МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
ДНІПРОВСЬКИЙ ДЕРЖАВНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ

К.В. Яшина, К.М. Ялова, Н.М. Лимар

КОНСПЕКТ ЛЕКЦІЙ

з дисципліни **«Конструювання програмного забезпечення»**

для здобувачів вищої освіти

першого (бакалаврського) рівня

спеціальності **121 – «Інженерія програмного забезпечення»**

очної і заочної форм навчання

Затверджено редакційно–видавничою
секцією науково–методичної ради ДДТУ
«_21_» __02__ 2019 р., протокол № _2_

Кам’янське
2019

Конспект лекцій з дисципліни «Конструювання програмного забезпечення» для для здобувачів вищої освіти першого (бакалаврського) рівня спеціальності 121 -«Інженерія програмного забезпечення» очної і заочної форм навчання / Укладачі К.В. Яшина, К.М. Ялова, Н.М. Лимар – Кам'янське: ДДТУ, 2019 р. – 75 с.

Укладачі:
кандидат технічних наук, доцент **Яшина К.В.**
кандидат технічних наук, доцент **Ялова К.М.**
завідувач лабораторії **Лимар Н.М.**

Рецензент
канд.фіз-мат.наук, доцент **СтроєваВ.О.**

Відповідальний за випуск: зав. кафедри програмного забезпечення систем Шумейко О.О.

Затверджено на засіданні кафедри програмного забезпечення систем
(протокол № _2_ від _13.02._ 2019р.)

Коротка анотація видання. Містить основні теоретичні відомості про сучасні технології конструювання програмного забезпечення (способи розробки якісних інтерфейсів класів, побудови віконних додатків, принципів роботи з бібліотекою стандартних шаблонів), контрольні питання, що сприяє подальшому самостійному поглибленню знань та навичок студентів у конструюванні програмного забезпечення.

ЗМІСТ

ВСТУП

Дисципліна «Конструювання програмного забезпечення» викладаєтьсядля студентівспеціальності 121 - «Інженерія програмного забезпечення» у продовж одного навчального семестру. Зміст дисципліни поділено на два модуля. Тематика лекцій охоплює весь зміст дисципліни, за винятком тем, передбачених робочою навчальною програмою дисципліни для самостійного викладання.

Метою даного конспекту є формування у майбутніх фахівців з розробки та тестування програмного забезпечення сучасного рівня інформаційної та програмістської культури, оволодіння основними принципами конструювання програмного забезпечення; набуття практичних навичок самостійного написання якісного коду для розв’язання різноманітних задач у практичній діяльності.

Робочою мовою конструювання при засвоєнні дисципліни «Конструювання програмного забезпечення» обрано популярну, потужну мову високого рівня C++.

Основні теми дисципліни передбачають ознайомлення з сучасними технологіями конструювання програмного забезпечення, способами розробки якісних інтерфейсів класів, побудовою віконних додатків, принципами роботи з бібліотекою стандартних шаблонів.

Освоєння дисципліни дозволить майбутнім фахівцям забезпечити необхідний рівень вивчення і аналізу фахових дисциплін за рахунок ефективного використання сучасних принципів конструювання програмного забезпечення.

ТЕМА 1. Предмет і зміст дисципліни

Розробка програмного забезпечення (ПЗ) - непростий процес, який може включати безліч компонентів.Ось які складові розробки ПЗ визначили вчені за останні 25 років:

- визначення проблеми;
- вироблення вимог;
- створення плану конструювання;
- розробка архітектури ПЗ, або високорівневе проектування;
- детальне проектування;
- кодування та налагодження;
- блочне тестування;
- інтеграційне тестування;
- інтеграція;
- тестування системи;
- коригуючий супровід.

Термін конструювання програмного забезпечення (ПЗ) (softwareconstruction) описує детальне створення робочої програмної системи за допомогою комбінації кодування, верифікації (перевірки), модульного тестування (unittesting), інтеграційного тестування і налагодження. На рис.1.1 показано місце конструювання серед інших процесів розробки ПЗ.

Рисунок 1.1 - Місце конструювання серед інших процесів розробки ПЗ

Головними компонентами конструювання є кодування та налагодження, однак воно включає і детальне проектування, тестування по блокам, інтеграційне тестування та інші процеси.

Область «Конструювання ПЗ» (рис. 1.2) тісно пов’язана з іншими областями. Найбільш сильний зв'язок існує з проектуванням (Software Design) і тестуванням (Software Testing). Причиною цього є те, що сам по собі процес конструювання програмного забезпечення зачіпає важливі аспекти діяльності з проектування та тестування. Крім того, конструювання відштовхується від результатів проектування, а тестування (у будь-якій своїй формі) передбачає роботу з результатами конструювання. Досить складно визначити межі між проектуванням, конструюванням і тестуванням, так як всі вони пов'язані в єдиний комплекс процесів життєвого циклу і, залежно від обраної моделі життєвого циклу і застосовуваних методів (методології), такий поділ може виглядати по різному.

Хоча ряд операцій з проектування детального дизайну може відбуватися до стадії конструювання, великий обсяг такого роду проектних робіт відбувається паралельно з конструюванням або як його частина. Це й є суть зв'язку з областю знань «Проектування програмного забезпечення».

У свою чергу, протягом всієї діяльності з конструювання, інженери використовують модульне і інтеграційне тестування. Таким чином пов'язана дана область знань з «Тестуванням програмного забезпечення».

У процесі конструювання зазвичай створюється більша частина активів програмного проекту - конфігураційних елементів (configuration items). Тому в реальних проектах просто неможливо розглядати діяльність з конструювання у відриві від галузі знань «конфігураційне управління» (Software Configuration Management).

Так як конструювання неможливо без використання відповідного інструментарію і, ймовірно, дана діяльність є найбільш інструментально-насиченою, важливу роль у конструюванні відіграє область знань «Інструменти і методи програмної інженерії» (Software Engineering Tools and Methods).

Безумовно, питання забезпечення якості значимі для всіх галузей знань та етапів життєвого циклу. У той же час, код є основним результуючим елементом програмного проекту. Таким чином, явно присутній зв'язок обговорюваних питань з областю знань «Якість програмного забезпечення» (Software Quality).

З пов'язаних дисциплін програмної інженерії (Related Disciplines of Software Engineering) найбільш тісний зв'язок даної галузі знань існує з комп'ютерними науками (computer scince). Саме в них, зазвичай, розглядаються питання побудови та використання алгоритмів і практик кодування. Нарешті, конструювання стосується і управління проектами (project management), причому, в тій мірі, наскільки діяльність з управління конструюванням важлива для досягнення результатів конструювання.

Конструювання ПЗ - створення працюючого ПЗ з залученням методів верифікації, кодування і тестування компонентів. До інструментів конструювання ПЗ віднесені мови програмування і конструювання, а також програмні методи та інструментальні системи (компілятори, системи

управління базами даних, генератори звітів, системи управління версіями, конфігурацією, тестуванням та ін.) До формальних засобів опису процесу конструювання ПЗ (взаємозв'язків між людиною і комп'ютером з урахуванням середовища оточення) віднесені мови конструювання.

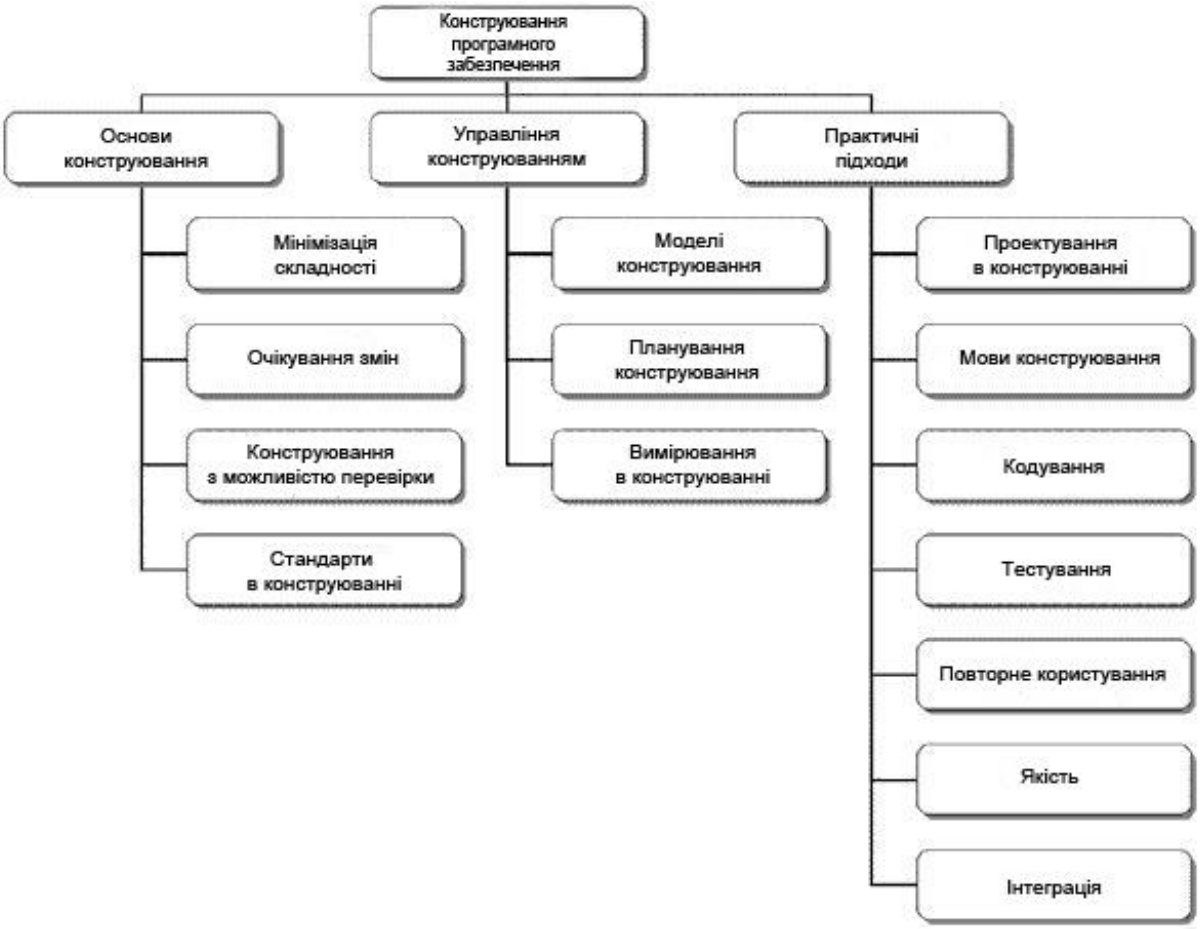

Рисунок 1.2 - Область знань «Конструювання програмного забезпечення»

Область знань «Конструювання ПЗ (Software Construction)» містить наступні розділи:

- зниженняскладності (Reduction in Complexity),
- попередженнявідхиленьвідстилю (Anticipation of Diversity),
- структуризаціядляперевірок (Structuring for Validation),
- використаннязовнішніхстандартів (Use of External Standards).

Основу даної області становлять завдання зниження складності конструювання програмного продукту, попередження відхилень від стилю (лінгвістичного, формального, візуального та ін.), яке забезпечується застосуванням найбільш відповідних стилів конструювання, структуризації ПЗ і використання зовнішніх стандартів.

Лінгвістичний стиль заснований на використанні словесних інструкцій і виразів для перегляду окремих елементів (конструкцій) програм. Він використовується при конструюванні нескладних конструкцій і може бути

приведений до виду традиційних функцій і процедур, логічному і функціональному їх програмуванні та ін.

Формальний стиль використовується для точного, однозначного і формального визначення компонентів системи. У результаті його застосування забезпечується конструювання складних систем з мінімальною кількістю помилок, які можуть виникнути у зв'язку з неоднозначністю визначень або узагальнень при конструюванні ПЗ неформальними методами.

Візуальний стиль є найбільш універсальним стилем конструювання ПЗ. Він дозволяє розробникам проекту представляти в наочному вигляді складні програмні конструкції.

Наприклад, графічний інтерфейс користувача звільняє розробника від підбору необхідних координат і властивостей об'єктів інтерфейсу. Візуальна мова проектування UML надає розробнику набір зручних діаграм для завдання статичної та динамічної структури ПЗ.

При застосуванні візуального стилю конструювання створюється текстовий і діаграмний опис структури ПЗ, який виводиться на екран дисплею не тільки для їх розгляду, але і корегування.

У процесі конструювання повинні використовуватися зовнішні стандарти (Ада 95, C++ і ін.), мов опису даних (XML, SQL та ін.), засобів комунікації (COM, CORBA та ін.), інтерфейсів компонентів (POSIX, IDL, APL), сценаріїв UML та ін.

Управління конструюванням базується на моделях конструювання, плануванні та внесенні змін.

Моделі конструювання містять набір операцій, послідовність дій і результати. Види моделей визначаються стандартом життєвого циклу, методологіями та практиками. Основні стандарти орієнтовані на екстремальне програмування і RUP.

Планування полягає у визначенні порядку створення компонентів і методів забезпечення якості. Вимірювання в конструюванні орієнтоване на кількісну оцінку обсягу коду, ступеню використання програмних інформаційних комплексів, ймовірності появи дефектів і кількісних показників якості ПЗ.

Внесення змін проводиться з метою збереження функціональної цілісності системи та рефакторінга коду на основі проведеного метричного аналізу необхідності виконання змін в ПЗ, яке конструюється.

Тестування в конструюванні. Найбільш поширені дві форми тестування створеногокоду - модульне і інтеграційне.Види тестування описані в спеціальнійгалузі знань.При цьому використовуються два стандарти (IEEE 829-1996 іIEEE 1008-1987)тестування елементів ПЗ і документації.Забезпечення якості конструювання базується не тільки на тестуванні й налагодженні окремих програм, а і на переглядах, інспектуванні, аналізі і оцінках результатів тестування.

Таким чином, розглянуті механізми конструювання дозволяють розробнику проекту прийняти рішення про використання методів

конструювання або проектування.Найбільш сучасним вважається метод моделювання UML.

Наведемо деякі конкретні завдання, пов'язані з конструюванням:

- перевірка виконання умов, необхідних для успішного конструювання;
- визначення способів подальшого тестування коду;
- проектування і написання класів та методів;
- створення і присвоєння імен змінним й іменованим константам;
- вибір керуючих структур і організація блоків команд;
- тестування по блокам, інтеграційне тестування і налагодження власного коду;
- взаємний огляд коду і низькорівневих програмних структур членами групи;
- «шліфовка» коду шляхом його ретельного форматування і коментування;
- інтеграція програмних компонентів, створених окремо;
- оптимізація коду, спрямована на підвищення його швидкодії, і зниження ступеня використання ресурсів.

Чому конструювання ПЗ так важливо?

Конструювання - крупна частина процесу розробки ПЗ. Залежно від розміру проекту на конструювання зазвичай йде 30-80% загального часу роботи.Все, що займає так багато часу роботи над проектом, неминуче впливає на його успішність.

Конструювання займає центральне місце в процесі розробки ПЗ. Вимоги до додатка і його архітектура розробляються до етапу конструювання, щоб гарантувати його ефективність.Тестування системи (в строгому сенсі незалежного тестування) виконується після конструювання і необхідне для перевірки його правильності.Конструювання - центр процесу розробки ПЗ.

Підвищена увага до конструювання може значно збільшити продуктивність праці. У своєму класичному дослідженні Секман, Еріксон та Грант показали, що продуктивність праці окремих програмістів під час конструювання змінюється в 10-20 разів (Sackman, Erikson, andGrant, 1968).З тих пір ці дані були підтверджені іншими дослідженнями (Curtis, 1981; Mills, 1983; Curtisetal., 1986; Card, 1987; ValettandMcGarry, 1989; DeMarcoandLister, 1999 №; Boehmetal., 2000).

Результат конструювання - вихідний код - часто є єдиним вірним описом програми Найчастіше єдиним видом доступної програмістам документації є сам вихідний код.Специфікації вимог і проектна документація можуть застаріти, але вихідний код актуальний завжди, тому він повинен бути максимально якісним. Послідовне застосування методів поліпшення вихідного коду - ось що відрізняє детальні, коректні і тому інформативні програми.

Конструювання - єдиний процес, який виконується в усіх випадках. Ідеальний програмний проект до початку конструювання проходить стадії ретельної вироблення вимог і проектування архітектури.Після конструювання в ідеалі має бути виконане вичерпне, статистично контрольоване тестування

системи.Однак, у реальних проектах нашого недосконалого світу розробники часто пропускають етапи вироблення вимог і проектування, починаючи прямо з конструювання програми.Тестування також часто випадає з розкладу через величезного числа помилок і браку часу.Але яким би терміновим чи погано спланованим не був проект, куди без конструювання подітися?Такимчином, підвищення ефективності конструювання ПЗ дозволяє оптимізувати будь-який проект, яким би недосконалим він не був.

КОНТРОЛЬНІ ПИТАННЯ:

1. Що описує термін «конструювання програмного забезпечення»?
2. У чому полягає зв'язок конструювання з проектуванням та тестуванням?
3. З чого складається область знань «Конструювання програмного забезпечення»?
4. Які існують стилі конструювання ПЗ? Чим характеризується кожен стиль?
5. Наведіть конкретні завдання, пов'язані з конструюванням. Чому конструювання ПЗ так важливо?

ТЕМА 2. Планування конструювання програмного забезпечення

Розглянемо етапи підготовки до конструювання ПЗ. Як і в будівництві, кінцевий успіх програмного проекту багато в чому визначається до початку конструювання. Якщо фундамент ненадійний або планування виконано недбало, на етапі конструювання ви в кращому випадку зможете тільки звести шкоду до мінімуму.

Важливість виконання попередніх умов

Спільною рисою всіх програмістів, що створюють високоякісне ПЗ, є використання високоякісних методів, що ставлять наголос на якості ПЗ на самому початку, середині і наприкінці проекту.

Якщо ви підкреслюєте якість в кінці проекту, це припадає на етап тестування системи.

Якщо ви приділяєте підвищену увагу якості всередині роботи над проектом, ви підкреслюєте методи конструювання.

Якщо ви підкреслюєте якість на початку проекту, ви якісно виконуєте планування, визначення вимог і проектування.

Конструювання - середній етап роботи, тому до початку конструювання успіх проекту вже частково зумовлений. І все ж під час конструювання ви хоча б повинні бути в змозі визначити, наскільки вдалою є ваша ситуація, і повернутися назад, якщо це потрібно.

Підготовка до проекту - одна з головних умов ефективного програмування, і це логічно. Обсяг планування залежить від масштабу проекту. З управлінської точки зору, планування передбачає визначення термінів, числа людей і комп'ютерів, необхідних для виконання робіт. З технічної - планування

це одержання представлення про створювану систему, що дозволяє не витратити гроші на створення невірної системи.

Дослідження останніх 25 років переконливо довели ефективність правильного виконання проектів з першого кроку і вартість внесення змін, яких можна було уникнути.

Вчені з компаній Hewlett-Packard, IBM, HughesAircraft, TRW та інших організацій виявили, що виправлення помилки до початку конструювання коштує в 10-100 разів дешевше, ніж її усунення в кінці роботи над проектом, під час тестування додатка або після його випуску (Fagan, 1976; Humphrey, Snyder, andWillis, 1991; Leffingwell 1997; Willisetal., 1998; Grady, 1999; Shulletal., 2002; BoehmandTurner, 2004).

Дані про відносну вартість виправлення дефектів в залежності від етапів їх внесення і виявлення (рис. 2.1):

	Час на виявлення дефекту				
Час на внесення дефекту	**Формулювання вимог**	**Проектування архітектури**	**Конструювання**	**Тестування системи**	**Після випуску ПЗ**
Формулювання вимог	1	3	5–10	10	10–100
Проектування архітектури	—	1	10	15	25–100
Конструювання	—	—	1	10	10–25

Рисунок 2.1 -Дані про відносну вартість виправлення дефектів в залежності від етапів їх внесення і виявлення

Джерело:адаптованозробіт «Design and Code Inspections to Reduce Errors in Program Development» (Fagan 1976), «Software Defect Removal» (Dunn, 1984), «Software Process Improvement at Hughes Aircraft» (Humphrey, Snyder, and Willis, 1991), «Calculating the Return on Investment from More Effective Requirements Management» (Leffingwell, 1997), «Hughes Aircraft's Widespread Deployment of a Continuously Improving Software Process» (Willis et al., 1998), «An Economic Release Decision Model: Insights into Software Project Management »(Grady, 1999),« What We Have Learned About Fighting Defects »(Shull et al., 2002) і« Balancing Agility and Discipline: A Guide for the Perplexed »(Boehm and TUrner, 2004).

Ці дані говорять, наприклад, про те, що дефект архітектури, виправлення якого при проектуванні архітектури обходиться в 1000 $, може під час тестуванні системи вилитися в 15 000 $ (рис. 2.2).

У більшості проектів основна частина зусиль щодо виправлення дефектів все ще припадає на праву частину (рис. 2.2), а значить, на налагодження і перероблення роботи йде близько 50% часу типового циклу розробки ПЗ (Mills, 1983; Boehm, 1987; CooperandMullen, 1993; Fishman, 1996; Haley, 1996; Wheeler, Brykczynski, andMeeson , 1996; Jones, 1998; Shulletal., 2002; Wiegers, 2002). У

десятках компаній було виявлено, що політика раннього виправлення дефектів може в два і більше разів знизити фінансові та часові витрати на розробку ПЗ (McConnell, 2013). Це дуже вагомий аргумент на користь раннього знаходження та вирішення проблем.

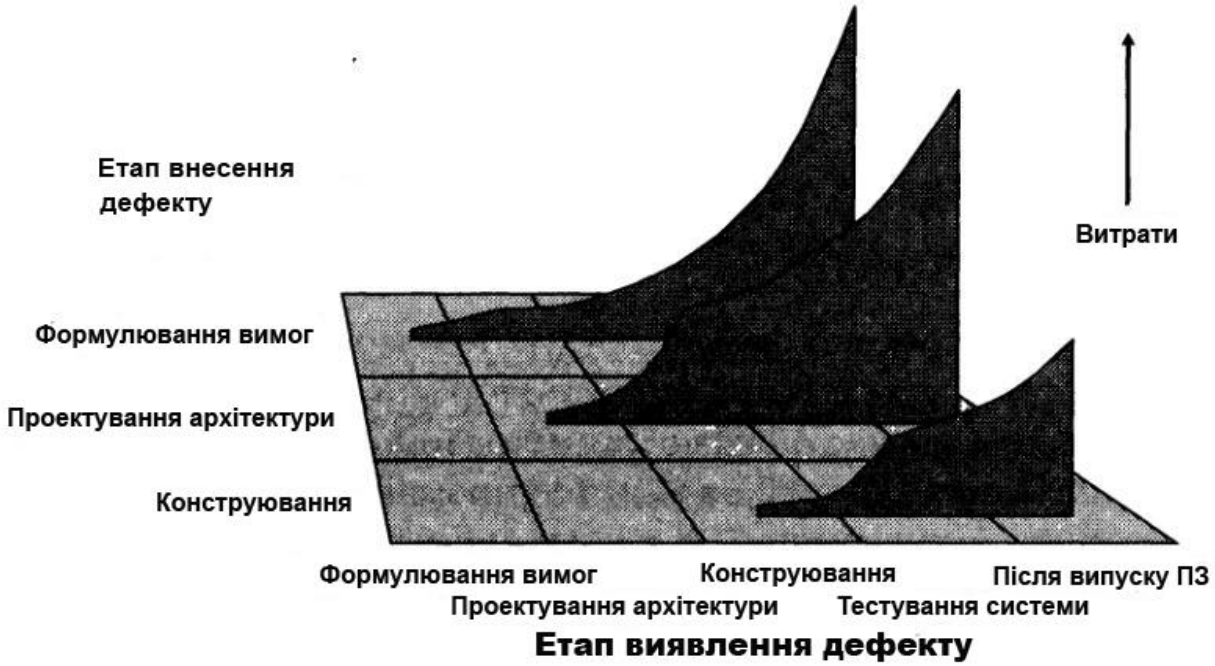

Рисунок 2.2 - Вартість виправлення дефекту в залежності від етапу його виявлення

Визначення типу програмного забезпечення

Узагальнюючи 20 років досліджень розробки ПЗ, Кейперс Джонс, керівник дослідницьких робіт у компанії Software Productivity Research, заявив, що він і його колеги стикалися з 40 різними методами збору вимог, 50 варіантами проектування ПЗ і 30 видами тестування, що застосовувалися в проектах, які реалізуються більш ніж на 700 мовах програмування (Jones, 2003).

Різні типи проектів закликають до різних сполученням підготовки та конструювання. Кожен проект унікальний, проте зазвичай проекти підпадають під загальні стилі розробки. Ось три найпопулярніші типи проектів, а також оптимальні в більшості випадків методи роботи над ними (рис. 2.3).

Працюючи над реальними проектами, ви зіткнетесь з незліченними варіаціями на три теми, зазначені в таблиці, однак, можна зробити і деякі загальні висновки. При розробці бізнес-систем переважно використовувати високоітеративні підходи, при яких планування, формулювання вимог і проектування архітектури перемежовуються з конструюванням, тестуванням системи та гарантією якості. Системи, від яких залежить життя людей, вимагають більш послідовних підходів, оскільки стабільність вимог - одна з умов найвищої надійності системи.

Вплив ітеративних підходів на попередні умови. Дехто стверджує, що при використанні ітеративних методів не потрібно займатися формулюванням попередніх умов, але ця точка зору хибна. Ітеративні підходи послаблюють недоліки неадекватної підготовки, але не усувають їх. При ітеративному

підході дефекти виправляються під час розвитку проекту, що дозволяє знизити загальні витрати.

	Тип ПЗ		
	Бізнес-системи	**Системи цільового призначення**	**Вбудовані системи, від яких залежить життя людей**
Типові програми	Інтернет-сайти. Сайти в інтрамережах. Системи управління матеріально-технічним постачанням. Ігри. Система управління інформацією. Система виплати заробітної плати.	Вбудоване ПЗ. Ігри. Інтрнет-сайти. Пакетне ПЗ. Програмні інструменти. Web-сервіси.	Авіаційне ПЗ. Вбудоване ПЗ. ПЗ для медичних пристроїв. Операційні системи. Пакетне ПЗ.
Моделі життєвого циклу	Гнучка розробка (екстремальне програмування, методологія Scrum, розробка на основі тимчасових вікон і т.д.). Еволюційне прототипування.	Поетапне постачання. Еволюційне постачання. Спіральна розробка.	Поетапне постачання. Еволюційне постачання. Спіральна розробка.
Планування і управління	Інкрементне планування проекту. Планування тестування і гарантії якості за необхідністю. Неформальний контроль за змінами.	Базове завчасне планування. Базове планування тестування. Планування тестування і гарантії якості за необхідністю. Формальний контроль за змінами.	Вичерпне завчасне планування. Вичерпне планування тестування. Вичерпне планування гарантії якості. Ретельний контроль за змінами.
Формулювання вимог	Неформальна специфікація вимог	Напівформальна специфікація вимог. Огляди вимог у міру потреби.	Формальна специфікація вимог. Формальні інспекції вимог.
	Бізнес-системи	**Призначення**	**Життя людей**
Проектування	Комбінація проектування і кодування.	Проектування архітектури. Неформальне детальне проектування. Огляди проекту за необхідності	Проектування архітектури. Формальні інспекції архітектури. Формальне детальне проектування. Формальні інспекції детального проекту.
Конструювання	Парне або індивідуальне програмування. Неформальна процедура реєстрації.	Парне або індивідуальне програмування. Неформальна процедура реєстрації коду. Огляди коду за необхідності.	Парне або індивідуальне програмування. Формальна процедура реєстрації коду. Формальні інспекції коду.
Тестування і гарантія якості	Розробники тестують власний код. Попередня розробка тестів. Тестування окремої групи проводиться в невеликому обсязі або не проводиться взагалі.	Розробники тестують власний код. Попередня розробка тестів. Окрема група тестування.	Розробники тестують власний код. Попередня розробка тестів. Окрема група тестування. Окрема група гарантії якості.
Впровадження програми	Неформальна процедура впровадження.	Формальна процедура впровадження.	Формальна процедура впровадження.

Рисунок 2.3 - Три найпопулярніші типи проектів

Ітеративний проект із скороченою програмою виконання попередніх умов або без неї відрізняється від аналогічного послідовного проекту двома

аспектами. По-перше, при ітеративному підході витрати на виправлення дефектів зазвичай нижче, тому що дефекти виявляються раніше. І все ж це відбувається в кінці кожної ітерації, і виправлення дефектів вимагає повторного проектування, кодування і тестування фрагментів ПЗ, що робить витрати більшими, ніж вони могли б бути (рис. 2.4).

	Підхід 1: послідовний підхід без виконання виконання попередніх умов		Підхід 2: ітеративний підхід без виконання попередніх умов	
Рівень завершеності проекту	**Витрати на роботу**	**Витрати на виправлення дефектів**	**Витрати на роботу**	**Витрати на виправлення дефектів**
20%	$100 000	$0	$100 000	$75 000
40%	$100 000	$0	$100 000	$75 000
60%	$100 000	$0	$100 000	$75 000
80%	$100 000	$0	$100 000	$75 000
100%	$100 000	$0	$100 000	$75 000
Витрати на виправлення дефектів в кінці проекту	$0	$500 000	$0	$0
СУМА	$500 000	$500 000	$500 000	$375 000
ЗАГАЛЬНА СУМА	$1 000 000		$875 000	

Рисунок 2.4 – Витрати на розробку проекту при послідовному та ітеративному підходах без виконання попередніх умов

По-друге, при ітеративних підходах витрати розподіляються по всьому проекту, а не групуються в його кінці. Зрештою і при ітеративних, і при послідовних підходах загальна сума витрат виявиться схожою, але в першому випадку вона не здаватиметься настільки великої, бо буде сплачена по частинах.

Адекватна увага до виконання попередніх умов дозволяє знизити витрати незалежно від типу підходу, що використовується (рис. 2.5). Ітеративні підходи зазвичай за багатьма параметрами краще послідовних, однак ітеративний підхід, при якому нехтують попередніми умовами, може в підсумку виявитися набагато дорожчим, ніж належним чином підготовлений послідовний проект.

	Підхід 3: послідовний підхід з виконанням попередніх умов		Підхід 4: ітеративний підхід з виконанням попередніх умов	
Рівень завершеності проекту	**Витрати на роботу**	**Витрати на виправлення дефектів**	**Витрати на роботу**	**Витрати на виправлення дефектів**
20%	$100 000	$20 000	$100 000	$10 000
40%	$100 000	$20 000	$100 000	$10 000
60%	$100 000	$20 000	$100 000	$10 000
80%	$100 000	$20 000	$100 000	$10 000
100%	$100 000	$20 000	$100 000	$10 000
Витрати на виправлення дефектів в кінці проекту	$0	$0	$0	$0
СУМА	$500 000	$100 000	$500 000	$50 000
ЗАГАЛЬНА СУМА	$600 000		$550 000	

Рисунок 2.5 – Витрати на розробку проекту при послідовному та ітеративному підходах з виконанням попередніх умов

Попередні умови, пов'язані з визначенням проблеми

Перша попередня умова, яку потрібно виконати перед конструюванням, - ясне формулювання проблеми, яку система повинна вирішувати. Це ще іноді називають «баченням продукції», «формулюванням точки зору», «формулюванням завдання» або «визначенням продукції». Визначення проблеми передує виробленню детальних вимог, яке є більш глибоким дослідженням проблеми (рис. 2.6).

Рисунок 2.6 - Визначення проблеми

Проблему слід формулювати мовою, зрозумілою користувачеві, а сама проблема повинна бути описана з користувацької точки зору.

Попередні умови, пов'язані з формулюванням вимог

Вимоги докладно описують, що повинна робити програмна система, а їх формулювання - перший крок до вирішення проблеми. Формулювання вимог також відоме як «розробка вимог», «аналіз вимог», «аналіз», «визначення вимог», «специфікація», «функціональна специфікація».

Важливість явного набору вимог пояснюється кількома причинами:

1. Явні вимоги допомагають гарантувати, що функціональність системи визначається користувачем, а не програмістом.

2. Увага до вимог допомагає звести до мінімуму зміни системи після початку розробки. Виявивши при кодуванні помилку в коді, ви зміните кілька рядків, і робота продовжиться. Якщо ж під час кодування ви знайдете помилку у вимогах, доведеться змінити проект програми, щоб він відповідав зміненим вимогам. Можливо, при цьому доведеться відмовитися від частини старого проекту, а оскільки відповідно до неї вже написаний деякий код, на реалізацію нового проекту піде більше часу, ніж могло б. Ви також повинні будете відмовитися від коду і тестів, на які вплинула зміна вимог, і написати їх наново. Навіть код, що залишився недоторканим, потрібно буде заново протестувати для гарантії того, що зміна не призвела до появи нових помилок.

Адекватне визначення вимог - одна з найважливіших умов успіху проекту, можливо, навіть більш важливе, ніж використання ефективних методів конструювання.

Дослідження, проведені в IBM і інших компаніях, показали, що при реалізації середнього проекту вимоги під час розробки змінюються приблизно на 25% (Boehni, 1981; Jones, 1994; Jones, 2000), на що припадає 70-85% обсягу повторної роботи над типовим проектом (Leffingwell, 1997; Wiegers, 2003).

Наступні дії дозволяють максимально легко перенести зміни вимог під час конструювання:

1. Оцініть якість вимог за допомогою контрольного списку.
2. Переконайтесь, що всім відома ціна зміни вимог.
3. Задайте процедуру контролю змін.
4. Використовуйте ті підходи до розробки, які адаптуються до змін.
5. Пам'ятайте про бізнес-моделі проекту.

Попередні умови, пов'язані з розробкою архітектури

Архітектура - це високорівнева частина проекту програми, каркас, що складається з деталей проекту (Buschman et al, 1996; Fowler, 2002; Bass Clements, Kazman 2003; Clements et al., 2003). Архітектуру також називають «архітектурою системи», «високорівневим проектом» і «проектом високого рівня».

Якість архітектури визначає концептуальну цілісність системи, яка в свою чергу визначає підсумкову якість системи. Якісна архітектура надає структуру, потрібну для підтримки концептуальної цілісності в масштабі системи. Вона надає програмістам керівництво, рівень детальності якого відповідає їх навичкам і виконуваній роботі. Вона дозволяє розділити роботу на частини, над якими окремі розробники і групи можуть працювати незалежно.

Якісна архітектура полегшує конструювання. Погана архітектура робить його майже неможливим.

Внесення змін в архітектуру при конструюванні і на подальших етапах коштує недешево. Час, необхідний для виправлення помилки в архітектурі ПЗ, дорівнює часу, потрібному для виправлення помилки у вимогах, тобто перевищує часові витрати на виправлення помилки в коді (Basili and Perricone, 1984; Willis, 1998). Зміни архітектури схожі на зміни вимог ще й тим, що невеликі зміни можуть мати великі наслідки. Чим би не були обумовлені зміни архітектури - виправленням помилок або внесенням покращень, - чим раніше ви усвідомлюєте їх необхідність, тим краще.

Типові компоненти архітектури:

1. Організація програми. У першу чергу архітектура повинна містити загальний опис системи. Архітектура має містити підтвердження того, що при її розробці були розглянуті альтернативні варіанти, і обґрунтовувати вибір остаточної організації системи. Архітектура має визначати основні компоненти програми. Залежно від розміру програми її компонентами можуть бути окремі класи або підсистеми, що складаються з декількох класів. Кожен компонент є класом або набором класів / методів, які в сукупності реалізують високорівневі функції програми, такі як: взаємодія з користувачем, відображення Web-

сторінок, інтерпретація команд, інкапсуляція бізнес-правил або доступ до даних. За кожну функцію програми, зазначену у вимогах, повинен відповідати хоча б один компонент. Якщо функцію реалізують декілька компонентів, вони повинні співпрацювати, а не конфліктувати.

Архітектура має чітко визначати відповідальність кожного компонента. Компонент повинен мати одну область відповідальності і якомога менше знати про області відповідальності інших компонентів.

Архітектура повинна ясно визначати правила комунікації для кожного компонента. Вона повинна описувати, які інші компоненти даний компонент може використовувати безпосередньо, які побічно, а які взагалі не повинен використовувати.

2. Основні класи. Архітектура має визначати основні класи, додатки, їх галузі відповідальності та механізми взаємодії з іншими класами. Вона повинна описувати ієрархії класів, зміни станів і час існування об'єктів. Якщо система досить велика, архітектура повинна описувати організацію класів у підсистеми.

При цьому не всі класи системи потрібно описувати в специфікації архітектури. Орієнтуйтеся на правило 80/20: описуйте 20% класів, якими на 80% визначається поведінка системи (Jacobsen, Booch, and Rumbaugh, 1999; Kruchten, 2000).

3. Організація даних. Архітектура повинна описувати основні види формату файлів і таблиць. Вона повинна описувати розглянуті альтернативи і обґрунтовувати підсумкові варіанти. Якщо програма використовує список ідентифікаторів клієнтів і розробники архітектури вирішили реалізувати його за допомогою списку з послідовним доступом, в документації має бути вказано, чому цей вид списку краще, ніж список з довільним доступом, стек або хеш-таблиця. Ця інформація надасть вам неоціненну допомогу під час конструювання й супроводу програми, підказавши, чим керувалися розробники архітектури. Без неї ви будете почувати себе глядачем, який дивиться іноземний фільм без субтитрів.

Прямий доступ до даних звичайно треба надавати тільки одній підсистемі або класу; винятки можливі при використанні класів або методів доступу, що забезпечують доступ до даних контрольованим абстрактним чином.

Архітектура має визначати високорівневу організацію та утримання всіх баз даних (БД), що використовуються.

4. Бізнес-правила. Архітектура, залежна від специфічних бізнес-правил. Вона повинна їх визначати і описувати їх вплив на проект системи. Візьмемо для прикладу бізнес-правило, згідно з яким інформація про клієнтів повинна застарівати не більше ніж на 30 секунд. В даному випадку в специфікації архітектури має бути зазначено, як це правило вплинуло на вибір методу забезпечення актуальності даних та їх синхронізації.

5. Інтерфейс користувача. Інтерфейс користувача (GUI) часто проектується на етапі формулювання вимог. Якщо це не так, його слід визначити на етапі розробки архітектури. Архітектура повинна описувати головні елементи формату Web-сторінок, GUI, інтерфейс командного рядка і т.д.

Комфортність GUI може в підсумку визначити популярність або провал програми.

Архітектура має бути модульною, щоб GUI можна було змінити, не торкнувшись бізнес-правил і модулів програми, що відповідають за виведення даних. Наприклад, архітектура повинна забезпечувати можливість порівняно легкої заміни групи класів інтерактивного інтерфейсу на групу класів інтерфейсу командного рядка. Така можливість дуже корисна; багато в чому це пояснюється тим, що інтерфейс командного рядка зручний для тестування ПЗ на рівні блоків або підсистем.

6. Управління ресурсами. Архітектура повинна включати план управління обмеженими ресурсами, такими як з'єднання з БД, потоки і дескриптори. При розробці драйверів, вбудованих систем та інших програм, які будуть працювати в умовах обмеженої пам'яті, архітектура повинна також визначати спосіб управління пам'яттю. Архітектура має містити оцінку ресурсів, що використовуються в номінальному режимі і при екстремальному навантаженні.

7. Безпека. Архітектура має визначати підхід до безпеки на рівні проекту програми та на рівні коду. Якщо модель погроз до сих пір не розроблена, це слід зробити при проектуванні архітектури. Про безпеку потрібно пам'ятати і при розробці принципів кодування, в тому числі методик обробки буферів і ненадійних даних (даних, що вводяться користувачами, файлів «cookie», конфігураційних даних і даних інших зовнішніх інтерфейсів), підходів до шифрування, рівню повідомлень про помилки, захист секретних даних, що знаходяться в пам'яті, та інших питань.

8. Продуктивність. У вимогах слід визначити показники продуктивності. Якщо вони пов'язані з використанням ресурсів, треба визначити пріоритети для різних ресурсів, в тому числі співвідношення швидкодії, використання пам'яті і витрат.

Архітектура повинна включати оцінки продуктивності і пояснювати, чому розробники архітектури вважають ці показники досяжними. Якщо вони можуть бути не досягнуті, це теж повинно бути відображено в архітектурі. Якщо для досягнення деяких показників потрібні специфічні алгоритми або типи даних, також вкажіть це в специфікації архітектури. Крім того, в архітектурі можна вказати обсяг простору і час, що виділяються кожному класу або об'єкту.

9. Масштабованість. Масштабованістю називають можливість системи адаптуватися до зростання вимог. Архітектура має описувати, як система буде реагувати на зростання числа користувачів, серверів, мережевих вузлів, записів у БД, транзакцій і т. д. Якщо розвиток системи не передбачається і її масштабованість не грає ролі, це має бути явно вказано в архітектурі.

10. Взаємодія з іншими системами. Якщо деякі дані або ресурси будуть загальними для розроблюваної системи та інших програм або пристроїв, в архітектурі потрібно вказати, як це буде реалізовано.

11. Інтернаціоналізація / локалізація. «Інтернаціоналізацією» називають реалізацію у програмі підтримки регіональних стандартів.

«Локалізацією» називають переклад інтерфейсу програми і реалізацію в ній підтримки конкретної мови.

12. Введення-виведення. Архітектура має визначати схему читання даних: попередній виклик, читання із затримкою або на вимогу. Крім того, вона повинна описувати рівень, на якому визначатимуться помилки введення-виведення: на рівні полів, записів, потоків даних або файлів.

13. Обробка помилок.

14. Відмовостійкість. При розробці архітектури системи слід вказати очікуваний рівень її відмовостійкості.

15. Можливість реалізації архітектури. Архітектура має підтверджувати, що система технічно здійсненна. Якщо неможливість реалізації якогось компонента може зробити проект непрацездатним, в архітектурі має бути відображено, як вивчалися ці питання: за допомогою прототипів, досліджень чи інакше. Ці аспекти ризику слід усунути до початку повномасштабного конструювання.

16. Надлишкова функціональність. Надійністю називають здатність системи продовжувати роботу після виявлення помилки. Дуже часто в специфікації архітектури розробники визначають більш надійну систему, ніж вказано у вимогах. Одна з причин цього полягає в тому, що система, яка складається з багатьох частин, що задовольняють мінімальним вимогам до надійності, в цілому може виявитися менш надійною, ніж потрібно. У специфікації архітектури має бути явно вказано, чи можуть програмісти реалізувати у своїх блоках програми надлишкову функціональність або вони повинні створити найпростішу працездатну систему.

Визначити відношення до реалізації надлишкової функціональності особливо важливо тому, що багато програмісти роблять це автоматично, з почуття професійної гідності. Явно висловивши очікування в архітектурі, ви зможете уникнути феномена, при якому деякі класи виключно надійні, а інші лише відповідають вимогам.

17. Повторне використання. Якщо план передбачає застосування існуючого коду, тестів, форматів даних і т. д., архітектура повинна пояснювати, як повторно використані ресурси будуть адаптовані до інших архітектурних особливостей.

18. Стратегія змін. Архітектура повинна підтверджувати, що можливі покращення розглядалися. Якщо вірогідні зміни форматів введення або виведення даних, стилю взаємодії з користувачами або вимог до обробки, архітектура повинна показувати, що всі ці зміни були передбачені і кожна з них буде обмежена невеликим числом класів. Архітектурний план внесення змін може бути зовсім простим: включити у файли даних номери версій, зарезервувати поля на майбутнє, спроектувати файли так, щоб в них можна було додати нові таблиці і т. д. Якщо застосовується генератор коду, архітектура повинна показувати, що він підтримує можливість внесення передбачуваних змін.

В архітектурі повинні бути відображені стратегії, які дозволяють програмістам не обмежувати наявні у них вибір завчасно. Так, архітектура

може визначати, що замість жорстко закодованих тестів буде застосовуватися метод, заснований на перевірці таблиць. Дані таблиць можна зберігати в зовнішньому файлі, а не включати в програму, що дозволить вносити до неї зміни без перекомпіляції.

Скільки часу слід присвятити виконанню попередніх умов

Час, що йде на визначення проблеми, формулювання вимог і проектування архітектури ПЗ, залежить від особливостей проекту. Як правило, якщо проект розвивається без проблем, робота над вимогами, архітектурою проекту і попереднім плануванням поглинає 10-20% зусиль і 20_30% часу (McConnell, 2013; Kruchten, 2000). Ці показники не включають витрати на детальне проектування - воно є частиною конструювання.

КОНТРОЛЬНІ ПИТАННЯ:

1. Що таке попередні вимоги? У чому полягає важливість їх виконання?
2. Які існують найпопулярніші типи програмних проектів?
3. Які оптимальні методи слід використовувати при роботі над кожним типом програмного проекту?
4. Що таке попередні умови, пов’язані з визначенням проблеми?
5. Що такепопередні умови, пов'язані з формулюванням вимог?
6. Що таке попередні умови, пов'язані з розробкою архітектури?

Скільки часу слід присвятити виконанню попередніх умов?

ТЕМА 3. Проектування при конструюванні

«Проектування» може виражатися в:

- простому написанні інтерфейсу класу на псевдокоді до розробки його деталей;
- малюванні діаграм відносин декількох класів перед написанням їх коду;
- обговоренні оптимального шаблону проектування разом з колегою.

Проблеми, пов'язані з проектуванням ПЗ

Під «проектуванням ПО» розуміють розробку або винахід схеми перетворення специфікації програми у готовий додаток. Проектування - це той процес, який пов'язує вироблення вимог з кодуванням і налагодженням. Структура вдалого високорівневого проекту програми може успішно охоплювати цілий ряд більш низькорівневих проектів. Проектування корисно при роботі над невеликими додатками і необхідно при роботі над великими.

Однак з проектуванням пов'язано декілька проблем:

Проектування - «брудна» проблема. Хорст Ріттель і Мелвін Веббер визначили «брудну» проблему як проблему, яку можна явно визначити тільки шляхом повного або часткового вирішення (Rittel and Webber, 1973).

Проектування - неохайний процес (навіть якщо воно призводить до акуратного результату). Завершений проект програми повинен виглядати добре організованим і ясним, але процес розробки цього проекту зовсім не такий акуратний, як кінцевий результат.

Проектування неохайно тому, що ви виконуєте багато невірних дій і потрапляєте в безліч тупиків, тобто здійснюєте масу помилок. У дійсності помилки є суттю проектування: дешевше допустити помилки і виправити проект програми, ніж знайти їх після кодування і виправляти готовий код. Проектування неохайно тому, що вдале рішення часто лише трохи відрізняється від невдалого.

Проектування пов'язане з визначенням компромісів і пріоритетів. Один з найважливіших аспектів роботи проектувальника - аналіз конкуруючих характеристик проекту і досягнення балансу між ними. Якщо швидкість відгуку системи важливіше, ніж мінімізація часу розробки, проектувальник вибере один варіант. Якщо на чолі кута швидкість розробки, оптимальним може виявитися інший варіант проекту.

Проектування передбачає обмеження можливостей. Проектування передбачає не тільки забезпечення можливостей, але і їх обмеження.

Проектування – не детермінований процес. Спроектувати комп'ютерну програму можна десятками різних способів.

Проектування - евристичний процес. Так як проектування не детерміновано, методи проектування найчастіше є евристичними методами, тобто «практичними правилами» або «способами, які можуть спрацювати», а не відтворюваними процесами, які завжди приводять до передбачуваних результатів. Проектування - метод проб і помилок.

Проектування - поступовий процес. Можна досить вдало узагальнити названі аспекти проектування, сказавши, що проектування - «поступовий процес». Проекти додатків не виникають у розумі розробників відразу в готовому вигляді. Вони розвиваються і поліпшуються під час оглядів, неформальних обговорень, написання коду та виконання його ревізій.

Основні концепції проектування

Успішне проектування ПЗ вимагає розуміння кількох важливих концепцій.

Головний Технічний Імператив Розробки ПЗ: управління складністю

Важливість управління складністю. Програмні проекти рідко зазнають краху з технічних причин. Найчастіше провал пояснюється неадекватним виробленням вимог, невдалим плануванням або неефективним управлінням. Якщо ж провал зумовлений таки переважно технічною причиною, дуже часто нею виявляється неконтрольована складність. Інакше кажучи, додаток став таким складним, що розробники перестали по-справжньому розуміти, що ж вони роблять. Якщо робота над проектом досягає моменту, після якого вже ніхто не може повністю зрозуміти, як зміна одного фрагмента програми вплине на інші фрагменти, прогрес припиняється.

Найчастіше причинами неефективності є: складне рішення простої проблеми; просте, але невірне рішення складної проблеми; неадекватне складне рішення складної проблеми.

Підхід до управління складністю:

- намагайтеся звести до мінімуму обсяг суттєвої складності, з яким доведеться працювати в кожен конкретний момент часу;

- стримуйте необов'язкове зростання несуттєвою складності.

Бажані характеристики проекту

Список внутрішніх характеристик проекту:

1. Мінімальна складність.
2. Простота супроводу.
3. Loose coupling - зведення до мінімуму числа з'єднань між різними частинами програми.
4. Розширюваність.
5. Можливість повторного використання.
6. Високий коефіцієнт об'єднання по входу.
7. Низький або середній коефіцієнт розгалуження по виходу.
8. Портуємість. Мінімальна, але повна функціональність.
9. Стратифікація.
10. Відповідність стандартним методикам.

Рівні проектування

Проектування програмної системи вимагає декількох рівнів детальності. Деякі методи проектування використовуються на всіх рівнях, а інші тільки на одному-двох (рис. 3.1).

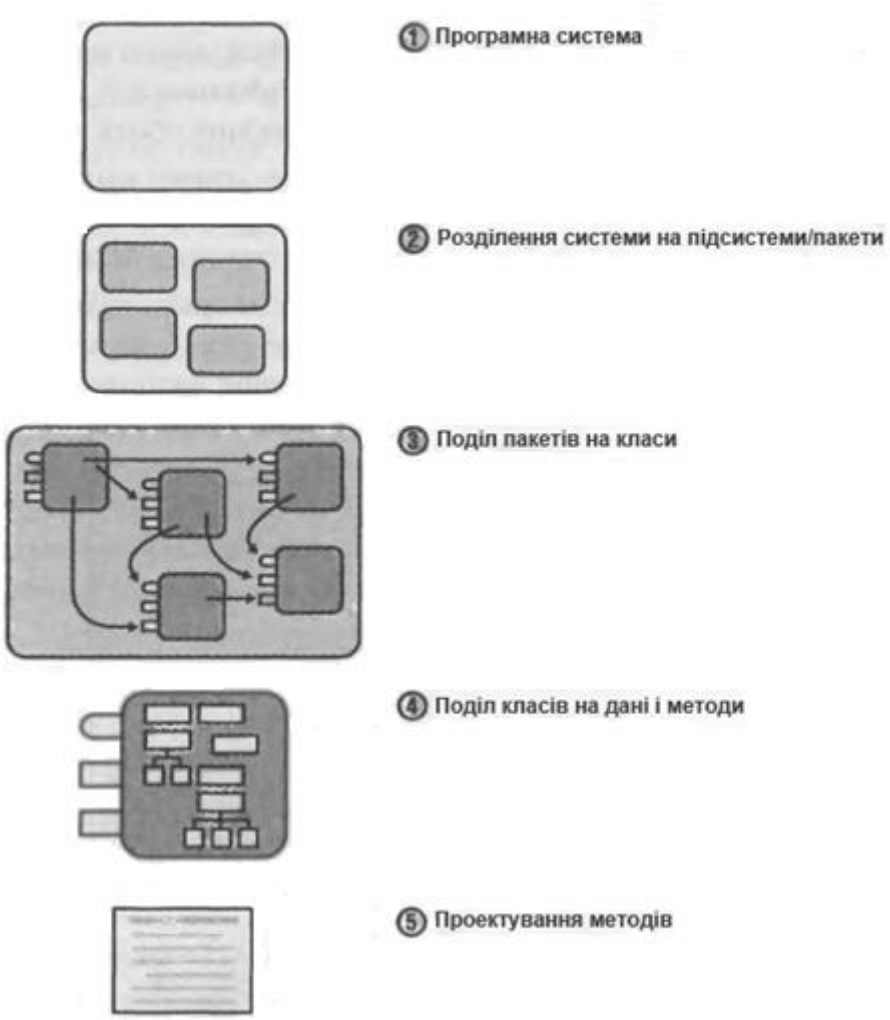

Рисунок 3.1 - Рівні проектування програми

Рівень 1: програмна система

Першому рівню проектування відповідає вся система. Деякі програмісти з системного рівня відразу переходять до проектування класів, але звичайно доцільно спроектувати більш високорівневі комбінації класів, такі як підсистеми або пакети.

Рівень 2: поділ системи на підсистеми або пакети

Головний результат проектування на цьому рівні - визначення основних підсистем. Підсистеми можуть бути досить великими, такими як модуль роботи з базами даних, модулі GUI, модулі бізнес-правил або створення звітів, інтерпретатор команд і т. д. Суть проектування на даному рівні полягає в поділі програми на основні підсистеми та визначенні взаємодій між ними. Зазвичай цей рівень потрібен при роботі над будь-якими проектами, реалізація яких займає більше декількох тижнів. При проектуванні окремих підсистем можна застосовувати різні підходи: обирайте той, який здається вам оптимальним у кожному конкретному випадку. На рис. 3.1 даний рівень проектування позначений цифрою 2.

Особливо важливий аспект цього рівня - визначення правил взаємодії підсистем. Якщо всі підсистеми можуть взаємодіяти, необхідність їх поділу зникає. Підкреслюйте суть підсистем, обмежуючи їх взаємодію між собою.

Припустимо, ви визначили систему з шести підсистем (рис. 3.2). За відсутності будь-яких обмежень чинності другого закону термодинаміки ентропія системи повинна збільшитися. Один із способів збільшення ентропії є абсолютно вільна взаємодія між підсистемами (рис. 3.3).

Рисунок 3.2. – Приклад системи, яка включає в себе шість підсистем

Стрілки між підсистемами можна розглядати як шланги з водою. Якщо вам необхідно «висмикнути» одну з підсистем, до неї напевно будуть підключені декілька шлангів. Чим більше шлангів вам потрібно буде від'єднати і підключити заново, тим гірше. Архітектура системи повинна бути такою, щоб заміна підсистем вимагала якомога менше метушні зі шлангами.

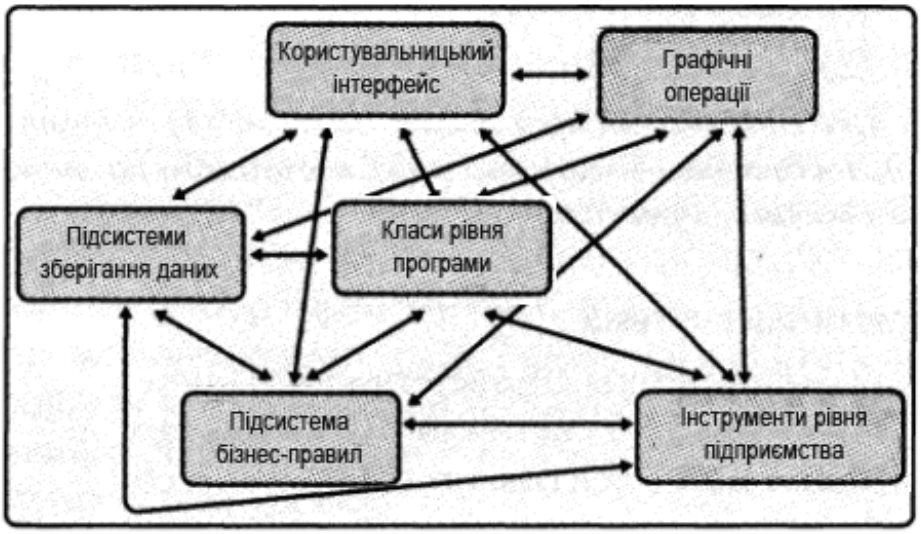

Рисунок 3.3. – Результат відсутності правил, які обмежують взаємодію підсистем

При належній передбачливості всі ці питання можна вирішити, виконавши трохи додаткової роботи. Реалізуйте комунікацію між підсистемами на основі принципу «необхідного знання». Пам'ятайте: простіше спочатку обмежити взаємодію, а потім зробити її більш вільною, ніж намагатися ізолювати підсистеми після написання декількох сотень викликів між ними. На рис. 3.4 показано, як кілька правил комунікації можуть змінити систему, зображену на рис. 3.3.

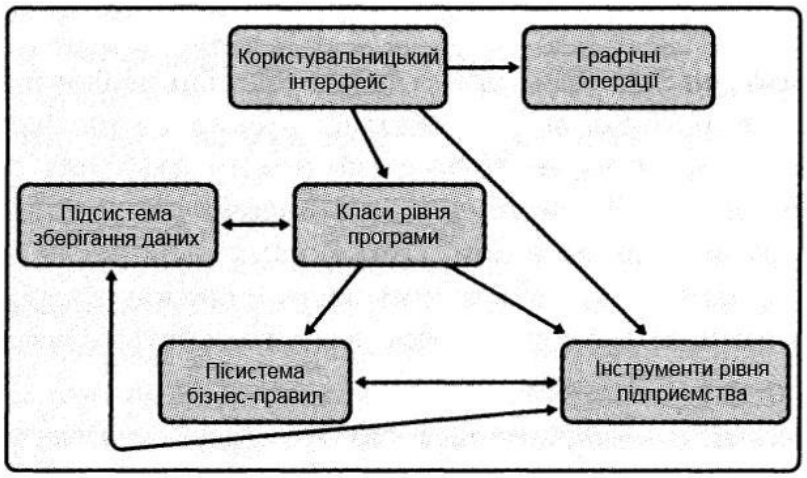

Рисунок 3.4. - Спрощення взаємодії підсистем

Щоб з'єднання підсистем були зрозумілими і легкими у супроводі, намагайтеся підтримувати простоту відносин між підсистемами. Найпростішим відношенням є таке, при якому одна підсистема викликає методи іншої. Більш складне відношення має місце, коли одна підсистема містить класи іншої. Найскладніше відношення - успадкування класів однієї підсистеми від класів іншої.

Пам’ятайте важливе правило: діаграма системного рівня на зразок тієї, що показана на рис. 3.4, повинна бути ациклічним графом. Інакше кажучи, програма не повинна містити циклічних відносин, за яких клас А використовує клас В, клас В використовує клас С, а клас С - клас А.

Підсистеми, що часто використовуються. Деякі типи підсистем знову і знову використовуються в різних системах. Нижче наведені ті, що зустрічаються найчастіше.

- Підсистема бізнес-правил. Бізнес-правилами називають закони, директиви, політики та процедури, реалізовані в комп'ютерній системі. Наприклад, у випадку системи розрахунку заробітної плати бізнес-правилами могли б бути директиви податкового управління, що визначають різноманітні види податків. Додатковим джерелом правил могла б бути угода з профспілкою, що регламентує оплату понаднормової роботи, відпустки і т. д. При створенні програми для агентства по страхуванню автомобілів правила можуть бути засновані на відповідних державних законах.
- Підсистема користувальницького інтерфейсу. Ізоляція компонентів інтерфейсу користувача в окрему підсистему дозволяє змінювати його, не впливаючи на решту програми. Як правило, підсистема користувача інтерфейсу містить кілька підлеглих підсистем або класів, що відповідають за GUI, інтерфейс командного рядка, роботу з меню, управління вікнами, довідкову систему і т. д.
- Підсистема доступу до БД. Ви може приховати деталі реалізації доступу до БД, щоб більша частина програми не потребувала знань «брудних» подробиць операцій над низькорівневими структурами і могла працювати з даними в термінах бізнес-проблеми. Підсистеми, що приховують деталі реалізації, забезпечують важливий рівень абстракції, що знижує складність програми. Вони концентрують операції над БД в одному місці і знижують ймовірність помилок при роботі з даними, а також дозволяють легко змінювати структуру БД без зміни великої частини програми.
- Підсистема ізоляції залежностей від операційної системи (ОС). Залежності від ОС слід ізолювати в підсистемі з тієї ж причини, що і залежності від обладнання. Якщо, наприклад, ви розробляєте програму для Microsoft Windows, навіщо обмежувати себе середовищем Windows? Ізолюйте виклики Windows у спеціалізованій інтерфейсній підсистемі, і якщо вам пізніше захочеться перенести програму на платформу Mac OS або Linux, то доведеться змінити тільки цю підсистему.

Рівень 3: поділ підсистем на класи

Цей рівень проектування передбачає визначення всіх класів системи. Наприклад, підсистема доступу до БД може бути далі розділена на класи доступу до даних і класи зберігання даних, а також метадані БД.

Крім того, на цьому рівні слід визначити деталі взаємодії кожного класу з іншими елементами системи, особливо інтерфейс класу. Суттю проектування

на даному рівні є декомпозиція підсистем до такого рівня детальності, який дозволить реалізувати частини підсистем у формі окремих класів.

Поділ підсистем на класи зазвичай потрібен у всіх проектах, на реалізацію яких піде більше декількох днів. У великих проектах необхідні і другий, і третій рівні проектування. При роботі над зовсім невеликим проектом другий рівень проектування можна пропустити.

Класи і об'єкти. Один з найважливіших аспектів об'єктно-орієнтованого проектування - відмінності між об'єктами і класами. Об'єкт - це будь-яка конкретна динамічна сутність, що має конкретні значення і атрибути та існуюча в період виконання програми. Клас - це статична сутність, з якою ви маєте справу, переглядаючи лістинг програми. Наприклад, ви можете оголосити клас Person (людина), що має такі атрибути, як прізвище, вік, стать і т. д. У період виконання ви будете працювати з об'єктами nancy, hank, diane, tony і т. д. - інакше кажучи, зі специфічними екземплярами класу. Відмінність між класом і об'єктом аналогічна відмінності між «схемою» і «екземпляром».

Рівень 4: поділ класів на методи

Даний рівень проектування полягає у поділі кожного класу на методи. Деякі методи вже будуть визначені на рівні 3, при проектуванні інтерфейсів класів. На рівні 4 ви маєте детально визначити закриті методи класів.

Повне визначення методів класу часто дозволяє краще зрозуміти його інтерфейс, що може підштовхнути до відповідної зміни інтерфейсу, тобто до повернення на рівень 3.

Четвертий рівень декомпозиції і проектування часто залишається окремим програмістам і необхідний у будь-якому проекті, реалізація якого вимагає більше декількох годин.

Рівень 5: проектування методів

На цьому рівні проектування полягає в детальному визначенні функціональності окремих методів, за що зазвичай відповідають окремі програмісти, що працюють над конкретними методами. Даний рівень може включати такі дії, як написання псевдокоду, пошук алгоритмів у книгах, роздум над оптимальною організацією фрагментів методу і написання коду. Цей рівень проектування виконується у всіх випадках, але не завжди усвідомлено і якісно.

Компоненти проектування: евристичні принципи

Визначте об'єкти реального світу

При проектуванні з використанням об'єктів визначте:

- об’єкти та їх атрибути (методи й дані);
- дії, які можуть бути виконані над кожним об’єктом;
- дії, які кожен об’єкт може виконувати над іншими об’єктами;
- частини кожного об’єкта, які можуть бачити інші об’єкти (відкриті й закриті частини);
- відкритий інтерфейс кожного об’єкту.

Визначте об'єкти та їх атрибути. В основі створення програм зазвичай лежать сутності реального світу. Наприклад, система розрахунку погодинної оплати може бути заснована на таких сутностях, як співробітники, клієнти, карти обліку часу і рахунки (рис. 13).

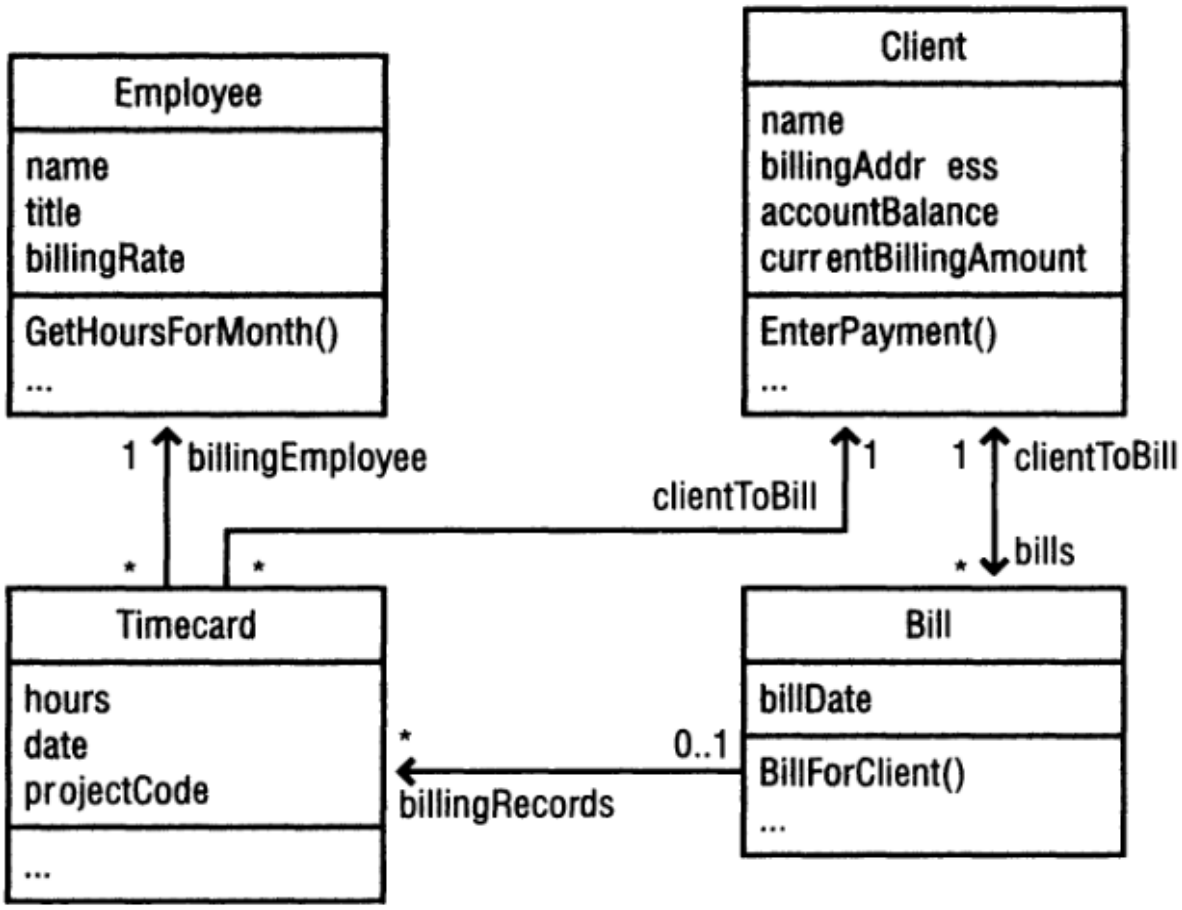

Рисунок 3.5 - Система розрахунку оплати, яка складається з чотирьох основних об'єктів

Визначити атрибути об'єктів не складніше, ніж самі об'єкти. Кожен об'єкт має характеристики, релевантні для комп'ютерної програми. Скажімо, в системі розрахунку погодинної оплати об'єкт «співробітник» володів би такими атрибутами, як ім'я/прізвище, посада та рівень оплати. З об'єктом «рахунок» були б пов'язані такі атрибути, як сума, ім'я/прізвище клієнта, дата і т. д.

Об'єктами системи GUI були б різноманітні вікна, кнопки, шрифти та інструменти малювання.

Визначте дії, які можуть бути виконані над кожним об'єктом. Об'єкти можуть підтримувати різноманітні операції. У нашій системі розрахунку оплати об'єкт «співробітник» міг би підтримувати зміну посади або рівня оплати, об'єкт «клієнт» - зміну реквізитів рахунку і т. д.

Визначте дії, які кожен об'єкт може виконувати над іншими об'єктами. Двома універсальними діями, які об'єкти можуть виконувати один над одним, є включення (containment) і спадкування. «Карта обліку часу» може включати об'єкт «співробітник» і об'єкт «клієнт», а об'єкт «рахунок» може включати карти обліку часу. Крім того, рахунок може повідомляти, що клієнт оплатив послуги, а клієнт - оплачувати зазначену в рахунку суму. Більш складна система включала б додаткові взаємодії.

Визначте частини кожного об'єкта, які можуть бачити інші об'єкти. Один з головних аспектів проектування - визначення частин об'єкта, які слід зробити відкритими, і частин, які слід тримати закритими. Цього рішення вимагають і дані, і методи.

Визначте інтерфейс кожного об'єкта. Для кожного об'єкта треба визначити формальний синтаксичний інтерфейс на рівні мови програмування. Дані та методи, які об'єкт надає в розпорядження решті об'єктів, називаються «відкритим інтерфейсом».

Ці часто повторювані дії не обов'язково виконувати в зазначеному порядку. Пам'ятайте про важливість ітерації.

Визначте узгоджені абстракції. Абстракція дозволяє задіяти концепцію, ігноруючи деякі її деталі і працюючи з різними деталями на різних рівнях. Маючи справу з складеним об'єктом, ви маєте справу з абстракцією. Якщо ви розглядаєте об'єкт як «будинок», а не як комбінацію скла, деревини і цвяхів, ви вдаєтеся до абстракції. Якщо ви розглядаєте безліч будинків як «місто», ви вдаєтеся до іншої абстракції.

Базові класи являють собою абстракції, що дозволяють концентруватися на загальних атрибутах похідних класів і ігнорувати деталі конкретних класів при роботі з базовим класом. Вдалий інтерфейс класу - це абстракція, що дозволяє зосередитися на інтерфейсі, не турбуючись про внутрішні механізми роботи класу. Інтерфейс грамотно спроектованого методу забезпечує таку ж вигоду на більш низькому рівні детальності, а інтерфейс грамотно спроектованого пакета або підсистеми - на більш високому.

Інкапсулюйте деталі реалізації. Разом з абстракцією на допомогу приходить інкапсуляція. Абстракція каже: «Ви можете розглянути об'єкт із загальної точки зору». Інкапсуляція додає: «Більше того, ви не можете розглянути об'єкт з іншої точки зору».

Продовжимо нашу аналогію: інкапсуляція дозволяє вам дивитися на будинок, але не дає підійти досить близько, щоб дізнатися, з чого зроблені двері. Інкапсуляція дозволяє вам знати про існування двері, про те, відкрита вона або замкнена, але при цьому ви не можете дізнатися, з чого вона зроблена (з дерева, скловолокна, сталі або іншого матеріалу), і вже ніяк не зможете розглянути окремі волокна деревини.

Використовуйте спадкування, якщо воно спрощує проектування. При проектуванні ПЗ часто з'ясовується, що одні об'єкти аналогічні іншим за винятком декількох відмінностей. Так, при створенні системи розрахунку зарплати потрібно врахувати, що одні співробітники працюють повний день, а інші - неповний. У цьому випадку набори даних, асоційовані з співробітниками обох категорій, будуть відрізнятися лише кількома аспектами. Об'єктно-орієнтований підхід дозволяє створити загальний тип «співробітник» і визначити співробітників, що працюють повний день, як співробітників загального типу за винятком декількох відмінностей. Якщо операція над об'єктом «співробітник» не залежить від його категорії, вона виконується так само як для співробітника загального типу. Якщо ж операція залежить від типу співробітника, вона виконується різними способами.

Визначення подібностей і відмінностей між такими об'єктами називається «спадкуванням», тому що окремі типи співробітників, що працюють повний і неповний день, успадковують властивості загального типу «співробітник».

Користь спадкування в тому, що воно доповнює ідею абстракції. Абстракція дозволяє представити об'єкти з різним рівнем детальності. Спадкування спрощує програмування. Спадкування - одне з найбільш потужних засобів об'єктно-орієнтованого програмування. При правильному застосуванні воно може принести велику користь, однак у зворотному випадку збиток буде чималим.

Приховуйте інформацію. Приховання інформації - один з основних принципів і структурного, і об'єктно-орієнтованого проектування. У першому випадку приховування інформації лежить в основі ідеї «чорних ящиків». У другому воно дає початок концепціям інкапсуляції і модульності і пов'язане з концепцією абстракції. Приховання інформації - одна з найбільш конструктивних ідей у світі розробки ПЗ.

Вперше приховання інформації було представлено на суд громадськості в 1972 р. Девідом Парнасом (David Parnas) у статті «On the Criteria to Be Used in Decomposing Systems Into Modules (Про критерії, що використовуються при декомпозиції систем на модулі)». З приховванням інформації тісно пов'язана ідея «секретів» - аспектів проектування та реалізації, які розробник ПЗ вирішує приховати в деякому місці від решти частини програми.

При прихованні інформації кожен клас (пакет, метод) характеризується аспектами проектування або конструювання, які він приховує від решти класів. Секретом може бути джерело ймовірних змін, формат файлу, реалізація типу даних чи область, ізоляція якої потрібна для зведення до мінімуму шкоди від можливих помилок. Клас повинен приховувати цю інформацію і захищати своє право на «особисте життя». Невеликі зміни системи можуть впливати на кілька методів класу, але не повинні поширюватися на його інтерфейс.

Один з найважливіших аспектів проектування класу - прийняття рішення про те, які властивості зробити доступними поза класом, а які залишити секретними. Клас може включати 25 методів, надаючи доступ тільки до п'яти з них і використовувати інші 20 внутрішньо. Клас може використовувати декілька типів даних, не розкриваючи відомостей про них.

Інтерфейс класу повинен повідомляти якомога менше про внутрішню роботу класу. Клас багато в чому схожий на айсберг, більша частина якого схована під водою.

Визначте області ймовірних змін:

- визначте елементи, зміна яких здається ймовірним;
- відокремте елементи, зміна яких здається ймовірним;
- ізолюйте елементи, зміна яких здається ймовірним.

Нижче наведені кілька областей, що змінюються найчастіше:

- Бізнес-правила.
- Залежності від устаткування.
- Введення-виведення.
- Нестандартні можливості мови.

- Складні аспекти проектування і конструювання.
- Змінні статусу. Змінні статусу характеризують стан програми і змінюються частіше, ніж більшість інших видів даних. Використання змінних статусу можна зробити більш гнучким і зрозумілим мінімум двома способами:

в якості змінних статусу застосовувати не булеві змінні, а перерахування;

замість безпосередньої перевірки змінної використовуйте методи доступу.

Підтримуйте сполучення слабким.Сполучення характеризує силу зв'язку класу або методу з іншими класами чи методами. Наша мета - створити класи і методи, що мають нечисленні, безпосередні, явні і гнучкі відносини з іншими класами, що ще називають «слабким сполученням» (loose coupling) ».

Критерії оцінки сполучення

Нижче описані критерії, що дозволяють оцінити пару модулів.

- Об'єм. Обсяг зв'язку характеризує число з'єднань між модулями. Чим їх менше, тим краще, оскільки модуль, що має більш компактний інтерфейс, легше пов'язати з іншими модулями. Метод, який приймає один параметр, слабкіше пов'язаний з зухвалими його модулями, ніж метод, який приймає шість параметрів. Клас, що має чотири грамотно певних відкритих методів, слабкіше пов'язаний з модулями, які його використовують, ніж клас, що надає 37 відкритих методів.
- Видимість. Видимістю називають помітність зв'язку між двома модулями. Програмування не служба в ЦРУ - ніхто не похвалить вас за вдале маскування. Воно більше схоже на рекламу: вам слід робити зв'язку між модулями як можна більш крикливими. Передача даних за допомогою списку параметрів формує очевидний зв'язок, і це вдалий варіант. Передача інформації іншого модуля в глобальних даних є замаскованою і тому невдалим зв'язком. Опис зв'язку, що здійснюється через глобальні дані, в документації робить її більш явною і є трохи більш вдалим підходом.
- Гнучкість. Гнучкість характеризує легкість зміни зв'язку між модулями. Ідеальний зв'язок має бути якомога гнучкіше. Гнучкість частково визначається іншими аспектами пов'язаності, але в той же час відрізняється від них. Нехай, у вас є метод LookupVacationBenefit (), що визначає тривалість відпустки співробітника на підставі дати його прийому на роботу і посади. Припустимо далі, що в іншому модулі у вас є об'єкт employee (співробітник), що містить, крім усього іншого, інформацію про посади і дату прийому на роботу, і що цей модуль передає об'єкт employee в метод LookupVacationBenefit ().

Види сполучення

Найпоширеніші види сполучення описані нижче:

- Просте створення пари за допомогою об'єкта. Модуль пов'язаний з об'єктом цим способом, якщо він створює екземпляр даного об'єкта. З цим видом сполучення також все гаразд.

- Сполучення за допомогою об'єкта-параметра. Два модулі сполучені один з одним об'єктом-параметром, якщо Об'єкт 1 вимагає, щоб Об'єкт 2 передав йому Об'єкт 3. Цей вид сполучення жорсткіше, ніж той вид, при якому Об'єкт 1 вимагає від Об'єкта 2 тільки примітивних типів даних, тому що Об'єкт 2 повинен володіти інформацією про Об'єкті 3.
- Семантичне сполучення. Найпідступніший тип сполучення має місце тоді, коли один модуль використовує не якийсь синтаксичний елемент іншого модуля, а деякі семантичні знання про внутрішню роботу цього модуля.

Намагайтеся використовувати популярні шаблони проектування (рис. 3.6).

Шаблони знижують складність, надаючи готові абстракції. Шаблони знижують кількість помилок, стандартизуючи деталі популярних рішень. Шаблони мають евристичну цінність, вказуючи на можливі варіанти проектування. Шаблони спрощують взаємодію між розробниками, дозволяючи їм спілкуватися на більш високому рівні.

Шаблон	Опис
Абстрактна фабрика (Abstract Factory)	Підтримує створення наборів родинних об'єктів шляхом визначення виду набору, але не виду кожного окремого об'єкту.
Адаптер (Adapter)	Перетворює інтерфейс класу в інший інтерфейс.
Мост (Bridge)	Створює інтерфейс і реалізацію таким чином, що їх можна змінити незалежно один від одного.
Компонувальник (Composite)	Складається з об'єкту, який містить додаткові об'єкти того ж типу, дозволяючи клієнтському коду взаємодіяти з об'єктом верхнього рівня і не турбуватися про детальні об'єкти.
Декоратор (Decorator)	Динамічно призначає об'єкту види відповідальності без створення окремих підкласів для кожної можливої конфігурації видів відповідальності.
Фасад (Facade)	Надає узгоджений інтерфейс коду, який в іншому випадку не представляв би узгодженого інтерфейсу.
Фабричний метод (Factory Method)	Створює екземпляри класів, похідних від конкретного базового класу, причому окремі похідні класи відстежуються тільки «Фабричним методом».
Итератор (Iterator)	Цей серверний об'єкт представляє доступ до кожного елементу набору в посідовному порядку.
Спостерігач (Observer)	Підтримує сінхронізацію декількох об'єктів, за допомогою якої об'єкт повідомляє набір зв'язаних об'єктів про зміни будь-якого члену набору.
Одиночка (Singleton)	Надає глобальний доступ до класу, який може мати один і тільки один екземпляр.

Рисунок 3.6 – Популярні шаблони проектування

Інші евристичні принципи

- Прагніть до максимальної зв'язності. Поняття зв'язності (cohesion) виникло в області структурного проектування і зазвичай обговорюється в тому ж контексті, що і сполучення (coupling). Зв'язність характеризує те, наскільки добре всі методи класу або всі фрагменти методу відповідають головної мети, - інакше кажучи, наскільки сфокусований клас. Класи, що складаються з дуже схожих за функціональністю блоків, мають високий ступінь зв'язності, і наша евристична мета полягає в тому, щоб цілісність була якомога вище. Зв'язність - корисний інструмент управління складністю, тому що чим краще код класу відповідає головній меті, тим простіше запам'ятати все, що код виконує.
- Формуйте ієрархії. Ієрархія - це багаторівнева структура організації інформації, при якій найбільш загальна або абстрактна репрезентація концепції відповідає вершині, а більш детальні спеціалізовані репрезентації - більше низьких рівнів. При розробці ПЗ ієрархії виявляються, наприклад, в наборах класів і в послідовностях викликів методів.
- Формалізуйте контракти класів. На більш детальному рівні корисну інформацію можна отримати, розглядаючи інтерфейс кожного класу як контракт з іншими частинами програми. Зазвичай контракт має форму «Якщо ви обіцяєте надати дані x, y і z і гарантуєте, що вони будуть мати характеристики a, b і з, я зобов'язуюсь виконати операції 1, 2 і 3 з обмеженнями 8, 9 і 10». Обіцянки клієнтів класу зазвичай називаються предумовою (preconditions), а зобов'язання класу перед клієнтами - постумовою (postconditions).
- Грамотно призначайте сфери відповідальності.
- Розробляйте систему для тестування.
- Малюйте діаграми.
- Підтримуйте модульність проекту системи. Цей принцип передбачає, що кожен метод або клас повинен бути схожий на «чорний ящик»: ви знаєте, що в нього надходить і що з нього виходить, але не знаєте, що відбувається всередині.

Методики проектування

Використовуйте ітерацію.Проектування - ітеративний процес. Якщо перша спроба створення проекту здається цілком вдалою, не зупиняйтеся! Друга спроба майже завжди виявляється краще першої, і при кожній спробі ви будете дізнаватися щось таке, що допоможе вам поліпшити загальний проект.

«Розділяй і володарюй». Розділіть програму на різні області та спроектуйте їх окремо.

«Top-down» і «bottom-up» підходи до проектування. «Top-down» проектування починається на високому рівні абстракції. Наприклад, ви спочатку визначаєте базові класи або інші неспецифічні елементи проекту.

«Bottom-up» проектування починається зі специфіки і поступово переходить до все більшої спільності. Як правило, воно починається з

визначення конкретних об'єктів, на основі яких потім розробляються більш загальні об'єднання об'єктів і базові класи.

Експериментальне прототипування. Іноді адекватність конкретного проекту неможливо оцінити, не маючи додаткових відомостей про деталі реалізації. Ви можете не знати, чи прийнятна конкретна організація бази даних, поки не дізнаєтеся, чи буде вона задовольняти конкретним вимогам до продуктивності. Ви можете не знати, чи прийнятний проект конкретної підсистеми, поки не будуть обрані конкретні бібліотеки GUI. Це приклади суттєвого «бруду» при проектуванні ПЗ: ви не можете повністю визначити проблему проектування, поки не вирішите її хоч частково.

Добре відомий недорогий спосіб отримати відповіді на ці питання - експериментальне прототипування. У слово «прототипування» люди вкладають різний зміст (McConnell, 1996). У даному контексті воно означає написання абсолютно мінімального обсягу підлягає викиданню коду, потрібного для відповіді на окреме питання проектування.

Спільне проектування.

Реєстрація процесу проектування. Традиційним способом реєстрації проекту є його опис у формальній проектній документації. Однак є маса інших способів, ефективних при використанні неформального підходу, при створенні невеликих систем або коли потрібна «полегшена» методика реєстрації проекту:

- Включайте проектну документацію прямо в код. Документуйте основні аспекти проектування в коментарях - як правило, в заголовках файлів або класів.
- Реєструйте протоколи обговорення проекту та прийняті рішення за допомогою Wiki.
- Пишіть резюме дискусій у формі електронної пошти. Обговоривши проект системи, доручіть комусь записати резюме бесіди - особливо ухвалені рішення - і відправити його кожному члену групи. Зберігайте копії листів у папці, доступній всім учасникам проекту.
- Використовуйте цифровий фотоапарат. Одним частим бар'єром, що перешкоджає документування проекту, є утомливо малювання проектів традиційним способом. Однак способи документування не обмежуються варіантами «реєстрацію проекту з використанням красиво відформатованої формальної нотації» і «повної відсутності проектної документації». Фотографуйте діаграми, які розробники малюють на дошці, і включайте їх в традиційні документи: це набагато простіше, ніж малювати схеми вручну, але не менш ефективно.
- Зберігайте плакати зі схемами проекту.
- Використовуйте картки CRC {Class, Responsibility, Collaborator - клас, відповідальність, співробітництво). Ще один простий варіант документування проекту - використовувати картки. Напишіть на кожній картці ім'я класу, аспекти його відповідальності та імена класів, з якими він співпрацює. Продовжуйте працювати з картками, поки не будете задоволені результатом. У цей момент ви можете просто зберегти картки

на майбутнє. Цей спосіб майже не вимагає витрат, не лякає своєю складністю і заохочує взаємодію членів групи (Beck, 1991).

- Створюйте діаграми UML з доречним рівнем детальності.

Описані способи працюють і в різних комбінаціях, так що можете вільно змішувати їх, пристосовуючи до конкретних проектів і навіть різним областям одного проекту.

КОНТРОЛЬНІ ПИТАННЯ:

1. У чому може виражатися проектування ПЗ?
2. Які проблеми пов’язані з проектуванням ПЗ?
3. Скільки існує основних концепцій проектування?
4. Які існують рівні проектування ПЗ?
5. Опишіть евристичні принципи проектування ПЗ?
6. Які існують методики проектування ПЗ?

ТЕМА 4. Ефективне використання класів

На зорі комп'ютерної епохи програмісти думали про програмування в термінах операторів. У 1970-80-ті про програми стали думати в термінах методів. У XXI столітті ми розглядаємо програмування в термінах класів.

Клас - це набір даних і методів, що мають загальну, цілісну, добре впевнену сферу відповідальності. Дані - необов'язковий компонент класу: клас може включати тільки методи, що мають цілісний набір послуг. Однією з головних умов ефективного програмування є максимізація частини програми, яку можна ігнорувати при роботі над конкретними фрагментами коду. Класи - головний засіб досягнення цієї мети.

Основи класів: абстрактні типи даних

Абстрактний тип даних (АТД) - це набір, що включає дані і операції, що виконуються над ними. Операції описують дані для іншої частини програми і дозволяють їх змінювати.

Розуміння концепції АТД необхідно для розуміння об'єктно-орієнтованого програмування. Не маючи чіткого уявлення про АТД, програмісти створюють класи, які тільки називаються «класами», будучи насправді лише зручними контейнерами, що містять набори даних і методів, які погано узгоджуються один з одним. АТД що дозволяють працювати з сутностями реального світу, а не з низькорівневими сутностями реалізації. Завдяки цьому замість вставки вузла в зв'язний список можна додати комірку в електронну таблицю, новий тип вікна в список типів вікон або черговий пасажирський автомобіль в програму, що моделює потік руху. Можливість працювати в проблемній області, а не у низькорівневій області реалізації програми дуже зручна.

Наведемо приклад ситуації, в якій застосування АТД було б корисним. Припустимо, ви пишете програму, керуючу виводом тексту на екран з використанням різноманітних гарнітур шрифтів, їх розмірів і атрибутів

(наприклад, «напівжирний» і «курсив»).За роботу зі шрифтами відповідає конкретна частина програми. При використанні АТД дані - назви гарнітур, розміри і атрибути шрифтів - будуть об'єднані в одну групу з методами які їх оброблюють . Набір даних і методів, що необхідні для досягнення певної однієї мети, - це і є АТД.

Без АТД вам довелося б прийняти спеціалізований підхід до роботи зі шрифтами. Скажімо, для вибору шрифту розміром 12 пт, яким могли б відповідати 16 пікселів, ви написали б:

currentFont.size = 16

Створивши набір бібліотечних методів, код можна було б зробити трохи зрозумілішим:

currentFont.size = PointsToPixels (12)

Крім того, атрибуту шрифту можна було б присвоїти більш певне ім'я, наприклад:

currentFont.sizelnPixels = PointsToPixels (12)

Однак, при цьому ви не змогли б включити в програму відразу два поля, що визначають розмір шрифта: currentFont.sizelnPixels (розмір шрифту в пікселах) і currentFont.sizelnPoints (розмір шрифту в пунктах), - бо тоді структура currentFont не змогла б дізнатися, яке з них використовувати. Крім того, змінюючи розміри шрифтів у декількох місцях, ви поширили б схожі рядки по всій програмі.

Для вибору напівжирного ви могли б написати код, що використовує логічне АБО і шістнадцяткову константу 0x02:

currentFont.attribute = currentFont.attribute or 0x02

Цей код можна трохи поліпшити, але найкраще, що ви отримаєте, використовуючи спеціалізований підхід, буде схоже на:

currentFont.attribute = currentFont.attribute or BOLD

або на що-небудь таке:

currentFont.bold = True

Як і у випадку з розміром шрифту, проблема тут у тому, що клієнтський код повинен контролювати елементи даних безпосередньо, а це обмежує число можливих способів застосування структури currentFont.

Такий підхід до програмування сприяє поширенню подібні рядків коду по всій програмі. Проблема не в тому, що спеціалізований підхід - погана методика програмування. Просто ви можете замінити його на кращу методику, переваги якої описані нижче.

Переваги використання АТД:

1. Можливість приховати деталі реалізації. Приховування інформації про типи даних шрифту має на увазі, що при необхідності зміни типу даних ви зможете змінити його в одному місці, не впливаючи на всю програму. Наприклад, якщо ви не приховаєте деталі реалізації в АТД, то при зміні одного виду представлення напівжирного шрифту на інший вам доведеться змінити кожен фрагмент коду, в якому задається напівжирний шрифт. Приховування інформації захистить іншу частину програми і в тих випадках, якщо ви

вирішите зберігати дані в зовнішньому сховищі, а не в пам'яті або переписати всі методи, які виконують операції над шрифтами, на іншій мові.

2. Обмеження області змін. Якщо ви захочете урізноманітнити шрифти і реалізувати для них додаткові операції (такі як перемикання на надрядковий шрифт, перекреслення і т. д.), ви зможете змінити один фрагмент коду, і це не вплине на решту програми.

3. Більш висока інформативність інтерфейсу. Код currentFont.size = 16 неоднозначний, так як число 16 може визначатирозмір шрифту і в пікселах, і в пунктах. Контекст про це нічого не говорить. Об'єднання всіх схожих операцій у АТД дозволяє визначити весь інтерфейс в термінах пунктів, в термінах пікселів або чітко розділити обидва варіанти, допомагаючи уникнути плутанини.

4. Простота оптимізації коду. Для підвищення швидкодії операцій над шрифтами ви зможете переписати кілька чітко визначених методів, а не блукати по всій програмі.

5. Простота перевірки коду. Нудну перевірку правильності команд виду *currentFont.attribute = currentFont.attribute or 0x02*ви зможете замінити більш простою перевіркою правильності викликів *currentFontSetBoldOn ()*. У першому випадку можна вказати невірне ім'я структури, невірне ім'я поля, невірну операцію (and замість or) або невірне значення атрибута 0x20 замість 0x02). У разі виклику*currentFont.SetBoldOn ()*помилкою може бути лише вказівка невірного імені методу, так що помітити її легше.

6. Простота розуміння коду. Команду виду*currentFont.attribute or 0x02*можна поліпшити, замінивши 0x02 на BOLD (або що там представляє константа 0x02), але навіть після цього по легкості читання вона не зрівняється з викликом методу *currentFont.SetBoldOn ()*. Вудфілд, Дансмор і Шен провели дослідження, учасники якого - аспіранти та студенти старших курсів факультету інформатики - повинні були відповісти на питання про дві програми: одна була розділена на вісім методів у функціональному стилі, а друга - на вісім методів АТД (Woodfield, Dunsmore, and Shen, 1981). Студенти, які відповідали на питання про другу програму, отримали оцінки вищі на 30%.

7. Обмеження галузі використання даних. У щойно представлених прикладах структур*currentFont*потрібно змінювати безпосередньо або передавати в кожен метод, який працює зі шрифтами. При використанні АТД вам не довелося б ні передавати її в методи, ні перетворювати на глобальні дані. АТД просто включав би структуру, що містить дані*currentFont.* Прямий доступ до цих даних мали б лише методи зі складу АТД, але не які б то не було інші методи.

Можливість роботи з сутностями реального світу, а не з низькорівневими деталями реалізації АТД дозволяє визначити операції над шрифтами так, що більша частина програми буде сформульована виключно в термінах шрифтів, а не доступу до масивів, визначень структур або значень*True*і *False.*

У нашому випадку в АТД можна було б включити методи:

currentFont.SetSizeInPoints (sizelnPoints)

currentFont.SetSizeInPixels (sizelnPixels)

currentFont.SetBoldOn ()

currentFont.SetBoldOff ()

currentFont.SetltalicOn ()

currentFont.SetltalicOff ()

currentFont.SetTypeFace (faceName)

Ці методи, ймовірно, були б короткими - мабуть, вони нагадували б код, наведений при обговоренні спеціалізованого підходу до управління шрифтами. Різниця двох підходів в тому, що, використовуючи АТД ви ізолюєте операції над шрифтами в наборі методів, який надає іншим частинам програми, що працюють з шрифтами, покращений рівень абстракції і захищає решту коду від змін операцій над шрифтами.

Приклади АТД

Припустимо, ви розробляєте додаток, який здійснює управління системою охолодження ядерного реактора. З системою охолодження можна працювати як з АТД, визначивши наступні операції:

coolingSystem.GetTemperature ()
coolingSystem.SetCirculationRate (rate)
coolingSystem.OpenValve (valveNumber)
coolingSystem.CloseValve (valveNumber)

Конкретна реалізація даних операцій залежала б від конкретного середовища. Решта фрагментів програми взаємодіяли б із системою охолодження за допомогою цих методів і могли б не турбуватися про внутрішні деталі реалізації структур даних, їх обмеження, зміни і т. д.

Додаткові приклади абстрактних типів даних і операцій, які можна було б для них визначити наведені на рис. 4.1.

Принципи використання АТД

1. Уявляйте у формі АТД поширені низькорівневі типи даних.Зазвичай при обговоренні АТД мова йде про подання у формі АТД популярних низькорівневих типів даних. Як можна помітити з прикладів, у формі АТД можна уявити стек, список, чергу і майже будь-який інший популярний тип даних. Якщо стек представляє набір співробітників, розглядайте АТД як набір співробітників, а не як стек. Якщо список відповідає набору рахунків, розглядайте його як набір рахунків. Використовуйте якомога вищий рівень абстракції.

2. Уявляйте у формі АТД навіть прості елементи. Одним з АТД в списку прикладів (рис. 4.1) був ліхтар, що підтримує тільки дві операції: увімкнення і вимкнення. Може здаватися, що створювати для операцій «увімкнути» і «вимкнути» окремі методи недоцільно. Однак, насправді АТД вигідно використовувати навіть у разі найпростіших операцій. Представлення ліхтаря і його операцій у формі АТД полегшує розуміння і зміну коду, обмежує потенційні слідства змін методів*TurnLightOn ()*і*TurnLightOff ()*і знижує число елементів даних, які потрібно передавати у методи.

Система регулювання швидкості	**Кавомолка**	**Паливний бак**
Встановити швидкість	Ввімкнути	Заповнити бак
Отримати поточні параметри	Вимкнути	Злити паливо
Відновити попередні значення швидкості	Встановити швидкість	Отримати ємкість паливного баку
Відключити систему	Почати перемелювання кави	Отримати статус паливного баку
	Припинити перемелювання кави	

Список		**Стек**
Ініціалізувати список	**Ліхтар**	Ініціалізувати стек
Додати елемент	Ввімкнути	Помістити елемент в стек
Видалити елемент	Вимкнути	Видалити елемент зі стеку
Прочитати наступний елемент		Прочитати верхній елемент стеку

Система довідкової інформації	**Меню**	**Файл**
Додати розділ	Створити нове меню	Відкрити файл
Видалити розділ	Знищити меню	Прочитати файл
Встановити поточний розділ	Додати в меню новий елемент	Записати файл
Відобразити вікно довідкової системи	Видалити елемент меню	Встановити покажчик файлу
Знищити вікно довідкової системи	Активувати елемент меню	Закрити файл
Відобразити показчик інформаційних розділів	Деактивувати елемент меню	
Повернутися до попереднього розділу	Відобразити меню	**Ліфт**
	Приховати меню	Перемістится на один поверх вгору
Покажчик	Отримати індекс обраного елементу меню	Перемістится на один поверх вниз
Виділити блок пам'яті		Перемістится на конкретний поверх
Звільнити блок пам'яті		Повідомити поточний номер поверху
Змінити об'єм виділеної пам'яті		Повернутися на перший поверх

Рисунок 4.1 – Прикдади АТД

3. Звертайтесь до АТД так, щоб це не залежало від середовища, яке використовується для їх збереження. Припустимо: таблиця страхових тарифів настільки велика, що її потрібно завжди зберігати на диску. Можливо представити її як «тарифів (rate *file)»* та створити такі методи доступу, як*RateFile.Read ().*Однак, посилаючись на таблицю як на файл, ви повідомляєте про неї більше інформації, ніж варто було б. Якщо ви коли-небудь зміните програму так, щоб таблиця зберігалася в пам'яті, а не на диску, код, який

звертається до неї як до файлу, стане некоректним. Тому намагайтеся присвоювати класам і методам доступу імена, які не залежать від способу збереження даних, і звертайтеся не до конкретних сутностей, а до АТД, таких як таблиця страхових тарифів. У даному випадку клас і метод доступу слід було б назвати *rateTable.Read ()* або просто *rates.Read ()*.

Робота з кількома примірниками даних при використанні абстрактних типів даних у середовищах, які не є об'єктно-орієнтованими

Об'єктно-орієнтовані мови автоматично підтримують роботу з кількома примірниками АТД. Якщо ви використовуєте С або іншу мову, яка не є об'єктно-орієнтованою, підтримку роботи з кількома примірниками даних потрібно реалізувати вручну. У цілому це означає, що ви маєте розробити для АТД сервіси, які можуть працювати з декількома екземплярами.

АТД «шрифт» спочатку пропонував такі сервіси:

currentFont.SetSize (sizelnPoints)
currentFont.SetBoldOn ()
currentFont.SetBoldOffO
currentFont.SetltalicOnO
currentFont.SetltalicOff ()
currentFont.SetTypeFace (faceName)

Усередовищі, щонеєоб'єктно-орієнтованим, ціметодинебулибпов'язанізкласомівиглядалибтак:

SetCurrentFontSize (sizelnPoints)
SetCurrentFontBoldOn ()
SetCurrentFontBoldOff ()
SetCurrentFontItalicOn ()
SetCurrentFontltalicOff ()
SetCurrentFontTypeFace (faceName)

Якбивихотілипрацюватизкількомашрифтамиодночасно, топовиннібулибстворитисервісистворенняатавилученняпримірниківшрифтівназразокцих:

CreateFont (fontId)
DeleteFont (fontId)
SetCurrentFont (fontId)

Ідентифікаторшрифту *fontId* дозволяєстежитизадекількомашрифтамипідчасїхстворенняівикористання. Щостосуєтьсяіншихоперацій, товцьомувипадкувиможетевибиратиодинзтрьохваріантівреалізаціїінтерфейсуАТД.

Варіант 1: явно вказувати примірник даних при кожному зверненні до сервісів АТД. У цьому випадку «поточний шрифт (current font)» не потрібен.У кожен метод, який працює зі шрифтами, ви передаєте *fontId*. Методи АТД Font стежать за всіма даними шрифту, а клієнтський код - лише за ідентифікатором *fontId*. Цей варіант вимагає, щоб кожен метод, який працює зі шрифтами, отримував додатковий параметр *fontId*.

Варіант 2: явно надавати дані, якими користуються сервіси АТД. В даному випадку ви маєте вказати потрібні АТД в кожному методі, що

використовує сервіс АТД. Інакше кажучи, ви створюєте тип даних Font, який передаєте в кожен із сервісних методів АТД. Ви повинні спроектувати сервісні методи АТД так, щоб вони використовували дані Font, передані в них при кожному виклику. При цьому клієнтський код не потребує ідентифікатора шрифту, тому що він стежить за даними шрифтів сам. (Хоча дані типу Font доступні безпосередньо, до них треба звертатися тільки через сервісні методи АТД. Це називається підтриманням структури «у закритому вигляді».)

Перевага цього підходу в тому, що сервісним методам АТД не доводиться переглядати інформацію про шрифт, спираючись на його ідентифікатор. Є і недолік: такий спосіб надає доступ до даних шрифту іншим частинам програми, через що підвищується ймовірність того, що клієнтський код буде використовувати деталі реалізації АТД, яким слід було б залишатися прихованими всередині АТД.

Варіант 3: використовувати неявні примірники (з великою обережністю). Ви повинні створити новий сервіс - скажімо,*SetCurrentFont (fontld),*- при виклику якого заданий примірник шрифту стає поточним. Після цього всі інші сервіси використовують поточний шрифт, завдяки чому в них не потрібно передавати параметр*fontId.* При розробці простих додатків такий підхід може полегшити використання декількох екземплярів даних. У складних додатках подібна залежність від стану в масштабі всієї системи. Тобто ви повинні стежити за поточним екземпляром шрифту у всьому коді, що викликає методи Font. Складність програми при цьому підвищується. Яким би не був розмір програми, завжди можна знайти більш вдалі альтернативи даному підходу.

Усередині АТД ви можете реалізувати роботу з декількома екземплярами даних як завгодно, але за межами АТД при використанні мови, яка не є об'єктно-орієнтованою, можливі тільки три зазначені варіанти.

Абстрактні типи даних і класи

Абстрактні типи даних лежать в основі концепції класів. У мовах, що підтримують класи, кожен АТД можна реалізувати як окремий клас. Проте зазвичай з класами пов'язують ще дві концепції: успадкування і поліморфізм. Можете розглядати клас як АТД, що підтримує спадкування і поліморфізм.

Якісні інтерфейси класів

1. Якісна абстракція. Інтерфейс класу - це абстракція реалізації класу. Інтерфейс класу повинен надавати групу методів, чітко узгоджуються один з одним. Розглянемо клас «співробітник». Він може містити такі дані, як прізвище співробітника, адресу, номер телефону і т. д., і пропонувати методи ініціалізації та використання цих даних. Приклад інтерфейсу, який формує якісну абстракцію (C + +)

```
class Employee {
  public:
// Відкриті конструктори і деструктори
        Employee ();
      Employee (
            FullName name,
            String address,
```

```
                    String workPhone,
                    String homePhone,
                    TaxId taxIdNumber,
                    JobClassification jobClass
                    );
            virtual ~ Employee ();
// Відкритіметоди
            FullName GetName () const;
            String GetAddress () const;
            String GetWorkPhone () const;
            String GetHomePhone () const;
            TaxId GetTaxIdNumber () const;
            JobClassification GetJobClassification () const;
            ...
        private:
            ...
        };
```

Запропонованаінтерфейсомцьогокласуабстракціячудова, томущовсіметодиінтерфейсуслужатьєдинійузгодженійметі. Інтерфейс, щопредставляєпогануабстракцію, містивбинабіррізноріднихметодів.

Приклад інтерфейсу, що формує погану абстракцію (C + +):

```
    class Program {
    public:
            ...
// Відкритіметоди
            void InitializeCommandStack ();
            void PushCommand (Command command);
            Command PopCommand ();
            void ShutdownCommandStack ();
            void InitializeReportFormatting ();
            void FormatReport (Report report);
            void PrintReport (Report report);
            void InitializeGlobalData ();
            void ShutdownGlobalData ();
            ...
    private:
            ...
    };
```

Схоже, цейкласміститьметодироботизістекомкоманд, форматуваннязвітів, друкузвітівіініціалізаціїглобальнихданих. Важко побачити зв'язок між стеком команд, обробкою звітів і глобальними даними. Інтерфейс такого класу не формує узгоджену абстракцію, і клас має погану зв'язність. У даному випадку методи слід реорганізувати в більш чіткі класи, інтерфейси яких представлятимуть більш вдалі абстракції.

Якби ці методи були частиною класу *Program,* для формування узгодженої абстракції їх можна було б змінити так:

Приклад інтерфейсу, формуючого більш вдалу абстракцію (C + +)

```
class Program {
public:
        ...
/ /Відкритіметоди
        void InitializeUserInterface ();
        void ShutDownUserlnterface ();
        void InitializeReports ();
        void ShutDownReports ();
        ...
private:
        ...
};
```

Уходіочищенняінтерфейсуоднійогометодибулипереміщенівбільшвідповідникласи, аіншібулиперетворенівзакритіметоди, використовуваніметодом*InitializeUserlnterface ()*іншимиметодами.

Даний спосіб оцінки абстракції класу заснований на вивченні відкритих методів класу, тобто його інтерфейсу.Однак, з того, що клас в цілому формує якісну абстракцію, зовсім не випливає, що його окремі методи також представляють вдалі абстракції.

Принципи створення якісної абстракції:

1. Висловлюйте в інтерфейсі класу узгоджений рівень абстракції. Кожен клас повинен бути реалізацією тільки одного АТД. Якщо клас реалізує більше одного АТД або якщо ви не можете визначити реалізацією якого АТД є клас, саме час реорганізувати клас в один або декілька якісних АТД.

Наступний клас має неузгоджений інтерфейс, тому що рівень абстракції, який формується їм, непостійний:

Приклад інтерфейсу, що включає різні рівні абстракції (C + +)

```
class EmployeeCensus: public ListContainer {
public:
        ...
/ /Відкритіметоди
/ /Абстракція, формованацимиметодами, відноситьсядорівня«employee» (співробітник).
        void AddEmployee (Employee employee);
        void RemoveEmployee (Employee employee);
/ /Абстракція, формована цими методами, відноситься до рівня«list» (список).
        Employee NextltemlnlnList ();
        Employee Firstltem ();
        Employee Lastltem ();
        ...
private:
        ...
```

```
};
```

ЦейкласпредставляєдваАТД: *Employee*і*ListContainer* (список-контейнер).Подібні змішані абстракції часто виникають, коли програміст реалізує клас за допомогою класу-контейнеру або інших бібліотечних класів і не приховує цей факт.Запитайте себе, чи повинна інформація про використання класу-контейнера бути частиною абстракції.Зазвичай це є деталлю реалізації, яку слід приховати від інших частин програми, наприклад так:

Приклад інтерфейсу, що формує узгоджену абстракцію (C + +)

```
class EmployeeCensus {
public:
        ...
// Відкритіметоди
//  Абстракція, формованаусімацимиметодами, тепервідноситьсядорівня«employee».
        void AddEmployee (Employee employee);
        void RemoveEmployee (Employee employee);
        Employee NextEmployee ();
        Employee FirstEmployee ();
        Employee LastEmployee ();
        ...
private:
// Той факт, що клас використовує бібліотеку ListContainer, тепер прихований.
        ListContainer m_EmployeeList;
        ...
};
```

Програмісти можуть стверджувати, що спадкування від *ListContainer* зручно, тому що воно підтримує поліморфізм, дозволяючи створити зовнішній метод пошуку або сортування, приймаючий об'єкт *ListContainer*. Але цей аргумент не проходить головний тест на доречність спадкування: «Чи використовується спадкування тільки для моделювання відносини "є"?». Спадкування класу *EmployeeCensus* (каталог особистих справ співробітників) від класу *ListContainer* означало б, що *EmployeeCensus* «Є» *ListContainer*, що, очевидно, невірно. Якщо абстракція об'єкта *EmployeeCensus* полягає в тому, що він підтримує пошук або сортування, ці можливості повинні бути явними узгодженими частинами інтерфейсу класу.

Якщо уявити відкриті методи класу як люк, що запобігає потраплянню води в підводний човен, неузгоджені відкриті методи - це щілини.Вода не буде протікати через них так швидко, як через відкритий люк, але пізніше човен все ж потоне.На практиці при змішуванні рівнів абстракції саме це і відбувається. В міру змін програми змішані рівні абстракції роблять її все менш і менш зрозумілою, поки код не стане зовсім загадковим.

2. Переконайтесь, що ви розумієте, реалізацією якої абстракції є клас. Деякі класи дуже схожі, тому при розробці класу потрібно розуміти, яку абстракцію повинен представляти його інтерфейс.

3. Надавайте методи разом з протилежними їм методами. Більшість операцій має відповідні протилежні операції. При проектуванні класу перевірте кожен відкритий метод на предмет того, чи потрібно вам його протилежність.Створювати протилежні методи, не маючи на те причин, не слід, але перевірити їх доцільність потрібно.

4. Прибирайте сторонню інформацію в інші класи.Іноді ви можете виявити, що одні методи класу працюють з однією половиною даних, а інші - з другою. Це означає, що ви маєте справу з двома класами, що ховаються під маскою одного.

5. Робіть інтерфейси програмними, а не семантичними. Кожен інтерфейс складається з програмної та семантичної частин.Перша містить типи даних та інші атрибути інтерфейсу, які можуть бути перевірені компілятором.Друга складається з припущень про використання інтерфейсу, які компілятор перевірити не може. Семантичний інтерфейс слід документувати в коментарях, але взагалі інтерфейси повинні якомога менше залежати від документації.Будь-який аспект інтерфейсу, який не може бути перевірений компілятором, є потенційним джерелом помилок.Намагайтеся перетворювати семантичні елементи інтерфейсу в програмні, використовуючи твердження (assertions) або іншими способами.

Приклад інтерфейсу, що робить неможливим супровод програми (C + +).

```
class Employee {
public:

/ /Відкритіметоди
        FullName GetName () const;
        Address GetAddress () const;
        PhoneNumber GetWorkPhone () const;
        ...
        bool IsJobClassificationValid (JobClassification jobClass);
        bool IsZipCodeValid (Address address);
        bool IsPhoneNumberValid (PhoneNumber phoneNumber);

        SqlQuery GetQueryToCreateNewEmployee () const;
        SqlQuery GetQueryToModifyEmployee () const;
        SqlQuery GetQueryToRetrieveEmployee () const;
        ...
private:
      ...
};
```

Таким чином ясна абстракція, перетворилась на суміш майже неузгоджених методів. Між співробітниками і методами, які перевіряють коректність поштового індексу, номера телефону або ставки зарплати (job classification), немає логічного зв'язку. Методи, що мають доступ до деталей SQL-запитів, відносяться до набагато більш низькому рівню абстракції, ніж клас *Employee,* порушуючи загальну абстракцію класу.

6.Уникайте порушення цілісності інтерфейсу при зміні класу.

7. Не додавайте до класу відкриті члени, які погано узгоджуються з абстракцією інтерфейсу.

8. Розглядайте абстракцію і зв'язність разом. Поняття абстракції і зв'язності (cohesion) тісно пов'язані: інтерфейс класу, що представляє якісну абстракцію, зазвичай відрізняється високою зв'язністю.І навпаки: класи, що мають високу зв'язність, зазвичай представляють якісні абстракції.

2. Якісна інкапсуляція. Принципи якісної інкапсуляції:

1. Мінімізуйте доступність класів та їх членів. Мінімізація доступності - одне з головних правил, що підтримують інкапсуляцію.Якщо ви невпевнені, яким слід зробити конкретний метод: відкритим, закритим або захищеним треба обирати найсуворіший з можливих рівнів захисту (Meyers, 1998; Bloch, 2001).

2. Не робіть дані-члени відкритими.

Надання доступу до даних-членам порушує інкапсуляцію та обмежує контроль над абстракцією.Як вказує Артур Ріель, клас *Point* (крапка), який надає доступ до даних:

float x;
floaty;
float z;

порушуєінкапсуляцію, томущоклієнтськийкодможевільноробитизданими Point щозавгодно, прицьомусамкласможенавітьнедізнатисяпроїхзміну (Riel, 1996).У той же час клас *Point,* що містить члени:

floatGetX ();
float GetY ();
float GetZ ();
void SetX (float x);
void SetY (float y);
void SetZ (float z);

підтримуєчудовуінкапсуляцію. Винемаєтеуявленняпроте, чиреалізованіданіяк*float x, yiz,*чизберігаєклас*Point*цієлементиях*double,*перетворюючиїху*float,*абожвінзберігаєїхнаМісяцііотримуєчерезсупутник.

3. Інтерфейс класу не повинен містити закриті деталі реалізації.

Приклад оприлюднення деталей реалізації класу (C + +).

```
class Employee {
    public:
        ...
        Employee (
        FullName name,
        String address,
        String workPhone,
        String homePhone,
        Taxld taxIdNumber,
        JobClassification jobClass
```

```
            );
            FullName GetNameO const;
            String GetAddressO const;
            ...
        private:
/ /Оприлюдненідеталіреалізації.
            String m_Name;
            String m_Address;
            int m_jobClass;
            ...
        };
```

ЗагальнийспосібвирішенняцієїпроблемиописавСкоттМейерсукнизі «Effective C + +, 2d ed» (Meyers, 1998). Відокремте інтерфейс класу від його реалізації, після чого оголошення классу має містити покажчик на його реалізацію, але не містити інших деталей реалізації.

Приклад приховування деталей реалізації класу (C + +).

```
    class Employee {
    public:
            ...
Employee (...);
            ...
            FullNane GetName () const;
            String GetAddress () const;
            ...
        private:
/ /Деталіреалізаціїприхованізадопомогоюпокажчика.
            EmployeeImplementation * m_implementation;
        };
```

Тепервиможетепомістититидеталіреалізаціївклас*EmployeeImplementation,*як ийбудедоступнийтількикласу*Employee.*

4. Не робіть припущень стосовно клієнтів класу. Клас слід спроектувати і реалізувати так, щоб він дотримувався контракту, сформульованого за допомогою інтерфейсу.Висловивши свої вимоги в інтерфейсі, клас не повинен робити припущень про те, як цей інтерфейс буде чи не буде використовуватись. Подібні коментарі вказують на те, що клас вимагає від своїх клієнтів більше, ніж слід:

-Ініціалізується x, y і z значенням 1.0, тому що

-При ініціалізації значенням 0.0 DerivedClass не працює

5. Уникайте використання дружніх класів. Іноді - наприклад, при реалізації шаблону State - дисципліноване використання дружніх класів допомагає керувати складністю (Gammaetal., 1995).

Проте зазвичай дружні класи порушують інкапсуляцію. Вони збільшують обсяг коду, підвищуючи тим самим складність програми.

6. Не робіть метод відкритим лише тому, що він використовує лише відкриті методи.

7. Цінуйте легкість читання коду більше, ніж зручність його написання. Навіть під час первісної розробки програми код доводиться читати набагато частіше, ніж писати.

8. Не використовуйте занадто жорстке сполучення. «Сполучення» (coupling) характеризує силу зв'язку між двома класами. Для мінімізації сполучення:

- мінімізуйте доступність класів та їх членів;
- уникайте дружніх класів;
- робіть дані базового класу закритими, а не захищеними: це послаблює сполучення похідних класів з базовим;
- не використовуйте дані-члени у відкритому інтерфейсі класу.

Проектування та реалізація класів

Для створення високоякісної програми недостатньо визначити вдалі інтерфейси класів - не менш важливо грамотно спроектувати і реалізувати внутрішній устрій класів. При цьому постатють питання, пов'язані з включенням, спадкуванням, методами / даними-членами, сполученням класів, конструкторами, а також об'єктами-значеннями і об'єктами-посиланнями.

Включення (відношення «містить»). Сутність включення (containment): один клас містить примітивний елемент даних або інший клас. При цьому включення - один з головних інструментів об'єктно-орієнтованого програмування. Включення можна розглядати як відношення «містить». Наприклад, об'єкт «співробітник» може «утримувати» прізвище, номер телефону, ідентифікаційний номер платника податків і т. д. Це відношення можна реалізувати, зробивши прізвище, номер телефону та номер платника податків даними-членами класу Employee. Якщо це необхідно, реалізуйте ставлення «містить» за допомогою закритого спадкування. Іноді включення не можливо реалізувати, роблячи один об'єкт членом іншого.Деякі експерти радять при цьому виконувати закрите успадкування класу-контейнера від класу, який повинен в ньому міститися (Meyers, 1998; Sutter, 2000). На практиці цей підхід встановлює занадто близькі відносини між дочірнім і батьківським класом, порушуючи інкапсуляцію.Зазвичай це вказує на помилки проектування, які слід вирішити інакше, не вдаючись до закритого спадкоємства.

При реалізації включення обережно використовуйте класи, які містять більше семи елементів даних-членів. При виконанні завдань людина може утримувати в пам'яті 7 ± 2 дискретних елементів (Miller, 1956). Якщо клас містить більше семи елементів даних-членів, подумайте, чи не розділити чи його на кілька менш великих класів (Riel, 1996). Орієнтуйтися на верхню межу діапазону «7 +2», якщо дані-члени є примітивними типами, такими як цілі числа і рядки, і на нижню, якщо вони є складними об'єктами.

*Спадкування (відношення «є»).*Мета спадкування - створити більш простий код, що досягається шляхом визначення базового класу, який ідентифікує загальні елементи двох або більше похідних класів.Спільними елементами можуть бути інтерфейси методів, їх реалізація, дані-члени або типи

даних.Спадкування допомагає уникати повторення коду й даних в декількох місцях програми, централізуючи їх у базовому класі.

Принципи ефективного спадкування:

1. Реалізуйте за допомогою відкритого спадкування ставлення «є». Базовий клас формулює очікування і обмеження, яким повинен буде відповідати похідний клас (Meyers, 1998).

Якщо похідний клас не має повністю дотримуватися правил, визначених інтерфейсом базового класу, спадкування виконувати не варто. Спробуйте замість цього застосувати включення або внести зміну на більш високому рівні ієрархії успадкування.

2. Розробляйте і документуйте класи з урахуванням можливості спадкування або забороніть його. Спадкування підвищує складність програми і може бути небезпечним.Тому гуру програмування на Java Джошуа Блох сказав: «Розробляйте і документуйте класи з урахуванням можливості спадкування або забороніть його». Якщо при проектуванні класу ви вирішили, що він не повинен підтримувати спадкування, то не оголошуйте його члени як *virtual* у разі C + + або *overridable* у разі MicrosoftVisualBasic; якщо ви програмуєте на Java, оголосіть члени такого класу як *final.*

3. Дотримуйтесь принцип підстановки Лісков (LiskovSubstitutionPrinciple, ISP). Барбара Лісков вважала, що спадкування варто використовувати, тільки якщо похідний клас дійсно «є» більш спеціалізованою версією базового класу (Liskov, 1988). Енді Хант і Дейв Томас сформулювали LSP так: «Клієнти повинні мати можливість використання підкласів через інтерфейс базового класу, не помічаючи ніяких відмінностей» (HuntandThomas, 2000).

Інакше кажучи, всі методи базового класу повинні мати в кожному похідному класі те ж значення.

Якщо у вас є базовий клас *Account* (рахунок) і похідні класи *CheckingAccount* (рахунок до запитання), *SavingsAccount* (депозитний рахунок) і *AutoLoanAccount* (рахунок позик), то при виклику яких би то не було методів класу *Account*в будь-якому з його підтипів програміст не повинен дбати про підтип конкретного об'єкту «рахунок».

При дотриманні принципу підстановки Лісков спадкування - потужний засіб зниження складності, що дозволяє програмісту зосередитися на загальних атрибутах об'єкта, що не хвилюючись про його деталях. Якщо ж програміст повинен постійно пам'ятати про семантичних відмінностях реалізацій підкласів, спадкування тільки підвищує складність.Так, у нашому прикладі програмісту довелося б думати: «Якщо я викликаю метод *InterestRate ()* (процентна ставка) класу *CheckingAccount* або *SavingsAccount,* він повертає відсоток, який банк виплачує клієнтові, однак метод *InterestRate ()* класу *AutoLoanAccount* повертає відсоток, що виплачується клієнтом банку, тому я повинен змінити знак результату ». Відповідно до LSP, в даному випадку клас *AutoLoanAccount* не повинен бути похідним від класу *Account,* тому що методи *InterestRate ()* в цих класах мають різні семантичні значення.

Переконайтеся, що ви наслідуєте тільки те, що хочете успадковувати Похідний клас може успадковувати інтерфейси методів-членів, їх реалізації або і те, й інше (рис. 4.2)

	Перевизначення методу можливе	Перевизначення методу неможливе
Реалізація за умовчанням є	Перевизначений метод	Неперевизначений метод
Реалізація за умовчанням немає	Абстрактний перевизначений метод	Цей варіант не використовується (немає сенсу в тому, щоб залишити метод без визначення, не дозволивши його перевизначити)

Рисунок 4.2 – Різновиди успадкованих методів

Як випливає з рис.4.2, успадковані методи можуть відноситися до однієї з трьох категорій:

абстрактний переобумовленої метод: похідний клас успадковує інтерфейс методу, але не його реалізацію;

перевизначений метод: похідний клас успадковує інтерфейс методу і його реалізацію за замовчуванням, а також може перевизначити цю реалізацію;

неперевизначений метод: похідний клас успадковує інтерфейс методу і його реалізацію за замовчуванням, перевизначити яку не може.

Створюючи новий клас за допомогою спадкування, обміркуйте тип спадкування кожного методу-члена.Чи не наслідуйте реалізацію тільки тому, що ви наслідуєте інтерфейс, і не наслідуйте інтерфейс тільки для того, щоб успадкувати реалізацію. Якщо вам потрібна реалізація класу, але не його інтерфейс, використовуйте включення, а не успадкування.

4. Переміщайте загальні інтерфейси, дані та форми поведінки на якомога вищий рівень ієрархії успадкування.

5. Намагайтесь не використовувати класів, об'єкти яких створюються в єдиному екземплярі. Використання єдиного примірника класу може вказувати на те, що ви сплутали об'єкти з класами.Подумайте, чи можна просто створити об'єкт замість нового класу.Чи можна конкретний похідний клас представити тільки даними, а не окремим класом?Шаблон Одинак (Singleton) - виняток з цього правила.

6. Намагайтесь не використовувати базових класів, які мають тільки один похідний клас.

Як правило, це говорить про невдале проектування базового класу. Припустимо, ви створили клас *Cat*, що включає метод *Scratchb ()*, але після виявили, що деякі коти позбавлені кігтів і не можуть дряпатися. Ви могли б успадкувати від класу *Cat* клас *ScratchlessCat,* перевизначивши в ньому метод *Scratcb ()* так, щоб він нічого не робив. Однак, цей підхід пов'язаний з низкою проблем:

- він порушує абстракцію (контракт інтерфейсу) класу *Cat*, змінюючи семантику його інтерфейсу;
- при розширенні на інші похідні класи цей підхід швидко стає некерованим. Що буде, коли ви знайдете кота без хвоста? Чи кота, який не ловить мишей? Чи кота, який не п'є молоко?У результаті у вас можуть з'явитися похідні класи *ScratcblessTaillessMicelessMilklessCat;*
- код, написаний за цією методикою, важко супроводжувати, бо з часом поведінка похідних класів починає сильно відрізнятися від інтерфейсів і форм поведінки базових класів.

Виправляти цю проблему слід у первісному класі *Cat.* Створіть клас *Claws* (кігті) і включіть його до класу *Cats.* Основа наших помилок - припущення, що всі коти дряпаються; запропонований спосіб дозволить усунути причину проблеми, а не боротися з її наслідками.

7. Уникайте багаторівневих ієрархій успадкування. Об'єктно-орієнтоване програмування підтримує масу способів управління складністю.Але використання будь-якого потужного засобу пов'язане з ризиком, тому деякі об'єктно-орієнтовані підходи часто підвищують складність замість того, щоб знижувати її. Артур Ріель у книзі «Object-OrientedDesignHeuristics» (Riel, 1996) пропонує обмежувати ієрархії спадкоємства максимум шістьма рівнями.

8. Віддавайте перевагу поліморфізму, а не великомасштабній перевірці типів.

Наявність в коді великого числа блоків case може вказувати на те, що програму краще було б спроектувати, використовуючи спадкування, хоча це вірно не завжди.

Приклад коду, який треба замінити викликом поліморфного методу (C + +).

```
switch (shape.type) {
        case Shape_Circle:
                shape.DrawCircle ();
                break;
        case Shape_Square:
                shape.DrawSquare ();
                break;
        ...
}
```

Тут методи *shape.DrawCircle ()* і *shape.DrawSquare ()* слід замінити на єдиний метод *shape.Draw (),* що підтримує малювання і кіл, і прямокутників.

З іншого боку, іноді блоки case служать для поділу по-справжньому різних видів об'єктів або форм поведінки.Так, наступний фрагмент цілком доречний в рамках об'єктно-орієнтованої програми:

Приклад коду, який не треба замінювати викликом поліморфного методу (C + +).

```
switch(ui.CommandO) {
        case Conmand_OpenFile:
                OpenFile ();
```

```
            break;
        case Command_Print:
            Print ();
            break;
        case Command_Save;
            Save ();
            break;
        case Command_Exit:
            ShutDown ();
           break;
       ...
    }
```

9. Робіть всі дані закритими, а не захищеними. Як говорить Джошуа Блох, «успадкування порушує інкапсуляцію» (Bloch, 2001).Виконуючи спадкування від класу, ви отримуєте привілейований доступ до його захищених методівта даних.Якщо похідному класу насправді потрібен доступ до атрибутів базового класу, включіть до базового класу захищені методи доступу.

10. Обережно використовуйте множинне спадкування.

Методи-члени і дані-члени.

1. Включайте в клас як можна менше методів.

В одному з дослідженьпрограм на C ++ було виявлено, що більшому числу методів у розрахунку на один клас відповідає більше число вад (Basili, Briand, and Melo, 1996). Однак, важливішими виявилися інші фактори, в тому числі багаторівневі ієрархії спадкоємства й велика кількість методів, що викликаються з класу. Розробляючи клас, прагніть до оптимальної відповідності між цими факторами і мінімальним числом методів.

2. Блокуйте неявно згенеровані методи та оператори, які вам не потрібні. Іноді деякі можливості, такі як створення об'єкта або його присвоювання, доцільно блокувати.Вам може здатися, що зробити це неможливо, тому що компілятор генерує ці операції автоматично.Однак, ви можете заборонити їх використання в клієнтському коді, оголосивши конструктор, оператор присвоювання або інший метод чи оператор як *private.*(Створення закритого конструктора - стандартний спосіб визначення класу-одиначки).

3. Мінімізуйте число різних методів, що викликаються класом. Число дефектів в коді класу статистично корелює із загальним числом методів, що викликаються класом (Basili, Briand, andMelo, 1996). Крім того, число дефектів в коді класу підвищується і при збільшенні числа класів, які в ньому використовуються. Ці концепції іноді називають «коефіцієнтом розгалуження по виходу (fanout)».

4. Мінімізуйте співпрацю класу з іншими класами. Безпосередні зв'язки досить небезпечні. При цьому зв'язки типу *account.ContactPerson ().DaytimeContactlnfo ().PhoneNumber ()* є ще більш небезпечними. У зв'язку з цим вчені сформулювали «Правило Деметри (LawofDemeter)» (LieberherrandHolland, 1989), яке свідчить, що Об'єкт А може викликати будь-які з власних методів. Якщо він створює Об'єкт В, він може викликати будь-які

методи Об'єкта В, але йому не слід викликати методи об'єктів, що повертаються Об'єктом В. У нашому випадку це означає, що виклик *account.ContactPerson ()* прийнятний, однак виклику *account.ContactPerson ().DaytimeContactlnfo ()* слід було б уникнути. Взагалі намагайтеся звести до мінімуму всі наступні показники:

- число видів створюваних об'єктів;
- число безпосередньо викликаються методів створених об'єктів;
- число викликів методів, що належать об'єктам, повернутим іншими створеними об'єктами.

Конструктори.

Поради щодо використання конструкторів майже не залежать від мови.

1. Намагайтесяініціалізувати всі дані-члени у всіх конструкторах.

Створюйте класи-одиначки за допомогою закритого конструктора Якщо ви хочете визначити клас, що дозволяє створити лише один об'єкт, скрійте всі конструктори класу і створіть статичний метод *Getlnstance (),* що надає доступ до єдиного екземпляру класу:

Приклад створення класу-одиначки за допомогою закритого конструктора (Java).

```
public class Maxld {
/ /Конструктори і деструктори
/ /Закритий конструктор.
        private Maxld () {
                ...
        }
        ...
/ /Відкриті методи
/ /Відкритий метод, який надає доступ до єдиного екземпляру класу.
        public static Maxld Getlnstance () {
                return m_instance;
        }
        ...
/ /Закриті члени
/ /Єдиний екземпляр класу.
        private static final Maxld m_instance = new Maxld ();
        ...
}
```

Закритийконструкторвикликаєтьсятількиприініціалізаціїстатичногооб'єкта*m_nstance*.Длязверненнядокласу-одиночці*Maxld*потрібнопростовикликатиметод*Maxld.Getlnstance ().*

2. Виконуйте повне копіювання, а не обмежене. Одним з головних аспектів роботи зі складними об'єктами є вибір типу їх копіювання: повного або обмеженого.Повна копія (deepcopy) - це почленована копія даних-членів об'єкта; обмежена копія (shallowcopy) зазвичай просто вказує або посилається на вихідний об'єкт, хоча конкретні значення «повного» і «обмеженого» копіювання можуть відрізнятися.

Метою створення обмежених копій зазвичай буває підвищення швидкодії програми.Однак, створення декількох копій великих об'єктів рідко призводить до помітного зниження швидкодії, хоча і виглядає естетично непривабливо.Підвищення складності навряд можна виправдати сумнівним поліпшенням швидкодії коду, тому, якщо не доведено зворотнє, краще виконувати повне копіювання.

Повні копії легше в реалізації та супроводі, ніж обмежені.При обмеженому копіюванні потрібно написати не тільки специфічний для об'єкта код, але й код підрахунку посилань, безпечного порівняння об'єктів, їх безпечного знищення і т.д.Такий код може бути джерелом помилок, тому без вагомої причини створювати його не слід.

КОНТРОЛЬНІ ПИТАННЯ:

1. Що таке абстрактні типи даних? Чому вони лежать в основі класів?
2. У чому полягають переваги використання АТД? Наведіть приклади використання АТД?
3. Які існують принципи використання АТД?
4. У чому полягають якісні абстракція та інкапсуляція?
5. Коли слід використовувати включення, а коли спадкування?
6. Наведіть основні принципи роботи з методами-членами і данини-членами класів?
7. Як ефективно використовувати конструктори?

Тема 5. Бібліотека стандартних шаблонів

STL – стандартна бібліотека шаблонів, яка складається з допоміжних засобів та контейнерів.

Серед допоміжних засобів STL розглянемо ітератори.

Ітератор – це об’єкт, який виконує у контейнері роль покажчика. За допомогою ітератора можна переміщатися всередині контейнера та отримувати доступ до окремих його елементів.

Для контейнерів існують наступні типи ітераторів (табл. 5.1).

Таблиця 5.1 - Типи ітераторів

Тип літератора	Опис
Iterator	Ітератор. При збільшенні значення ітератор переміщається до кінця контейнеру
const_ iterator	Константний ітератор. Змінити значення, на яке посилається ітератор неможливо.
Reverse_iterator	Зворотній ітератор. При збільшенні значення літератор переміщується до початку контейнера.
const_reverse_iterator	Константний зворотній ітератор. Змінити значення, на яке посилається ітератор неможливо.

Надати значення змінним дозволяють наступні методи (табл. 5.2).

Таблиця 5.2 - Методи для надання значення змінним

Метод	Опис	Прототип
begin ()	Повертає ітератор, який встановлений на перший елемент контейнера.	iterator begin (); const_ iterator begin () const;
cbegin ()	Повертає константний ітератор, який встановлений на перший елемент контейнера.	const_ iterator cbegin () const;
rbegin ()	Повертає зворотній ітератор, який встановлений на останній елемент контейнера.	revers_iterator rbegin (); const_ revers_iterator rbegin () const;
Crbegin ()	Повертає зворотній константний ітератор, який встановлений на останній елемент контейнера.	const_ revers_iterator crbegin () const;
end ()	Повертає ітератор, який встановлений на позицію після останнього елемента контейнера.	iterator end (); const_ iterator end () const;
cend ()	Повертає константний ітератор, який встановлений на позицію після останнього елемента контейнера.	const_ iterator cend () const;
rend ()	Повертає зворотній ітератор, який встановлений на позицію перед першим елементом контейнера.	revers_iterator rend (); const_ revers_iterator rend () const;
crend ()	Повертає константний зворотній ітератор, який встановлений на позицію перед першим елементом контейнера.	const_ revers_iterator crend () const;

Над ітераторами виконуються ті самі дії, що й над покажчиками.
Контейнери у бібліотеці STL поділяються на три типи:

- послідовні (deque, list, vector);
- асоціативні(map, multimap, set, multiset);
- контейнери-адаптери (priority_queue, queue, stack).

Усі контейнери є динамічними.
Розглянемо клас deque- двостороння черга.Перед використанням класу необхідно додати# include<deque>

Створення об’єкта класу deque

Існують наступні способи створити екземпляр класу deque:

1. Оголосити екземпляр класу без ініціалізації.

std :: deque <int> d;

2. Вказати кількість елементів

std :: deque<in>d (5); // 5 елементів dequeіззначенням 0
std :: deque<int>d (5, 1); // 5 елементів dequeіззначенням 1

3. Вказати об’єкт класу всередині круглих дужок або після =

```
std :: deque <int> d1 (5,1);
std :: deque <int> d2 (d1);
std :: deque <int> d3=d1;
```

4. Вказати діапазон всередині контейнера за допомогою ітераторів

```
std :: deque <int> d1 (5,1);
std :: deque <int> d2 (d1.begin (), d1.end () );
```

Над елементами dequeвизначені дії ==, !=, <, <=, >, >=.

Вставка елементів (табл. 5.3).

Таблиця 5.3 - Методи вставки елементів у deque

Метод	Опис	Прототип
push_back()	Додає елемент у кінець черги	voidpush_back (const T &Val);
push_front()	Додає елемент у початок черги	voidpush_front (const T &Val);
insert()	Додає елемент у визначену позицію	iterator insert (constiterator); void insert (const_iterator Where, size_type Count, const T &Val); template <class It> void insert (const_iterator Where,It First, It Last);
swap()	Змінює елементи двох контейнерів місцями	void swap (deque&Right);

Видалення елементів (табл. 5.4).

Таблиця 5.4 - Методи видалення елементів з deque

Метод	Опис	Прототип
pop_back ()	Видаляє останній елемент	voidpop_back ();
pop_front ()	Видаляє перший елемент	voidpop_front ()
erase ()	Видаляє один елемент або елементи з діапазону	iterator erase (const_iterator Where);
clear ()	Видаляє усі елементи	void clear ();
empty ()	Повертає 1, якщо контейнер не містить елементів та 0 – у зворотному випадку	void empty () const;
size ()	Повертає кількість елементів контейнера	size_type size () const;
max_size ()	Повертає максимальну кількість елементів контейнера	size_type max_size () const;

Доступ до елементів (табл. 5.5).

Таблиця 5.5 - Методи доступу до елементів deque

Метод	Опис	Прототип
at ()	Повертає посилання на елемент, що знаходиться за індексом Pos	reference at (size_type Pos); const_reference at(size_type Pos) const;
front ()	Повертає посилання на перший елемент	reference front (); const_reference front () const;
back ()	Повертає посилання на останній елемент	reference back (); const_reference back () const;

*Приклад 1.*Порівняння двох об'єктів deque.

```
std :: deque <int> d1 ( 5, 1 );
std :: deque <int> d2 ( d1.begin(), d1.end() );
if ( d1 == d2 )
{
        std :: cout << ”d1=d2” << std :: endl;
}
```

*Приклад 2.*Присвоєння одного об'єкта іншому.

```
std :: deque <int> d1 ( 3, 1 ), d2;
std :: deque <int>:: const_iterator i;
d2 = d1;
for ( i = d2.begin(); i != d2.end(); ++ i)
{
        std :: cout << *i << ” ”;

} // 1 1 1
```

*Приклад 3.*Використання методу assign ().

void assign (size_type Count, const T &Val); // Видаляє існуючі елементи, додає Count елементів Val.

template <class It>

void assign (It First, It Last);// Видаляє існуючі елементи, додає елементи з діапазону Fist, Last.

```
std :: deque <int> d1 ( 2, 1 );
std :: deque <int>:: const_iterator i;
d1.assign ( 3, 5 );
for ( i=d2.begin(); i != d2.end(); ++ i)
{
        std::cout << *i<<” ”;

} // 5 5 5

std :: deque <int> d1 ( 2, 1 );
std :: deque <int> d2 ( 3, 5 );
```

```
std :: deque <int> :: const_iterator i;
d1.assign ( d2.begin(), d2.end() );
for ( i = d1.begin(); i != d1.end(); ++ i)
{
        std::cout << *i << ” ”;

} // 5 5 5
```

*Приклад 4.*Використання методу push_back ().

```
std :: deque <int> d ( 2, 1 );
std :: deque <int> :: const_iterator i;
d.push_back ( 5 );
for ( i = d.begin(); i != d.end(); ++i)
{
        std :: cout << *i<< ” ”;

} // 1 1 5
```

*Приклад 5.*Використання методу push_front().

```
std :: deque <int> d (2,1);
std :: deque <int>:: const_iterator i;
d.push_ front (5);
for (i=d.begin(); i!=d.end(); ++i)
{
        std::cout <<*i<<” ”;
}// 5 1 1
```

*Приклад 6.*Використання методу insert ().

```
std :: deque <int> d ( 2, 1 );
std :: deque <int> :: const_iterator i;
std :: deque <int> :: iterator it = d.begin();
++ it;
d.insert ( it, 5 );
for ( i = d.begin(); i != d.end(); ++ i)
{
        std :: cout << *i << ” ”;
} // 1 5 1

std :: deque <int> d ( 2, 1);
std :: deque <int> :: const_iterator i;
std :: deque <int> :: iterator it = d.begin();
++ it;
d.insert ( it, 3, 5 );
for ( i = d.begin(); i != d.end(); ++ i)
{
        std :: cout << *i << ” ”;
} // 1 5 5 5 1
```

```
std :: deque <int> d1 ( 2, 1 );
std :: deque <int> d2 ( 2, 5);
std :: deque <int> :: const_iterator i;
std :: deque <int> :: iterator it = d1.begin();
++ it;
d1.insert ( it, d2.begin(), d2.end() );
for ( i = d1.begin(); i != d1.end(); ++ i)
{
        std :: cout << *i << ” ”;
} // 1 5 5 5 1
```

*Приклад 7.*Використання методу swap ().

```
std :: deque <int> d1 ( 2, 1 ), d2 ( 3, 5 );
std :: deque <int> :: const_iterator i;
d1.swap ( d2 );
for ( i = d1.begin(); i != d1.end(); ++ i)
{
        std :: cout << *i << ” ”;

} // 5 5 5
std::cout << std :: endl;
for ( i = d2.begin(); i != d2.end(); ++ i)
{
        std :: cout << *i << ” ”;

} // 1 1
```

*Приклад 8.*Використання методу pop_back ().

```
std :: deque <int> d;
std :: deque <int> :: const_iterator i;
d.push_back ( 1 );
d.push_back ( 2 );
d.pop_back ();
for ( i = d.begin(); i != d.end(); ++ i)
{
        std :: cout << *i << ” ”;
} // 1
```

*Приклад 9.*Використання методу pop_front ().

```
std :: deque <int> d;
std :: deque <int> :: const_iterator i;
d.push_back ( 1 );
d.push_back ( 2 );
d.pop_front ();
for ( i = d.begin(); i != d.end(); ++ i)
{
        std :: cout << *i << ” ”;
```

```
} // 2
```
*Приклад 10.*Використання методу erase ().
```
std :: deque <int> d;
std :: deque <int> :: const_iterator i, it;
d.push_back ( 1 );
d.push_back ( 2 );
it = --d.end ();
d.erase ( it );
for ( i = d.begin(); i != d.end(); ++ i)
{
        std :: cout << *i << ” ”;
} // 1

std :: deque <int> d;
std :: deque <int> :: const_iterator i;
d.push_back ( 1 );
d.push_back ( 2 );
d.push_back ( 3 );
d.erase (d.begin(), -- d.end());
for ( i = d.begin(); i != d.end(); ++ i)
{
        std :: cout << *i << ” ”;
} // 3
```
*Приклад 11.*Використання методів clear (), empty ().
```
std :: deque <int> d;
d.push_back ( 1 );
d.push_back ( 2 );
d.push_back ( 3 );
d. clear ();
if ( d.empty () )
{
        std :: cout << “Немає елементів”<< std :: endl;
}
```
*Приклад 12.*Використання методу size ().
```
std :: deque <int> d;
d.push_back ( 1 );
d.push_back ( 2 );
d.push_back ( 3 );
std :: cout << d.size () << std :: endl; // 3
```
*Приклад 13.*Використання методу max_size ().
```
std :: deque <int> d1;
std :: cout << d1.max_size () << std :: endl; // 1073741823
std :: deque <char> d2;
std :: cout << d2.max_size () << std :: endl; // 4294967295
```
*Приклад 14.*Використання методу resize ().

```
std :: deque <int> d;
std :: deque <int> :: const_iterator i;
d.push_back ( 1 );
d.push_back ( 2 );
d.push_back ( 3 );
d. resize ( 2 );
for ( i = d.begin(); i != d.end(); ++ i)
{
        std :: cout << *i << ” ”;
} // 1 2
std :: cout << std :: endl;
d. resize ( 5, 0 );
for ( i = d.begin(); i != d.end(); ++ i)
{
        std :: cout << *i << ” ”;
} // 1 2 0 0 0
```

*Приклад 15.*Доступ до елементів deque за індексом.

```
std :: deque <int> d (2, 1);
std ::cout<<d[ 0 ] << “ ” << d[ 1 ] << std :: endl; // 1 1
d[ 1 ] = 2;
std :: cout <<d[0] << “ ”<< d[1] << std :: endl; // 1 2
```

*Приклад 16.*Використання методу at ().

```
std :: deque<int>d (2, 1);
std :: cout <<d.at ( 0 ) << “ ” << d. at ( 1 ) << std :: endl; // 1 1
d.at ( 1 ) = 2;
std :: cout <<d.at ( 0 ) << “ ” << d. at ( 1 ) << std :: endl; // 1 2
```

*Приклад 17.*Використання методу front ().

```
std :: deque <int> d (2, 1);
d.front () = 5;
std :: cout<<d.front () <<std :: endl; // 5
std :: cout<<d[ 0 ] << “ ” <<d[ 1 ] <<std :: endl; // 5 1
```

Приклад 18. Використання методу back ().

```
std :: deque<int>d ( 2, 1 );
d. back () = 5;
std :: cout << d. back () << std :: endl; // 5
std :: cout << d[ 0 ] << “ ” << d[ 1 ] << std :: endl; // 1 5
```

*Приклад 19.*Перебір елементів deque за допомогою ітераторів.

```
# include <iostream>
# include <deque>

int main ()
{
        std :: deque <int> d;
        std :: deque <int> :: const_iterator it1;
        std :: deque <int> :: reverse_iterator it2;
```

```
        for ( int i = 1; i <= 10; ++i) d.push_back ( i );
        it1 = d.begin ();
        *it1 = 800;                    //Зміна значення
        // Перебір елементів у прямому порядку
        for( it1 = d.begin (); it1 != d.end (); ++ it1)
        {
            std :: cout << *it1 << std :: endl;
        }
        // Перебір елементів у оберненому порядку
        for ( it2 = d.rbegin (); it2 != d.rend (); ++ it2)
        {
            std :: cout << *it2 << std :: endl;
        }
        std :: cin.get ();
        return 0;
    }
```

Розглянемо клас **vector.** Перед використанням класу необхідно додати# include<vector>

Створення об'єкта класу vector

Способи створення екземплярів класу vectoraналогічні способам створення екземплярів класуdeque.

Над елементами vector визначені дії ==, !=, <, <=, >, >=.

Вставка елементів

Методи вставки елементів у екземпляр класу vectoraналогічні методам вставки елементів у екземпляр класуdeque.Але vector не має методу push_front().

Видалення елементів

Методи видалення елементів з екземпляру класу vector аналогічні методам видалення елементів з екземпляру класуdeque.Але vector не має методу pop_front ().

Доступ до елементів

Методи доступу до елементів екземпляру класу vectoraналогічні методам доступу до елементів екземпляру класуdeque. Крім того доступ до елементів може виконуватись за допомогою ітераторів.

*Приклад 20.*Доступ до елементів vector за допомогою покажчиків.

```
# include <iostream>
# include <vector>

int main ()
{
    std :: vector<int> v;
    std :: vector<int> :: pointer p;
    std :: vector<int> :: const_pointer cp;
    v.reserve ( 10 );
```

```
        for ( int i = 1; i <= 10; ++ i)
                v.push_back ( i );
        p = v.data ();
        *p = 800;
        cp = v.data ();
        for ( int j = 0; c = v.size (); j < c; ++ j; ++ cp)
        {
                std :: cout << *cp << " ";
        } // 800 2 3 4 5 6 7 8 9 10
        std :: cout << std :: endl;
        std :: cin.get ();
        return 0;
}
```

Розглянемо клас list- список**.** Перед використанням класу необхідно додати# include<list>

Створення об'єкта класу list

Способи створити екземплярів класу listаналогічні способам створення екземплярів класуdeque.

Над елементами listвизначені дії ==, !=, <, <=, >, >=.

Вставка елементів

Методи вставки елементів у екземпляр класу list аналогічні методам вставки елементів у екземпляр класуdeque.

Видалення елементів

Окрім методів видалення елементів з екземпляру класу list аналогічних методам видалення елементів **з** екземпляру класу dequeіснують наступні (табл. 5.6):

Таблиця 5.6 - Методи видалення елементів з list

Метод	Опис	Прототип
Remove ()	Видаляєусі елементи, що мають значення Val	void remove (const T & Val)
Remove_if()	Видаляє елементи, для Pred яких повернув значення true	template < class Pr > void remove_if (Pr Pred);
unique ()	Видаляє елементи, що повторюються	voidunique (); template < class Pr > void unique (Pr Pred);

Доступ до елементів.

Для доступу до елементівекземпляру класу list призначені методи front (),back (). Крім того доступ до методів може виконуватись за допомогою ітераторів.

Сортування, об'єднання, перевертання (табл. 5.7)

Таблиця 5.7 - Методи сортування, об'єднання, перевертання елементів list

Метод	Опис	Прототип
sort()	Сортує список	void sort (); template < class Pr > void sort (Pr Pred);
splice() merge()	Об'єднують список	void splice (const_iterator Where, list &Right); void splice (const_iterator Where, list &Right, const_iterator First); void splice (const_iterator Where, list &Right, const_iterator First, const_iterator Last); void merge (list&Right); template < class Pr > void merge (list&Right, Pr Pred);
Reverse()	Змінює порядок елементів списку на протилежний	void reverse();

*Приклад 21.*Використання методу remove ().

```
std :: list<int>L ( 3, 1 );
std :: list <int>:: const_iterator i;
L.push_front ( 0 );
L.push_back ( 2 );
for ( i = L.begin(); i != L.end(); ++ i)
{
        std :: cout << *i << " ";
} // 0 1 1 1 2
std :: cout << std :: endl;
L.remove ( 1 );
for ( i = L.begin(); i != L.end(); ++ i)
{
        std :: cout << *i << " ";
} // 0 2
```

*Приклад 22.*Використання методу unique ().

```
std :: list <int> L1 ( 3, 1 ), L2;
std :: list <int>:: const_iterator i;
L1.push_front ( 0 );
L1.push_back ( 2 );
L1.push_back ( 1 );
L2 = L1;
L1.unique ();
for ( i = L.begin(); i != L.end(); ++ i)
{
        std :: cout << *i << " ";
} // 0 1 2 1
```

```
std :: cout << std :: endl;
L2. unique ( std :: equal_to <int> () );
for ( i = L2.begin(); i != L2.end(); ++ i)
{
        std :: cout<< *i<< " ";
} // 0 1 2 1
```

*Приклад 23.*Сортування списку у прямому порядку.

```
# include <iostream>
# include <list>
int main ()
{
        std :: list <int> L;
        std :: list <int>:: const_iterator i;
        L.push_back ( 5 );
        L.push_back ( 2 );
        L.push_back ( 4 );
        L.sort ();
        for ( i = L.begin(); i != L.end(); ++ i)
        {
                std :: cout<< *i<< " ";
        } // 2 4 5

}
```

*Приклад 24.*Додавання елементів перед останнім елементом списку.

```
std :: list <int> L1 ( 3, 1 ), L2 ( 3, 2 );
std :: list <int>:: const_iterator i;
L1.splice ( -- L1.end (), L2 );
for ( i = L1.begin(); i != L1.end(); ++ i)
{
        std :: cout << *i << " ";
} // 1 1 2 2 2 1
std :: cout << std :: endl << L2.size () << std :: endl;
```

*Приклад 25.*Використання другого прототипу методу splice ().

```
std :: list <int> L1 ( 3, 1 ), L2;
std :: list <int>:: const_iterator i;
L2.push_front ( 1 );
L2.push_back ( 2 );
L2.push_back ( 3 );
L1.splice ( L1.end (), L2, ++ L2.begin () );
for ( i = L1.begin(); i != L1.end(); ++ i)
{
        std :: cout << *i << " ";
} // 1 1 1 2
std :: cout << std :: endl;
for ( i = L2.begin(); i != L2.end(); ++ i)
```

```
{
        std :: cout << *i << “ ”;
} // 1 3
```

*Приклад26.*Використання третього прототипу методу splice ().

```
std :: list <int> L1 ( 3, 1 ), L2;
std :: list <int>:: const_iterator i;
L2.push_front ( 1 );
L2.push_back ( 2 );
L2.push_back ( 3 );
L1.splice ( -- L1.end (), L2, ++ L2.begin (), L2.end() );
for ( i = L1.begin(); i != L1.end(); ++ i)
{
        std :: cout << *i << “ ”;
} // 1 1 2 3 1
std :: cout << std :: endl;
for ( i = L2.begin(); i != L2.end(); ++ i)
{
        std :: cout<< *i<< “ ”;
} // 1
```

*Приклад27.*Використання третього прототипу методу merge ().

```
std :: list <int> L1, L2;
std :: list <int>:: const_iterator i;
L1.push_front ( 3 );
L1.push_back ( 6 );
L1.push_back ( 1 );
L2.push_front ( 5 );
L2.push_back ( 2 );
L2.push_back ( 4 );
L1.sort ();
L2.sort ();
L1.merge ( L2 );
for ( i = L1.begin(); i != L1.end(); ++ i)
{
        std :: cout << *i << “ ”;
} // 1 2 3 4 5 6
std :: cout << std::endl << L2.size () << std::endl;
```

*Приклад28.*Використання третього прототипу методу reverse ().

```
std :: list <int> L;
std :: list <int>:: const_iterator i;
L.push_front ( 3 );
L.push_back ( 6 );
L.push_back ( 1 );
L.push_front ( 5 );
L.push_back ( 2 );
L.push_back ( 4 );
```

```
for ( i = L.begin(); i != L.end(); ++ i)
{
        std :: cout << *i << " ";
} // 3 6 1 5 2 4
std :: cout << std :: endl;
L. reverse ();
for ( i = L.begin(); i != L.end(); ++i)
{
        std :: cout << *i << " ";
} // 4 2 5 1 6 3
```

Клас pair

Клас pair, який оголошується у заголовному файлі utility, реалізує пару ключ/значення. До ключа можна звернутися через атрибут first до значення – через атрибут second. Створити екземпляр класу можна наступними способами:

1. Оголосити екземпляр класу без ініціалізації.

```
std :: pair <int, int> pr;
std :: cout << pr. first << std :: endl; // 0
std :: cout << pr. second << std :: endl; // 0
```

2. Вказати після назви змінної всередині круглих дужок два значення через кому.

3. Скористатися функцією make_pair ().

Над елементами pairвизначені дії ==, !=, <, <=, >, >=.

Клас mapреалізує асоціативний масив, у якому кожному ключу відповідає одне значення. Перед тим як використовувати клас необхідно додати # include <map>.

Створення об'єкта класу map

Існують наступні способи створити екземпляр класу map**:**

1. Оголосити екземпляр класу без ініціалізації.

```
std :: map < std :: string, int > m;
```

2. Вказати об'єкт класу всередині круглих дужок або після =
3. Вказати діапазон всередині контейнера за допомогою ітераторів

Над елементами mapвизначені дії ==, !=, <, <=, >, >=.

Вставка елементів (табл. 5.8)

Вставити елемент можна, вказавши ключ всередині квадратних дужок. Значення елементу задається після оператору присвоєння. Якщо ключ існує, то замість вставки елементу виконується заміна значення.

Видалення елементів

Методи видалення елементів з екземпляру класу map аналогічні методам видалення елементів з екземпляру класуdeque.Але map не має методу pop_back (), pop_front ().

Таблиця 5.8 - Методи вставки елементів у map

Метод	Опис	Прототип
insert ()	Додає один або декілька елементів	pair <iterator, bool> insert (const pair &Val); iterator insert (const_iterator Where, const pair &Val); template < class It> void insert (It First, It Last);
swap ()	Змінює елементи двох контейнерів місцями	void swap (map &Right);

Доступ до елементів (табл. 5.9)

Таблиця 5.9 - Методи доступу до елементів map

Метод	Опис	Прототип
at ()	Повертає посилання на елемент, ключ якого відповідає значенню Keyval.	mapped_type &at (const key_type &Keyval); const mapped_type &at (const key_type &Keyval);
count ()	Повертає кількість елементів, ключ яких відповідає значенню Keyval.	size_type count (const key_type &Keyval) const;
find ()	Повертає ітератор, встановлений на елемент, ключ якого відповідає значенню Keyval.	iterator find (const key_type &Keyval); const_iterator find (const key_type &Keyval) const;
lower_bound( )	Повертає ітератор, встановлений на елемент, ключ якого більше або дорівнює значенню Keyval.	iterator lower_bound((const key_type &Keyval); const_iterator lower_bound((const key_type &Keyval) const;
upper_bound( )	Повертає ітератор, встановлений на елемент, ключ якого більше значення Keyval.	iterator lower_bound((const key_type &Keyval); const_iterator lower_bound((const key_type &Keyval) const;
equal_range()	Повертає екземпляр класу pair.	

Приклад 29. Створення екземпляру класу pair.

```
typedef struct std :: pair< std :: string, int >PAIR;
PAIR pr ( “Key” , 10);
std :: cout << pr.first << std::endl; // Key
std :: cout << pr.second << std::endl; // 10

typedef struct std :: pair< std :: string, int >PAIR;
PAIR pr1 ( “Key” , 10 );
```

```
PAIR pr2 ( pr1 );
std :: cout << pr2.first << std::endl; // Key
std :: cout << pr2.second << std::endl; // 10

typedef struct std :: pair< int , int >PAIR;
PAIR pr = std :: make_pair ( 5 , 10 );
PAIR pr2 ( pr1 );
std :: cout << pr.first << std::endl; // 5
std :: cout << pr.second << std::endl; // 10
```

*Приклад 30.*Створення екземпляру класу map.

```
std :: map < int, int > m1;
std :: map < int, int > m2 ( m1 );
std :: map < int, int > m3 = m1;

typedef struct std :: pair< int , int >PAIR;
std :: map < int , int > m1;
m1.insert ( PAIR ( 0 , 10 ) );
m1.insert ( PAIR ( 1 , 20 ) );
std :: map < int , int > m2 ( m1 . begin () , m1 . end () );
```

*Приклад 31.*Присвоєння одного об’єкту класу map іншому.

```
typedef struct std :: pair< int , int >PAIR;
std :: map < int , int > m1 , m2;
std :: map < int , int > :: const_iterator i;
m1.insert ( PAIR ( 0 , 10 ) );
m1.insert ( PAIR ( 1 , 20 ) );
m2 = m1;
for ( i = m2 .begin () ; i != m2 .end (); ++i )
{
        std :: cout << i-> first << “->” << i-> second << “;”
} // 0 -> 10; 1 -> 20;
```

*Приклад 32.*Використання методу insert().

```
typedef struct std::pair< std::string, int > PAIR;
std::map < std::string, int > m;
std::map < std::string, int >:: const_iterator i;
std::pair< std::map< std::string, int >::iterator, bool > pr;
pr = m.insert( PAIR( “one”, 1 ) );
if ( pr.second )
{
        std :: cout <<( pr.first )-> first << “->” <<( pr.first )-> second <<
std::endl;
} // one->1
else std :: cout << “Error” << std::endl;
m.insert( std::map < std::string, int >::value_type ( “two”, 2 ) );
for ( i = m. begin () ; i != m. end (); ++i )
{
```

```
        std :: cout << i-> first << “->” << i-> second << “;”;
} // one -> 1; two -> 2;
```

Розглянути роботу з другим прототипом методу insert().

*Приклад 33.*Використання методу swap ().

```
typedef struct std::pair< std::string, int > PAIR;
std::map < std::string, int > m1, m2;
std::map < std::string, int > :: const_iterator i;
m1.insert( PAIR ( “0”, 1 ) );
m1.insert( PAIR ( “1”, 2 ) );
m2.insert( PAIR ( “one”, 1 ) );
m2.insert( PAIR (“two”, 2 ) );
m2.swap ( m1 );
for ( i = m1.begin (); i != m1.end (); ++i )
{
        std :: cout << i-> first << “->” << i-> second << “;”;
} // one -> 1; two -> 2;
std :: cout << std :: endl;
for ( i = m2.begin (); i != m2.end (); ++i )
{
        std :: cout << i-> first << “->” << i-> second << “;”;
} // 0 -> 1; 1 -> 2;
```

*Приклад 34.*Використання методу erase()

```
typedef struct std::pair< std::string, int > PAIR;
std::map < std::string, int > m;
std::map < std::string, int > :: const_iterator i;
m.insert( PAIR ( “one”, 1 ) );
m.insert( PAIR ( “two”, 2 ) );
m.erase ( “one” );
for ( i = m.begin (); i != m.end (); ++i )
{
        std :: cout << i-> first << “->” << i-> second << “;”;
} // two -> 2;

typedef struct std::pair< std::string, int > PAIR;
std::map < std::string, int > m;
std::map < std::string, int > :: const_iterator i;
m.insert( PAIR ( “one”, 1 ) );
m.insert( PAIR ( “two”, 2 ) );
m.erase ( --m.end() );
for ( i = m.begin (); i != m.end (); ++i )
{
        std :: cout << i-> first << “->” << i-> second << “;”;
} // one -> 1;

typedef struct std::pair< std::string, int > PAIR;
```

```
std::map < std::string, int > m;
std::map < std::string, int > :: const_iterator i;
m.insert( PAIR ( “0”, 1 ) );
m.insert( PAIR ( “1”, 2 ) );
m.insert( PAIR ( “2”, 1 ) );
m.insert( PAIR ( “3”, 2 ) );
m.erase ( m.begin(), --m.end() );
for ( i = m.begin (); i != m.end (); ++i )
{
        std :: cout << i-> first << “->” << i-> second << “;”;
} // 3 -> 4;
```

*Приклад 35.*Використання методів clear(), empty(), size()

```
typedef struct std::pair< std::string, int > PAIR;
std::map < std::string, int > m;
m.insert( PAIR ( “0”, 1 ) );
m.insert( PAIR ( “1”, 2 ) );
std :: cout << m.size () << std :: endl; // 2
m.clear();
if ( m.empty () )
{
        std :: cout << “Немає елементів” << std ::endl;
}
std:: cout << m.size () << std::endl; // 0
```

*Приклад 36.*Доступ до елементів map.

```
std::map < std::string, int > m;
std::map < std::string, int > :: const_iterator i;
m[“one”] = 1;
m[“two”] = 2;
m[“one”] = 100;
std :: cout << m[“one”] << std ::endl; // 100
std :: cout << m[“two”] << std ::endl; // 2
std :: cout << m[“key”] << std ::endl; // 0
for ( i = m.begin (); i != m.end (); ++i )
{
        std :: cout << i-> first << “->” << i-> second << “;”;
} // key -> 4; one -> 100; two -> 2;
```

*Приклад 37.*Використання методу at()

```
std::map < std::string, int > m;
m[“one”] = 1;
m.at( “one” ) = 100;
std :: cout << m.at ( “one” ) << std :: endl; //100
```

*Приклад 38.*Використання методу count()

```
std::map < std::string, int > m;
m[“one”] = 1;
std :: cout << m.count ( “one” ) << std :: endl; // 1
```

```
std :: cout << m.count ( “two” ) << std :: endl; // 0
```

*Приклад 39.*Використання методу find()

```
std::map < std::string, int > m;
std::map < std::string, int > :: const_iterator i;
m[“one”] = 1;
i = m.find ( “one” );
if ( i != m.end() )
{
        std :: cout << i -> second << std::endl; // 1
}
i = m.find ( “two” );
if (i == m.end() )
{
        std :: cout << “No” << std::endl; // No
}
```

*Приклад 40.*Використання методу lower_bound ()

```
std::map < std::string, int > m;
std::map < std::string, int > :: const_iterator i;
m[0] = 0;
m[1] = 1;
m[4] = 4;
m[5] = 5;
i = m.lower_bound ( 2 );
if ( i != m.end() )
{
        std :: cout << i -> second << std::endl; // 4
}
i = m.lower_bound ( 1 );
if ( i != m.end() )
{
        std :: cout << i -> second << std::endl; // 1
}
```

*Приклад 41.*Використання методу equal_range ()

```
typedef std::map < int, int > M;
M m;
std::pair < M::iterator, M::iterator > pr;
m[0] = 0;
m[1] = 1;
m[4] = 4;
m[5] = 5;
pr = m.equal_range ( 1 );
if ( pr.first != m.end() )
{
        std :: cout << pr.first -> second << std::endl; // 1
```

```
}
if ( pr.second != m.end() )
{
        std :: cout << pr.second -> second << std::endl; // 4
}
pr = m.equal_range ( 2 );
if ( pr.first != m.end() )
{
        std :: cout << pr.first -> second << std::endl; // 4
}
if ( pr.second != m.end() )
{
        std :: cout << pr.second -> second << std::endl; // 4
}
```

Клас setреалізує множину, що складається з унікальних елементів. Перед тим як використовувати клас необхідно додати # include<set>.

Створення об'єкта класу set

Способи створити екземпляр класу setаналогічні способам створити екземпляр класуmap

Над елементами **set**визначені дії ==, !=, <, <=, >, >=.

Вставка елементів

Способи вставки елементів у об'єкт класуsetаналогічні способам вставки елементів у об'єкт класу map

Видалення елементів

Способи видалення елементів з об'єкту класуsetаналогічні способам видалення елементів з об'єкту класу map

Доступ до елементів

Способи доступу до елементів об'єкту класуsetаналогічні способам доступу до елементів об'єкту класу map.Окрім використання методу at().

Для використання STL контейнерів нестандартних типів даних (Button, Label, TextBox) необхідно у заголовному файлі додати #include<cliext/vector>. При описі STL контейнерів використовувати простів імен cliext.

Приклад 42

```
cliext :: vector < TextBox^ > data ;
cliext::vector<Label^> rezult;
textTemp =gcnew TextBox();
data.push_back(textTemp);
this -> Controls -> Add(textTemp);
```

КОНТРОЛЬНІ ПИТАННЯ:

1. Що таке стандартна бібліотека шаблонів?
2. Що таке ітератои?

3. Що таке контейнери?
4. Яких типів бувають контейнери?
5. Які методи використовують для надання значення змінним?
6. Що таке deque?
7. Які методи використовують для вставки елементів у deque?
8. Які методи використовують для видалення елементів з deque?
9. Які методи використовують для доступу до елементів deque?
10. Що таке vector?
11. Які методи використовують для вставки елементів у vector?
12. Які методи використовують для видалення елементів з vector?
13. Які методи використовують для доступу до елементів vector?
14. Що таке list?
15. Які методи використовують для вставки елементів у list?
16. Які методи використовують для видалення елементів з list?
17. Які методи використовують для доступу до елементів list?
18. Які методи використовують для сортування, об’єднання, перевертання елементів list?
19. Що таке pair?
20. Що таке map?
21. Які методи використовують для вставки елементів у map?
22. Які методи використовують для видалення елементів з map?
23. Які методи використовують для доступу до елементів map?
24. Що таке set?
25. Які методи використовують для вставки елементів у set?
26. Які методи використовують для видалення елементів з set?
27. Які методи використовують для доступу до елементів set?

РЕКОМЕНДОВАНА ЛІТЕРАТУРА

Базова

1. Макконнелл С. Совершенный код. Мастер-класс / С. Мак-коннелл. – Москва : Русская редакция, 2013. – 896 с.
2. Маршал Д. Надежный код / Д. Маршал, Д. Бруно. – Москва :Русская редакция ; Санкт-Петербург : БХВ-Петербург, 2010. – 320 с.
3. Соммервилл И. Инженерия программного обеспечения / И. Соммервилл. – 6-е изд. – Москва : Вильямс, 2002. – 624 с.
4. Дюваль П. Непрерывная интеграция. Улучшение качества программного обеспечения и снижение риска / П. Дюваль, Э. Гло-вер. – Москва : Вильямс, 2008. – 240 с.
5. Шпак З.Я. Програмування мовою С. –Львів: Оріяна-Нова, 2006. – 432с. (25 примірників у бібліотеці).

Допоміжна

5. Зыль С. Проектирование, разработка и анализ программ-ного обеспечения систем реального времени / С. Зыль. – Санкт-Петербург : БХВ-Петербург, 2010. – 336 с.
6. Спинеллис Д. Анализ программного кода.– «Вильямс», 2004. – 524с.
7. ISO 10002:2014. Quality management. Customer satis-faction. Guidelines for complaints handling in organizations – Revises ISO 10002:2004: introduced 15.07.2014. – ISO, 2014. – 26 p.
8. ISO/IEC 25010:2011. Systems and software engineering. Systems and software Quality Requirements and Evaluation (SQuaRE). System and software quality models – Revises ISO/IEC 9126-1:2001: introduced 01.03.2011. – ISO/IEC, 2011. – 34 p.
9. ISO/IEC 25030:2007. Software engineering. Software product Quality Requirements and Evaluation (SQuaRE). Quality requirements – Introduced 01.06.2007. – ISO/IEC, 2007. – 36 p.
10. Bourque P. Guide to the software engineering body of knowledge (SWEBOK). Version 3.0 / P.Bourque, R.E.Fairley – A project of the IEEE Computer Society, 2014. – 335 p.

НАВЧАЛЬНЕ ВИДАННЯ

Конспект лекцій з дисципліни «Конструювання програмного забезпечення» для для здобувачів вищої освіти першого (бакалаврського) рівня спеціальності 121 -«Інженерія програмного забезпечення» очної і заочної форм навчання.

Укладачі:
Яшина Ксенія Володимирівна
Ялова Катерина Миколаївна
Лимар Ніна Миколаївна

Підписано до друку __21.02._ 2019 р.
Формат _А4_ Обсяг_3,4_ др. арк.
Тираж _40__ прим. Замовлення_37_ .
51918, м.Кам’янське,
вул. Дніпробудівська, 2.