Звичайна функція могла б створювати необхідний об’єкт:

```
i: COMPLEX
do
   create Result.make_cartesian (0, 1)
end
```

Проте кожне звернення до і створювало б новий однаковий об’єкт.

Для таких ситуацій використовується **одноразова функція** *(once function)*:

```
i: COMPLEX
once
   create Result.make_cartesian (0, 1)
end
```

Під час першого виклику виконується тіло функції та створюється об’єкт. Усі наступні виклики повертають результат першого виконання без повторного створення об’єкта.

Це дозволяє отримати один спільний об’єкт, доступний багатьом компонентам.

Важливо розуміти різницю між **постійним посиланням і постійним об’єктом**. Одноразова функція гарантує, що клієнти отримуватимуть посилання на той самий об’єкт. Але сам об’єкт може залишатися змінюваним.

Наприклад, якщо *Message_window* повертає спільне вікно, всі компоненти працюють з одним екземпляром, однак його текст може змінюватися:

Message_window.put_text ("Помилка з'єднання")

Тому одноразова функція створює насамперед **спільний об’єкт**, а не автоматично незмінний об’єкт.

Ця різниця є принциповою. Якщо об’єкт має бути справді незмінним, це повинно забезпечуватися його інтерфейсом, контрактами або іншими обмеженнями, а не лише фактом одноразового створення.

14.4 Спільні об’єкти та одноразова ініціалізація

Одноразові операції корисні не лише для створення постійних об’єктів. Вони дозволяють організовувати ресурси, які повинні бути створені або налаштовані один раз, незалежно від того, який компонент звернувся до них першим.

Типовими прикладами є:

- спільне вікно системних повідомлень;
- доступ до системного журналу;
- параметри апаратного середовища;
- кількість доступних процесорів;
- обсяг пам’яті;
- початкове налаштування графічної підсистеми;
- інші ресурси, що ініціалізуються за потреби.

Параметр може бути невідомим під час компіляції, але після визначення залишатися незмінним протягом виконання програми.

Наприклад:

```
Available_memory: INTEGER
once
   Result := system_available_memory
end
```

Перший компонент, якому знадобиться значення Available_memory, виконає його обчислення. Наступні компоненти отримають уже готовий результат. Такий механізм дає змогу реалізувати **відкладену одноразову ініціалізацію** без спеціального центрального модуля запуску.

Подібний підхід застосовується й до процедур. Якщо певну операцію потрібно виконати лише один раз, використовується одноразова процедура.

Наприклад:

```
check_setup
once
   terminal_setup
end
```

Кожна операція графічної бібліотеки може викликати *check_setup*. Під час першого звернення буде виконано налаштування термінала, а всі наступні виклики нічого не робитимуть. Клієнт при цьому не повинен пам’ятати правило «перед першою графічною операцією викликати процедуру ініціалізації».

Такий підхід має дві важливі переваги. По-перше, він зменшує кількість правил, які повинен знати користувач класу або бібліотеки. По-друге, логіка ініціалізації залишається в тому модулі, якому вона належить, а не переноситься до зовнішнього керуючого коду.

Основні варіанти застосування наведено в таблиці 14.2.

Таблиця 14.2 – Основні механізми роботи зі спільними значеннями

Потреба	Механізм	Результат
Незмінне значення базового типу	Символічна константа	Одне зрозуміле ім’я для фіксованого значення
Значення, яке визначається під час виконання один раз	Одноразова функція	Динамічно обчислена константа
Один спільний складний об’єкт	Одноразова функція класового типу	Усі клієнти отримують той самий об’єкт
Одноразове системне налаштування	Одноразова процедура	Ініціалізація виконується лише за першої потреби
Група пов’язаних констант	Окремий клас констант	Централізоване та логічно організоване зберігання

Таким чином, глобальна інформація може залишатися доступною багатьом компонентам без перетворення системи на набір неконтрольованих глобальних змінних.

14.5 Рядкові константи та фіксовані набори значень

Рядки використовуються в повідомленнях, назвах, шаблонах, параметрах інтерфейсу та багатьох інших частинах програмного забезпечення.

Замість повторного використання:

print ("File not found")

доцільно визначити:

File_not_found: STRING = "File not found"

і далі використовувати:

print (File_not_found)

Це полегшує зміну повідомлень, централізоване редагування текстів та локалізацію програмного забезпечення. Якщо всі повідомлення розкидані безпосередньо по коду, переклад системи іншою мовою потребуватиме пошуку великої кількості рядкових літералів.

При цьому рядок є об’єктом класу *STRING*. Тому значення, записане як: *Message: STRING = "Syntax error"* логічно поводиться як спільний об’єкт. Якщо використовуваний тип рядка допускає зміну символів, необхідно враховувати, що спільне посилання саме по собі не гарантує незмінності об’єкта.

Інша задача виникає, коли атрибут може мати одну з невеликої кількості фіксованих альтернатив. Наприклад, результат операції читання може бути:

- успішним;
- помилкою відкриття;

- помилкою читання.

Для цього можна використовувати символічні коди:

- *Successful: INTEGER = 1*
- *Open_error: INTEGER = 2*
- *Read_error: INTEGER = 3*

або механізм *unique*:

Successful, Open_error, Read_error: INTEGER unique

У цьому випадку конкретні числові значення призначаються автоматично, а програмний код працює з їхніми змістовними назвами.

Такі коди виправдані лише тоді, коли набір можливих випадків справді є фіксованим. Якщо ж різні варіанти мають власну поведінку і система повинна регулярно розширюватися новими типами, доцільніше використовувати окремі класи, успадкування та динамічне зв'язування.

Наприклад, коди:

- *Circle*
- *Rectangle*
- *Square*

не повинні замінювати класи *CIRCLE, RECTANGLE і SQUARE,* якщо кожний тип фігури має власну реалізацію *draw, rotate, perimeter* тощо. Для відкритої класифікації система класів є гнучкішою за фіксований список кодів.

14.6 Проблеми надмірного використання глобальних даних

Глобальні об'єкти вирішують реальну проблему спільного доступу, але надмірне їх використання може зруйнувати переваги об'єктної декомпозиції.

Перша проблема — **приховані залежності**. Якщо клас використовує глобальний об'єкт, але ця залежність неочевидна з його інтерфейсу, складніше зрозуміти умови його роботи.

Друга проблема — **спільний змінний стан**. Якщо десятки компонентів можуть змінювати один глобальний об'єкт, результат роботи одного класу може залежати від дій іншого класу, який формально з ним не пов'язаний. Це ускладнює аналіз програми та пошук помилок.

Третя проблема — **порядок ініціалізації**. У традиційних системах глобальний ресурс часто потребує додаткового прапорця на зразок:

```
ready: BOOLEAN
і перевірок:
if not ready then
   initialize
   ready := True
end
```

Але сам *ready* також потребує коректної глобальної ініціалізації. Це породжує нові залежності й переносить відповідальність за початковий стан системи до централізованого коду.

Четверта проблема — **втрата автономності класів**. Якщо майже кожний компонент системи залежить від великого глобального об'єкта *SYSTEM_STATE*, клас стає важко повторно використовувати незалежно від конкретної програми.

П'ята проблема — **глобальний об'єкт як контейнер для всього**. Наприклад, створення одного класу *GLOBALS*, що містить налаштування, базу даних, повідомлення, користувача, журнали, мережеві ресурси та десятки інших значень, лише переносить проблему глобальних змінних у об'єктну форму. Формально дані стали атрибутами класу, але архітектурна проблема залишилася.

Тому спільні значення необхідно групувати за змістом. Параметри середовища можуть належати одному класу, текстові повідомлення — іншому, доступ до журналу — окремому сервісу. Кожна така група повинна представляти чітку абстракцію.

14.7 Практичні правила використання глобальних об'єктів і констант

Під час роботи з глобально доступними значеннями доцільно дотримуватися декількох правил.

1. Використовуйте символічні константи замість «магічних» значень.

Значення 30000 нічого не пояснює. Ім'я *Connection_timeout* одразу визначає його роль.

2. Константа повинна мати змістовне ім'я.

Ім'я повинно описувати призначення значення, а не дублювати його:

Max_attempts: INTEGER = 3

краще, ніж:

Three: INTEGER = 3

3. Не перетворюйте кожне просте значення на символічну константу.

Нульові елементи базових операцій та очевидні значення можуть залишатися безпосередньо в коді.

4. Пов'язані константи групуйте.

Якщо клас містить велику кількість параметрів одного призначення, варто розглянути окрему абстракцію для їх зберігання.

5. Для складного спільного об'єкта використовуйте контрольоване одноразове створення.

Один об'єкт не потрібно багаторазово створювати лише тому, що до нього звертаються різні клієнти.

6. Не плутайте спільність із незмінністю.

Те, що всі компоненти отримують той самий об'єкт, не означає, що його стан не може змінюватися.

7. Ініціалізуйте ресурс у тому модулі, який відповідає за нього.

Клієнт не повинен знати складний порядок службових викликів, необхідний для підготовки ресурсу.

8. Використовуйте одноразові процедури для операцій, що мають виконуватися лише один раз.

Це особливо корисно для автоматичної ініціалізації бібліотек і системних засобів.

9. Фіксований набір кодів використовуйте лише для справді фіксованих випадків.

Якщо кількість варіантів може зростати і кожний варіант має окрему поведінку, краще використовувати поліморфну систему класів.

10. Мінімізуйте глобальний змінний стан.

Глобальними повинні бути лише ті ресурси, для яких спільність є властивістю самої задачі, а не зручним способом уникнути правильного проєктування зв'язків між класами.

У результаті глобальний ресурс повинен залишатися винятком із загального принципу локальної відповідальності, а не перетворюватися на основний засіб передачі інформації між компонентами.

Висновки

У програмній системі іноді необхідно забезпечити доступ декількох незалежних класів до спільної інформації. До неї можуть належати константи, параметри середовища, системні ресурси та складні спільні об'єкти.

Для фіксованих значень доцільно використовувати символічні константи. Вони підвищують читабельність коду, усувають «магічні» числа та спрощують зміну параметрів. Значення, що визначаються лише під час виконання, але потім залишаються незмінними, можуть обчислюватися одноразово.

Для об'єктів класових типів важливо не порушувати інформаційне приховування спробами описувати константу через внутрішню структуру об'єкта. Один спільний екземпляр може створюватися під час першого звернення і надалі повторно використовуватися всіма клієнтами. При цьому необхідно пам'ятати, що постійне посилання не гарантує незмінності самого об'єкта.

Одноразова ініціалізація дозволяє прибрати з клієнтського коду зайві правила щодо порядку запуску компонентів. Ресурс може самостійно забезпечити своє початкове налаштування тоді, коли він уперше знадобиться.

Рядкові повідомлення та інші значення, які можуть змінюватися під час супроводу системи, також варто централізувати. Це полегшує редагування, локалізацію та підтримку програмного забезпечення.

Водночас глобальні об’єкти не повинні перетворюватися на універсальний засіб зв’язку між класами. Надмірний глобальний стан створює приховані залежності, послаблює автономність компонентів і робить систему складнішою для зміни та повторного використання. Правильний підхід полягає у використанні глобальних об’єктів **лише там, де сама природа ресурсу вимагає його спільності**, із чіткою відповідальністю, контрольованою ініціалізацією та мінімально необхідними можливостями зміни.

СПИСОК ВИКОРИСТАНОЇ ЛІТЕРАТУРИ

1. Куєвда Ю. В. Об’єктно-орієнтоване програмування. Курс лекцій : навчальний посібник. Київ : КПІ ім. Ігоря Сікорського, 2024. 164 с.
2. Порєв В. М. Об’єктно-орієнтоване програмування. Конспект лекцій : навчальний посібник. Київ : КПІ ім. Ігоря Сікорського, 2022. 271 с.
3. Мнушка О. В., Савченко В. М., Маций О. Б. Об’єктно-орієнтоване програмування мовою Python : навчальний посібник. Харків : ХНАДУ, 2021. 228 с.
4. Любченко Н. Ю., Соболь М. О., Пугачов Р. В., Подорожняк А. О., Івашко А. В. Основи ООП C++ в прикладах : навчально-методичний посібник. Ч. 1. Харків : НТУ «ХПІ», 2024. 223 с.
5. Назарчук І. В. Програмування та алгоритмічні мови : у 2 ч. Ч. 2. Програмування мовою C++. Вступ до об’єктно-орієнтованого програмування : підручник. Київ : КПІ ім. Ігоря Сікорського, 2024. 162 с.
6. Алхімова С. М. Об’єктно-орієнтоване програмування. Ч. 2. Об’єктно-орієнтований підхід до розробки програмного забезпечення : підручник. Київ : КПІ ім. Ігоря Сікорського, 2019. 194 с.
7. Коноваленко І. В., Марущак П. О. Платформа .NET та мова програмування C# 8.0 : навчальний посібник. Тернопіль : ФОП Паляниця В. А., 2020. 320 с.
8. Тихоход В. О., Гурін А. Л., Беспала О. М. Технології розробки програмного забезпечення. Курс лекцій : навчальний посібник. Київ : КПІ ім. Ігоря Сікорського, 2024. 230 с.
9. Pressman R. S., Maxim B. R. Software Engineering: A Practitioner’s Approach. 9th ed. New York : McGraw-Hill Education, 2020.
10. Budgen D. Software Design: Creating Solutions for Ill-Structured Problems. 3rd ed. Chapman & Hall/CRC, 2021. 364 p. Особливо доречне джерело для тем про модульність, об’єктне моделювання, класи, архітектуру та принципи проєктування.
11. Dingle A. Object-Oriented Design Choices. Chapman & Hall/CRC, 2021. 348 p. Праця безпосередньо розглядає вибір між успадкуванням, композицією, делегуванням, залежностями та іншими механізмами об’єктного проєктування.
12. Richards M., Ford N. Fundamentals of Software Architecture: An Engineering Approach. O’Reilly Media, 2020. 432 p.
13. Farley D. Modern Software Engineering: Doing What Works to Build Better Software Faster. Addison-Wesley Professional, 2021. 256 p.
14. Winters T., Manshreck T., Wright H. Software Engineering at Google: Lessons Learned from Programming Over Time. O’Reilly Media, 2020. 602 p.
15. Washizaki H. (ed.). Guide to the Software Engineering Body of Knowledge (SWEBOK Guide). Version 4.0. IEEE Computer Society, 2024. 413 p. Це особливо

сильна академічна позиція для методичного видання, оскільки SWEBOK систематизує загальновизнані знання з програмної інженерії, включно з проєктуванням, архітектурою, конструюванням і Agile.

16. Schwaber K., Sutherland J. The Scrum Guide. The Definitive Guide to Scrum: The Rules of the Game. November 2020. Офіційна чинна редакція Scrum Guide.

17. Coleman J., Vacanti D., Johnson C., Singh P., Wester J., Neverdal C., Firlit M., Gilb T., Tendon S. The Kanban Guide. May 2025. Офіційна редакція настанови з Kanban.

18. Meyer B. Object-Oriented Software Construction. 2nd ed. Upper Saddle River : Prentice Hall, 1997. — фундаментальне першоджерело, виняток із рекомендованого діапазону 2019–2025 років.