

МІНІСТЕРСТВО ОСВІТИ ТА НАУКИ УКРАЇНИ
НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ»

КОНСПЕКТ ЛЕКЦІЙ

“Інженерія програмного забезпечення”
для напряму підготовки (спеціальностей)
6.050102 Комп’ютерна інженерія

Розробник

к.т.н., с.н.с.,
ст.викладач каф. ОТ
Антонюк А.І.

Рекомендовано кафедрою
обчислювальної техніки

Протокол № від 2014 р.

Завідувач кафедри

_____Луцький Г.М.

(підпис)

Київ - 2014

Зміст

Розділ 1. Методології розробки ПЗ	6
Лекція 1. Життєвий цикл ПЗ	6
1. Основні поняття	6
2. Стандарт ISO/IEC 12207:1995.....	7
3. Стандарт на процеси ЖЦ систем (ISO/IEC 15288 System life cycle processes)	11
Лекція 2. Моделі та методології розробки	12
Водопадна (каскадна, послідовна) модель	13
Ітераційна модель.....	14
Спіральна модель	15
Методології розробки ПЗ	19
Розділ 2. Засоби та середовища створення ПЗ.	21
Лекція 3. Інтегровані середовища розробки ПЗ.....	21
Інтегровані середовища розробки ПЗ	21
Eclipse. Початок.....	21
Архітектура Eclipse	22
Проекти Eclipse.....	23
Лекція 4. Управління версіями та автоматизація збірки ПЗ.....	27
Системи управління версіями документів.....	27
Невеликий словник основних термінів-сленгів.....	28
Світовий досвід	28
Види систем контролю версії.....	29
Виникнення конфліктів та їх вирішення	30
Поширені системи керування версіями	30
Нове покоління інструментів.....	31
Переваги	31
Типи	32
Вимоги до систем збірки	32

Розділ 3. Проектування ПЗ з використанням шаблонів.....	34
Лекція 5. Шаблони проектування ПЗ.....	34
Проектування ПЗ.....	34
Лекція 6. Структурні шаблони проектування: Composite та Decorator.....	38
Компонувальник Composite	38
Декоратор (Decorator)	42
Лекція 7. Структурні шаблони проектування: Proxy та Flyweight.....	45
Шаблон Proxy	45
Шаблон Легковаговик (Flyweight) - козак.....	49
Лекція 8. Структурні шаблони проектування: Adapter, Bridge та Facade.	52
Шаблон Adapter	52
Шаблон Bridge	54
Шаблон Facade.....	58
Лекція 9. Шаблони поведінки: Iterator та Mediator. Ошибка! Закладка не определена.	
Шаблон Iterator	Ошибка! Закладка не определена.
Шаблон Mediator	Ошибка! Закладка не определена.
Лекція 10. Шаблони поведінки: Observer та Strategy. Ошибка! Закладка не определена.	
Шаблон Observer	Ошибка! Закладка не определена.
Шаблон Strategy (Стратегія)	Ошибка! Закладка не определена.
Лекція 11. Шаблони поведінки: Command та Visitor. Ошибка! Закладка не определена.	
Шаблон Command	Ошибка! Закладка не определена.
Лекція 12. Шаблони поведінки: State та Memento. Ошибка! Закладка не определена.	
Шаблон State	Ошибка! Закладка не определена.
Шаблон Memento.....	Ошибка! Закладка не определена.
Лекція 13. Шаблони поведінки: Interpreter та Chain of Responsibility. Ошибка! Закладка не определена.	
Шаблон Interpreter	Ошибка! Закладка не определена.
Шаблон Chain of Responsibility.....	Ошибка! Закладка не определена.
Лекція 14. Породжувальні шаблони Prototype, Factory Method та Abstract Factory.....	Ошибка! Закладка не определена.

Шаблон Prototype	Ошибка! Закладка не определена.
Шаблон Factory Method	Ошибка! Закладка не определена.
Шаблон Abstract Factory	Ошибка! Закладка не определена.
Лекція 15. Шаблиони Singleton та Builder.	Ошибка! Закладка не определена.
Шаблон Singleton.....	Ошибка! Закладка не определена.
Шаблон Builder	Ошибка! Закладка не определена.
Розділ 4. Проектування інтерфейсу користувача. .	Ошибка! Закладка не определена.
Лекція 16. Графічний інтерфейс користувача.	Ошибка! Закладка не определена.
Призначення та типи інтерфейсів користувача	Ошибка! Закладка не определена.
Принципи розробки графічного інтерфейсу користувача	Ошибка! Закладка не определена.
Приклад графічного інтерфейса користувача	Ошибка! Закладка не определена.
Лекція 17. Моделі подій та промальовування.	Ошибка! Закладка не определена.
Моделі обробки подій.....	Ошибка! Закладка не определена.
Моделі промальовування графічного інтерфейсу користувача	Ошибка! Закладка не определена.
Розділ 5. Моделювання програмного забезпечення	Ошибка! Закладка не определена.
Лекція 18. Інформаційне моделювання та моделювання бізнес-процесів	Ошибка! Закладка не определена.
Методології моделювання.....	Ошибка! Закладка не определена.
Сімейство стандартів	Ошибка! Закладка не определена.
Моделювання бізнес-процесів	Ошибка! Закладка не определена.
Лекція 19. Поведінкове, структурне та функціональне моделювання	Ошибка! Закладка не определена.
Поведінкове моделювання	Ошибка! Закладка не определена.
Структурне моделювання.....	Ошибка! Закладка не определена.
IDEF4	Ошибка! Закладка не определена.
Функціональне моделювання за допомогою IDEF0	Ошибка! Закладка не определена.
Лекція 20. Моделювання потоків даних	Ошибка! Закладка не определена.
Моделювання потоків даних.....	Ошибка! Закладка не определена.

Розділ 6. Методи забезпечення та контролю якості ПЗ **Ошибка! Закладка не определена.**

Лекція 21. Тестування ПЗ **Ошибка! Закладка не определена.**

Лекція 22. Постійна інтеграція ПЗ **Ошибка! Закладка не определена.**

Принципи постійної інтеграції ПЗ **Ошибка! Закладка не определена.**

Безперервна інтеграція - Continuous Integration (CI) **Ошибка! Закладка не определена.**

Профіти **Ошибка! Закладка не определена.**

Посилання та Публікації **Ошибка! Закладка не определена.**

Розділ 7. Менеджмент програмних проектів **Ошибка! Закладка не определена.**

Лекція 23. Управління проектами **Ошибка! Закладка не определена.**

Поняття проекту **Ошибка! Закладка не определена.**

Причини виникнення і сутність управління проектами **Ошибка! Закладка не определена.**

Управління процесом виконання проекту **Ошибка! Закладка не определена.**

Управління якістю **Ошибка! Закладка не определена.**

Лекція 24. Управління ресурсами, ризиками та конфігураціями **Ошибка! Закладка не определена.**

Планування в УП **Ошибка! Закладка не определена.**

Управління плануванням **Ошибка! Закладка не определена.**

Процес планування **Ошибка! Закладка не определена.**

План реалізації проекту **Ошибка! Закладка не определена.**

Лекція 25. Контроль та моніторинг стану проекту **Ошибка! Закладка не определена.**

Контроль в управлінні проектами **Ошибка! Закладка не определена.**

Моніторинг робіт по проекту **Ошибка! Закладка не определена.**

Вимірювання прогресу й аналіз робіт **Ошибка! Закладка не определена.**

Прийняття рішень **Ошибка! Закладка не определена.**

Розділ 1. Методології розробки ПЗ

Лекція 1. Життєвий цикл ПЗ

1. Основні поняття

- Основні поняття та проблеми розробки ПЗ.
- Життєвий цикл ПЗ;
- міжнародні стандарти життєвого циклу ПЗ [1, с. 12-14; 2, с. 41-60]

В наш час інформаційні системи (ІС) є важливою інфраструктурною складовою глобальної економіки, що динамічно розвивається. Одним з ключових інструментів успішного розвитку і ефективного застосування ІС є стандартизація, де первинне місце займають стандарти інженерії програмного забезпечення (ПЗ) - правила, які встановлюють для загального і багатократного використання, загальні принципи, процеси і інструменти створення ефективного програмного забезпечення. Ці стандарти формують методологічну основу діяльності по створенню ПЗ інформаційних систем різного масштабу і призначення

Програмне забезпечення — це усе або частина програм, процедур, правил та відповідної документації системи обробки інформації (ISO/IEC 2382-1: 1993. Information technology — Vocabulary — Part 1: Fundamental terms).

Інші визначення ПЗ з міжнародних та вітчизняних стандартів:

- Комп'ютерні програми, процедури і, можливо, відповідна документація, що відносяться до функціонування комп'ютерної системи (FCD ISO/IEC 24765. Systems and Software Engineering Vocabulary).
- Сукупність програм системи обробки інформації та програмних документів, необхідних для експлуатації цих програм (ГОСТ 19781-90).
- Програмний засіб (ГОСТ 28806-90) - це об'єкт, що складається з програм, процедур, правил, а також, якщо передбачено, супутніх їм документації та даних, що відносяться до функціонування конкретної системи обробки інформації. Поняття «Програмне забезпечення» (ГОСТ 19781) є окремим випадком терміну «програмні засоби» і відрізняється від останнього відсутністю даних та документації, що відносяться до функціонування конкретної системи обробки інформації.

У цих умовах «Інженерія ПЗ» — це системний підхід до аналізу, проектування, оцінки, реалізації, тестування, обслуговування та модернізації програмного забезпечення, тобто застосування інженерії до розробки програмного забезпечення.

Поняття життєвого циклу ПЗ ІС. Процеси життєвого циклу: основні, допоміжні, організаційні. Зміст і взаємозв'язок процесів життєвого циклу ПЗ ІС. Моделі життєвого циклу: **каскадна, модель з проміжним контролем, спіральна**. Стадії життєвого циклу ПЗ ІС. Регламентація процесів проектування у вітчизняних та міжнародних стандартах.

Методологія проектування інформаційних систем описує процес створення і супроводу систем у вигляді *життєвого циклу* (ЖЦ) ІС, представляючи його як деяку послідовність стадій і виконуваних на них процесів. Для кожного етапу визначаються склад і послідовність виконуваних робіт, одержувані результати, методи і засоби, необхідні для виконання робіт, ролі та відповідальність учасників і т.д. Такий формальний опис ЖЦ ІС дозволяє спланувати та організувати процес колективної розробки і забезпечити управління цим процесом.

Життєвий цикл ІС можна представити як ряд подій, що відбуваються з системою в процесі її створення та використання.

Серед найбільш відомих стандартів можна виділити наступні:

- ГОСТ 34.601-90 - поширюється на автоматизовані системи та встановлює стадії і етапи їх створення. Крім того, в стандарті міститься опис змісту робіт на кожному етапі. Стадії та етапи роботи, закріплені в стандарті, більшою мірою відповідають каскадній моделі життєвого циклу [4].
- ISO/IEC 12207:1995 - стандарт на процеси і організацію життєвого циклу. Поширюється на всі види замовленого ПЗ. Стандарт не містить опису фаз, стадій та етапів [5].

2. Стандарт ISO/IEC 12207:1995

Стандарт ISO/IEC 12207:1995 "Information Technology - Software Life Cycle Processes" є основним нормативним документом, який регламентує склад процесів життєвого циклу ПЗ. Він визначає структуру життєвого циклу, що містить процеси, дії і завдання, які повинні бути виконані під час створення ПЗ.

Кожен процес розділений на набір дій, кожна дія - на набір завдань. Кожен процес, дія або завдання ініціюється і виконується іншим процесом в міру необхідності, причому не існує заздалегідь визначених послідовностей виконання. Зв'язки за вхідними даними при цьому зберігаються.

Відповідно до базовим міжнародним стандартом ISO/IEC 12207 всі *процеси ЖЦ ПЗ* діляться на три групи:

1. Основні процеси:

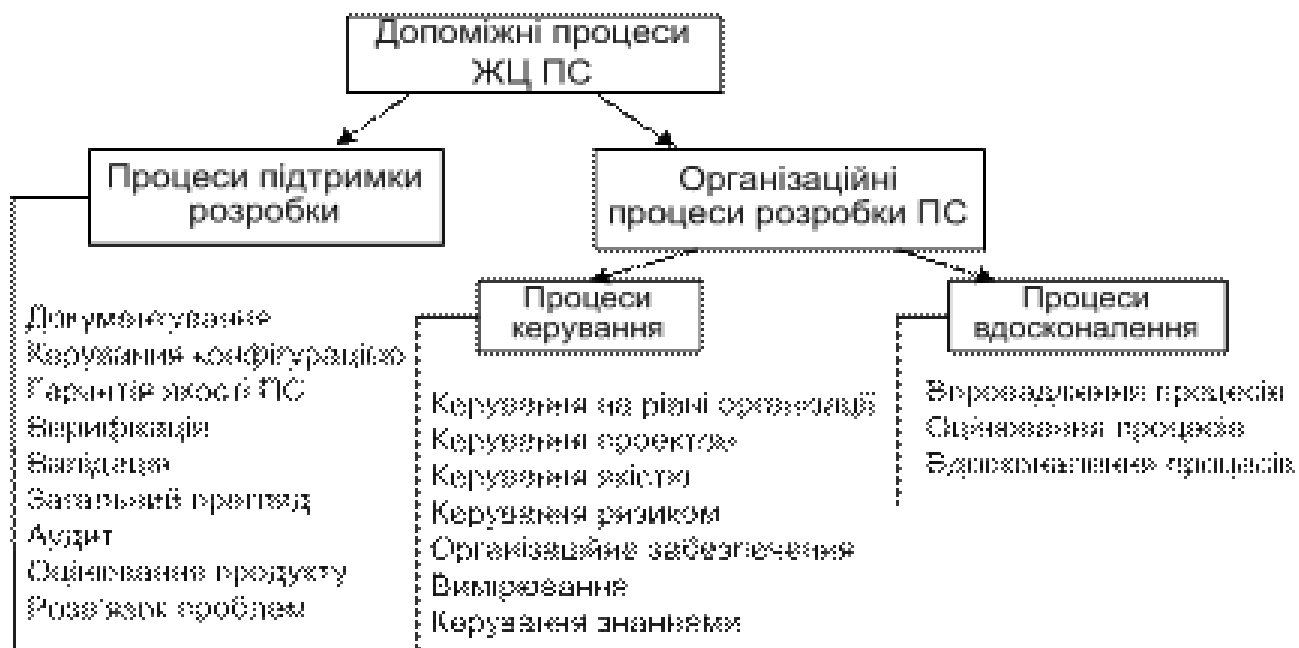
- Придбання (дії і завдання замовника, що здобуває ПЗ)
- Поставка (дії і завдання постачальника, який постачає замовнику програмний продукт або послугу)
- Розробка (дії і завдання, що виконуються розробником: створення ПЗ, оформлення проектної та експлуатаційної документації, підготовка тестових та навчальних матеріалів і т. д.)

- Експлуатація (дії і завдання оператора - організації, що експлуатує систему)
- Супровід (дії і завдання, що виконуються супроводжує організацією, тобто службою супроводу). Супровід - внесення змін в ПЗ з метою виправлення помилок, підвищення продуктивності або адаптації до нових умов роботи або вимог.



2. Допоміжні процеси:

- Документування (формалізований опис інформації, створеної протягом ЖЦ ПЗ)
- Управління конфігурацією (застосування адміністративних і технічних процедур на всьому протязі ЖЦ ПЗ для визначення стану компонентів ПЗ, управління його модифікаціями).
- Забезпечення якості (забезпечення гарантій того, що ІС і процеси її ЖЦ відповідають заданим вимогам та затвердженим планам)
- Верифікація (визначення того, що програмні продукти, які є результатами певної дії, повністю задовольняють вимогам або умовам, обумовленим попередніми діями)
- Атестація (визначення повноти відповідності заданих вимог і створеної системи їх конкретному функціональному призначенню)
- Спільна оцінка (оцінка стану робіт по проекту: контроль планування та управління ресурсами, персоналом, апаратурою, інструментальними засобами)
- Аудит (визначення відповідності вимогам, планам і умовам договору)
- Вирішення проблем (аналіз і рішення проблем, незалежно від їх походження чи джерела, які виявлені в ході розробки, експлуатації, супроводу або інших процесів)



3. Організаційні процеси:

- Управління (дії і завдання, які можуть виконуватися будь-якою стороною, що управляє своїми процесами)
- Створення інфраструктури (вибір та супровід технології, стандартів та інструментальних засобів, вибір і установка апаратних і програмних засобів, що використовуються для розробки, експлуатації чи супроводу ПЗ)
- Удосконалення (оцінка, вимір, контроль та вдосконалення процесів ЖЦ)
- Навчання (початкове навчання і подальше постійне підвищення кваліфікації персоналу)

4. Зміст основних процесів ЖЦ ПЗ ІС (ISO/IEC 12207)

У таблиці 2.1 наведені орієнтовні приклади опису основних процесів ЖЦ. Допоміжні процеси призначені для підтримки виконання основних процесів, забезпечення якості проекту, організації верифікації, перевірки та тестування ПЗ. Організаційні процеси визначають дії і завдання, що виконуються як замовником, так і розробником проекту для управління своїми процесами.

Для підтримки практичного застосування стандарту ISO/IEC 12207 розроблено ряд технологічних документів: Керівництво для ISO/IEC 12207 (ISO/IEC TR 15271:1998 Information technology - Guide for ISO/IEC 12207) і Настанова щодо застосування ISO/IEC 12207 до управління проектами (ISO/IEC TR 16326:1999 Software engineering - Guide for the application of ISO/IEC 12207 to project management).

Таблиця 2.1. Зміст основних процесів ЖЦ ПЗ ІС (ISO/IEC 12207)

Процес (виконавець процесу)	Дії	Вхід	Результат
-----------------------------------	-----	------	-----------

Придбання (замовник)	<ul style="list-style-type: none"> • ініціювання • Підготовка заявочних пропозицій • Підготовка договору • Контроль діяльності постачальника • Приймання ІС 	<ul style="list-style-type: none"> • Рішення про початок робіт по впровадженню ІС • Результати обстеження діяльності замовника • Результати аналізу ринку ІС/тендеру • План поставки/розробки • Комплексний тест ІС 	<ul style="list-style-type: none"> • Техніко-економічне обґрунтування впровадження ІС • Технічне завдання на ІС • Договір на поставку/розробку • Акти приймання етапів роботи • Акт приймально-здавальних випробувань
Поставка (розробник ІС)	<ul style="list-style-type: none"> • ініціювання • Відповідь на заявочні пропозиції • Підготовка договору • Планування виконання • Поставка ІС 	<ul style="list-style-type: none"> • Рішення керівництва про участь в розробці • Результати тендеру • Технічне завдання на ІС • План управління проектом • Розроблена ІС і документація 	<ul style="list-style-type: none"> • Рішення про участь в розробці • Комерційні пропозиції/конкурсна заявка • Договір на поставку/розробку • План управління проектом • Реалізація/корегування • Акт приймально-здавальних випробувань
Розробка (розробник ІС)	<ul style="list-style-type: none"> • Підготовка • Аналіз вимог до ІС • Проектування архітектури ІС • Розробка вимог до ПЗ • Проектування архітектури ПЗ • Детальне проектування ПЗ • Кодування і тестування ПЗ • Інтеграція ПЗ і кваліфікаційне 	<ul style="list-style-type: none"> • Технічне завдання на ІС • Технічне завдання на ІС, модель ЖЦ • Технічне завдання на ІС • Підсистеми ІС • Специфікації вимоги до компонентів ПЗ • Архітектура ПЗ • Матеріали детального проектування ПЗ • План інтеграції ПЗ, тести • Архітектура ІВ, 	<ul style="list-style-type: none"> • Використовувана модель ЖЦ, стандарти розробки • План робіт • Склад підсистем, компоненти обладнання • Специфікації вимоги до компонентів ПЗ • Склад компонентів ПЗ, інтерфейси з БД, план інтеграції ПЗ • Проект БД, специфікації інтерфейсів між компонентами ПЗ, вимоги до тестів • Тексти модулів ПЗ, акти автономного тестування • Оцінка відповідності

	тестування ПЗ • Інтеграція ІС і кваліфікаційне тестування ІС	ПЗ, документація на ІС, тести	комплексу ПЗ вимогам ТЗ • Оцінка відповідності ПЗ, БД, технічного комплексу та комплекту документації вимогам ТЗ
--	---	-------------------------------	---

3. Стандарт на процеси ЖЦ систем (ISO/IEC 15288 System life cycle processes)

Пізніше був розроблений і в 2002 р. опублікований стандарт на процеси *життєвого циклу* систем (ISO/IEC 15288 System life cycle processes). До розробки стандарту були залучені фахівці різних областей: системної інженерії, програмування, управління якістю, людськими ресурсами, безпекою та ін. Був врахований практичний досвід створення систем в урядових, комерційних, військових і академічних організаціях. Стандарт застосовний для широкого класу систем, але його основне призначення - підтримка створення комп'ютеризованих систем.

Відповідно до стандарту ISO/IEC серії 15288 [7] в структуру ЖЦ слід включати наступні групи процесів:

1. Договірні процеси:

- придбання (внутрішні рішення або рішення зовнішнього постачальника);
- поставка (внутрішні рішення або рішення зовнішнього постачальника).

2. Процеси підприємства:

- управління навколишнім середовищем підприємства;
- інвестиційне управління;
- управління ЖЦ ІС;
- управління ресурсами;
- управління якістю.

3. Проектні процеси:

- планування проекту;
- оцінка проекту;
- контроль проекту;
- управління ризиками;
- управління конфігурацією;
- управління інформаційними потоками;
- прийняття рішень.

4. Технічні процеси:

- визначення вимог;
- аналіз вимог;
- розробка архітектури;
- впровадження;
- інтеграція;
- верифікація;
- перехід;
- атестація;
- експлуатація;
- супровід;
- утилізація.

5. Спеціальні процеси:

- визначення та встановлення взаємозв'язків виходячи із завдань і цілей.

Стадії створення системи, передбачені в стандарті ISO/IEC 15288, дещо відрізняються від розглянутих вище. Перелік стадій і основні результати, які повинні бути досягнуті до моменту їх завершення, наведені в табл. 2.2.

6. Стадії створення систем (ISO/IEC 15288)

Таблиця 2.2. Стадії створення систем (ISO/IEC 15288)

№ з/п	Стадія	Опис
1	Формування концепції	Аналіз потреб, вибір концепції та проектних рішень
2	Розробка	Проектування системи
3	Реалізація	Виготовлення системи
4	Експлуатація	Введення в експлуатацію та використання системи
5	Підтримка	Забезпечення функціонування системи
6	Зняття з експлуатації	Припинення використання, демонтаж, архівування системи

Лекція 2. Моделі та методології розробки

- Моделі та методології розробки.
- Аналіз, специфікація, верифікація та валідація вимог до ПЗ.
- Функціональні та нефункціональні вимоги [1, с. 12-14; 2, с. 41-60]

Модель життєвого циклу відображає різні стани системи, починаючи з моменту виникнення необхідності в даній ІС і закінчуючи моментом її повного виходу з ужитку. *Модель життєвого циклу* - структура, що містить процеси, дії і завдання, які здійснюються в ході розробки, функціонування та супроводження

програмного продукту протягом всього життя системи, від визначення вимог до завершення її використання.

В даний час відомі і використовуються наступні *моделі життєвого циклу*:

Водопадна (каскадна, послідовна) модель

Водопадна модель (рис. 2.1) життєвого циклу (*waterfall model*) Була запропонована в 1970 р. Уїнстоном Ройсом. Вона передбачає послідовне виконання всіх етапів проекту в строго фіксованому порядку. Перехід на наступний етап означає повне завершення робіт на попередньому етапі. Вимоги, визначені на стадії формування вимог, суворо документуються у вигляді технічного завдання і фіксуються на весь час розробки проекту. Кожна стадія завершується випуском повного комплексу документації, достатньої для того, щоб розробка могла бути продовжена іншою командою розробників.



Рис. 2.1. Каскадна модель ЖЦ ІС

Етапи проекту відповідно до каскадної моделі:

1. Формування вимог;
2. Проектування;
3. Реалізація;
4. Тестування;
5. Впровадження;
6. Експлуатація та супровід.

Переваги:

- Повна і узгоджена документація на кожному етапі;
- Легко визначити терміни і витрати на проект.

Недоліки:

У Водоспадної моделі перехід від однієї фази проекту до іншого передбачає повну коректність результату (виходу) попередньої фази. Однак неточність вимог або некоректна їх інтерпретація в результаті призводить до того, що доводиться "відкочуватися" до ранньої фази проекту і необхідна переробка не просто вибиває проектну команду з графіка, але призводить часто до якісного зростання витрат і,

не виключено, до припинення проекту в тій формі, в якій він спочатку замислювався. На думку сучасних фахівців, основна помилка авторів Водоспадної моделі полягає в припущеннях, що проект проходить через весь процес один раз, спроектована архітектура добра і проста у використанні, проект здійснення розумний, а помилки в реалізації легко усуваються в міру тестування. Ця модель виходить з того, що всі помилки будуть зосереджені в реалізації, а тому їх усунення відбувається рівномірно під час тестування компонентів і системи [2]. Таким чином, Водоспадна модель для великих проектів мало реалістична і може бути ефективно використана тільки для створення невеликих систем [3].

Ітераційна модель

Альтернативою послідовної моделі є так звана модель ітеративної і інкрементальної розробки (**англ.** *iterative and incremental development, IID*) (рис.2.2), Що отримала також від Т. Гілба в 70-і рр.. назву *еволюційної моделі*. Також цю модель називають *ітеративною та інкрементальною моделлю* [4].



Рис. 2.2. Поетапна модель з проміжним контролем

Модель IID передбачає розбиття життєвого циклу проекту на послідовність ітерацій, кожна з яких нагадує "міні-проект", включаючи всі процеси розробки в застосуванні до створення менших фрагментів функціональності, порівняно з проектом в цілому. Мета кожної *ітерації* - отримання працюючої версії програмної системи, що включає функціональність, визначену інтегрованим змістом всіх попередніх та поточної ітерації. Результат фінальної ітерації містить всю необхідну функціональність продукту. Таким чином, із завершенням кожної ітерації продукт отримує приріст - *інкремент* - до його можливостей, які, отже, розвиваються *еволюційно*. Ітеративність, інкрементальність і еволюційність в даному випадку є вислів одного і те ж сенсу різними словами з різних точок зору [3].

За висловом Т. Гілба, "еволюція - прийом, призначений для створення видимості стабільності. Шанси успішного створення складної системи будуть максимальними, якщо вона реалізується в серії невеликих кроків і кожен крок містить в собі чітко певний успіх, а також можливість "відкочення" до попереднього успішному етапу в разі невдачі. Перед тим, як пустити в справу всі

ресурси, призначені для створення системи, розробник має можливість одержувати з реального світу сигнали зворотного зв'язку і виправляти можливі помилки в проєкті" [4].

Підхід ІІД має і свої негативні сторони, які, по суті, - зворотна сторона переваг. По-перше, цілісне розуміння можливостей і обмежень проєкту дуже довгий час відсутня. По-друге, при ітераціях доводиться відкидати частину зробленої раніше роботи. По-третє, сумлінність фахівців при виконанні робіт знижується, що психологічно зрозуміло, адже над ними постійно тяжіє відчуття, що "все одно все можна буде переробити і поліпшити пізніше" [3].

Різні варіанти ітераційного підходу реалізовані в більшості сучасних методологій розробки (**RUP, MSF, XP**).

Спіральна модель

Спіральна модель (англ. *spiral model*) Була розроблена в середині 1980-х років Баррі Боем. Вона заснована на класичному **циклі Демінга PDCA** (plan-do-check-act). При використанні цієї моделі ПЗ створюється в кілька **ітерацій** (витків спіралі) **методом прототипування**.

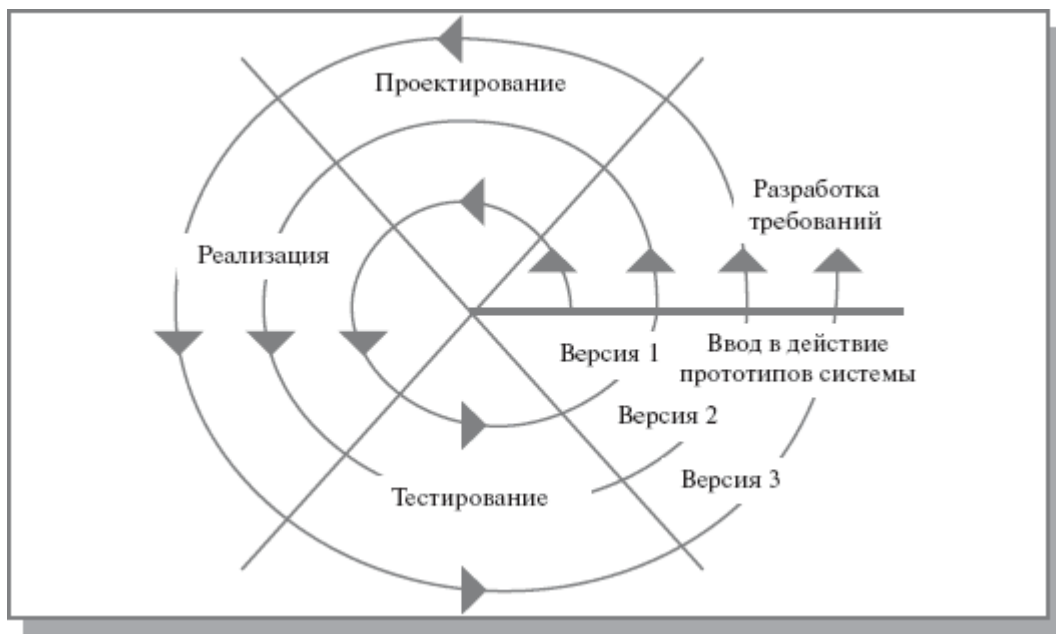


Рис. 2.3. Спіральна модель ЖЦ ІС

Кожна ітерація відповідає створенню фрагмента або версії ПЗ, на ній уточнюються цілі і характеристики проєкту, оцінюється якість отриманих результатів і плануються роботи наступної ітерації.

На кожній ітерації оцінюються:

- ризик перевищення термінів і вартості проєкту;
- необхідність виконання ще однієї ітерації;
- ступінь повноти і точності розуміння вимог до системи;
- доцільність припинення проєкту.

Важливо розуміти, що спіральна модель не є альтернативою еволюційної моделі (моделі IID), а спеціально опрацьованим варіантом. Іноді спіральну модель помилково використовують як синонім еволюційної моделі взагалі, або (не менш помилково) згадують як абсолютно самостійну модель поряд з IID [3].

Відмінною особливістю спіральної моделі є спеціальна увага, що приділяється ризикам, що впливає на організацію життєвого циклу, і контрольним точкам. Боем формулює 10 найбільш поширених (за пріоритетами) ризиків:

1. Дефіцит фахівців.
2. Нереалістичні терміни і бюджет.
3. Реалізація невідповідної функціональності.
4. Розробка неправильного користувацького інтерфейсу.
5. Перфекціонізм, непотрібна оптимізація і відточування деталей.
6. Безперервний потік змін.
7. Брак інформації про зовнішні компоненти, що визначають оточення системи або залучені до інтеграції.
8. Недоліки в роботах, виконуваних зовнішніми (по відношенню до проекту) ресурсами.
9. Недостатня продуктивність одержуваної системи.
10. Розрив у кваліфікації фахівців різних областей.

У сьогоденній спіральної моделі визначено такий загальний набір контрольних точок [5]:

1. Concept of Operations (COO) - концепція (використання) системи;
2. Life Cycle Objectives (LCO) - цілі та зміст життєвого циклу;
3. Life Cycle Architecture (LCA) - архітектура життєвого циклу; тут же можливо говорити про готовність концептуальної архітектури до цільової програмної системи;
4. Initial Operational Capability (IOC) - перша версія створюваного продукту, придатна для дослідної експлуатації;
5. Final Operational Capability (FOC) - готовий продукт, розгорнутий (встановлений і налаштований) для реальної експлуатації.

На практиці найбільшого поширення набули дві основні *моделі життєвого циклу*:

- *каскадна модель* (характерна для періоду 1970-1985 рр.);
- *спіральна модель* (характерна для періоду після 1986.г.).

У ранніх проектах досить простих ІС кожен додаток являло собою єдиний, функціонально та інформаційно незалежний блок. Для розробки такого типу додатків ефективним виявився каскадний спосіб. Кожен етап завершувався після повного виконання та документального оформлення всіх передбачених робіт. Можна виділити наступні позитивні сторони застосування каскадного підходу:

- *на кожному етапі формується закінчений набір проектної документації, який відповідає критеріям повноти і узгодженості;*
- *виконувані в логічній послідовності етапи робіт дозволяють планувати терміни завершення всіх робіт і відповідні витрати.*

Каскадний підхід добре зарекомендував себе при побудові відносно простих ІС, коли на самому початку розробки можна досить точно і повно сформулювати всі вимоги до системи. Основним недоліком цього підходу є те, що реальний процес створення системи ніколи повністю не вкладається в таку жорстку схему, постійно виникає потреба в поверненні до попередніх етапів і уточнення або перегляд раніше прийнятих рішень. У результаті реальний процес створення ІС виявляється відповідним *поетапній моделі з проміжним контролем*.

Однак і ця схема не дозволяє оперативно враховувати виникаючі зміни і уточнення вимог до системи. Узгодження результатів розробки з користувачами проводиться тільки в точках, що плануються після завершення кожного етапу робіт, а загальні вимоги до ІС зафіксовані у вигляді технічного завдання на весь час її створення. Таким чином, користувачі часто отримують систему, не задовольняє їхнім реальним потребам.

Спіральна модель ЖЦ була запропонована для подолання перелічених проблем. На етапах аналізу і проектування реалізовуваність технічних рішень і ступінь задоволення потреб замовника перевіряється шляхом створення прототипів. Кожен виток спіралі відповідає створенню працездатного фрагмента або версії системи. Це дозволяє уточнити вимоги, цілі і характеристики проекту, визначити якість розробки, спланувати роботи наступного витка спіралі. Таким чином поглиблюються і послідовно конкретизуються деталі проекту і в результаті вибирається обґрунтований варіант, який задовольняє дійсним вимогам замовника та доводиться до реалізації.

Ітеративна розробка відображає об'єктивно існуючий спіральний цикл створення складних систем. Вона дозволяє переходити на наступний етап, не чекаючи повного завершення роботи на поточному та вирішити головне завдання - якнайшвидше показати користувачам системи працездатний продукт, тим самим активізуючи процес уточнення і доповнення вимог.

Основна проблема спірального циклу - визначення моменту переходу на наступний етап. Для її рішення вводяться тимчасові обмеження на кожен з етапів *життєвого циклу*, і перехід здійснюється відповідно до плану, навіть якщо не вся запланована робота закінчена. Планування проводиться на основі статистичних даних, отриманих у попередніх проектах, і особистого досвіду розробників.

Незважаючи на наполегливі рекомендації компаній - вендорів і експертів в області проектування і розробки ІС, багато компаній продовжують використовувати *каскадну модель* замість якого-небудь варіанта ітераційної моделі. Основні причини, з яких *каскадна модель* зберігає свою популярність, наступні [3]:

1. Звичка - багато ІТ-фахівці здобували освіту в той час, коли вивчалася тільки *каскадна модель*, тому вона використовується ними і в наші дні.

2. Ілюзія зниження ризиків учасників проекту (замовника і виконавця). *Каскадна модель* припускає розробку закінчених продуктів на кожному етапі: технічного завдання, технічного проекту, програмного продукту і документації користувача. Розроблена документація дозволяє не тільки визначити вимоги до продукту наступного етапу, але і визначити обов'язки сторін, обсяг робіт і терміни, при цьому остаточна оцінка термінів і вартості проекту проводиться на початкових етапах, після завершення обстеження. Очевидно, що якщо вимоги до інформаційної системи змінюються в ході реалізації проекту, а якість документів виявляється невисокою (вимоги неповні і/або суперечливі), то в дійсності використання *каскадної моделі* створює лише ілюзію визначеності і на ділі збільшує ризики, зменшуючи лише відповідальність учасників проекту. При формальному підході менеджер проекту реалізує тільки ті вимоги, які містяться в специфікації, спирається на документ, а не на реальні потреби бізнесу. Є два основних типи контрактів на розробку ПЗ. Перший тип передбачає виконання певного обсягу робіт за певну суму у визначені терміни (*fixed price*). Другий тип припускає почасову оплату роботи (*time work*). Вибір того чи іншого типу контракту залежить від ступеня визначеності задачі. *Каскадна модель* з певними етапами та їх результатами краще пристосована для укладення контракту з оплатою за результатами роботи, а саме цей тип контрактів дозволяє отримати повну оцінку вартості проекту до його завершення. Більш імовірно укладання контракту з погодинною оплатою на невелику систему, з відносно невеликою вагою в структурі витрат підприємства. Розробка та впровадження інтегрованої інформаційної системи вимагає істотних фінансових витрат, тому використовуються контракти з фіксованою ціною, і, отже, *каскадна модель* розробки і впровадження. *Спіральна модель* частіше застосовується при розробці інформаційної системи силами власного відділу ІТ підприємства.
3. Проблеми впровадження при використанні ітераційної моделі. У деяких областях *спіральна модель* не може застосовуватися, оскільки неможливо використання/тестування продукту, що володіє неповною функціональністю (наприклад, військові розробки, атомна енергетика і т.д.). Поетапне ітераційне впровадження інформаційної системи для бізнесу можливо, але пов'язане з організаційними складнощами (перенесення даних, інтеграція систем, зміна бізнес-процесів, облікової політики, навчання користувачів). Трудовитрати при поетапному ітераційному впровадженні виявляються значно вищими, а управління проектом вимагає справжнього мистецтва. Передбачаючи вказані складнощі, замовники вибирають *каскадну модель*, щоб "впроваджувати систему один раз".

Кожна зі стадій створення системи передбачає виконання певного обсягу робіт, які представляються у вигляді *процесів ЖЦ*. *Процес* визначається як сукупність взаємопов'язаних дій, що перетворюють вхідні дані у вихідні. Опис

кожного процесу включає в себе перелік вирішуваних завдань, вихідних даних і результатів.

Існує цілий ряд стандартів, що регламентують ЖЦ ПЗ, а в деяких випадках і процеси розробки.

Значний внесок у теорію проектування та розробки інформаційних систем внесла компанія IBM, запропонувавши ще в середині 1970-х років методологію BSP (Business System Planning - методологія організаційного планування). Метод структурування інформації з використанням матриць перетину бізнес-процесів, функціональних підрозділів, функцій систем обробки даних (інформаційних систем), інформаційних об'єктів, документів і баз даних, запропонований в BSP, використовується сьогодні не тільки в IT-проектах, але і в проектах з реінжинірингу бізнес-процесів, зміни організаційної структури. Найважливіші кроки процесу BSP, їх послідовність (одержати підтримку вищого керівництва, визначити процеси підприємства, визначити класи даних, провести інтерв'ю, обробити і організувати дані інтерв'ю) можна зустріти практично у всіх формальних методиках, а також у проектах, що реалізуються на практиці.

Методології розробки ПЗ

- Rational Unified Process (RUP) пропонує ітеративну модель розробки, що включає чотири фази: початок, дослідження, побудова та впровадження. Кожна фаза може бути розбита на етапи (ітерації), в результаті яких випускається версія для внутрішнього або зовнішнього використання. Проходження через чотири основні фази називається циклом розробки, кожен цикл завершується генерацією версії системи. Якщо після цього робота над проектом не припиняється, то отриманий продукт продовжує розвиватися і знову минає ті ж фази. Суть роботи в рамках RUP - це створення і супровід моделей на базі UML [6].
- Custom Development Method, методика Oracle з розробки прикладних інформаційних систем - технологічний матеріал, деталізований до рівня заготовок проектних документів, розрахованих на використання в проектах із застосуванням Oracle. Застосовується CDM для класичної моделі ЖЦ (передбачені всі роботи/завдання й етапи), а також для технологій "швидкої розробки" (Fast Track) або "полегшеного підходу", рекомендованих у випадку малих проектів.
- Microsoft Solution Framework (MSF) схожа з RUP, так само включає чотири фази: аналіз, проектування, розробка, стабілізація - є ітераційною; припускає використання об'єктно-орієнтованого моделювання. MSF у порівнянні з RUP більшою мірою орієнтована на розробку бізнес-додатків.
- Extreme Programming (XP). Екстремальне програмування (найновіша серед розглянутих методологій) сформувалося в 1996 році. В основі методології командна робота, ефективна комунікація між замовником і виконавцем

протягом всього проекту з розробки ІС, а розробка ведеться з використанням послідовно допрацьовуваних прототипів.

Розділ 2. Засоби та середовища створення ПЗ.

Лекція 3. Інтегровані середовища розробки ПЗ

Інтегровані середовища розробки ПЗ

- Інтегровані середовища розробки ПЗ.
- Системи управління проектами (Redmine, JIRA). [1, с. 15-45; 2, с. 19-40]

Інтегроване Середовище Розробки (ICP) — від Integrated Development Environment (також можливі інтерпретації Integrated Design Environment — інтегроване середовище проектування; чи Integrated Debugging Environment — інтегроване середовище налагодження) — це комп'ютерна програма, що допомагає програмістові розробляти нове програмне забезпечення чи модифікувати (удосконалювати) вже існуюче.

Інтегровані середовища розробки зазвичай складаються з редактора сирцевого коду, компілятора чи/або інтерпретатора, засобів автоматизації збірки, та зазвичай налагоджувача. Іноді сюди також входять системи контролю версій, засоби для профілювання, а також різноманітні засоби та утиліти для спрощення розробки графічного інтерфейсу користувача. Багато сучасних ICP також включають оглядач класів, інспектор об'єктів та діаграм ієрархії класів для використання об'єктно-орієнтованого підходу у розробці програмного забезпечення. Сучасні ICP часто підтримують розробку на декількох мовах програмування.

Eclipse. Початок

Eclipse (вимовляється «і-клі́нс», від англійського «затемнення») — вільне модульне інтегроване середовище розробки програмного забезпечення. Розробляється і підтримується Eclipse Foundation. Написаний в основному на Java, і може бути використаний для розробки застосунків на Java і, за допомогою різних плагінів, на інших мовах програмування, включаючи Ada, C, C++, COBOL, Fortran, Perl, PHP, Python, R, Ruby (включно з каркасом Ruby on Rails), Scala, Clojure та Scheme. Середовища розробки зокрема включають Eclipse ADT (Ada Development Toolkit) для Ada, Eclipse CDT для C/C++, Eclipse JDT для Java, Eclipse PDT для PHP.

Початок коду йде від IBM VisualAge[1], він був розрахований на розробників Java, складаючи Java Development Tools (JDT). Але користувачі могли розширяти можливості, встановлюючи написані для програмного каркасу Eclipse плагіни, такі як інструменти розробки під інші мови програмування, і могли писати і вносити свої власні плагіни і модулі.

Випущена на умовах Eclipse Public License, Eclipse є вільним програмним забезпеченням. Він став одним з перших IDE під GNU Classpath і без проблем працює під IcedTea.

Eclipse являє собою фреймворк для розробки модульних кросс-платформових застосунків із низкою особливостей:

- можливість розробки ПЗ на багатьох мовах програмування (рідною є Java);
- крос-платформова;
- модульна, призначена для подальшого розширення незалежними розробниками;
- з відкритим вихідним кодом;
- розробляється і підтримується фондом Eclipse, куди входять такі постачальники ПЗ, як IBM, Oracle, Borland.

Спочатку проект розроблявся в IBM як корпоративний стандарт IDE, настановлений на розробки на багатьох мовах під платформи IBM. Потім проект було перейменовано на Eclipse і надано для подальшого розвитку спільноті.

Eclipse насамперед повноцінна Java IDE, націлена на групову розробку, має засоби роботи з системами контролю версій (підтримка CVS входить у поставку Eclipse, активно розвиваються кілька варіантів SVN модулів, існує підтримка VSS та інших). З огляду на безкоштовність, у багатьох організаціях Eclipse — корпоративний стандарт для розробки ПЗ на Java.

Друге призначення Eclipse — служити платформою для нових розширень. Такими стали C/C++ Development Tools (CDT), розроблювані інженерами QNX разом із IBM, засоби для підтримки інших мов різних розробників. Безліч розширень доповнює Eclipse менеджерами для роботи з базами даних, серверами застосунків та інших.

З версії 3.0 Eclipse став не монолітною IDE, яка підтримує розширення, а набором розширень. У основі лежать фреймворки OSGi, і SWT/JFace, на основі яких розроблений наступний шар — платформа і засоби розробки повноцінних клієнтських застосунків RCP (Rich Client Platform). Платформа RCP є базою для розробки різних RCP програм як торент-клієнт Azareus чи File Arranger. Наступний шар — платформа Eclipse, що є набором розширень RCP — редактори, панелі, перспективи, модуль CVS і модуль Java Development Tools (JDT).

Eclipse написана на Java, тому є платформи-незалежним продуктом, крім бібліотеки графічного інтерфейсу SWT, яка розробляється окремо для більшості поширених платформ. Бібліотека SWT використовує графічні засоби платформи (ОС), що забезпечує швидкість і звичний зовнішній вигляд інтерфейсу користувача.

Відповідно до IDC, із Eclipse працюють 2.3 мільйона розробників.

Архітектура Eclipse

Основою Eclipse є платформа розширеного клієнта (RCP — від англ. *rich client platform*). Її складають такі компоненти:

- Ядро платформи (завантаження Eclipse, запуск модулів);
- OSGi (стандартне середовище постачання комплектів);

- SWT (стандартний інструментарій віджетів);
- JFace (файлові буфери, робота з текстом, текстові редактори);
- Робоче середовище Eclipse (панелі, редактори, проекції, майстри).

GUI в Eclipse написаний з використанням інструментарію SWT. Останній, на відміну від Swing (який лише емулює окремі графічні елементи використовуваної платформи), дійсно використовує графічні компоненти даної системи. Призначений для користувача інтерфейс Eclipse також залежить від проміжного шару GUI, званого JFace, який спрощує побудову призначеного для користувача інтерфейсу, що базується на SWT.

Гнучкість Eclipse забезпечується за рахунок модулів, що підключаються, завдяки чому можлива розробка не тільки на Java, але і на інших мовах, таких як C/C++, Perl, Groovy, Ruby, Python, PHP, ErLang та інших.

Проекти Eclipse

Платформа

- Eclipse Project (Eclipse.org) (англ.) — власне, проект Eclipse, включає в себе
 - Platform (Eclipse Platform, Platform) — каркас
 - PDE (Plug-in Development Environment, PDE) — інструмент розширення Eclipse-платформи за допомогою Eclipse-плагінів
 - JDT (Java Development Tools, JDT) — інструмент розробки Java-програм та Eclipse-плагінів зокрема
- RCP (Rich Client Platform, RCP) — платформа розширеного клієнта, мінімальний набір плагінів (org.eclipse.core.runtime, org.eclipse.ui) для побудови програми з графічним інтерфейсом

Приклади проектів

Крім того, у склад Eclipse входять такі проекти (перелічені лише кілька [1]):

- Aperi (від латинського «відкривати») — open source система управління системами мережного зберігання даних
- BIRT (Business Intelligence and Reporting Tools) (англ.) — Web- і PDF-звіти
- DTP (Data Tools Platform) (англ.) — розробка систем, що управляються даними (data-centric systems), зокрема даними в реляційних базах; управління програмами з великою кількістю конекторів
- GEF (Graphical Editor Framework) (англ.) — фреймворк для побудови вбудованих графічних редакторів
- Jazz (Jazz.net(англ.) [2](рос.)) — інструмент для співпраці
- *Modeling* (eclipse.org/modeling/)

- EMF (eclipse.org/modeling/emf/) Середовище моделювання Eclipse — засіб для створення моделей і генерації коду для побудови інструментів та інших застосунків, що базуються на структурованій моделі даних, зі специфікації моделі, прописаної в XMI
- UML2 ([3]) — реалізація метамоделі UML 2.0 для підтримки розробки інструментів моделювання
- *Tools* (eclipse.org/tools/)
 - AspectJ ([4]) — аспектно-орієнтоване розширення мови Java
 - CDT (C/C++ Development Tools) (англ.) — середовище розробки на C/C++ (C/C++ IDE)
- TPTP (Test & Performance Tools Platform) (англ.) — розробка інструментів тестування, — зневаджувачі, профайлери тощо
- VE (Visual Editor Project) (англ.) — розробка інструментів GUI
- WTP (Web Tools Platform Project) (англ.) — інструменти розробки веб-застосунків J2EE
 - редактори HTML, JavaScript, CSS, JSP, SQL, XML, DTD, XSD і WSDL
 - графічні редактори для XSD и WSDL
 - майстри і провідник веб-служб, інструменти тестування WS-I
 - інструменти для доступу і побудови запитів і моделей баз даних
- Комунікаційне середовище Eclipse (ECF) націлене на створення комунікаційних застосунків на платформі Eclipse.
- Проект розробки програмного забезпечення для приладів (DSDP)
- Pulsar — інструментальна платформа для уніфікованої розробки застосунків для смартфонів
- Платформа паралельних інструментів (PTP) забезпечує портовану, масштабовану, засновану на стандартах платформу паралельних інструментів, яка дозволить полегшити інтеграцію інструментів, специфічних для паралельної комп'ютерної архітектури.
- Платформа вбудованого розширеного клієнта (eRCP) — призначена для розширення RCP на вбудовані пристрої. У eRCP входить набір компонентів, які є підмножиною компонентів RCP. Вона дозволить перенести модель застосунку, використовуючи на настільних комп'ютерах, на інші пристрої.
- DLTK (DLTK) — інтегроване середовище розробки для динамічних мов програмування.
- Jetty
- Eclipse Orion — інтегроване середовище розробки, що працює у веб-браузері

Кількість нових підпроектів (як керованих Eclipse Foundation, так і сторонніх) швидко збільшується. Доводиться координувати зусилля величезної кількості розробників і пропонувати загальні правила — «Eclipse Development Process» (Project Lifecycle).

Redmine

Redmine — вільний серверний веб-застосунок для управління проектами та відстежування помилок. До системи входить календар-планувальник та діаграми Ганта для візуального представлення ходу робіт за проектом та строків виконання. Redmine написано на Ruby і є застосунком розробленим з використанням відомого веб-фреймворку Ruby on Rails, що означає легкість в розгортанні системи та її адаптації під конкретні вимоги. Для кожного проекту можна вести свої вікі та форуми.

Функціональні можливості:

- Ведення декількох проектів
- Гнучка система доступу з використанням ролей
- Система відстеження помилок
- Діаграми Ганта та календар
- Ведення новин проекту, документів та управління файлами
- Сповіщення про зміни за допомогою RSS-потоків та електронної пошти
- Власна Wiki для кожного проекту
- Форуми для кожного проекту
- Облік часових витрат
- Налаштування власних (custom) полів для задач, затрат часу, проектів та користувачів
- Легка інтеграція із системами керування версіями (SVN, CVS, Git, Mercurial, Bazaar и Darcs)
- Створення записів про помилки на основі отриманих листів
- Підтримка LDAP автентифікації
- Можливість самореєстрації нових користувачів
- Багатомовний інтерфейс (у тому числі українська та російська мови)
- Підтримка СКБД: MySQL, PostgreSQL, SQLite.

Atlassian JIRA

Atlassian JIRA — система відстежування помилок, призначена для організації спілкування з користувачами, хоча в деяких випадках може бути використана для управління проектами.

Розроблена компанією Atlassian Software Systems. Платна. Має веб-інтерфейс. Назва системи (JIRA) отримано шляхом модифікації японської назви Годзіла ("Gojira"), що в свою чергу є алюзією на назву конкуруючого продукту — Bugzilla.[1] JIRA створювалася в якості заміни Bugzilla і багато в чому повторює архітектуру Bugzilla.

Система дозволяє працювати з декількома проектами. Для кожного з проектів створює та веде схеми безпеки і схеми оповіщення.

Всередині компанії «Atlassian Software Systems» для управління процесом розробки використовується «стіна смерті». «Стіна смерті» — це дошка, на яку чіпляються роздруківки запитів користувачів з JIRA і по стану якої відстежується хід розробки. Після закінчення розробки, програмісти інформують користувачів про результати за допомогою JIRA.

Лекція 4. Управління версіями та автоматизація збірки ПЗ

Системи управління версіями документів

- Системи управління версіями документів, архітектурні особливості (CVS, SVN, Git).
- Інструменти автоматизації зборки проєктів (утиліта make, системи CMake, Ant та Maven). [1, с. 15-45; 2, с. 19-40]

Система керування версіями (англ. *source code management*, SCM) — програмний інструмент для керування версіями одиниці інформації: вихідного коду програми, скрипту, веб-сторінки, веб-сайту, 3D моделі, текстового документу тощо.

Система керування версіями — це потужний інструмент, який дозволяє одночасно, без завад один одному, проводити роботу над груповими проєктами.

Системи керування версіями зазвичай використовуються при розробці програмного забезпечення для відстеження, документування та контролю над поступовими змінами в електронних документах: у сирцевого коду застосунків, кресленнях, електронних моделях та інших документах, над змінами яких одночасно працюють декілька людей.

Кожна версія позначається унікальною цифрою чи літерою, зміни документу занотовуються. Зазвичай також зберігається автор зробленої зміни та її час.

Інструменти для контролю версій входять до складу багатьох інтегрованих середовищ розробки.

Система керування версіями існують двох основних типів: з централізованим сховищем та розподіленням.

Система контролю дозволяє зберігати попередні версії файлів та завантажувати їх за потребою. Вона зберігає повну інформацію про версію кожного з файлів, а також повну структуру проєкту на всіх стадіях розробки. Місце зберігання даних файлів називають репозиторієм. В середині кожного з репозиторіїв можуть бути створені паралельні лінії розробки — гілки.

Гілки зазвичай використовують для зберігання експериментальних, незавершених(alpha, beta) та повністю робочих версій проєкту(final). Більшість систем контролю версії дозволяють кожному з об'єктів присвоювати тегі, за допомогою яких можна формувати нові гілки та репозиторії.

Використання системи контролю версії є необхідним для роботи над великими проєктами, над якими одночасно працює велика кількість розробників. Системи контролю версії надають ряд додаткових можливостей:

- Можливість створення різних варіантів одного документу;
- Документування всіх змін (коли ким було змінено/додано, хто який рядок змінив);
- Реалізує функцію контролю доступу користувачів до файлів. Є можливість його обмеження;
- Дозволяє створювати документацію проекту з поетапним записом змін в залежності від версії;
- Дозволяє давати пояснення до змін та документувати їх;

Невеликий словник основних термінів-сленгів

- Транк (trunk) — основна гілка коду
- Бранч (branch) — відгалуження
- Чекін (Check in (submit, commit)) — відправлення коду в репозиторій
- Чекаут (Check out) — одержання зміни з репозиторію
- Конфлікти — виникають, коли кілька людей правлять один і той же код, конфлікти можна вирішувати
- Патч — шматок з записаними змінами, які можна застосувати до сховища з кодом

Світовий досвід

На сьогодні у світі існує безліч організацій, які використовують системи контролю версій у своїй повсякденній роботі. Практично кожна фірма, що виробляє програмне забезпечення використовує їх. Але крім комерційних організацій системи контролю версій використовуються в університетах у всьому світі. Так наприклад можна виділити статтю двох професорів університету в Торонто: Gregory V. Wilson і Karen Reid. У своєму новому проекті вони намагаються створити середовище для студентів, в якому вони хочуть використати системи контролю версій, issue trackers, web-логи. Це не поодинокий випадок. Деякі університети прагнуть використовувати і створювати такі середовища, в яких усі студенти технічних спеціальностей змогли б мати систему контролю версій, web-логи та інше. Цікаво також, що в США департамент освіти також займається цим питанням. Впровадження подібних систем відбувається на державному рівні.

Види систем контролю версії

Централізовані системи контролю версії

Централізована система контролю версії (клієнт-серверна) — система, дані в якій зберігаються в єдиному «серверному» сховищі. Весь обмін файлами відбувається з використанням центрального сервера. Є можливість створення та роботи з локальними репозиторіями (робочими копіями).

Переваги:

- Загальна нумерація версій;
- Дані знаходяться на одному сервері;
- Можлива реалізація функції блокування файлів;
- Можливість керування доступом до файлів;

Недоліки:

- Потреба в мережевому з'єднанні для оновлення робочої копії чи збереження змін;

До таких систем відносять Subversion, Team Foundation Server.

Розподілені системи контролю версії

Розподілена система контролю версії (англ. Distributed Version Control System, DVCS) — система, яка використовує замість моделі клієнт-сервер, розподілену модель зберігання файлів. Така система не потребує сервера, адже всі файли знаходяться на кожному з комп'ютерів.

Переваги:

- Кожний з розробників працює зі своїм власним репозиторієм;
- Рішення щодо злиття гілок приймається керівникам проекту;
- Немає потреби в мережевому з'єднанні;

Недоліки:

- Немає можливості контролю доступу до файлів;
- Відсутня загальна нумерація версії файла;
- Значно більша кількість необхідного дискового простору;
- Немає можливості блокування файлів;

До таких систем відносять Git, Mercurial, SVK, Monotone, Codeville, BitKeeper.

Виникнення конфліктів та їх вирішення

Конфлікти можуть виникнути при операції злиття, розгалужені від різних джерел. При зміні одного і того ж рядка різними користувачами виникає конфлікт. Тобто якщо один користувач внесе зміни в документ та виконає злиття — то конфлікту не буде. Але якщо після цього інший користувач змінить документ в тих же рядках, що і перший — то виникне конфлікт. В такому випадку конфлікт повинен вирішувати другий користувач (власне, із-за якого і виник конфлікт), або система (як правило, вирішення конфліктів покладається на користувачів).

Вирішення конфліктів

Ручний режим:

- Шляхом ручних змін конфліктних рядків.
- Примусовий запис своєї версії поверх попередньої (зафіксованої).

Автоматичний режим:

- Ординальна заміна. Схожа на примусовий запис, але відрізняється тим, що кожен користувач має свій пріоритет. Записуються зміни того в кого пріоритет вище.

Зауваження. Але далеко не в кожній системі існує пріоритет користувачів.

Поширені системи керування версіями

- Concurrent Versions System (CVS)
- Subversion (SVN)
- Revision Control System (RCS)
- Perforce
- Microsoft Visual Source Safe (VSS)
- Mercurial
- Bazaar
- Darcs
- Git

Автоматизація збору — етап написання скриптів або автоматизація широкого спектра завдань стосовно до ПЗ, який застосовується розроблювачами в їхній повсякденній діяльності, включаючи такі дії, як:

- Компіляція вихідного коду в бінарний код

- збірка бінарного коду
- виконання тестів
- розгортання програми на виробничій платформі
- написання супровідної документації або опис змін нової версії

Історично так склалося, що розроблювачі застосовували автоматизацію збору для виклику компіляторів і лінковщиків зі скрипту збору, на відміну від виклику компілятора з командного рядка. Досить просто за допомогою командного рядка передати один вихідний модуль компіляторів, а потім і лінковщику для створення кінцевого об'єкта. Однак, при спробі скомпілювати або злінувати безліч модулів з вихідним кодом, причому в певному порядку, здійснення цього процесу вручну за допомогою командного рядка виглядає занадто незручним. Більш привабливою альтернативою є мова скриптів, яка підтримується утилітою Make. Даний інструмент дозволяє писати скрипти збірки, визначаючи порядок їхнього виклику, етапи компіляції й лінковки для збірки програми. GNU Make [1] також надає такі додаткові можливості, як наприклад, «залежності» («*makedepend*»), які дозволяють указати умови підключення вихідного коду на кожному етапі збору. Це й стало початком автоматизації збірки. Основною метою була автоматизація викликів компіляторів і лінковщиків. По мірі зростання і ускладнення процесу збірки розроблювачі почали додавати дії до й після викликів компіляторів, як наприклад, перевірку (*check-out*) версій об'єктів, які копіюються, на тестову систему. Термін «автоматизація збірки» уже містить у собі керування діями до і після компіляції й лінковки, так само як і дії при компіляції й лінковці.

Нове покоління інструментів

В останні роки рішення по керуванню збіркою зробили ще більш зручний і керованим процес автоматизованої збірки. Для виконання автоматизованої збірки й контролю цього процесу існують як комерційні, так і відкриті рішення. Деякі рішення націлені на автоматизацію кроків до і після виклику складальних скриптів, а інші виходять за рамки дій до й після обробки скриптів і повністю автоматизують процес компіляції й лінковки, рятуючи від ручного написання скриптів. Такі інструменти надзвичайно корисні для безперервної інтеграції, коли потрібні часті виклики компіляції й обробка проміжних збірок.

Переваги

- Поліпшення якості продукту
- Прискорення процесу компіляції й лінковки
- Рятування від зайвих дій
- Мінімізація «поганих (некоректних) збірок»
- Рятування від прив'язки до конкретної людини
- Ведення історії збірок і релізів для розбору випусків

- Економія часу й грошей завдяки причинам, зазначеним вище.[6]

Типи

- Автоматизація за запитом (On-Demand automation): запуск користувачем скрипта в командному рядку.
- Запланована автоматизація (Scheduled automation): безперервна інтеграція, що відбувається у вигляді нічних збірок.
- Умовна автоматизація (Triggered automation): безперервна інтеграція, що виконує збірки при кожному підтвердженні зміни коду (commit) у системі керування версіями.

Одна з особливих форм автоматизації збірки — автоматичне створення make-файлів (makefiles). Ці файли сумісні з такими інструментами як:

- GNU Automake
- CMake
- imake
- qmake
- nmake
- wmake
- Apache Ant
- Apache Maven
- OpenMake Meister

Вимоги до систем збірки

Базові вимоги:

1. Часті або нічні збірки для своєчасного виявлення проблем.[7][8][9]
2. Підтримка керування залежностями вихідного коду (Source Code Dependency Management)
3. Обробка різницевої збірки
4. Повідомлення при збігу вихідного коду (після збірки) з наявними бінарними файлами.
5. Прискорення збірки
6. Звіт про результати компіляції і ліנקовки.

Додаткові вимоги:[10]

1. Створення опису змін (release notes) і іншої супутньої документації (наприклад, керівництва).
2. Звіт про статус збірки
3. Звіт про успішне/неуспішне проходження тестів.
4. Підсумовування доданих/змінених/вилучених особливостей у кожній новій збірці

Apache Ant

Apache Ant (англ. *ant* — мураха і водночас акронім — «Another Neat Tool») — java-утиліта для автоматизації процесу збирання програмного продукту.

Ant — платформонезалежний аналог UNIX-утиліти *make*, але з використанням мови Java, він вимагає платформи Java, і краще пристосований для Java-проектів. Найпомітніша безпосередня різниця між Ant та Make те, що Ant використовує XML для опису процесу збирання і його залежностей, тоді як Make має свій власний формат Makefile. За замовчуванням XML-файл називається *build.xml*.

Ant був створений в рамках проекту Jakarta, сьогодні — самостійний проект першого рівня Apache Software Foundation.

Перша версія була розроблена інженером Sun Microsystems Джеймсом Девідсоном (James Davidson), який потребував утиліти подібної *make*, розробляючи першу референтну реалізацію J2EE.

На відміну від *make*, утиліта Ant повністю незалежна від платформи, потрібно лише наявність на застосовуваній системі встановленої робочого середовища Java — JRE. Відмова від використання команд операційної системи і формат XML забезпечують переносимість сценаріїв.

Управління процесом складання відбувається за допомогою XML-сценарію, який також називають Build-файлом. У першу чергу цей файл містить визначення проекту, що складається з окремих цілей (Targets). Цілі порівняні з процедурами в мовах програмування і містять виклики команд-завдань (Tasks). Кожне завдання являє собою неподільну, атомарному команду, що виконує певну елементарну дію.

Між цілями можуть бути визначені залежності — кожна мета виконується тільки після того, як виконані всі цілі, від яких вона залежить (якщо вони вже були виконані раніше, повторного виконання не здійснюється).

Типовими прикладами цілей є *clean* (видалення проміжних файлів), *compile* (компіляція всіх класів), *deploy* (розгортання програми на сервері). Конкретний набір цілей та їхнього взаємозв'язку залежать від специфіки проекту.

Ant дозволяє визначати власні типи завдань шляхом створення Java-класів, що реалізують певні інтерфейси.

Розділ 3. Проектування ПЗ з використанням шаблонів.

Лекція 5. Шаблони проектування ПЗ.

- Проектування архітектури ПЗ.
- Шаблони проектування ПЗ.
- Класифікація шаблонів проектування.
- Графічна нотація [1, с. 140-141, 162-173]

Проектування ПЗ

Проектування ПЗ – це процес визначення архітектури, набору компонентів, їх інтерфейсів, інших характеристик системи і кінцевого складу програмного продукту. Область знань «Проектування ПЗ (Software Design)» складається з таких розділів:

- базові концепції проектування ПЗ (Software Design Basic Concepts),
- ключові питання проектування ПЗ (Key Issue in Software Design),
- структура й архітектура ПЗ (Software Structure and Architecture),
- аналіз і оцінка якості проектування ПЗ (Software Design Quality Analysis and Evaluation),
- нотації проектування ПЗ (Software Design Notations),
- стратегія і методи проектування ПЗ (Software Design Strategies and Methods).

Базова концепція проектування ПЗ

це методологія проектування архітектури за допомогою різних методів (об'єктного, компонентного й ін.), процеси ЖЦ (стандарт ISO/IEC 12207) і техніки – декомпозиція, абстракція, інкапсуляція й ін. На початкових стадіях проектування предметна область декомпонується на окремі об'єкти (при об'єктно-орієнтованому проектуванні) або на компоненти (при компонентному проектуванні). Для подання архітектури програмного забезпечення вибираються відповідні артефакти (нотації, діаграми, блок-схеми і методи).

Ключові питання проектування

це декомпозиція програм на функціональні компоненти для незалежного і одночасного їхнього виконання, розподіл компонентів у середовищі функціонування і їх взаємодія між собою, забезпечення якості і живучості системи й ін.

Проектування архітектури ПЗ

проводиться архітектурним стилем, заснованим на визначенні основних елементів структури – підсистем, компонентів, об'єктів і зв'язків між ними.

Архітектура проекту – високорівневе подання структури системи і специфікація її компонентів. Архітектура визначає логіку системи через окремі компоненти системи настільки детально, наскільки це необхідно для написання коду, а також визначає зв'язки між компонентами. Існують і інші види подання структур, засновані на проектуванні зразків, шаблонів, сімейств програм і каркасів програм.

Один з інструментів проектування архітектури – **патерн (шаблон)**. Це типовий конструктивний елемент ПЗ, що задає взаємодію об'єктів (компонентів) проектованої системи, а також ролі і відповідальності виконавців. Основна мова опису – UML. Патерн може бути **структурним**, що містить у собі структуру типової композиції з об'єктів і класів, об'єктів, зв'язків і ін.; **поведінковим**, що визначає схеми взаємодії класів об'єктів і їх поведінку, задається діаграмами діяльності, взаємодії, потоків керування й ін.; **погоджувальним**, що відображає типові схеми розподілу ролей екземплярів об'єктів і способи динамічної генерації структур об'єктів і класів.

Аналіз і оцінка якості проектування ПЗ

– це заходи щодо аналізу сформульованих у вимогах атрибутів якості, функцій, структури ПЗ, з перевірки якості результатів проектування за допомогою *метрик* (функціональних, структурних і ін.) і *методів моделювання і прототипування*.

Нотації проектування

дозволяють представити опис об'єкта (елемента) ПЗ і його структуру, а також поведінку системи за цим об'єктом. Існує два типи нотацій: структурна, поведінкова, та множина їх різних представлень.

Структурні нотації – це структурне, блок-схемне або текстове подання аспектів проектування структури ПЗ з об'єктів, компонентів, їх інтерфейсів і взаємозв'язків. До нотацій відносять формальні мови специфікацій і проектування: ADL (Architecture Description Language), UML (Unified Modeling Language), ERD (Entity–Relation Diagrams), IDL (Interface Description Language) тощо. Нотації містять у собі мовний опис архітектури й інтерфейсу, діаграм класів і об'єктів, діаграм сутність–зв'язок, конфігурації компонентів, схем розгортання, а також структурні діаграми, що задають у наочному вигляді оператори циклу, розгалуження, вибору і послідовності.

Поведінкові нотації відбивають динамічний аспект роботи системи та її компонентів. Ними можуть бути діаграми потоків даних (Data Flow), діяльності (Activity), кооперації (Colloboration), послідовності (Sequence), таблиці прийняття рішень (Decision Tables), передумови і постумови (Pre-Post Conditions), формальні мови специфікації (Z, VDM, RAISE) і проектування.

Стратегія і методи проектування ПЗ.

До стратегій відносять: проектування вгору, вниз, абстрагування, використання каркасів і ін. Методи є функціонально-орієнтовані, структурні, які базуються на структурному аналізі, структурних картах, діаграмах потоків даних й ін. Вони орієнтовані на ідентифікацію функцій і їх уточнення знизу-вгору, після цього уточнюються діаграми потоків даних і проводиться опис процесів.

В об'єктно-орієнтованому проектуванні ключову роль відіграє спадкування, поліморфізм й інкапсуляція, а також абстрактні структури даних і відображення об'єктів. Підходи, орієнтовані на структури даних, базуються на методі Джексона і використовуються для подання вхідних і вихідних даних структурними діаграмами. Метод UML призначений для опису сценаріїв роботи проекту у наочному діаграмному вигляді. Компонентне проектування ґрунтується на використанні готових компонентів (reuse) з визначеними інтерфейсами і їх інтеграції в конфігурацію, як основи розгортання компонентної системи для її функціонування в операційному середовищі.

Формальні методи опису програм ґрунтуються на специфікаціях, аксіомах, описах деяких попередніх умов, твердженнях і постулатах, що визначають заключну умову одержання програмою правильного результату. Специфікація функцій і даних, якими ці функції оперують, а також умови і твердження – основа доведення правильності програми.

Шаблони проектування програмного забезпечення

Шаблони проектування програмного забезпечення (англ. *software design patterns*) — ефективні способи вирішення задач проектування програмного забезпечення. Шаблон не є закінченим зразком, який можна безпосередньо транслювати в програмний код. *Об'єктно-орієнтований шаблон найчастіше є зразком вирішення проблеми і відображає відношення між класами та об'єктами, без вказівки на те, як буде зреалізовано це відношення.*

У 70-х роках двадцятого сторіччя архітектор Кристофер Александр (англ. *Christopher Alexander*) склав перелік шаблонів проектування. В області архітектури ця ідея не отримала такого розвитку, котрого вона досягла пізніше в області розробки програмного забезпечення.

У 1987 році Кент Бек (англ. *Kent Beck*) і Вард Каннігем (англ. *Ward Cunningham*) узяли ідеї Крістофера Александра та розробили шаблони відповідно до розробки програмного забезпечення для розробки графічних оболонок мовою Smalltalk.

У 1988 році Ерік Гамма (англ. *Erich Gamma*) почав писати докторську роботу при цюрихському університеті про загальну переносимість цієї методики на розробку програм.

У 1989—1991 роках Джеймс Коплін (англ. *James Coplien*) трудився над розробкою ідіом для програмування мовою C++ та опублікував у 1991 році книгу «Advanced C++ Idioms».

У цьому ж році Ерік Гамма закінчує свою докторську роботу та переїздить до США, де у співробітництві з Річардом Хелмом (англ. *Richard Helm*), Ральфом Джонсоном (англ. *Ralph Johnson*) та Джоном Вліссідсом (англ. *John Vlissides*) публікує книгу «Design Patterns — Elements of Reusable Object-Oriented Software». У цій книзі описані 23 шаблони проектування. Також команда авторів цієї книги відома суспільству під назвою Банда чотирьох (англ. *Gang of Four - GoF*). Саме ця книга послужила приводом до прориву методу шаблонів.

Типи шаблонів GOF

- Твірні (*производящие*) шаблони
- Структурні шаблони
- Шаблони поведінки

Шаблони GRASP

Також існує інша група шаблонів проектування, що отримала назву GRASP - General Responsibility (ответственность) Assignment Software Patterns. Опис цих шаблонів наводить Крег Ларман у своїй книзі [1]. Шаблони GRASP формулюють найбільш базові принципи розподілу обов'язків між типами.

Лекція 6. Структурні шаблони проектування: Composite та Decorator.

- Шаблон Composite: мотивація, структура, учасники, відносини, шляхи застосування та результат використання.
- Приклад реалізації шаблону Composite.
- Шаблон Decorator: мотивація, структура, учасники, відносини, шляхи застосування та результат використання.
- Приклад реалізації шаблону Decorator. [1, с. 173-183, 203-213]

Структурні шаблони (англ. *structural patterns*) — шаблони проектування, у яких розглядається питання про те, як із класів та об'єктів утворюються більші за розмірами структури.

Структурні шаблони рівня *класу* використовують спадковість для утворення композицій із інтерфейсів та реалізацій.

Структурні шаблони рівня *об'єкта* компонують об'єкти для отримання нової функціональності. Додаткова гнучкість у цьому разі пов'язана з можливістю змінювати композицію об'єктів під час виконання, що є неприпустимим для статичної композиції класів.

Перелік структурних шаблонів

- Адаптер (Adapter)
- Декоратор (Decorator)
- Замісник (Proxy) - посередник
- Компонувальник (Composite)
- Міст (Bridge)
- Легковаговик (Flyweight) – легковес, противовес, грузик
- Фасад (Facade)

Компонувальник Composite

Компонувальник Composite — структурний шаблон який об'єднує об'єкти в ієрархічну деревовидну структуру, і дозволяє уніфіковане звертання для кожного елемента дерева.

Призначення

Дозволяє користувачам будувати складні структури з простіших компонентів. Проектувальник може згрупувати дрібні компоненти для формування більших, які, в свою чергу, можуть стати основою для створення ще більших.

Структура

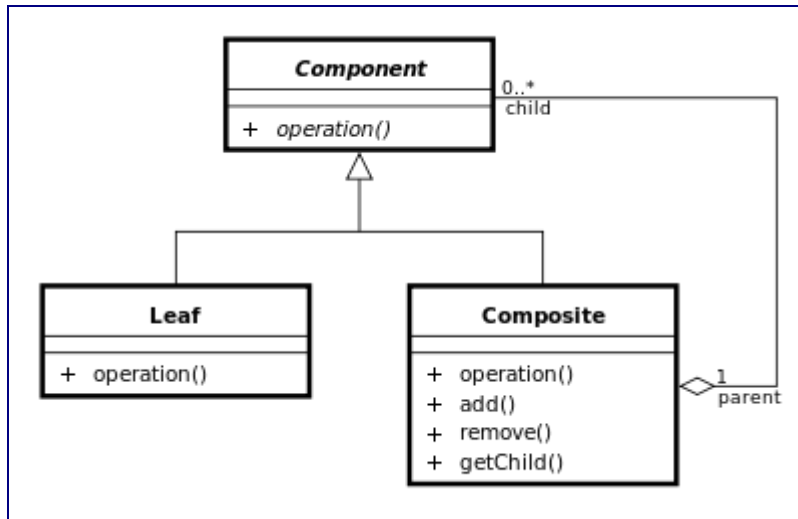


рис. 6.1

Ключем до паттерну компонування є абстрактний клас, який є одночасно і примітивом, і контейнером (Component). У ньому оголошені методи, специфічні для кожного виду об'єкта (такі як Operation) і загальні для всіх складових об'єктів, наприклад операції для доступу і управління нащадками. Підкласи Leaf визначає примітивні об'єкти, які не є контейнерами. У них операція Operation реалізована відповідно до їх специфічних потреб. Оскільки у примітивних об'єктів немає нащадків, то жоден з цих підкласів не реалізує операції, пов'язані з управлінням нащадками (Add, Remove, GetChild). Клас Composite складається з інших примітивніших об'єктів Component. Реалізована в ньому операція Operation викликає однойменну функцію відтворення для кожного нащадка, а операції для роботи з нащадками вже не порожні. Оскільки інтерфейс класу Composite відповідає інтерфейсу Component, то до складу об'єкта Composite можуть входити і інші такі ж об'єкти.

Учасники

- Component (Component)

Оголошує інтерфейс для компонуємих об'єктів; Надає відповідну реалізацію операцій за замовчуванням, загальну для всіх класів; Оголошує єдиний інтерфейс для доступу до нащадків та управління ними; Визначає інтерфейс для доступу до батька компонента в рекурсивній структурі і при необхідності реалізує його (можливість необов'язкова);

- Leaf (Leaf_1, Leaf_2) — лист.

Об'єкт того ж типу що і Composite, але без реалізації контейнерних функцій; Представляє листові вузли композиції і не має нащадків; Визначає поведінку примітивних об'єктів в композиції; Входить до складу контейнерних об'єктів;

- Composite (Composite) — складений об'єкт.

Визначає поведінку контейнерних об'єктів, у яких є нащадки; Зберігає ієрархію компонентів-нащадків; Реалізує пов'язані з управлінням нащадками (контейнерні) операції в інтерфейсі класу Component;

Приклад Реалізації

```
// Composite pattern -- Structural example

using System;
using System.Collections.Generic;

namespace DoFactory.GangOfFour.Composite.Structural
{
    /// <summary>
    /// MainApp startup class for Structural
    /// Composite Design Pattern.
    /// </summary>
    class MainApp
    {
        /// <summary>
        /// Entry point into console application.
        /// </summary>
        static void Main()
        {
            // Create a tree structure
            Composite root = new Composite("root");
            root.Add(new Leaf("Leaf A"));
            root.Add(new Leaf("Leaf B"));

            Composite comp = new Composite("Composite X");
            comp.Add(new Leaf("Leaf XA"));
            comp.Add(new Leaf("Leaf XB"));

            root.Add(comp);
            root.Add(new Leaf("Leaf C"));

            // Add and remove a leaf
            Leaf leaf = new Leaf("Leaf D");
            root.Add(leaf);
            root.Remove(leaf);

            // Recursively display tree
            root.Display(1);

            // Wait for user
            Console.ReadKey();
        }
    }

    /// <summary>
    /// The 'Component' abstract class
    /// </summary>
    abstract class Component
    {
```



```

protected string name;

// Constructor
public Component(string name)
{
    this.name = name;
}

public abstract void Add(Component c);
public abstract void Remove(Component c);
public abstract void Display(int depth);
}

/// <summary>
/// The 'Composite' class
/// </summary>
class Composite : Component
{
    private List<Component> _children = new List<Component>();

    // Constructor
    public Composite(string name)
        : base(name)
    {
    }

    public override void Add(Component component)
    {
        _children.Add(component);
    }

    public override void Remove(Component component)
    {
        _children.Remove(component);
    }

    public override void Display(int depth)
    {
        Console.WriteLine(new String('-', depth) + name);

        // Recursively display child nodes
        foreach (Component component in _children)
        {
            component.Display(depth + 2);
        }
    }
}

/// <summary>
/// The 'Leaf' class
/// </summary>
class Leaf : Component
{
    // Constructor
    public Leaf(string name)
        : base(name)
    {
    }

    public override void Add(Component c)
    {
        Console.WriteLine("Cannot add to a leaf");
    }
}

```

```

    }

    public override void Remove(Component c)
    {
        Console.WriteLine("Cannot remove from a leaf");
    }

    public override void Display(int depth)
    {
        Console.WriteLine(new String('-', depth) + name);
    }
}
}

```

Декоратор (Decorator)

Decorator — структурний шаблон проектування, призначений для динамічного підключення додаткових можливостей до об'єкта. Шаблон Decorator надає гнучку альтернативу методу визначення підкласів з метою розширення функціональності.

Основні характеристики

Завдання

Об'єкт, який передбачається використовувати, виконує основні функції. Проте може виникнути потреба додати до нього деяку додаткову функціональність, яка виконуватиметься до або після основної функціональності об'єкта.

Спосіб вирішення

Декоратор передбачає розширення функціональності об'єкта без визначення підкласів.

Учасники

Клас *ConcreteComponent* — клас, в який за допомогою шаблону Декоратор додається нова функціональність. В деяких випадках базова функціональність надається класами, похідними від класу *ConcreteComponent*. У подібних випадках клас *ConcreteComponent* є вже не конкретним, а абстрактним. Абстрактний клас *Component* визначає інтерфейс для використання всіх цих класів.

Наслідки

Функціональність, що додається, реалізується в невеликих об'єктах. Перевага полягає в можливості динамічно додавати цю функціональність до або після основної функціональності об'єкта *ConcreteComponent*.

Реалізація

Створюється абстрактний клас, що представляє як початковий клас, так і нові функції, що додаються в клас. У класах-декораторах нові функції викликаються в необхідній послідовності — до або після виклику подальшого об'єкта.

Зауваження і коментарі

- Хоча об'єкт-декоратор може додавати свою функціональність до або після функціональності основного об'єкта, ланцюжок створюваних об'єктів завжди повинен закінчуватися об'єктом класу `ConcreteComponent`.
- Базові класи мови Java широко використовують шаблон Декоратор для організації обробки операцій введення-виведення.

Приклад Реалізації

```
public interface InterfaceComponent {
    void doOperation();
}

abstract class Decorator implements InterfaceComponent{
    protected InterfaceComponent component;

    public Decorator (InterfaceComponent c){
        component = c;
    }

    public void doOperation(){
        component.doOperation();
    }
    public void newOperation(){
        System.out.println("Do Nothing");
    }
}

class MainComponent implements InterfaceComponent{
    @Override
    public void doOperation() {
        System.out.print("World!");
    }
}

class DecoratorSpace extends Decorator{

    public DecoratorSpace(InterfaceComponent c){
        super(c);
    }

    @Override
    public void doOperation() {
        System.out.print(" ");
        super.doOperation();
    }
    @Override
    public void newOperation(){
        System.out.println("New space operation");
    }
}
```

```

class DecoratorComma extends Decorator{

    public DecoratorComma(InterfaceComponent c){
        super(c);
    }

    @Override
    public void doOperation() {
        System.out.print(",");
        super.doOperation();
    }

    @Override
    public void newOperation(){
        System.out.println("New comma operation");
    }
}

class DecoratorHello extends Decorator{

    public DecoratorHello(InterfaceComponent c){
        super(c);
    }

    @Override
    public void doOperation() {
        System.out.print("Hello");
        super.doOperation();
    }

    @Override
    public void newOperation(){
        System.out.println("New hello operation");
    }
}

class Main {

    public static void main (String... s){
        Decorator c = new DecoratorHello(new DecoratorComma(new
DecoratorSpace(new MainComponent())));
        c.doOperation(); // Результат выполнения программы "Hello, World!"
        c.newOperation(); // New hello operation
    }
}

```

Лекція 7. Структурні шаблони проектування: Proxy та Flyweight.

- Шаблон Proxy: мотивація, структура, учасники, відносини, шляхи застосування та результат використання.
- Приклад реалізації шаблону Proxy.
- Шаблон Flyweight: мотивація, структура, учасники, відносини, шляхи застосування та результат використання.
- Приклад реалізації шаблону Flyweight. [1, с. 191-203, 141-152]

Шаблон Proxy

Шаблон Proxy (Заступник) — Шаблон проектування. Надає об'єкт, що контролює доступ, перехоплюючи всі виклики до нього.

Проблема

Необхідно управляти доступом до об'єкта так, щоб створювати громіздкі об'єкти «на вимогу».

Вирішення

Створити сурогат громіздкого об'єкта. «Заступник» зберігає посилання, яке дозволяє заступникові звернутися до реального суб'єкта (об'єкт класу «Заступник» може звертатися до об'єкта класу «Суб'єкт», якщо інтерфейси «Реального Суб'єкта» і «Суб'єкта» однакові). Оскільки інтерфейс «Реального Суб'єкта» ідентичний інтерфейсу «Суб'єкта», так, що «Заступника» можна підставити замість «Реального Суб'єкта», контролює доступ до «Реального Суб'єкта», може відповідати за створення або видалення «Реального Суб'єкта». «Суб'єкт» визначає загальний для «Реального Суб'єкта» і «Заступника» інтерфейс, так, що «Заступник» може бути використаний скрізь, де очікується «Реальний Суб'єкт».

«Заступник» може мати і інші обов'язки, а саме:

- видалений «Заступник» може відповідати за кодування запиту і його аргументів і відправку закодованого запиту реальному «Суб'єктові»
- віртуальний «Заступник» може кешувати додаткову інформацію про реального «Суб'єкта», щоб відкласти його створення
- захищаючий «Заступник» може перевіряти, чи має клієнтський об'єкт необхідні для виконання запиту права.

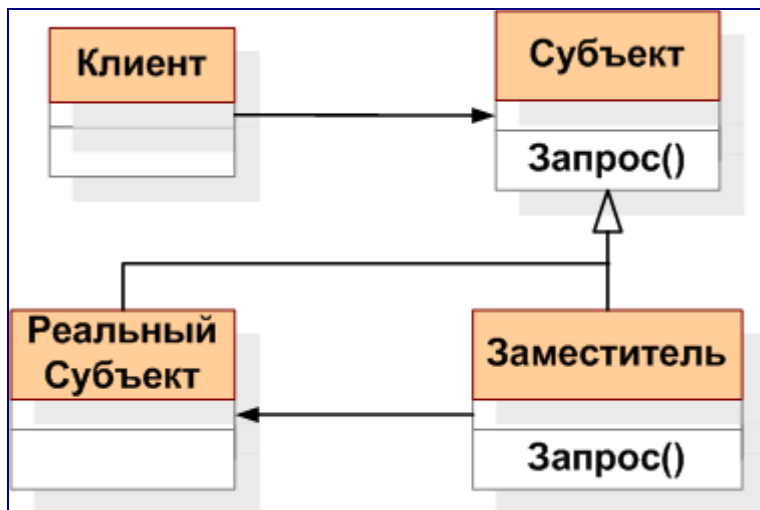


Рис. 7.1

Шаблон проху буває декількох видів, а саме:

- Що протоколює проксі : зберігає в лог всі виклики «Суб'єкта» з їхніми параметрами.
- Віддалений заступник (англ. *remote proxies*) : забезпечує зв'язок з «Суб'єктом», що перебуває в іншому адресному просторі або на віддаленій машині. Так само може відповідати за кодування запиту і його аргументів і відправлення закодованого запиту реальному «Суб'єктові»,
- Віртуальний заступник (англ. *virtual proxies*): забезпечує створення реального «Суб'єкта» тільки тоді, коли він дійсно знадобиться. Так само може кеширувати частину інформації про реальний «Суб'єкті», щоб відкласти його створення,
- Записи-запис^-записові-запису-копіюва-пер-запису: забезпечує копіювання «суб'єкта» при виконанні клієнтом певних дій (окремий випадок «віртуального проксі»).
- Захищаючий заступник (англ. *protection proxies*): може перевіряти, чи має об'єкт, який викликає, необхідні для виконання запиту права.
- Кешируючий проксі: забезпечує тимчасове зберігання результатів розрахунку до віддачі їхнім множинним клієнтам, які можуть розділити ці результати.
- Екрануючий проксі: захищає «Суб'єкт» від небезпечних клієнтів (або навпаки).
- Синхронізуючий проксі: робить синхронізований контроль доступу до «Суб'єкта» в асинхронному багатопотоковому середовищі.

- Smart reference проху: робить додаткові дії, коли на «Суб'єкт» створюється посилання, наприклад, розраховує кількість активних посилань на «Суб'єкт».

Переваги й недоліки від застосування

Переваги:

- Віддалений заступник;
- віртуальний заступник може виконувати оптимізацію;
- захищаючий заступник;
- "розумне" посилання;

Недоліки

- різке збільшення часу відгуку.

Сфера застосування

Шаблон Проху може застосовуватися у випадках роботи з мережевим з'єднанням, з величезним об'єктом у пам'яті (або на диску) або з будь-яким іншим ресурсом, що складно або важко копіювати. Добре відомий приклад застосування — об'єкт, що підраховує число посилань.

Проксі й близькі до нього шаблони[1]

- Адаптер забезпечує інтерфейс, що відрізняється, до об'єкта.
- Проксі забезпечує той же самий інтерфейс.
- Декоратор забезпечує розширений інтерфейс.

Приклад реалізації

```
public class Main {

    public static void main(String[] args) {
        // Create math proxy
        IMath p = new MathProxy();

        // Do the math
        System.out.println("4 + 2 = " + p.add(4, 2));
        System.out.println("4 - 2 = " + p.sub(4, 2));
        System.out.println("4 * 2 = " + p.mul(4, 2));
        System.out.println("4 / 2 = " + p.div(4, 2));
    }

}

/**
 * "Subject"
```

```

*/
public interface IMath {

    public double add(double x, double y);

    public double sub(double x, double y);

    public double mul(double x, double y);

    public double div(double x, double y);
}

/**
 * "Real Subject"
 */
public class Math implements IMath {

    public double add(double x, double y) {
        return x + y;
    }

    public double sub(double x, double y) {
        return x - y;
    }

    public double mul(double x, double y) {
        return x * y;
    }

    public double div(double x, double y) {
        return x / y;
    }

}

/**
 * "Proxy Object"
 */
public class MathProxy implements IMath {

    private Math math;

    public MathProxy() {
        math = new Math();
    }

    public double add(double x, double y) {
        return math.add(x, y);
    }

    public double sub(double x, double y) {
        return math.sub(x, y);
    }

    public double mul(double x, double y) {
        return math.mul(x, y);
    }

    public double div(double x, double y) {
        return math.div(x, y);
    }

}

```


Шаблон Легковаговик (Flyweight) - козак

Призначення

Використовується для ефективної підтримки (в першу чергу для зменшення затрат пам'яті) великої кількості дрібних об'єктів.

Опис

Шаблон Легковаговик (Flyweight) використовує загальнодоступний легкий об'єкт (flyweight, легковаговик), який одночасно може використовуватися у великій кількості контекстів. Стан цього об'єкта поділяється на внутрішній, що містить інформацію, незалежну від контексту, і зовнішній, який залежить або змінюється разом з контекстом легковаговика. Об'єкти клієнтів відповідають за передачу зовнішнього стану легковаговика, коли йому це необхідно.

Переваги

- Зменшує кількість об'єктів, що підлягають обробці.
- Зменшує вимоги до пам'яті.

Застосування

Шаблон Легковаговик можна використовувати коли:

- В програмі використовується велика кількість об'єктів.
- Затрати на збереження високі через велику кількість об'єктів.
- Більшість станів об'єктів можна зробити зовнішніми.
- Велика кількість груп об'єктів може бути замінена відносно малою кількістю загальнодоступних об'єктів, однократно видаливши зовнішній стан.
- Програма не залежить від ідентичності об'єктів. Оскільки об'єкти-легковаговики можуть використовуватися колективно, то тести на ідентичність будуть повертати значення "істина" ("true") для концептуально різних об'єктів.

Діаграма UML

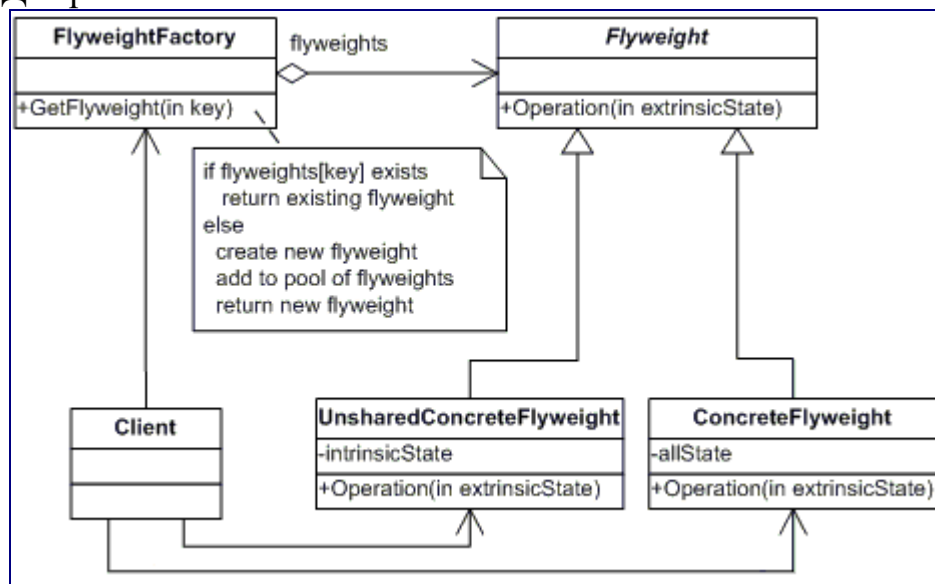


Рис. 7.2

Приклад реалізації

```

import java.util.*;

public enum FontEffect {
    BOLD, ITALIC, SUPERSCRIPT, SUBSCRIPT, STRIKETHROUGH
}

public final class FontData {
    /**
     * A weak hash map will drop unused references to FontData.
     * Values have to be wrapped in WeakReferences,
     * because value objects in weak hash map are held by strong references.
     */
    private static final WeakHashMap<FontData, WeakReference<FontData>>
flyweightData =
        new WeakHashMap<FontData, WeakReference<FontData>>();
    private final int pointSize;
    private final String fontFace;
    private final Color color;
    private final Set<FontEffect> effects;

    private FontData(int pointSize, String fontFace, Color color,
EnumSet<FontEffect> effects) {
        this.pointSize = pointSize;
        this.fontFace = fontFace;
        this.color = color;
        this.effects = Collections.unmodifiableSet(effects);
    }

    public static FontData create(int pointSize, String fontFace, Color color,
FontEffect... effects) {
        EnumSet<FontEffect> effectsSet = EnumSet.noneOf(FontEffect.class);
        effectsSet.addAll(Arrays.asList(effects));
        // We are unconcerned with object creation cost, we are reducing overall
memory consumption
        FontData data = new FontData(pointSize, fontFace, color, effectsSet);
        if (!flyweightData.containsKey(data)) {

```

```

        flyweightData.put(data, new WeakReference<FontData> (data));
    }
    // return the single immutable copy with the given values
    return flyweightData.get(data).get();
}

@Override
public boolean equals(Object obj) {
    if (obj instanceof FontData) {
        if (obj == this) {
            return true;
        }
        FontData other = (FontData) obj;
        return other.pointSize == pointSize && other.fontFace.equals(fontFace)
            && other.color.equals(color) && other.effects.equals(effects);
    }
    return false;
}

@Override
public int hashCode() {
    return (pointSize * 37 + effects.hashCode() * 13) * fontFace.hashCode();
}

// Getters for the font data, but no setters. FontData is immutable.
}

```

Лекція 8. Структурні шаблони проектування: Adapter, Bridge та Facade.

- Шаблон Adapter: мотивація, структура, учасники, відносини, шляхи застосування та результат використання.
- Приклад реалізації шаблону Adapter.
- Шаблон Bridge: мотивація, структура, учасники, відносини, шляхи застосування та результат використання.
- Приклад реалізації шаблону Bridge.
- Шаблон Facade: мотивація, структура, учасники, відносини, шляхи застосування та результат використання.
- Приклад реалізації шаблону Facade. [1, с. 152-162, 183-191]

Шаблон Adapter

Адаптер, Adapter — структурний шаблон проектування, призначений для організації використання функцій об'єкта, недоступного для модифікації, через спеціально створений інтерфейс.

Призначення

Адаптує інтерфейс одного класу в інший, очікуваний клієнтом. Адаптер забезпечує роботу класів з несумісними інтерфейсами, та найчастіше застосовується тоді, коли система підтримує необхідні дані і поведінку, але має невідповідний інтерфейс.

Застосування

Адаптер передбачає створення класу-оболонки з необхідним інтерфейсом.

Структура

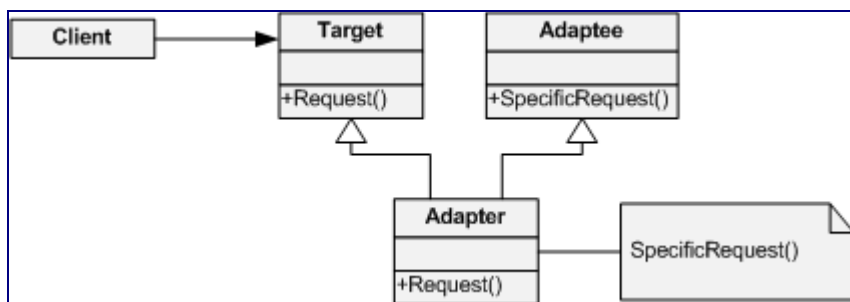


Рис. 8.1

UML діаграма, що ілюструє структуру шаблону проектування Адаптер (з використанням множинного наслідування)

Учасники

Клас Adapter приводить інтерфейс класу Adaptee у відповідність з інтерфейсом класу Target (спадкоємцем якого є Adapter). Це дозволяє об'єктові Client використовувати об'єкт Adaptee так, немов він є екземпляром класу Target.

Наслідки

Шаблон Адаптер дозволяє включати вже існуючі об'єкти в нові об'єктні структури, незалежно від відмінностей в їхніх інтерфейсах.

Приклад

```
// Target
public interface Chief
{
    public Object makeBreakfast();
    public Object makeLunch();
    public Object makeDinner();
}

// Adaptee
public class Plumber
{
    public Object getScrewNut()
    { ... }
    public Object getPipe()
    { ... }
    public Object getGasket()
    { ... }
}

// Adapter
public class ChiefAdapter extends Plumber implements Chief
{
    public Object makeBreakfast()
    {
        return getGasket();
    }
    public Object makeLunch()
    {
        return getPipe();
    }
    public Object makeDinner()
    {
        return getScrewNut();
    }
}

// Client
public class Client
{
    public static void eat(Object dish)
    { ... }

    public static void main(String[] args)
    {
        Chief ch = new ChiefAdapter();
    }
}
```

```

Object dish = ch.makeBreakfast();
eat(dish);
dish = ch.makeLunch();
eat(dish);
dish = ch.makeDinner();
eat(dish);
callAmbulance();
}
}

```

Шаблон Bridge

Міст (англ. *Bridge*) - шаблон проектування, відноситься до класу структурних шаблонів.

Призначення

Відокремити абстракцію від її реалізації таким чином, щоб перше та друге можна було змінювати незалежно одне від одного.

Мотивація

Якщо для деякої абстракції можливо кілька реалізацій, зазвичай застосовують спадкування. Абстрактний клас визначає інтерфейс абстракції, а його конкретні підкласи по-різному реалізують його. Але такий підхід не завжди є достатньо гнучким. Спадкування жорстко прив'язує реалізацію до абстракції, що перешкоджає незалежній модифікації, розширенню та повторному використанню абстракції та її реалізації.

Застосовність

Слід використовувати шаблон *Міст* у випадках, коли:

- треба запобігти постійній прив'язці абстракції до реалізації. Так, наприклад, буває коли реалізацію необхідно обрати під час виконання програми;
- як абстракції, так і реалізації повинні розширюватись новими підкласами. У цьому разі шаблон *Міст* дозволяє комбінувати різні абстракції та реалізації та змінювати їх незалежно одне від одного;
- зміни у реалізації не повинні впливати на клієнтів, тобто клієнтський код не повинен перекомпілюватись;
- треба повністю сховати від клієнтів реалізацію абстракції;
- треба розподілити одну реалізацію поміж кількох об'єктів (можливо застосовуючи підрахунок посилань), і при цьому приховати це від клієнту.

Структура

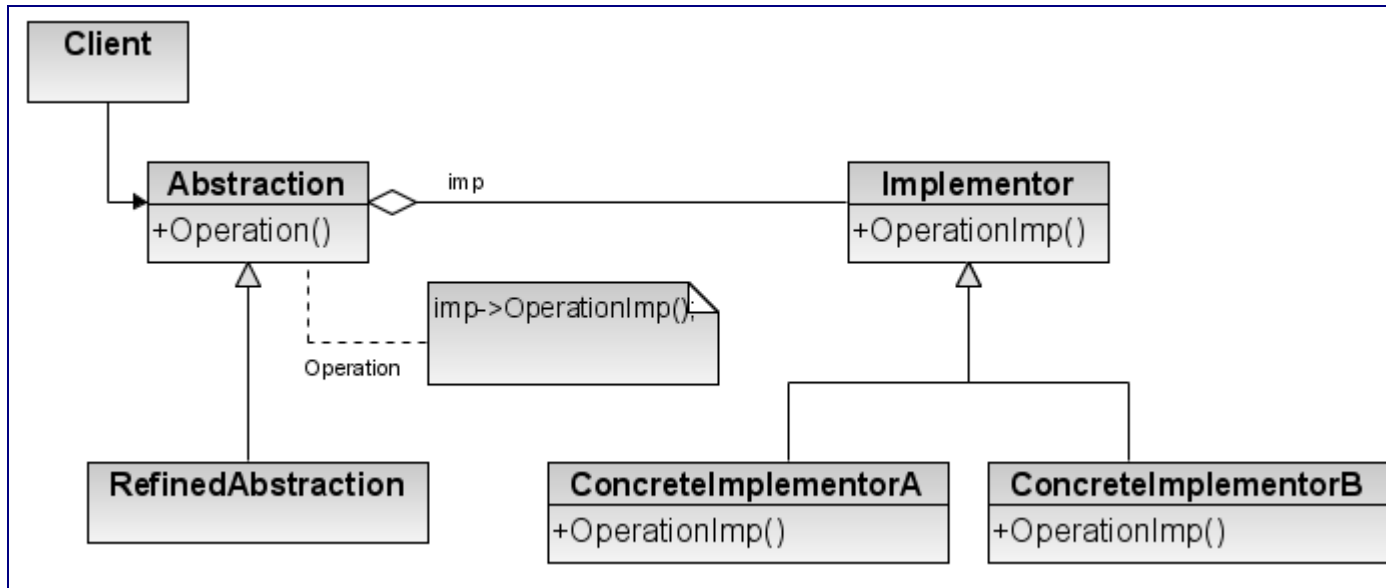


Рис. 8.2

UML діаграма, що описує структуру шаблону проектування *Bridge*

- **Abstraction** – абстракція:
 - визначає інтерфейс абстракції;
 - зберігає посилання на об'єкт типу *Implementor*;
- **RefinedAbstraction** – уточнена абстракція:
 - розширює інтерфейс, означений абстракцією *Abstraction*;
- **Implementor** – реалізатор:
 - визначає інтерфейс для класів реалізації. Він не зобов'язаний точно відповідати інтерфейсу класу *Abstraction*. Насправді обидва інтерфейси можуть бути зовсім різними. Зазвичай, інтерфейс класу *Implementor* надає тільки примітивні операції, а клас *Abstraction* визначає операції більш високого рівня, що базуються на цих примітивах;
- **ConcreteImplementor** – конкретний реалізатор:
 - містить конкретну реалізацію інтерфейсу класу *Implementor*.

Відносини

Об'єкт *Abstraction* містить у собі *Implementor* і перенаправляє йому запити клієнта

```
package com.designpatterns.bridge;
```

```
/* Файл Drawer.java
 *
```

```

    * */

public interface Drawer {

    public void drawCircle(int x, int y, int radius);

}

package com.designpatterns.bridge;

/* Файл SmallCircleDrawer.java
 *
 * */

public class SmallCircleDrawer implements Drawer{

    public static final double radiusMultiplier = 0.25;

    @Override
    public void drawCircle(int x, int y, int radius) {
        System.out.println("Small circle center = " + x + "," + y + "
radius = " + radius*radiusMultiplier);
    }

}

package com.designpatterns.bridge;

/* Файл LargeCircleDrawer.java
 *
 * */

public class LargeCircleDrawer implements Drawer{

    public static final int radiusMultiplier = 10;

    @Override
    public void drawCircle(int x, int y, int radius) {
        System.out.println("Large circle center = " + x + "," + y + "
radius = " + radius*radiusMultiplier);
    }

}

package com.designpatterns.bridge;

/* Файл Shape.java
 *
 * */

public abstract class Shape {

    protected Drawer drawer;

    protected Shape(Drawer drawer){
        this.drawer = drawer;
    }

    public abstract void draw();

    public abstract void enlargeRadius(int multiplier);

```



```

}

package com.designpatterns.bridge;

/* Файл Circle.java
 *
 * */

public class Circle extends Shape{

    private int x;

    private int y;

    private int radius;

    public Circle(int x, int y, int radius, Drawer drawer) {
        super(drawer);
        setX(x);
        setY(y);
        setRadius(radius);
    }

    @Override
    public void draw() {
        drawer.drawCircle(x, y, radius);
    }

    @Override
    public void enlargeRadius(int multiplier) {
        radius *= multiplier;
    }

    public int getX() {
        return x;
    }

    public int getY() {
        return y;
    }

    public int getRadius() {
        return radius;
    }

    public void setX(int x) {
        this.x = x;
    }

    public void setY(int y) {
        this.y = y;
    }

    public void setRadius(int radius) {
        this.radius = radius;
    }

}

package com.designpatterns.bridge;

/* Класс, показывающий работу шаблона проектирования "Мост".

```

```

* Файл Application.java
*
* */

public class Application {

    public static void main (String [] args){
        Shape [] shapes = {
            new Circle(5,10,10, new LargeCircleDrawer()),
            new Circle(20,30,100, new SmallCircleDrawer())};

        for (Shape next : shapes){
            next.draw();
        }
    }
}

// Output
Large circle center = 5,10 radius = 100
Small circle center = 20,30 radius = 25.0

```

Шаблон Facade

Фасад — шаблон проектування, призначений для об'єднання групи підсистем під один уніфікований інтерфейс, надаючи доступ до них через одну точку входу. Це дозволяє спростити роботу з підсистемами.

Фасад відноситься до структурних шаблонів проектування.

Складові шаблону

Класи, з яких складається шаблон можна розділити на 3 частини:

1. фасад;
2. підсистеми;
3. клієнти.

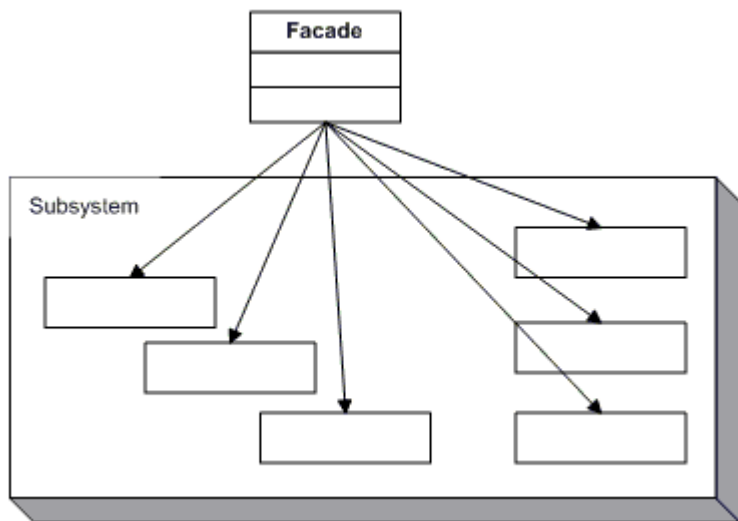


Рис. 8.3

Ролі складових

Фасад

- Визначає певним підсистемам інтерфейс, отже знає кому адресувати запити;
- делегує запити клієнтів потрібним об'єктам підсистеми;
- створює нові методи, котрі об'єднують виклики об'єктів системи і\або додають свою логіку;
- приховує підсистеми;
- зменшує кількість параметрів методів, шляхом попередньої підстановки визначених значень.

Підсистема

- реалізує функціонал, закритий та не видимий для зовнішніх компонентів
- виконує роботу, запитану клієнтом через фасад.
- не зберігає посилання на фасад - це означає що одна підсистема може мати довільну кількість фасадів.

Клієнт

- здійснює запити фасаду;
- не знає про існування підсистем.

Випадки використання

Фасад використовується у випадках, коли потрібно:

- спростити доступ до складної системи;
- створити рівні доступу до системи;
- додати стійкість до змін підсистем;
- зменшити кількість сильних зв'язків між клієнтом та підсистемою, але залишити доступ до повної функціональності.

```
/* Complex parts */
function SubSystem1() {
    this.method1 = function() {
        alert("вызван SubSystem1.method1");
    };
}
function SubSystem2() {
    this.method2 = function() {
        alert("вызван SubSystem2.method2");
    };
    this.methodB = function() {
        alert("вызван SubSystem2.methodB");
    };
};
```

```
}

/* Facade */
function Facade() {
    var s1 = new SubSystem1();
    var s2 = new SubSystem2();

    this.m1 = function() {
        alert("вызван Facade.m1");
        s1.method1();
        s2.method2();
    };

    this.m2 = function() {
        alert("вызван Facade.m2");
        s2.methodB();
    };
}

/* Client */
function Test() {
    var facade = new Facade();
    facade.m1();
    facade.m2();
}

var obj = new Test();
```