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. 61
Шаблон Iterator 61
Шаблон Mediator 63
Лекція 10. Шаблони поведінки: Observer та Strategy. 67
Шаблон Observer 67
Шаблон Strategy (Стратегія) 70
Лекція 11. Шаблони поведінки: Command та Visitor. 73
Шаблон Command 73
Лекція 12. Шаблони поведінки: State та Memento. 81
Шаблон State 81
Шаблон Memento 83
Лекція 13. Шаблони поведінки: Interpreter та Chain of Responsibility. 87
Шаблон Interpreter 87
Шаблон Chain of Responsibility 90
Лекція 14. Породжувальні шаблони Prototype, Factory Method та Abstract Factory. 93

Шаблон Prototype 93
Шаблон Factory Method 95
Шаблон Abstract Factory 97
Лекція 15. Шаблони Singleton та Builder. 100
Шаблон Singleton 101
Шаблон Builder 102
Розділ 4. Проектування інтерфейсу користувача. 106
Лекція 16. Графічний інтерфейс користувача. 106
Призначення та типи інтерфейсів користувача 106
Принципи розробки графічного інтерфейсу користувача 109
Приклад графічного інтерфейса користувача 116
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! Зак**
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! Заклад**
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 определен**

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 н**

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 каскадної 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

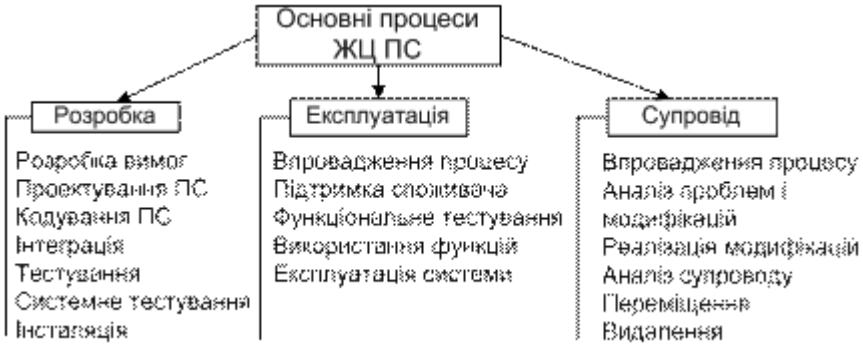
Стандарт ISO/IEC 12207:1995 "Information Technology - Software Life Cycle Processes" є основним нормативним документом, який регламентує склад процесів життєвого циклу ПЗ. Він визначає структуру життєвого циклу, що містить процеси, дії і завдання, які повинні бути виконані під час створення ПЗ.

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.

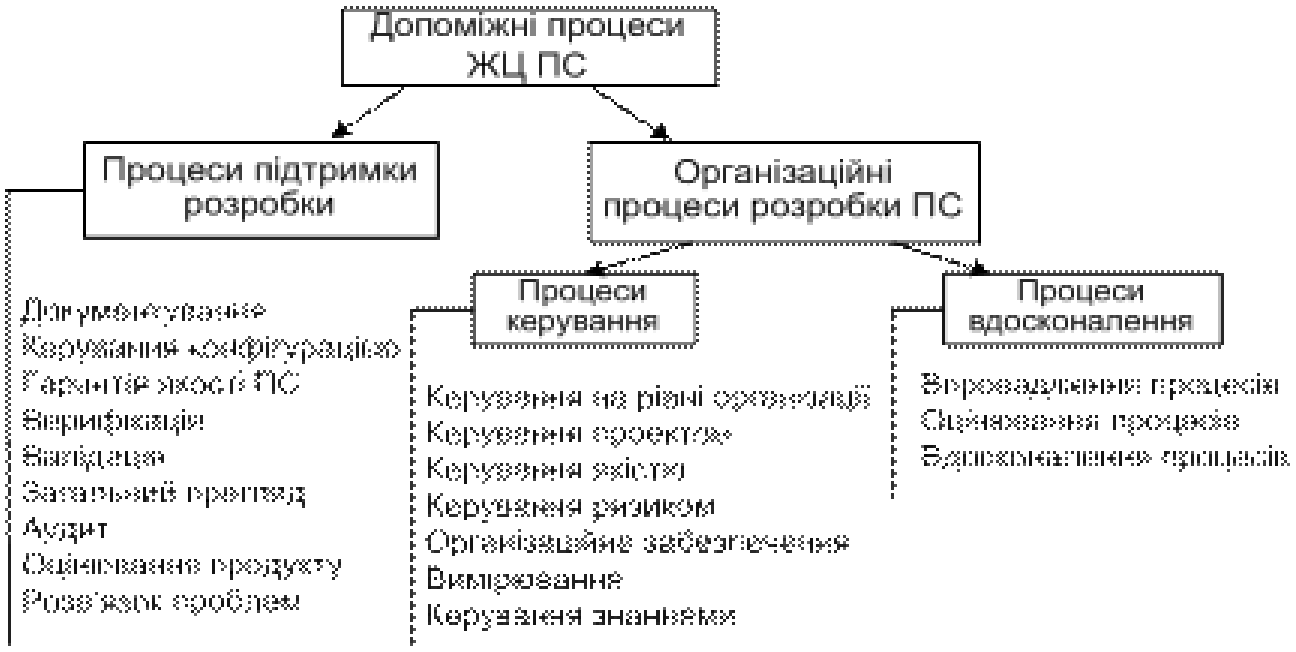
Відповідно до базовим міжнародним стандартом ISO/IEC 12207 всі *процеси ЖЦ ПЗ* діляться на три групи:

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)	• ініціювання • Підготовка заявочних пропозицій • Підготовка договору • Контроль діяльності постачальника • Приймання ІС	• Рішення про початок робіт по впровадженню ІС • Результати обстеження діяльності замовника • Результати аналізу ринку ІС/ тендеру • План поставки/ розробки • Комплексний тест ІС	• Техніко-економічне обгрунтування впровадження ІС • Технічне завдання на ІС • Договір на поставку/ розробку • Акти приймання етапів роботи • Акт приймально-здавальних випробувань
Поставка (розробник ІС)	• ініціювання • Відповідь на заявочні пропозиції • Підготовка договору • Планування виконання • Поставка ІС	• Рішення керівництва про участь в розробці • Результати тендеру • Технічне завдання на ІС • План управління проектом • Розроблена ІС і документація	• 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)	• 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	• 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,	• 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 комплекту 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 омана 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, Водопадна 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 інкрементального 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 проекті" **[4]**.

Підхід IID має і свої негативні сторони, які, по суті, - зворотна сторона переваг. По-перше, цілісне розуміння можливостей і обмежень проекту 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**.

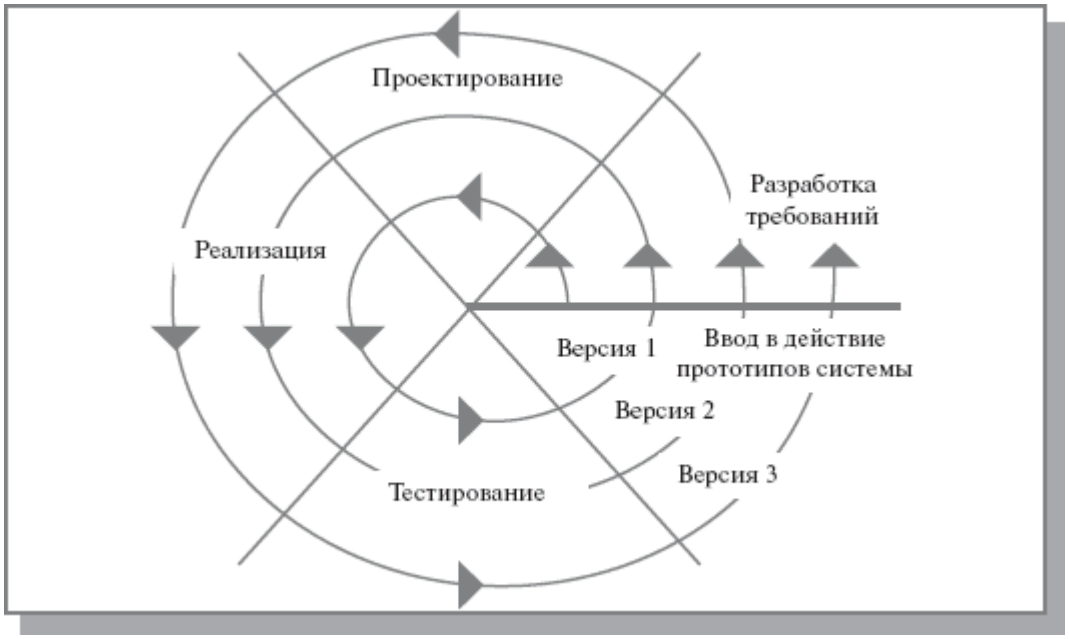


Рис. 2.3. Спіральна модель ЖЦ ІС

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

На кожній ітерації оцінюються:

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

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 користувальницького 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 сьогоднішній 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 обгрунтований 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 IT 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 реалізовуються 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-клі́пс*», 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 платформо-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, використовного на настільних комп'ютерах, на інші пристрої.
- DLTK (DLTK) — інтегроване середовище розробника 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 проектів (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 проектами.

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 розподіленим.

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 проекту 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 проекту(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 проектами, 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 зміненні 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 компіляторові, а потім і лінковщику для створення кінцевого об'єкта. Однак, при спробі скомпілювати або злінкувати 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 лінковщиков. 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 лінковкці.

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 лінковки.

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 проходженння 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 декомпозується 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 постумовах, 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 зрештою реалізоване 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 Крістофер Александра та розробили шаблони відповідно до розробки програмного забезпечення для розробки графічних оболонок мовою Smalltalk.
У 1988 році Ерік Ґамма (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 Ґамма 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 шаблона проектування. Також команда авторів цієї книги відома суспільству під назвою Банда чотирьох (англ. *Gang of Four - GoF*). Саме ця книга послужила привідом 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) - посредник
- 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

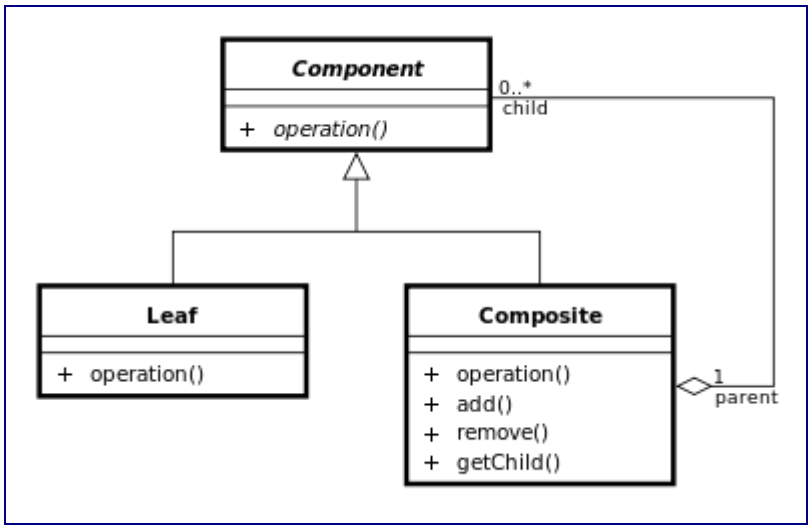


рис. 6.1

Ключем до паттерну компонувальник 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 управління 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 компонуемих 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 рекурсивної 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.

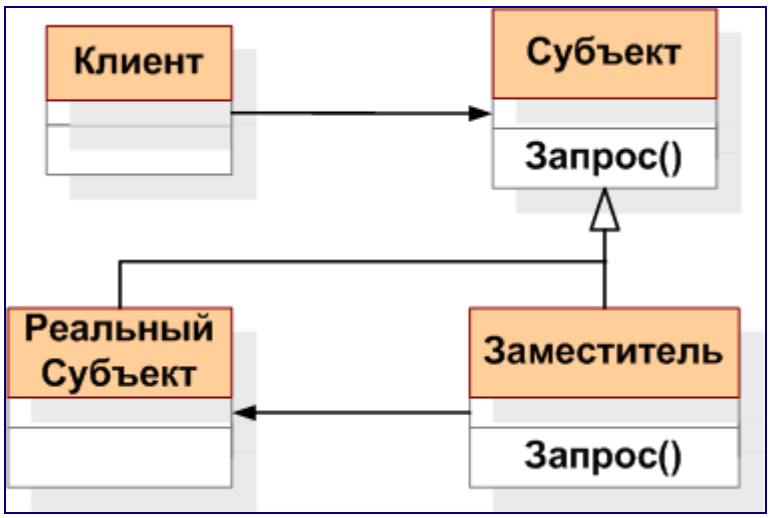


Рис. 7.1

Шаблон proxy 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 кешірувати 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 proxy: 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 Proxy 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

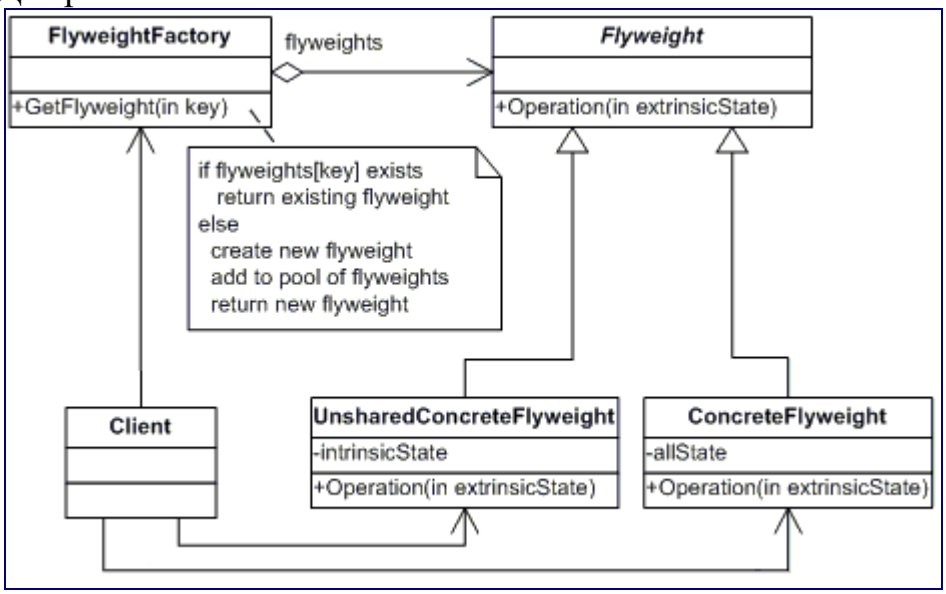


Рис. 7.2

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

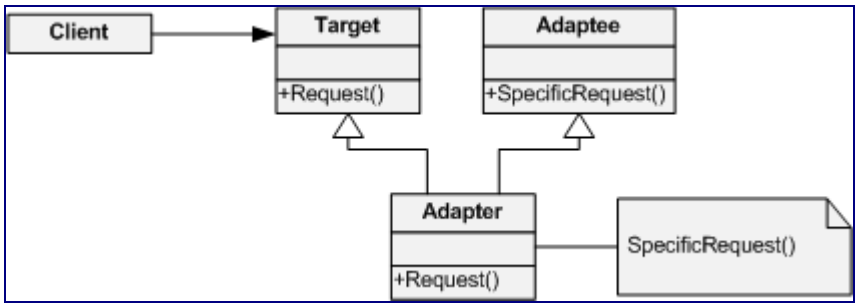


Рис. 8.1

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 *Micm* 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 *Micm* 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

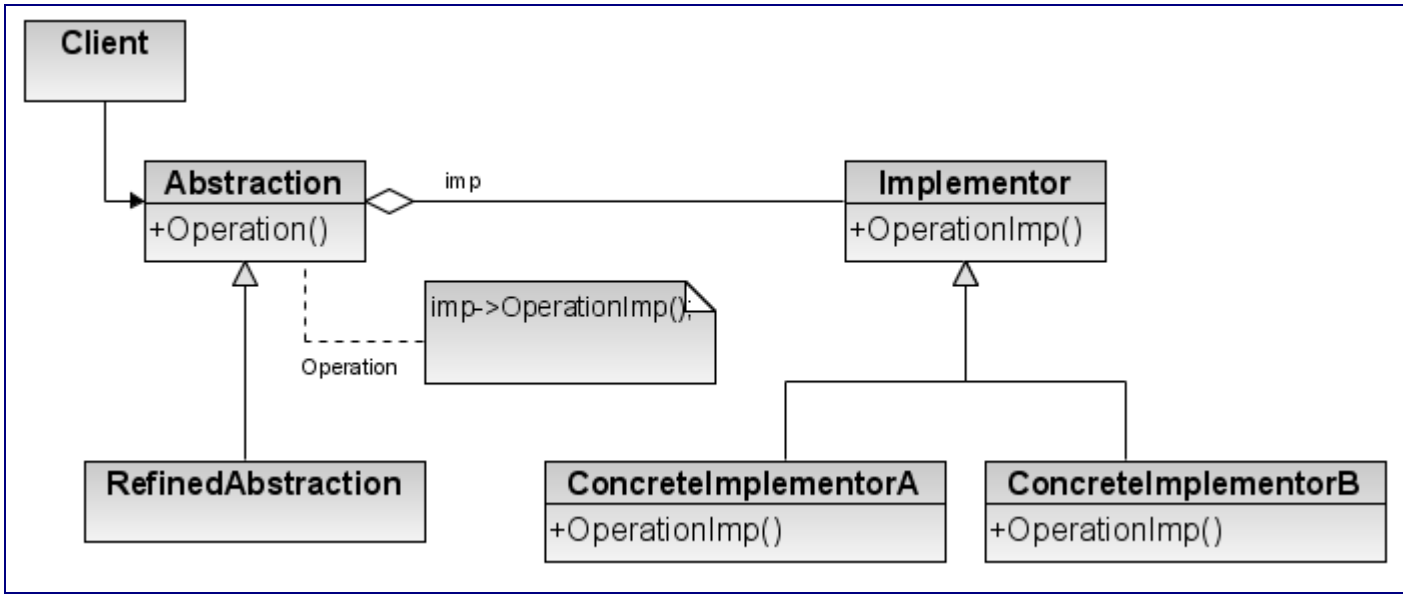


Рис. 8.2
UML діаграма, що описує структуру шаблону проектування *Micm*

- 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* визначує 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

Фаса́д — 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.

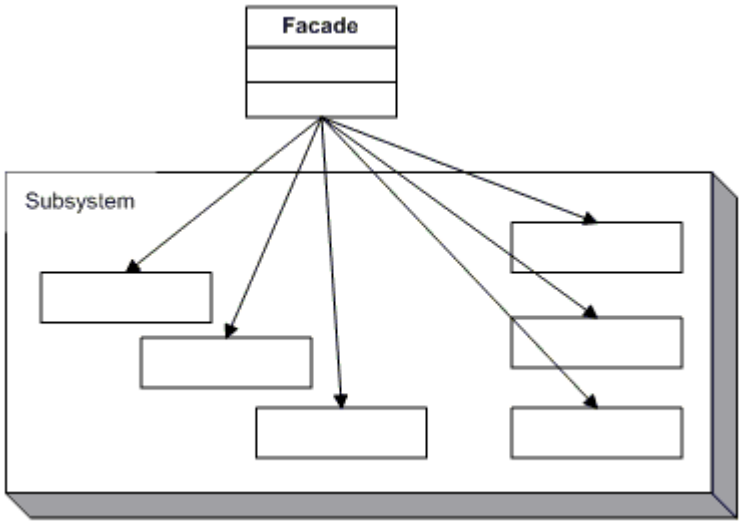


Рис. 8.3

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 використовує+ться 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();
```

Лекція 9. Шаблони поведінки: Iterator та Mediator.

- Призначення шаблонів поведінки.
- Шаблон Iterator: мотивація, структура, учасники, відносини, шляхи застосування та результат використання.
- Приклад реалізації шаблону Iterator.
- Шаблон Mediator: мотивація, структура, учасники, відносини, шляхи застосування та результат використання.
- Приклад реалізації шаблону Mediator. [1, с. 216-217, 249-263]

Шаблон Iterator

Ітератор (англ. *Iterator*) — шаблон проектування, належить до класу шаблонів поведінки.

Призначення
Надає спосіб послідовного доступу до всіх елементів складеного об'єкта, не розкриваючи його внутрішнього улаштування.

Мотивація
Складений об'єкт, скажімо список, повинен надавати спосіб доступу до своїх елементів, не розкриваючи їхню внутрішню структуру. Більш того, іноді треба обходити список по-різному, у залежності від задачі, що вирішується. При цьому немає ніякого бажання засмічувати інтерфейс класу *Список* усілякими операціями для усіх потрібних варіантів обходу, навіть якщо їх усі можна передбачити заздалегідь. Крім того, іноді треба, щоб в один момент часу існувало декілька активних операцій обходу списку.

Все це призводить до необхідності реалізації шаблону *Ітератор*. Його основна ідея у тому, щоб за доступ до елементів та обхід списку відповідав не сам список, а окремий об'єкт-ітератор. У класі *Ітератор* означений інтерфейс для доступу до елементів списку. Об'єкт цього класу прослідковує поточний елемент, тобто він володіє інформацією, які з елементів вже відвідувались.

Застосовність
Можна використовувати шаблон *Ітератор* у випадках:

- для доступу до змісту агрегованих об'єктів не розкриваючи їхнє внутрішнє улаштування;

- для підтримки декількох активних обходів одного й того ж агрегованого об'єкта;
- для подання уніфікованого інтерфейсу з метою обходу різноманітних агрегованих структур (тобто для підтримки поліморфної ітерації).

Структура

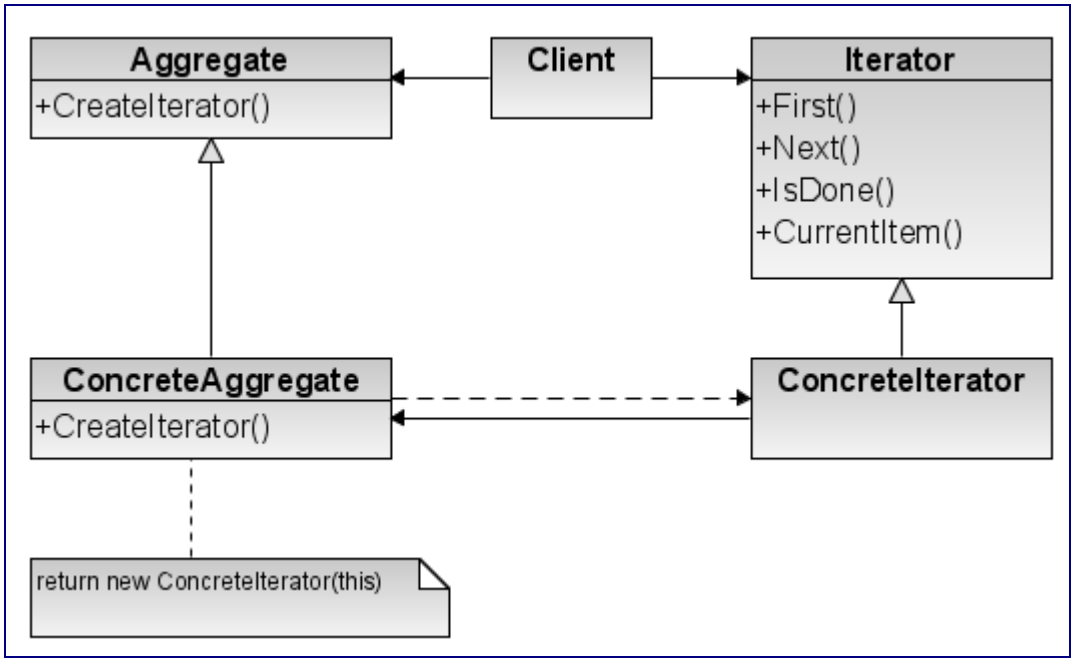


Рис. 9.1
UML діаграма, що описує структуру шаблону проектування Ітератор

- Iterator
  - визначає інтерфейс для доступу та обходу елементів
- ConcreteIterator
  - реалізує інтерфейс класу *Iterator*;
  - слідкує за поточною позицією під час обходу агрегату;
- Aggregate
  - визначає інтерфейс для створення об'єкта-ітератора;
- ConcreteAggregate
  - реалізує інтерфейс створення *ітератора* та повертає екземпляр відповідного класу *ConcreteIterator*

Відносини
ConcreteIterator відслідковує поточний об'єкт у агрегаті та може вирахувати наступний.

Шаблон Mediator

Посередник (англ. *Mediator*) - шаблон проектування, відноситься до класу шаблонів поведінки.

Призначення
Визначає об'єкт, що інкапсулює спосіб взаємодії множини об'єктів. *Посередник* забезпечує слабку зв'язаність системи, звільняючи об'єкти від необхідності явно посилатися один на одного, і дозволяючи тим самим незалежно змінювати взаємодії між ними.

Мотивація

Застосовність
Слід використовувати шаблон *Посередник* у випадках, коли:

- існують об'єкти, зв'язки між котрими досить складні та чітко задані. Отримані при цьому залежності не структуровані та важкі для розуміння;
- не можна повторно використовувати об'єкт, оскільки він обмінюється інформацією з багатьма іншими об'єктами;
- поведінка, розподілена між кількома класами, повинна піддаватися налагодженню без створювання множини підкласів.

Структура

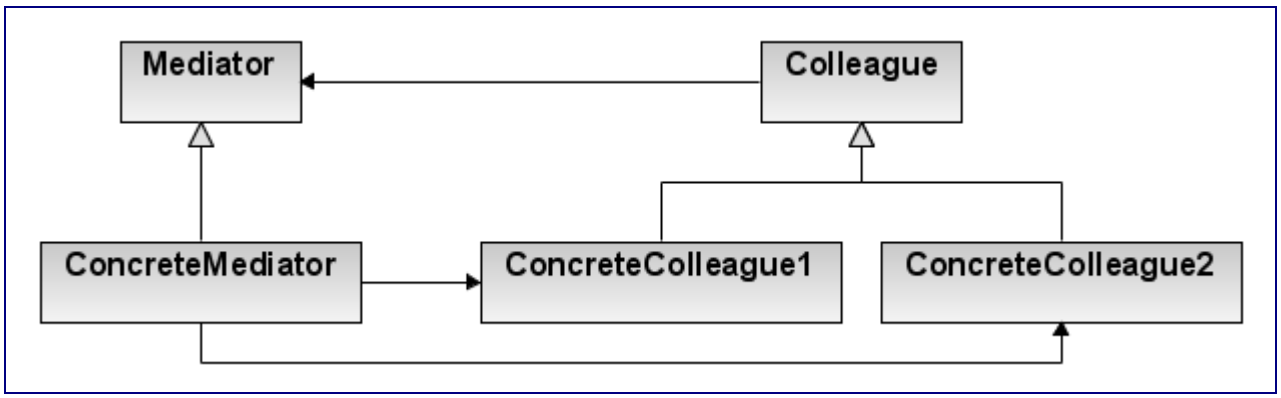


Рис. 9.2
UML діаграма, що описує структуру шаблону проектування *Посередник*

- Mediator – посередник:

- визначає інтерфейс для обміну інформацією з об'єктами *Colleague*;

- ConcreteMediator – конкретний посередник:
  - реалізує кооперативну поведінку, координуючи дії об'єктів *Colleague*;
  - володіє інформацією про колег, та підраховує їх;
- Класи Colleague – колеги:
  - кожному класу *Colleague* відомо про свій об'єкт *Mediator*;
  - усі колеги обмінюються інформацією виключно через посередника, інакше за його відсутності їм довелося б спілкуватися між собою напряму.

Відносини

Колеги посилають запити посередникові та отримують запити від нього. *Посередник* реалізує кооперативну поведінку шляхом переадресації кожного запиту відповідному колезі (або декільком з них).

Приклади

```
      package example.pattern.mediator;
import java.util.Random;

// Mediator
public class TankCommander {
        private TankDriver driver;
        private TankGunner gunner;
        private TankLoader loader;

        public void setDriver(TankDriver driver) {
                this.driver = driver;
        }

        public void setGunner(TankGunner gunner) {
                this.gunner = gunner;
        }

        public void setLoader(TankLoader loader) {
                this.loader = loader;
        }

        public void targetDetected(String target) {
                System.out.println("Commander: new target detected!");
                gunner.setWeapon(
                                loader.prepareWeapon(target));
                driver.halt();
                gunner.fire(target);
        }

        public void noTarget() {
                System.out.println("Commander: no target.");
                gunner.stopFire();
                driver.move();
        }
```

```
}

// Colleague
public class TankDriver {
        private TankCommander commander;
        private int fuel = 4;

        public TankDriver(TankCommander commander) {
                this.commander = commander;
                this.commander.setDriver(this);
        }

        public void move() {
                if (fuel <= 0 ) {
                        System.out.println("Driver: no fuel!");
                        return;
                }
                System.out.println("Driver: tank is moving!");
                fuel--;
                // looking for a new target
                if ((new Random()).nextInt(2) == 0) {
                        commander.targetDetected("armored");
                } else {
                        commander.targetDetected("infantry");
                }
        }

        public void halt() {
                System.out.println("Driver: tank has halted!");
        }
}

// Colleague
public class TankGunner {
        private TankCommander commander;
        private String weapon = "MachineGun";

        public TankGunner (TankCommander commander) {
                this.commander = commander;
                this.commander.setGunner(this);
        }

        public void fire(String target) {
                System.out.println("Gunner ["+weapon+"]: " + target + " is under
fire!");
                commander.noTarget(); // target is destroyed
        }

        public void stopFire() {
                System.out.println("Gunner ["+weapon+"]: Fire is stoped.");
        }

        public void setWeapon(String weapon) {
                this.weapon = weapon;
        }
}

// Colleague
public class TankLoader {
        private TankCommander commander;

        public TankLoader (TankCommander commander) {
```

```
                this.commander = commander;
                this.commander.setLoader(this);
        }

        public String prepareWeapon(String target) {
                String weapon = "MachineGun";
                if (target.equals("armored")) {
                        weapon = "Cannon";
                }
                System.out.println("Loader: " + weapon + " is ready!");
                return weapon;
        }
}
```

Лекція 10. Шаблони поведінки: Observer та Strategy.

- Шаблон Observer: мотивація, структура, учасники, відносини, шляхи застосування та результат використання.
- Приклад реалізації шаблону Observer.
- Шаблон Strategy: мотивація, структура, учасники, відносини, шляхи застосування та результат використання.
- Приклад реалізації шаблону Strategy. [1, с. 263-272, 280-291]

Шаблон Observer

Спостерігач, Observer - поведінковий шаблон проектування. Також відомий як «підлеглі» (Dependents), «видавець-передплатник» (Publisher-Subscriber).

Призначення

Визначає залежність типу «один до багатьох» між об'єктами таким чином, що при зміні стану одного об'єкта всі залежних від нього сповіщаються про цю подію.

Структура

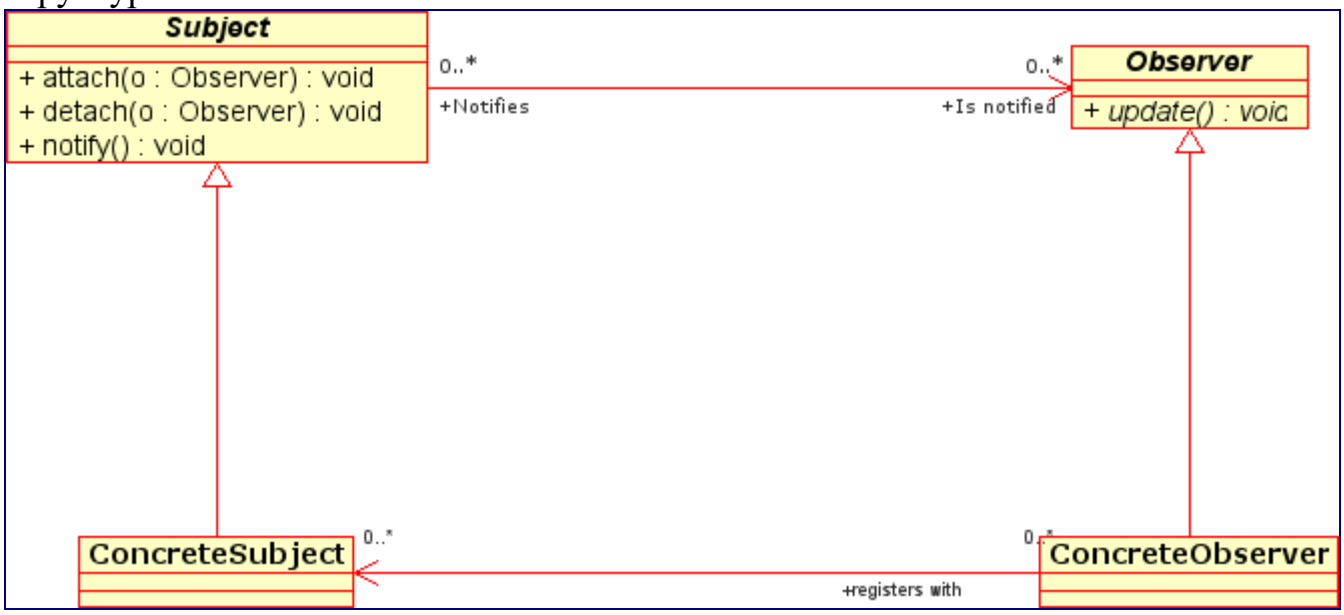


Рис. 10.1

При реалізації шаблону «спостерігач» зазвичай використовуються такі класи:

- Observable — інтерфейс, що визначає методи для додавання, видалення та оповіщення спостерігачів.
- Observer — інтерфейс, за допомогою якого спостережуваний об'єкт оповіщає спостерігачів.
- ConcreteObservable — конкретний клас, який реалізує інтерфейс Observable.

- ConcreteObserver — конкретний клас, який реалізує інтерфейс Observer.

При зміні спостережуваного об’єкту, оповіщення спостерігачів може бути реалізоване за такими сценаріями:

- Спостережуваний об’єкт надсилає, кожному із зареєстрованих спостерігачів, всю потенційно релевантну інформацію (примусове розповсюдження).
- Спостережуваний об’єкт надсилає, кожному із зареєстрованих спостерігачів, лише повідомлення про те що інформація була змінена, а кожен із спостерігачів, за необхідності, самостійно здійснює запит необхідної інформації у спостережуваного об’єкта (розповсюдження за запитом).

Область застосування

Шаблон «спостерігач» застосовується в тих випадках, коли система володіє такими властивостями:

- існує, як мінімум, один об'єкт, що розсилає повідомлення
- є не менше одного одержувача повідомлень, причому їхня кількість і склад можуть змінюватися під час роботи програми.

Цей шаблон часто застосовують в ситуаціях, в яких відправника повідомлень не цікавить, що роблять одержувачі з наданою їм інформацією.

```
      package example.pattern.observer;
import java.util.ArrayList;
import java.util.List;

public interface Subject {
        void attach(Observer o);
        void detach(Observer o);
        void notifyObserver();
}

public interface Observer {
        void update();
}

public class ConcreteSubject implements Subject {
        private List<Observer> observers = new ArrayList<Observer>();
        private int value;

        public void setValue(int value) {
                this.value = value;
                notifyObserver();
        }

        public int getValue() {
                return value;
        }

        @Override
```

```
        public void attach(Observer o) {
                observers.add(o);
        }

        @Override
        public void detach(Observer o) {
                observers.remove(o);
        }

        @Override
        public void notifyObserver() {
                for (Observer o : observers) {
                        o.update();
                }
        }
}

public class ConcreteObserver1 implements Observer {

        private ConcreteSubject subject;

        public ConcreteObserver1(ConcreteSubject subject) {
                this.subject = subject;
        }

        @Override
        public void update() {
                System.out.println("Observer1: " + subject.getValue());
        }

}

public class ConcreteObserver2 implements Observer {

        private ConcreteSubject subject;

        public ConcreteObserver2(ConcreteSubject subject) {
                this.subject = subject;
        }

        @Override
        public void update() {
                String out = "";
                for (int i = 0; i < subject.getValue(); i++) {
                        out += "*";
                }
                System.out.println("Observer2: " + out);
        }
}

public class Program {
        public static void main(String[] args) {
                ConcreteSubject subject = new ConcreteSubject();
                ConcreteObserver1 observer1 = new ConcreteObserver1(subject);
                ConcreteObserver2 observer2 = new ConcreteObserver2(subject);
                subject.attach(observer1);
                subject.attach(observer2);
                subject.setValue(3);
                subject.setValue(8);
        }
}
```

Шаблон Strategy (Стратегія)

Цей патерн відомий ще під іншою назвою - "Policy". Його суть полягає у тому, щоб створити декілька схем поведінки для одного об'єкту та винести в окремий клас.

Призначення патерну Strategy

Існують системи, поведінка яких визначається відповідно до певного роду алгоритмів. Всі вони подібні між собою: призначені для вирішення спільних задач, мають однаковий інтерфейс для користування, але відрізняються тільки "поведінкою", тобто реалізацією. Користувач, налаштувавши програму на потрібний алгоритм - отримує потрібний результат.

Приклад. Є програма (інтерфейс) через яку обраховується ціна на товар для покупців у яких є знижка та ціна за сезонною знижкою - обираємо необхідний алгоритм. Об'єктно-орієнтованиий дизайн такої програми будується на ідеї використання поліморфізму. Результатом є набір "класів-родичів" - у яких єдиний інтерфейс та різна реалізація алгоритмів.

Недоліками такого алгоритму є те, що реалізація жорстко прив'язана до підкласу, що ускладнює внесення змін.

Вирішенням даної проблеми є використання патерну Стратегія (Strategy).

Опис патерну Strategy

Реалізувати програму, якою рахуватиметься знижка для покупця, можна за допомогою патерну Strategy. Для цього створюється декілька класів "Стратегія", кожен з яких містить один і той же поліморфний метод "Порахувати вартість". Як параметри в метод передаються дані про продаж. Об'єкт Strategy має зв'язок з конкретним об'єктом - для якого використовується алгоритм.

Переваги

1. Можливість позбутися умовних операторів.
2. Клієнт може вибирати найбільш влучну стратегію залежно від вимог щодо швидкодії і пам'яті.

Недоліки

1. Збільшення кількості об'єктів.
2. Клієнт має знати особливості реалізацій стратегій для вибору найбільш вдалої.

Висновки

Останнім часом розроблено багато мов програмування, але в кожній з них для досягнення найкращого результату роботи необхідно використовувати шаблони програмування, одним з яких є Стратегія (Strategy).

```
// Класс реализующий конкретную стратегию, должен наследовать этот интерфейс
// Класс контекста использует этот интерфейс для вызова конкретной стратегии
interface Strategy {
    int execute(int a, int b);
}

// Реализуем алгоритм с использованием интерфейса стратегии
class ConcreteStrategyAdd implements Strategy {

    public int execute(int a, int b) {
        System.out.println("Called ConcreteStrategyAdd's execute()");
        return a + b;  // Do an addition with a and b
    }
}

class ConcreteStrategySubtract implements Strategy {

    public int execute(int a, int b) {
        System.out.println("Called ConcreteStrategySubtract's execute()");
        return a - b;  // Do a subtraction with a and b
    }
}

class ConcreteStrategyMultiply implements Strategy {

    public int execute(int a, int b) {
        System.out.println("Called ConcreteStrategyMultiply's execute()");
        return a * b;   // Do a multiplication with a and b
    }
}

// Класс контекста использующий интерфейс стратегии
class Context {

    private Strategy strategy;

    // Constructor
    public Context(Strategy strategy) {
        this.strategy = strategy;
    }

    public int executeStrategy(int a, int b) {
        return strategy.execute(a, b);
    }
}

// Тестовое приложение
class StrategyExample {

    public static void main(String[] args) {

        Context context;

        context = new Context(new ConcreteStrategyAdd());
        int resultA = context.executeStrategy(3,4);

        context = new Context(new ConcreteStrategySubtract());
        int resultB = context.executeStrategy(3,4);

        context = new Context(new ConcreteStrategyMultiply());
        int resultC = context.executeStrategy(3,4);
```

```
        }
    }
```

Лекція 11. Шаблони поведінки: Command та Visitor.

- Шаблон Command: мотивація, структура, учасники, відносини, шляхи застосування та результат використання.
- Приклад реалізації шаблону Command.
- Шаблон Visitor: мотивація, структура, учасники, відносини, шляхи застосування та результат використання.
- Приклад реалізації шаблону Visitor. [1, с. 227-236, 300-309]

Шаблон Command

Команда (англ. *Command*) — шаблон проектування, відноситься до класу шаблонів поведінки. Також відомий як *Дія* (англ. *Action*), *Транзакція* (англ. *Transaction*).

Призначення

Інкапсулює запит у формі об'єкта, дозволяючи тим самим задавати параметри клієнтів для обробки відповідних запитів, ставити запити у чергу або протоколювати їх, а також підтримувати скасовування операцій.

Мотивація

Застосовність

Слід використовувати шаблон *Команда* коли:

- треба параметризувати об'єкти дією. У процедурній мові таку параметризацію можна виразити за допомогою функції оберненого виклику, тобто такою функцією, котра реєструється, щоби бути викликаною пізніше. Команди є об'єктно-орієнтованою альтернативою функціям оберненого виклику;
- визначати, ставити у чергу та виконувати запити у різний час. Строк життя об'єкта *Команди* не обов'язково залежить від строку життя початкового запиту. Якщо отримувача вдається реалізувати таким чином, щоб він не залежав від адресного простору, то об'єкт-команду можна передати іншому процесу, котрий займеться його виконанням;
- потрібна підтримка скасовування операцій. Операція *Execute* об'єкта *Команда* може зберегти стан, що необхідний для відкочення дій, виконаних *Командою*. У цьому разі у інтерфейсі класу *Command* повинна бути додаткова операція *Unexecute*, котра скасовує дії, виконанні попереднім викликом операції *Execute*. Виконані команди зберігаються у списку історії. Для реалізації довільної кількості рівней скасування та повтору команд треба обходити цей список відповідно в оберненому та прямому напрямках, викликаючи під час відвідування кожного елементу операцію *Unexecute* або *Execute*;

- підтримати протоколювання змін, щоб їх можна було виконати повторно після аварійної зупинки системи. Доповнивши інтерфейс класу *Command* операціями зберігання та завантаження, можна вести протокол змін у внутрішній пам'яті. Для поновлення після збою треба буде завантажити збереженні команди з диску та повторно виконати їх за допомогою операції *Execute*;
- треба структурувати систему на основі високорівневих операцій, що побудовані з примітивних. Така структура є типовою для інформаційних систем, що підтримують транзакції. Транзакція інкапсулює множину змін даних. Шаблон *Команда* дозволяє моделювати транзакції. В усіх команд є спільний інтерфейс, що надає можливість працювати однаково з будь-якими транзакціями. За допомогою цього шаблону можна легко додавати у систему нові види транзакцій.

Структура

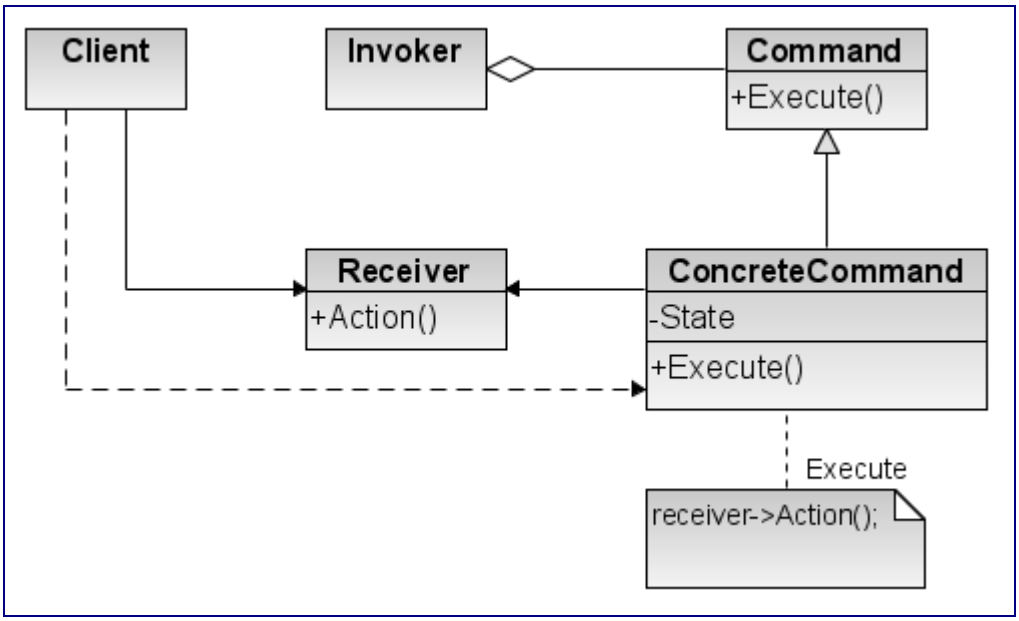


Рис. 11.1
UML діаграма, що описує структуру шаблону проектування *Команда*

- Command — команда:
  - оголошує інтерфейс для виконання операції;
- ConcreteCommand — конкретна команда:
  - визначає зв'язок між об'єктом-отримувачем *Receiver* та дією;
  - реалізує операцію *Execute* шляхом виклику відповідних операцій об'єкта *Receiver*;
- Client — клієнт:
  - створює об'єкт класу *ConcreteCommand* та встановлює його отримувача;
- Invoker — викликач:

    - звертається до команди щоб та виконала запит;
- Receiver — отримувач:
    - має у своєму розпорядженні усю інформацію про способи виконання операцій, необхідних для задоволення запиту. У ролі отримувача може виступати будь-який клас.

Відносини

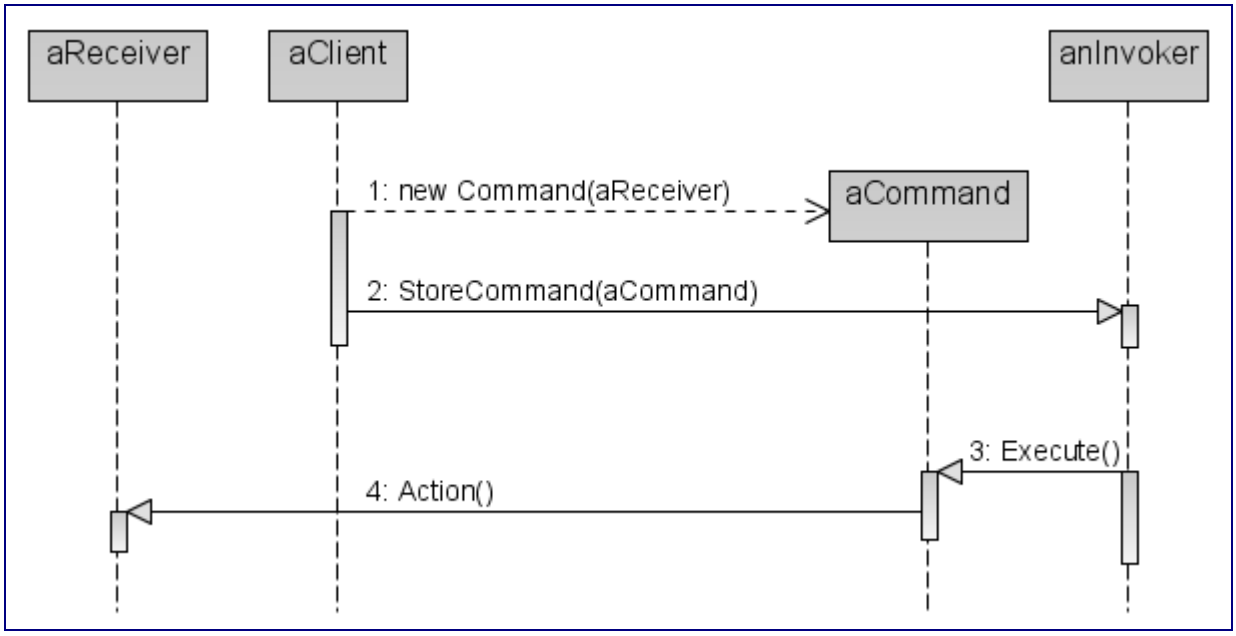


Рис. 11.2
UML діаграма, що описує взаємовідносини поміж об'єктів шаблону проектування *Команда*

- клієнт створює об'єкт *ConcreteCommand* та встановлює для нього отримувача;
- викликач *Invoker* зберігає об'єкт *ConcreteCommand*;
- викликач надсилає запит, викликаючи операцію команди *Execute*. Якщо підтримується скасування виконаних дій, то *ConcreteCommand* перед викликом *Execute* зберігає інформацію про стан, достатню для виконання скасування;
- об'єкт *ConcreteCommand* викликає операції отримувача для виконання запиту

На діаграмі видно, як *Command* розриває зв'язок між викликачем та отримувачем (а також запитом, що повинен бути виконаний останнім).

```
      /*the Invoker class*/

public class Switch {
    private Command flipUpCommand;
```

```
    private Command flipDownCommand;

    public Switch(Command flipUpCmd,Command flipDownCmd){
            this.flipUpCommand=flipUpCmd;
            this.flipDownCommand=flipDownCmd;
           }

    public void flipUp(){
         flipUpCommand.execute();
    }

    public void flipDown(){
         flipDownCommand.execute();
    }
}

/*Receiver class*/

public class Light{
     public Light(){  }

     public void turnOn(){
        System.out.println("The light is on");
     }

     public void turnOff(){
        System.out.println("The light is off");
     }
}


/*the Command interface*/

public interface Command{
    void execute();
}


/*the Command for turning on the light*/

public class TurnOnLightCommand implements Command{
   private Light theLight;

   public TurnOnLightCommand(Light light){
        this.theLight=light;
       }

   public void execute(){
      theLight.turnOn();
   }
}

/*the Command for turning off the light*/

public class TurnOffLightCommand implements Command{
   private Light theLight;

   public TurnOffLightCommand(Light light){
        this.theLight=light;
       }

   public void execute(){
```

```
      theLight.turnOff();
   }
}

/*The test class*/
public class TestCommand{
   public static void main(String[] args){
        Light l=new Light();
        Command switchUp=new TurnOnLightCommand(l);
        Command switchDown=new TurnOffLightCommand(l);

        Switch s=new Switch(switchUp,switchDown);

        s.flipUp();
        s.flipDown();
   }
}
```

Відвідувач (Visitor)

Відвідувач (Visitor) - шаблон проектування, який дозволяє відділити певний алгоритм від елементів, на яких алгоритм має бути виконаний, таким чином можливо легко додати або ж змінити алгоритм без змін щодо елементів системи. Практичним результатом є можливість додавання нових операцій в існуючі структури об'єкта без зміни цих структур.

Відвідувач дозволяє додавати нові віртуальні функції в родинні класи без зміни самих класів, натомість, один відвідувач створює клас, який реалізує всі відповідні спеціалізації віртуальної функції. Відвідувач бере приклад посилання в якості вхідних даних і реалізується шляхом подвійної диспетчеризації.Призначення шаблону

- Шаблон Відвідувач визначає операцію, виконувану над кожним елементом деякої структури. Дозволяє, не змінюючи класи цих об'єктів, додавати в них нові операції.
- Є класичною технікою для відновлення втраченої інформації про тип.
- Шаблон Відвідувач дозволяє виконати потрібні дії в залежності від типів двох об'єктів.

Проблема

Над кожним об'єктом деякої структури виконується одна або більше операцій. Визначити нову операцію, не змінюючи класи об'єктів.

Опис шаблону

Основним призначенням патерну Відвідувач є введення абстрактної функціональності для сукупної ієрархічної структури об'єктів "елемент", а саме, патерн відвідувач дозволяє, не змінюючи класи елементів, додавати в них нові

операції. Для цього вся обробна функціональність переноситься з самих класів елементів в ієрархію спадкування Відвідувача.

Шаблон відвідувач дозволяє легко додавати нові операції - потрібно просто додати новий похідний від відвідувача клас. Однак патерн Відвідувач слід використовувати тільки в тому випадку, якщо підкласи елементів сукупної ієрархічної структури залишаються стабільними (незмінними). В іншому випадку, потрібно докласти значних зусиль на оновлення всієї ієрархії.

Іноді наводяться заперечення з приводу використання патерну Відвідувач, оскільки він розділяє дані та алгоритми, що суперечить концепції об'єктно-орієнтованого програмування . Однак успішний досвід застосування STL, де поділ даних і алгоритмів покладено в основу, доводить можливість використання патерну відвідувач.

Особливості шаблону

- Сукупна структура об'єктів елементу може визначатися за допомогою патерну Компонувальник (Composite).
- Для обходу може використовуватися Ітератор (Iterator).
- Шаблон Відвідувач демонструє класичний прийом відновлення інформації про втрачені типи, не вдаючись до понижуючого приведення типів(динамічне приведення).

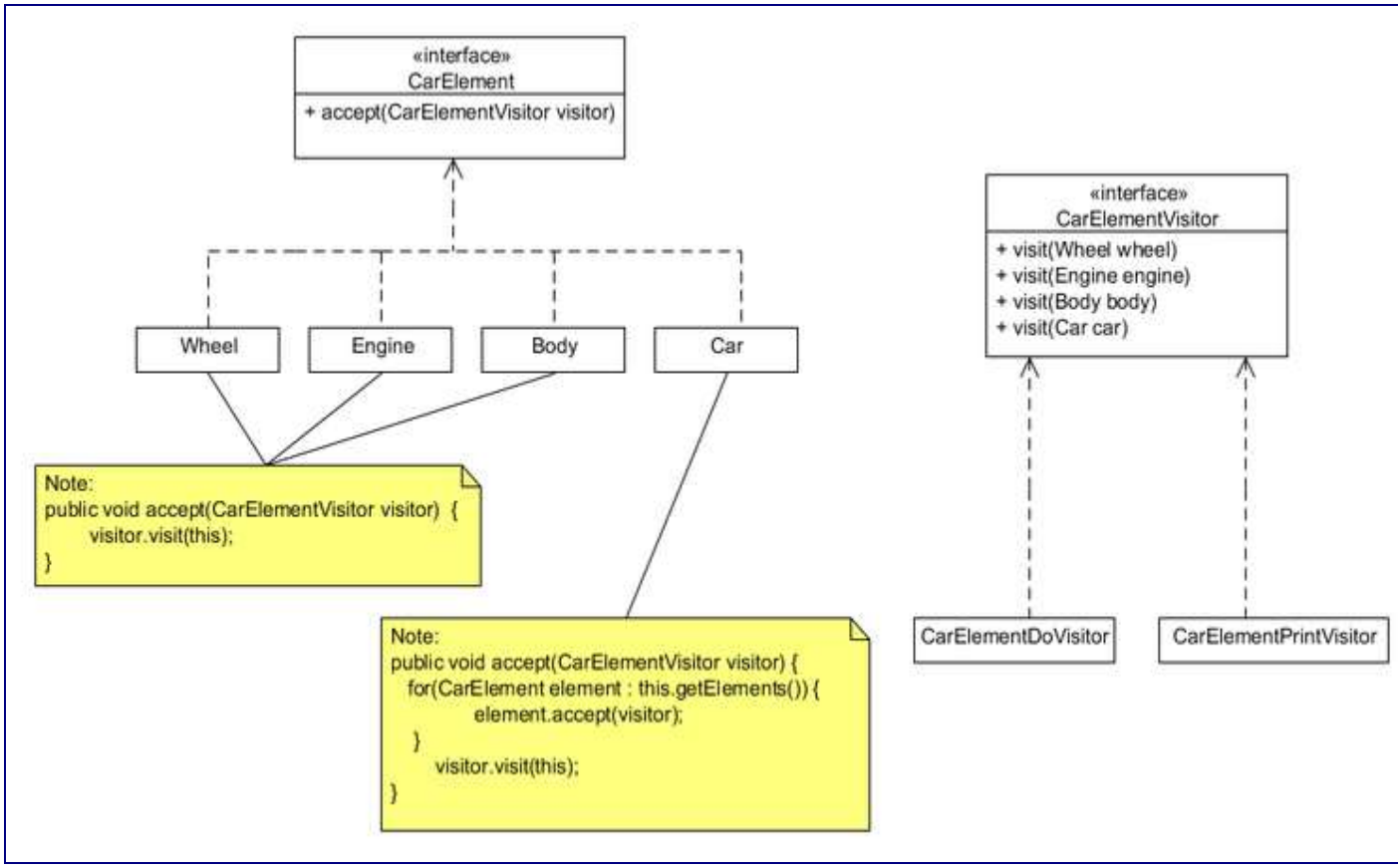


рис. 11.3

Приклад

```
interface CarElementVisitor {
    void visit(Wheel wheel);
    void visit(Engine engine);
    void visit(Body body);
    void visit(Car car);
}

interface CarElement {
    void  accept(CarElementVisitor  visitor);  //  CarElements  have  to  provide
accept().
}

class Wheel implements CarElement {
    private String name;

    public Wheel(String name) {
        this.name = name;
    }

    public String getName() {
        return this.name;
    }

    public void accept(CarElementVisitor visitor) {
        /*
         * accept(CarElementVisitor) in Wheel implements
         * accept(CarElementVisitor) in CarElement, so the call
         * to accept is bound at run time. This can be considered
         * the first dispatch. However, the decision to call
         * visit(Wheel) (as opposed to visit(Engine) etc.) can be
         * made during compile time since 'this' is known at compile
         * time to be a Wheel. Moreover, each implementation of
         * CarElementVisitor implements the visit(Wheel), which is
         * another decision that is made at run time. This can be
         * considered the second dispatch.
         */
        visitor.visit(this);
    }
}

class Engine implements CarElement {
    public void accept(CarElementVisitor visitor) {
        visitor.visit(this);
    }
}

class Body implements CarElement {
    public void accept(CarElementVisitor visitor) {
        visitor.visit(this);
    }
}

class Car implements CarElement {
    CarElement[] elements;

    public Car() {
        //create new Array of elements
        this.elements = new CarElement[] { new Wheel("front left"),
            new Wheel("front right"), new Wheel("back left") ,
            new Wheel("back right"), new Body(), new Engine() };
    }
```

```
    public void accept(CarElementVisitor visitor) {
        for(CarElement elem : elements) {
            elem.accept(visitor);
        }
        visitor.visit(this);
    }
}

class CarElementPrintVisitor implements CarElementVisitor {
    public void visit(Wheel wheel) {
        System.out.println("Visiting " + wheel.getName() + " wheel");
    }

    public void visit(Engine engine) {
        System.out.println("Visiting engine");
    }

    public void visit(Body body) {
        System.out.println("Visiting body");
    }

    public void visit(Car car) {
        System.out.println("Visiting car");
    }
}

class CarElementDoVisitor implements CarElementVisitor {
    public void visit(Wheel wheel) {
        System.out.println("Kicking my " + wheel.getName() + " wheel");
    }

    public void visit(Engine engine) {
        System.out.println("Starting my engine");
    }

    public void visit(Body body) {
        System.out.println("Moving my body");
    }

    public void visit(Car car) {
        System.out.println("Starting my car");
    }
}

public class VisitorDemo {
    static public void main(String[] args) {
        Car car = new Car();
        car.accept(new CarElementPrintVisitor());
        car.accept(new CarElementDoVisitor());
    }
}
```

Лекція 12. Шаблони поведінки: State та Memento.

- Шаблон State: мотивація, структура, учасники, відносини, шляхи застосування та результат використання.
- Приклад реалізації шаблону State.
- Шаблон Memento: мотивація, структура, учасники, відносини, шляхи застосування та результат використання.
- Приклад реалізації шаблону Memento. [1, с. 272-280, 291-300, 314-328]

Шаблон State

Стан (англ. *State*) — шаблон проектування, відноситься до класу шаблонів поведінки.Призначення
Дозволяє об'єктові варіювати свою поведінку у залежності від внутрішнього стану. Ззовні здається, що змінився клас об'єкта.

Застосовність
Слід використовувати шаблон *Стан* у випадках, коли:

- поведінка об'єкта залежить від його стану та повинно змінюватись під час виконання програми;
- у коді операцій зустрічаються умовні оператори, що складаються з багатьох частин, у котрих вибір гілки залежить від стану. Зазвичай у такому разі стан представлено константами, що перелічуються. Часто одна й та ж структура умовного оператору повторюється у декількох операціях. Шаблон *Стан* пропонує помістити кожну гілку у окремий клас. Це дозволить трактувати стан об'єкта як самостійний об'єкт, котрий можна змінитися незалежно від інших.

Структура

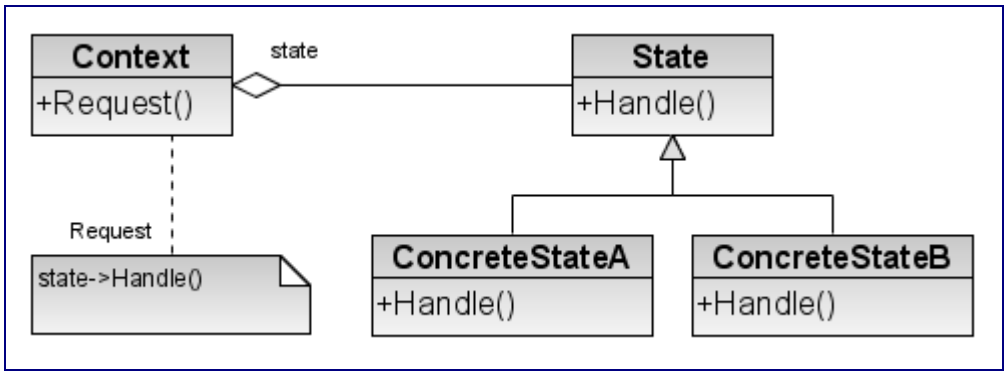


Рис. 12.1. UML діаграма, що описує структуру шаблону проектування *Стан*

- Context — контекст:
  - визначає інтерфейс, що є корисним для клієнтів;

    - зберігає екземпляр підкласу *ConcreteState*, котрим визначається поточний стан;
- State — стан:
    - визначає інтерфейс для інкапсуляції поведінки, асоційованої з конкретним станом контексту *Context*;
- Підкласи ConcreteState — конкретні стани:
    - кожний підклас реалізує поведінку, асоційовану з деяким станом контексту *Context*.

Відносини

- клас *Context* делегує залежні від стану запити до поточного об'єкта *ConcreteState*;
- контекст може передати себе у якості аргументу об'єкта *State*, котрий буде обробляти запит. Це надає можливість об'єкта-стану при необхідності отримати доступ до контексту;
- *Context* — це головний інтерфейс для клієнтів. Клієнти можуть конфігурувати контекст об'єктами стану *State*. Один раз зконфігурувавши контекст, клієнти вже не повинні напряму зв'язуватися з об'єктами стану;
- або *Context*, або підкласи *ConcreteState* можуть вирішити, за яких умов та у якій послідовності відбувається зміна станів.

```
// "интерфейс" State

  function State() {
          this.someMethod = function() { };
          this.nextState = function() { };
  }


// реализация State

        // первое состояние
        function StateA(widjet) {
                var dublicate = this; // ссылка на инстанцирующийся объект (т.к. this может меняться)

                this.someMethod = function() {
                        alert("StateA.someMethod");
                        dublicate.nextState();
                };
                this.nextState = function() {
                        alert("StateA > StateB");
                        widjet.onNextState( new StateB(widjet) );
                };
        }
        StateA.prototype = new State();
        StateA.prototype.constructor = StateA;

        // второе состояние
        function StateB(widjet) {
                var dublicate = this;

                this.someMethod = function() {
                        alert("StateB.someMethod");
```

```
                        dublicate.nextState();
                };
                this.nextState = function() {
                        alert("StateB > StateA");
                        widjet.onNextState( new StateA(widjet) );
                };
        }
        StateB.prototype = new State();
        StateB.prototype.constructor = StateB;


// "интерфейс" Widget

        function Widget() {
                this.someMethod = function() { };
                this.onNextState = function(state) { };
        }


// реализация Widget

        function Widget1() {
                var state = new StateA(this);

                this.someMethod = function() {
                        state.someMethod();
                };
                this.onNextState = function(newState) {
                        state = newState;
                };
        }
        Widget1.prototype = new Widget();
        Widget1.prototype.constructor = Widget1;


// использование

var widget = new Widget1();
widget.someMethod(); // StateA.someMethod
                     // StateA > StateB
widget.someMethod(); // StateB.someMethod
                     // StateB > StateA
```

Шаблон Memento

Зні́мок (англ. *Memento*) — шаблон проектування, відноситься до класу шаблонів поведінки.

Призначення

Не порушуючи інкапсуляції, фіксує та виносить за межі об'єкта його внутрішній стан так, щоб пізніше можна було відновити з нього об'єкт.

Мотивація

Застосовність

Слід використовувати шаблон *Знімок* у випадках, коли:

- необхідно зберегти миттєвий знімок стану об'єкта (або його частини), щоб згодом об'єкт можна було відтворити у тому ж самому стані;
- безпосереднє вилучення цього стану розкриває деталі реалізації та порушує інкапсуляцію об'єкта.

Структура

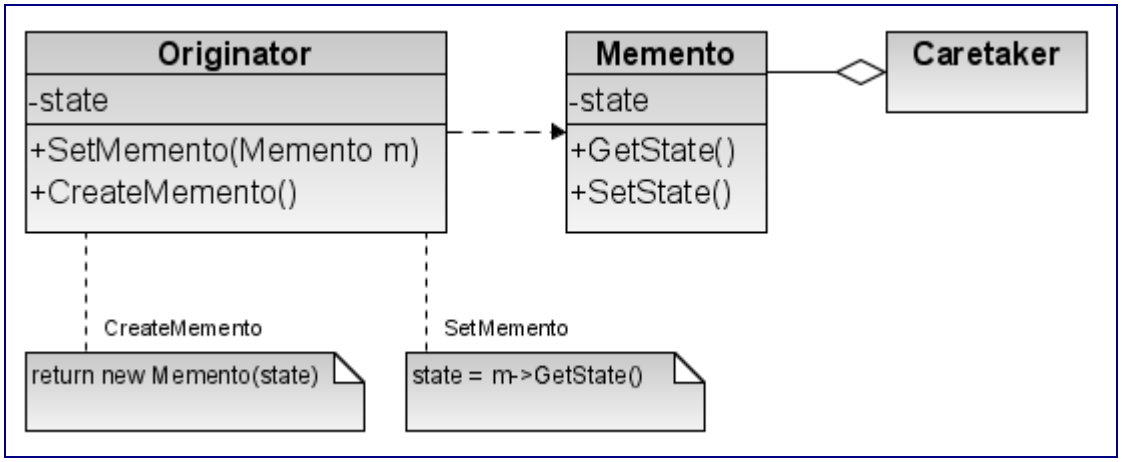


Рис.12.2. UML діаграма, що описує структуру шаблону проектування *Знімок*

- Memento — контекст:
  - зберігає внутрішній стан об'єкта *Originator*. Обсяг інформації, що зберігається, може бути різним та визначається потребами хазяїна;
  - забороняє доступ усім іншим об'єктам окрім хазяїна. По суті *знімок* має два інтерфейси. Опікун *Caretaker* користується лише вузьким інтерфейсом *знімку* — він може лише передавати *знімок* іншим об'єктам. Напроти, хазяїн користується широким інтерфейсом, котрий забезпечує доступ до всіх даних, необхідних для відтворення об'єкта (чи його частини) у попередньому стані. Ідеальний варіант — коли тільки хазяїну, що створив знімок, відкритий доступ до внутрішнього стану знімку;
- Originator — хазяїн:
  - створює знімок, що утримує поточний внутрішній стан;
  - використовує знімок для відтворення внутрішнього стану;
- CareTaker — опікун:
  - відповідає за зберігання знімку;
  - не проводить жодних операцій над знімком та не має уяви про його внутрішній зміст.

Відносини

- опікун запитує знімок у хазяїна, деякий час тримає його у себе, опісля повертає хазяїну. Іноді цього не відбувається, бо хазяїн не має необхідності відтворювати свій попередній стан;

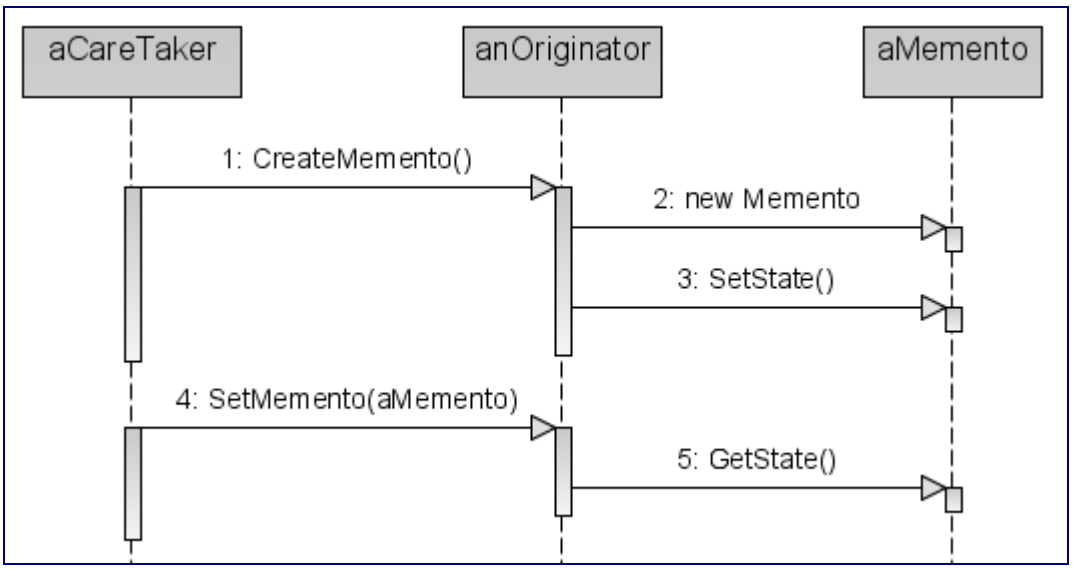


Рис. 12.3. UML діаграма, що описує відносини між об'єктами шаблону проектування *Знімок*

- знімки пасивні. Тільки хазяїн, що створив знімок, має доступ до інформації про стан.

```
      import java.util.List;
import java.util.ArrayList;
class Originator {
    private String state;
    // The class could also contain additional data that is not part of the
    // state saved in the memento.

    public void set(String state) {
        System.out.println("Originator: Setting state to " + state);
        this.state = state;
    }

    public Memento saveToMemento() {
        System.out.println("Originator: Saving to Memento.");
        return new Memento(state);
    }

    public void restoreFromMemento(Memento memento) {
        state = memento.getSavedState();
        System.out.println("Originator: State after restoring from Memento: " +
state);
    }

    public static class Memento {
        private final String state;

        public Memento(String stateToSave) {
            state = stateToSave;
        }

        public String getSavedState() {
            return state;
        }
    }
}
```

```
class Caretaker {
    public static void main(String[] args) {
        List<Originator.Memento> savedStates = new
ArrayList<Originator.Memento>();

        Originator originator = new Originator();
        originator.set("State1");
        originator.set("State2");
        savedStates.add(originator.saveToMemento());
        originator.set("State3");
        // We can request multiple mementos, and choose which one to roll back to.
        savedStates.add(originator.saveToMemento());
        originator.set("State4");

        originator.restoreFromMemento(savedStates.get(1));
    }
}
```

The output is:

```
Originator: Setting state to State1
Originator: Setting state to State2
Originator: Saving to Memento.
Originator: Setting state to State3
Originator: Saving to Memento.
Originator: Setting state to State4
Originator: State after restoring from Memento: State3
```

Лекція 13. Шаблони поведінки: Interpreter та Chain of Responsibility.

- Шаблон Interpreter: мотивація, структура, учасники, відносини, шляхи застосування та результат використання.
- Приклад реалізації шаблону Interpreter.
- Шаблон Chain of Responsibility: мотивація, структура, учасники, відносини, шляхи застосування та результат використання.
- Приклад реалізації шаблону Chain of Responsibility. [1, с. 238-249, 217-226]

Шаблон Interpreter

Інтерпретатор (англ. *Interpreter*) — шаблон проектування, відноситься до класу шаблонів поведінки.

Призначення

Для заданої мови визначає представлення її граматики, а також інтерпретатор речень цієї мови.

Мотивація

У разі, якщо якась задача виникає досить часто, є сенс подати її конкретні проявлення у вигляді речень простою мовою. Потім можна буде створити інтерпретатор, котрий вирішує задачу, аналізуючи речення цієї мови.
Наприклад, пошук рядків за зразком — досить розповсюджена задача. Регулярні вирази — це стандартна мова для задання зразків пошуку.

Застосовність

Шаблон Інтерпретатор слід використовувати, коли є мова для інтерпретації, речення котрої можна подати у вигляді абстрактних синтаксичних дерев. Найкраще шаблон працює коли:

- граматика проста. Для складних граматик ієрархія класів стає занадто громіздкою та некерованою. У таких випадках краще застосовувати генератори синтаксичних аналізаторів, оскільки вони можуть інтерпретувати вирази, не будуючи абстрактних синтаксичних дерев, що заощаджує пам'ять, а можливо і час;
- ефективність не є головним критерієм. Найефективніші інтерпретатори зазвичай не працюють безпосередньо із деревами, а спочатку транслюють їх в іншу форму. Так, регулярний вираз часто перетворюють на скінченний автомат. Але навіть у цьому разі сам транслятор можна реалізувати за допомогою шаблону інтерпретатор.

Структура

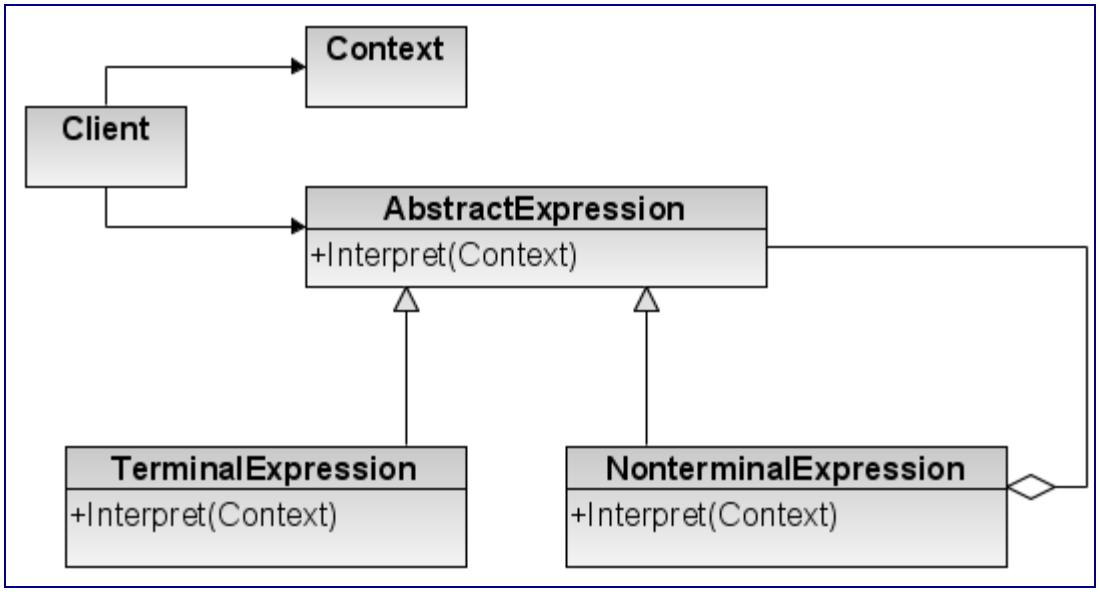


Рис.13.1. UML діаграма, що описує структуру шаблону проектування Інтерпретатор

- AbstractExpression — абстрактний вираз:
  - оголошує абстрактну операцію *Interpret*, загальну для усіх вузлів у абстрактному синтаксичному дереві;
- TerminalExpression — термінальний вираз:
  - реалізує операцію *Interpret* для термінальних символів граматики;
  - необхідний окремий екземпляр для кожного термінального символу у реченні;
- NonterminalExpression — нетермінальний вираз:
  - по одному такому класу потребується для кожного граматичного правила;
  - зберігає змінні екземпляру типу *AbstractExpression* для кожного символу;
  - реалізує операцію *Interpret* для нетермінальних символів граматики. Ця операція рекурсивно викликає себе для змінних, зберігаючих символи;
- Context — контекст:
  - містить інформацію, глобальну по відношенню до інтерпретатору;
- Client — клієнт:
  - будує (або отримує у готовому вигляді) абстрактне синтаксичне дерево, репрезентуюче окреме речення мовою з даною граматикою. Дерево складено з екземплярів класів *NonterminalExpression* та *TerminalExpression*;
  - викликає операцію *Interpret*.

Відносини

- клієнт будує (або отримує у готовому вигляді) речення у вигляді абстрактного синтаксичного дерева, у вузлах котрого знаходяться об'єкти класів *NonterminalExpression* та *TerminalExpression*. Далі клієнт ініціалізує контекст та викликає операцію *Interpret*;
- у кожному вузлі виду *NonterminalExpression* через операції *Interpret* визначається операція *Interpret* для кожного підвиразу. Для класу *TerminalExpression* операція Interpret визначає базу рекурсії;
- операції *Interpret* у кожному вузлі використовують контекст для зберігання та доступу до стану інтерпретатору.

```
import java.util.Map;

interface Expression {
    public int interpret(Map<String,Expression> variables);
}

class Number implements Expression {
    private int number;
    public Number(int number)       { this.number = number; }
    public int interpret(Map<String,Expression> variables)  { return number; }
}

class Plus implements Expression {
    Expression leftOperand;
    Expression rightOperand;
    public Plus(Expression left, Expression right) {
        leftOperand = left;
        rightOperand = right;
    }

    public int interpret(Map<String,Expression> variables)  {
        return                  leftOperand.interpret(variables)                +
rightOperand.interpret(variables);
    }
}

class Minus implements Expression {
    Expression leftOperand;
    Expression rightOperand;
    public Minus(Expression left, Expression right) {
        leftOperand = left;
        rightOperand = right;
    }

    public int interpret(Map<String,Expression> variables)  {
        return                  leftOperand.interpret(variables)                -
rightOperand.interpret(variables);
    }
}

class Variable implements Expression {
    private String name;
    public Variable(String name)       { this.name = name; }
    public int interpret(Map<String,Expression> variables)  {
        if(null==variables.get(name)) return 0; //Either return new Number(0).
        return variables.get(name).interpret(variables);
    }
```

```
}

import java.util.Map;
import java.util.Stack;

class Evaluator implements Expression {
    private Expression syntaxTree;

    public Evaluator(String expression) {
        Stack<Expression> expressionStack = new Stack<Expression>();
        for (String token : expression.split(" ")) {
            if  (token.equals("+")) {
                Expression   subExpression   =   new   Plus(expressionStack.pop(),
expressionStack.pop());
                expressionStack.push( subExpression );
            }
            else if (token.equals("-")) {
                // it's necessary remove first the right operand from the stack
                Expression right = expressionStack.pop();
                // ..and after the left one
                Expression left = expressionStack.pop();
                Expression subExpression = new Minus(left, right);
                expressionStack.push( subExpression );
            }
            else
                expressionStack.push( new Variable(token) );
        }
        syntaxTree = expressionStack.pop();
    }

    public int interpret(Map<String,Expression> context) {
        return syntaxTree.interpret(context);
    }
}

import java.util.Map;
import java.util.HashMap;

public class InterpreterExample {
    public static void main(String[] args) {
        String expression = "w x z - +";
        Evaluator sentence = new Evaluator(expression);
        Map<String,Expression> variables = new HashMap<String,Expression>();
        variables.put("w", new Number(5));
        variables.put("x", new Number(10));
        variables.put("z", new Number(42));
        int result = sentence.interpret(variables);
        System.out.println(result);
    }
}
```

Шаблон Chain of Responsibility

Ланцюжок відповідальностей - шаблон об'єктно-орієнтованого дизайну у програмуванні.

В об'єктно-орієнтованому дизайні, шаблон «ланцюжок відповідальностей» є шаблоном, який складається з об'єктів «команда» і серії об'єктів-виконавців. Кожен об'єкт-виконавець має логіку, що описує типи об'єктів «команда», які він

може обробляти, а також як передати далі ланцюжком ті об'єкти-команди, що він не може обробляти. Крім того існує механізм для додавання нових призначених для обробки об'єктів у кінець ланцюжка.

У варіаціях стандартного ланцюжка відповідальностей, деякі обробники можуть бути в ролі диспетчерів, які здатні відсилати команди в різні напрямки формуючи Дерево відподальності. У деяких випадках це можна організувати рекурсивно, коли об'єкт який оброблюється викликає об'єкт вищого рівня обробки з командою що пробує вирішити меншу частину проблеми; у цьому випадку рекурсія продовжує виконуватися поки команда не виконається, або поки дерево повністю не буде оброблене. XML-інтерпретатор (проаналізований, але який ще не було поставлено на виконання) може бути хорошим прикладом.

Цей шаблон застосовує ідею слабкого зв'язку, який розглядається як програмування у найкращих практиках.

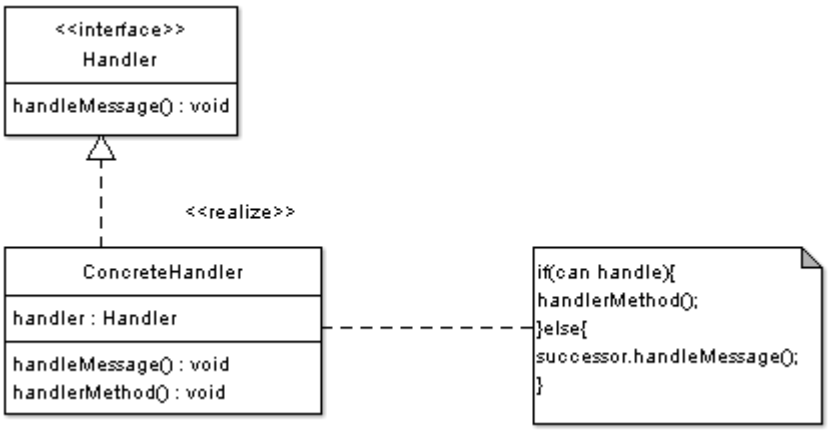


Рис. 13.2

```
package chainofresp;

abstract class Logger {
    public static int ERR = 3;
    public static int NOTICE = 5;
    public static int DEBUG = 7;
    protected int mask;

    // The next element in the chain of responsibility
    protected Logger next;

    public Logger setNext(Logger log) {
        next = log;
        return log;
    }

    public void message(String msg, int priority) {
        if (priority <= mask) {
            writeMessage(msg);
        }
        if (next != null) {
            next.message(msg, priority);
        }
    }

    abstract protected void writeMessage(String msg);
```

```
}

class StdoutLogger extends Logger {
    public StdoutLogger(int mask) {
        this.mask = mask;
    }

    protected void writeMessage(String msg) {
        System.out.println("Writing to stdout: " + msg);
    }
}


class EmailLogger extends Logger {
    public EmailLogger(int mask) {
        this.mask = mask;
    }

    protected void writeMessage(String msg) {
        System.out.println("Sending via email: " + msg);
    }
}

class StderrLogger extends Logger {
    public StderrLogger(int mask) {
        this.mask = mask;
    }

    protected void writeMessage(String msg) {
        System.err.println("Sending to stderr: " + msg);
    }
}

public class ChainOfResponsibilityExample {
    public static void main(String[] args) {
        // Build the chain of responsibility
        Logger logger, logger1,logger2;
        logger = new StdoutLogger(Logger.DEBUG);
        logger1 = logger.setNext(new EmailLogger(Logger.NOTICE));
        logger2 = logger1.setNext(new StderrLogger(Logger.ERR));

        // Handled by StdoutLogger
        logger.message("Entering function y.", Logger.DEBUG);

        // Handled by StdoutLogger and EmailLogger
        logger.message("Step1 completed.", Logger.NOTICE);

        // Handled by all three loggers
        logger.message("An error has occurred.", Logger.ERR);
    }
}
/*
The output is:
   Writing to stdout:   Entering function y.
   Writing to stdout:   Step1 completed.
   Sending via e-mail:  Step1 completed.
   Writing to stdout:   An error has occurred.
   Sending via e-mail:  An error has occurred.
   Writing to stderr:   An error has occurred.
*/
```

Лекція 14. Породжувальні шаблони Prototype, Factory Method та Abstract Factory.

- Призначення шаблонів, що породжують.
- Шаблон Prototype: мотивація, структура, учасники, відносини, шляхи застосування та результат використання.
- Приклад реалізації шаблону Prototype. [1, с. 89-93, 121-130]
- Шаблон Factory Method: мотивація, структура, учасники, відносини, шляхи застосування та результат використання.
- Приклад реалізації шаблону Factory Method.
- Шаблон Abstract Factory: мотивація, структура, учасники, відносини, шляхи застосування та результат використання.
- Приклад реалізації шаблону Abstract Factory. [1, с. 111-121, 93-102]

Шаблон Prototype

Прототип (англ. *Prototype*) - шаблон проектування, відноситься до класу твірних шаблонів.

Призначення

Задає види об'єктів, що створюються, за допомогою екземпляру-прототипу, та створює нові об'єкти шляхом копіювання цього прототипу.

Застосування

Слід використовувати шаблон *Прототип* коли:

- класи, що інстанціюються, визначаються під час виконання, наприклад за допомогою динамічного завантаження;
- треба запобігти побудові ієрархій класів або фабрик, паралельних ієрархій класів продуктів;
    - екземпляри класу можуть знаходитись у одному з не дуже великої кількості станів. Може статися, що зручніше встановити відповідну кількість прототипів та клонувати їх, а не інстанціювати кожний раз клас вручну в слушному стані.

Структура

- UML діаграма, що описує структуру шаблону проектування *Прототип*
- Prototype – прототип:
- визначає інтерфейс для клонування самого себе;
- ConcretePrototype – конкретний прототип:
- реалізує операцію клонування самого себе;
- Client – клієнт:

- створює новий об'єкт, звертаючись до *прототипу* із запитом клонувати себе.

Відносини

Клієнт звертається до *прототипу*, щоб той створив свого клона.

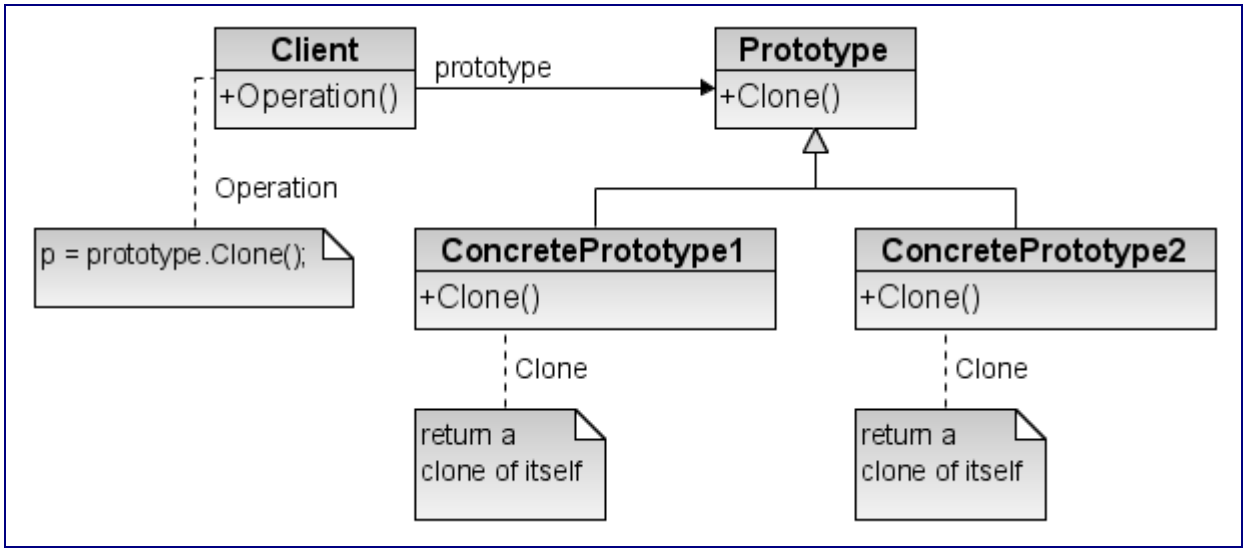


рис. 14.1

```
      /**
 * Prototype Class
 */
public class Cookie implements Cloneable {

    @Override
    public Cookie clone() throws CloneNotSupportedException {
        // call Object.clone()
        Cookie copy = (Cookie) super.clone();

        //In an actual implementation of this pattern you might now change
references to
        //the expensive to produce parts from the copies that are held inside the
prototype.

        return copy;
    }
}

/**
 * Concrete Prototypes to clone
 */
public class CoconutCookie extends Cookie { }

/**
 * Client Class
 */
public class CookieMachine {

    private Cookie cookie; // Could have been a private Cloneable cookie.

    public CookieMachine(Cookie cookie) {
```

```
        this.cookie = cookie;
    }

    public Cookie makeCookie() throws CloneNotSupportedException {
        return (Cookie) this.cookie.clone();
    }

    public static void main(String args[]) throws CloneNotSupportedException {
        Cookie tempCookie = null;
        Cookie prot = new CoconutCookie();
        CookieMachine cm = new CookieMachine(prot);
        for (int i = 0; i < 100; i++)
            tempCookie = cm.makeCookie();
    }
}
```

Шаблон Factory Method

Фабричний метод (англ. *Factory Method*) — шаблон проектування, відноситься до класу твірних шаблонів.

Призначення

Визначає інтерфейс для створення об'єкта, але залишає підкласам рішення про те, який саме клас інстанціювати. Фабричний метод дозволяє класу делегувати інстанціювання підкласам.

Застосування

Слід використовувати шаблон *Фабричний метод* коли:

- класу не відомо заздалегідь, об'єкти яких саме класів йому потрібно створювати;
- клас спроектовано так, щоб об'єкти, котрі він створює, специфікувалися підкласами;
- клас делегує свої обов'язки одному з кількох допоміжних підкласів, та потрібно локалізувати знання про те, який саме підклас приймає ці обов'язки на себе.

Структура

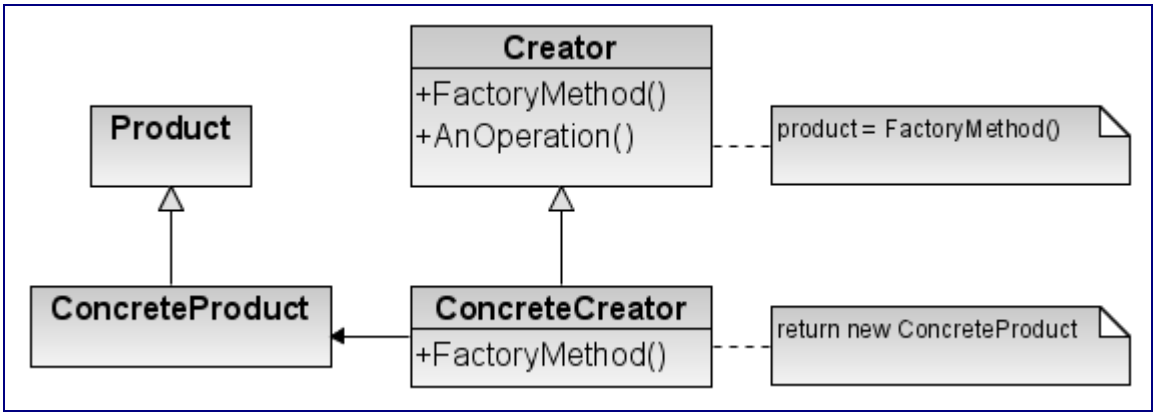


Рис.14.2. UML діаграма, що описує структуру шаблону проектування *Фабричний метод*

- Product — продукт: визначає інтерфейс об'єктів, що створюються фабричним методом;
- ConcreteProduct — конкретний продукт: реалізує інтерфейс *Product*;
- Creator — творець: оголошує фабричний метод, що повертає об'єкт класу *Product*. *Creator* може також визначати реалізацію за замовчанням фабричного методу, що повертає об'єкт *ConcreteProduct*;
- може викликати фабричний метод для створення об'єкта *Product*;
- ConcreteCreator — конкретний творець: заміщує фабричний метод, що повертає об'єкт *ConcreteProduct*.

```
// Product
abstract class Product {
}

// ConcreteProductA
class ConcreteProductA extends Product {
}

// ConcreteProductB
class ConcreteProductB extends Product {
}

// Creator
abstract class Creator {
    public abstract Product factoryMethod();
}

// У этого класса может быть любое кол-во наследников.
//  Для создания нужного нам объекта можно написать следующие Фабрики:
ConcreteCreatorA, ConcreteCreatorB

// ConcreteCreatorA
class ConcreteCreatorA extends Creator {
    @Override
    public Product factoryMethod() {
        return new ConcreteProductA();
    }
}

// ConcreteCreatorB
class ConcreteCreatorB extends Creator {
    @Override
    public Product factoryMethod() {
        return new ConcreteProductB();
    }
}


public class FactoryMethodExample {

    public static void main(String[] args) {
        // an array of creators
        Creator[] creators = {new ConcreteCreatorA(), new ConcreteCreatorB()};

        // iterate over creators and create products
        for (Creator creator: creators) {
            Product product = creator.factoryMethod();
            System.out.printf("Created {%s}\n", product.getClass());
```

```
            }
        }
    }
```

Шаблон Abstract Factory

Абстрактна фабрика (англ. *Abstract Factory*) — шаблон проектування, відноситься до класу твірних шаблонів.

Призначення

Подає інтерфейс для утворення родин взаємозв'язаних або взаємозалежних об'єктів, не специфікуючи їхніх конкретних класів.

Застосування

Слід використовувати шаблон *Абстрактна фабрика* коли:

- система не повинна залежати від того, як утворюються, компонуються та представляються вхідні до неї об'єкти;
- вхідні до родини взаємозв'язані об'єкти повинні використовуватися разом і необхідно забезпечити виконання цього обмеження;
- система повинна конфігуруватися однією з родин складаючих її об'єктів;
- треба подати бібліотеку об'єктів, розкриваючи тільки їхні інтерфейси, але не реалізацію.

Структура

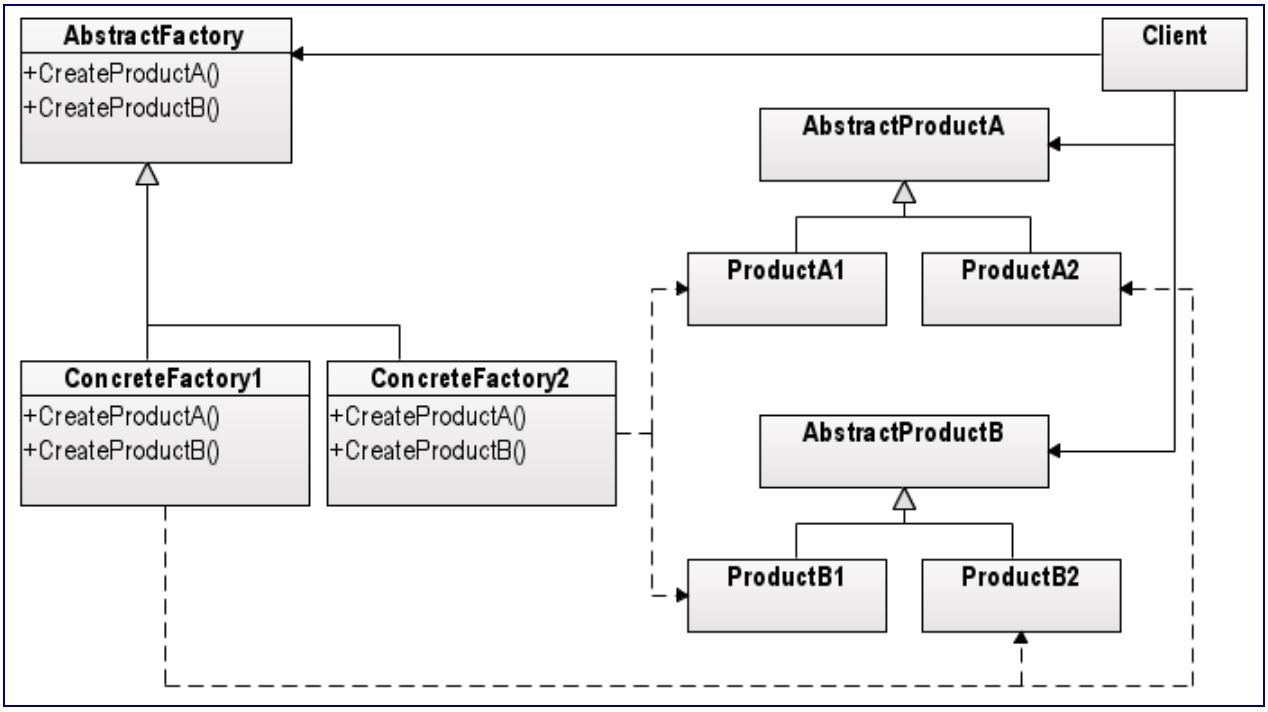


рис. 14.3 UML діаграма, що описує структуру шаблону проектування *Абстрактна фабрика*

- AbstractFactory — абстрактна фабрика: оголошує інтерфейс для операцій, що створюють абстрактні об'єкти-продукти;

- ConcreteFactory — конкретна фабрика: реалізує операції, що створюють конкретні об'єкти-продукти;
- AbstractProduct — абстрактний продукт: оголошує інтерфейс для типу об'єкта-продукту;
- ConcreteProduct — конкретний продукт: визначає об'єкт-продукт, що створюється відповідною конкретною *фабрикою*; реалізує інтерфейс *AbstractProduct*;
- Client — клієнт: користується виключно інтерфейсами, котрі оголошенні у класах *AbstractFactory* та *AbstractProduct*.

Відносини

- Зазвичай під час виконання створюється єдиний екземпляр класу *ConcreteFactory*. Ця конкретна фабрика створює об'єкти продукти, що мають досить визначену реалізацію. Для створення інших видів об'єктів клієнт повинен користуватися іншою конкретною фабрикою;
- *AbstractFactory* передоручає створення об'єктів продуктів своєму підкласу *ConcreteFact*

```
    // AbstractProducts
// создаем абстрактные классы продуктов

        // AbstractProductA
        // "интерфейс" пошлины за перевозку
        function ShipFeeProcessor() {
                this.calculate = function(order) { };
        }

        // AbstractProductB
        // "интерфейс" налога
        function TaxProcessor() {
                this.calculate = function(order) { };
        }


// Products
// создаем реализацию абстрактных классов

        // Product A1
        // класс для расчета пошлины за перевозку для Европы
        function EuropeShipFeeProcessor() {
                this.calculate = function(order) {
                        // перегрузка метода calculate ShipFeeProcessor
                        return 11 + order;
                };
        }
        EuropeShipFeeProcessor.prototype = new ShipFeeProcessor();
        EuropeShipFeeProcessor.prototype.constructor = EuropeShipFeeProcessor;

        // Product A2
        // класс для расчета пошлины за перевозку для Канады
        function CanadaShipFeeProcessor() {
                this.calculate = function(order) {
                        // перегрузка метода calculate ShipFeeProcessor
                        return 12 + order;
```

```
                };
        }
        CanadaShipFeeProcessor.prototype = new ShipFeeProcessor();
        CanadaShipFeeProcessor.prototype.constructor = CanadaShipFeeProcessor;

        // Product B1
        // класс для расчета налогов для Европы
        function EuropeTaxProcessor() {
                this.calculate = function(order) {
                        // перегрузка метода calculate TaxProcessor
                        return 21 + order;
                };
        }
        EuropeTaxProcessor.prototype = new TaxProcessor();
        EuropeTaxProcessor.prototype.constructor = EuropeTaxProcessor;

        // Product B2
        // класс для расчета налогов для Канады
        function CanadaTaxProcessor() {
                this.calculate = function(order) {
                        // перегрузка метода calculate TaxProcessor
                        return 22 + order;
                };
        }
        CanadaTaxProcessor.prototype = new TaxProcessor();
        CanadaTaxProcessor.prototype.constructor = CanadaTaxProcessor;


// AbstractFactory
// "интерфейс" фабрики
        function FinancialToolsFactory() {
                this.createShipFeeProcessor = function() {};
                this.createTaxProcessor = function() {};
        };


// Factories

        // ConcreteFactory1
        // Европейская фабрика будет возвращать нам экземпляры классов...
        function EuropeFinancialToolsFactory() {
                this.createShipFeeProcessor = function() {
                        // ...для расчета пошлины за перевозку для Европы
                        return new EuropeShipFeeProcessor();
                };
                this.createTaxProcessor = function() {
                        // ...для расчета налогов для Европы
                        return new EuropeTaxProcessor();
                };
        };
        EuropeFinancialToolsFactory.prototype = new FinancialToolsFactory();
        EuropeFinancialToolsFactory.prototype.constructor =
EuropeFinancialToolsFactory;

        // ConcreteFactory2
        //  аналогично,  Канадская  фабрика  будет  возвращать  нам  экземпляры
классов...
        function CanadaFinancialToolsFactory() {
                this.createShipFeeProcessor = function() {
                        // ...для расчета пошлины за перевозку для Канады
                        return new CanadaShipFeeProcessor();
                };
```

```
                this.createTaxProcessor = function() {
                        // ...для расчета налогов для Канады
                        return new CanadaTaxProcessor();
                };
        };
        CanadaFinancialToolsFactory.prototype = new FinancialToolsFactory();
        CanadaFinancialToolsFactory.prototype.constructor                    =
CanadaFinancialToolsFactory;


// Client
// класс для заказа товара
        function OrderProcessor() {
                var taxProcessor;
                var shipFeeProcessor;

                this.orderProcessor = function(factory) {
                        // метод создает экземпляры классов для:
                        shipFeeProcessor  =  factory.createShipFeeProcessor();  //
расчета пошлин за перевозку
                        taxProcessor  =  factory.createTaxProcessor();  //  расчета
налогов
                };

                this.processOrder = function(order)     {
                        // когда экземпляры классов для расчета созданы, нам надо
только воспользоваться ими
                        // ...
                        var resShipFee = shipFeeProcessor.calculate(order);
                        var resTax = taxProcessor.calculate(order);
                        // ...
                };

                this.debug = function(str, order) {
                        // для наглядности
                        alert(str +": "+ shipFeeProcessor.calculate(order) +", "+
taxProcessor.calculate(order));
                };
        }


// использование

var eu = new OrderProcessor(); // создаем "пустой" класс для заказа товара
eu.orderProcessor(new  EuropeFinancialToolsFactory());  //  передаем  ему  новый
экземпляр нужной нам фабрики
eu.processOrder(100); // производим заказ
eu.debug("eu", 100); // тест выведет: "eu: 111, 121"

var ca = new OrderProcessor();
ca.orderProcessor(new CanadaFinancialToolsFactory());
ca.processOrder(0);
ca.debug("ca", 0); // тест выведет: "ca: 12, 22"
```

Лекція 15. Шаблони Singleton та Builder.

- Шаблон Singleton: мотивація, структура, учасники, відносини, шляхи застосування та результат використання.
- Приклад реалізації шаблону Singleton.

- Шаблон Builder: мотивація, структура, учасники, відносини, шляхи застосування та результат використання.
- Приклад реалізації шаблону Builder. [1, с. 130-138, 102-111]

Шаблон Singleton

Одинак (англ. *Singleton*) — шаблон проектування, відноситься до класу твірних шаблонів. Гарантує, що клас матиме тільки один екземпляр, і забезпечує глобальну точку доступу до цього екземпляра.

Мотивація

Для деяких класів важливо, щоб існував тільки один екземпляр. Наприклад, хоча у системі може існувати декілька принтерів, може бути тільки один спулер. Повинна бути тільки одна файлова система та тільки один активний віконний менеджер.

Глобальна змінна не вирішує такої проблеми, бо не забороняє створити інші екземпляри класу.

Рішення полягає в тому, щоб сам клас контролював свою «унікальність», забороняючи створення нових екземплярів, та сам забезпечував єдину точку доступу. Це є призначенням шаблону *Одинак*.

Застосування

Слід використовувати шаблон *Одинак* коли:

- повинен бути тільки один екземпляр деякого класу, що легко доступний всім клієнтам;
- єдиний екземпляр повинен розширюватись шляхом успадкування, та клієнтам потрібно мати можливість працювати з розширеним екземпляром не змінюючи свій код.

Структура

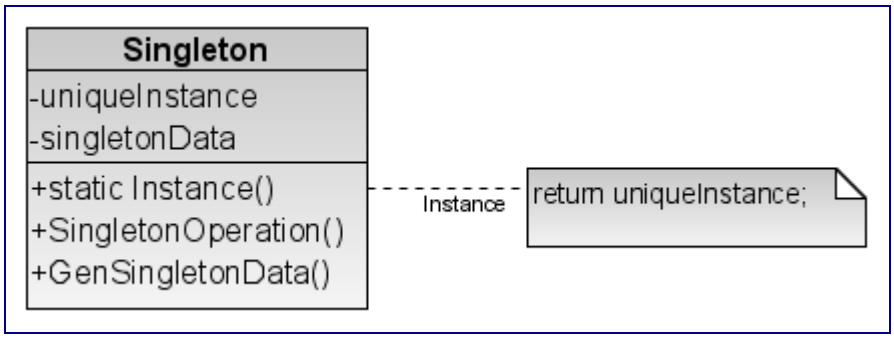


Рис. 15.1. Діаграма класів, що описує структуру шаблону проектування *Одинак*

- Singleton — одинак:
- визначає операцію *Instance*, котра дозволяє клієнтам отримувати доступ до єдиного екземпляру. *Instance* — це операція класу;
- може нести відповідальність за створення власного унікального екземпляру.

Відносини

Клієнти отримують доступ до єдиного об'єкта класу *Singleton* лише через його операцію *Instance*.

```
public class Singleton {

  private static Singleton instance;

  private Singleton (){
  }

  public static Singleton getInstance(){
          if (instance == null){
                  instance = new Singleton();
          }
          return instance;
  }

}
```

Шаблон Builder

Будівник (англ. *Builder*) — шаблон проектування, відноситься до класу твірних шаблонів.

Призначення

Відокремлює конструювання складного об'єкта від його подання, таким чином у результаті одного й того ж процесу конструювання можуть бути отримані різні подання.

Мотивація

Застосування

Слід використовувати шаблон *Будівник* коли:

- алгоритм створення складного об'єкта не повинен залежати від того, з яких частин складається об'єкт та як вони стикуються поміж собою;
- процес конструювання повинен забезпечити різні подання об'єкта, що конструюється.

Структура

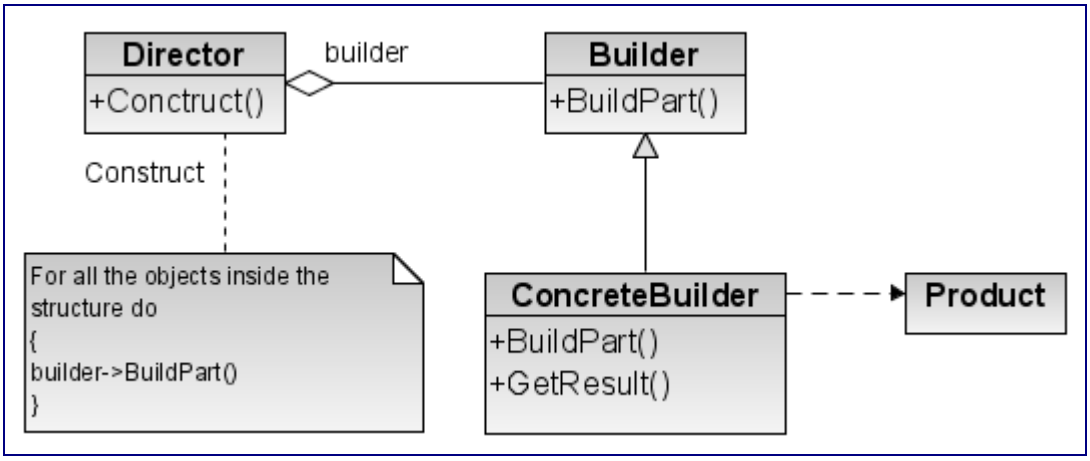


Рис. 15.2. UML діаграма, що описує структуру шаблону проектування *Будівник*

- Builder — будівник: визначає абстрактний інтерфейс для створення частин об'єкта *Product*;
- ConcreteBuilder — конкретний будівник: конструює та збирає докупи частини продукту шляхом реалізації інтерфейсу *Builder*; визначає подання, що створюється, та слідкує на ним; надає інтерфейс для доступу до продукту;
- Director — управитель: конструює об'єкт, користуючись інтерфейсом *Builder*;
- Product — продукт: подає складний конструйований об'єкт. *ConcreteBuilder* будує внутрішнє подання продукту та визначає процес його зборки; вносить класи, що визначають складені частини, у тому числі інтерфейси для зборки кінцевого результату з частин.

Відносини

- *клієнт* створює об'єкт-управитель *Director* та конфігурує його потрібним об'єктом-будівником *Builder*;
- *управитель* повідомляє *будівника* про те, що потрібно побудувати наступну частину *продукту*;
- *будівник* оброблює запити управителя та додає нові частини до *продукту*;
- *клієнт* забирає продукт у *будівника*.

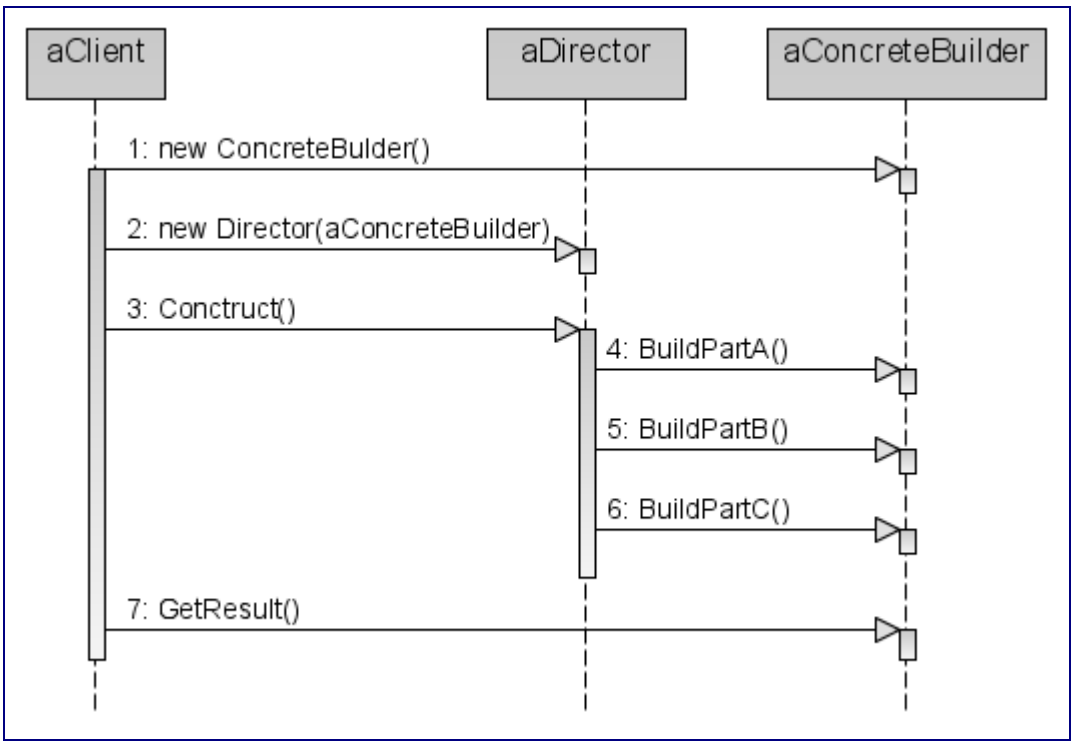


Рис. 15.3

```
      /** "Product" */
class Pizza {
   private String dough = "";
   private String sauce = "";
   private String topping = "";

   public void setDough(String dough)     { this.dough = dough; }
   public void setSauce(String sauce)     { this.sauce = sauce; }
   public void setTopping(String topping) { this.topping = topping; }
}


/** "Abstract Builder" */
abstract class PizzaBuilder {
   protected Pizza pizza;

   public Pizza getPizza() { return pizza; }
   public void createNewPizzaProduct() { pizza = new Pizza(); }

   public abstract void buildDough();
   public abstract void buildSauce();
   public abstract void buildTopping();
}

/** "ConcreteBuilder" */
class HawaiianPizzaBuilder extends PizzaBuilder {
   public void buildDough()   { pizza.setDough("cross"); }
   public void buildSauce()   { pizza.setSauce("mild"); }
   public void buildTopping() { pizza.setTopping("ham+pineapple"); }
}

/** "ConcreteBuilder" */
class SpicyPizzaBuilder extends PizzaBuilder {
   public void buildDough()   { pizza.setDough("pan baked"); }
   public void buildSauce()   { pizza.setSauce("hot"); }
```

```
   public void buildTopping() { pizza.setTopping("pepperoni+salami"); }
}


/** "Director" */
class Waiter {
   private PizzaBuilder pizzaBuilder;

   public void setPizzaBuilder(PizzaBuilder pb) { pizzaBuilder = pb; }
   public Pizza getPizza() { return pizzaBuilder.getPizza(); }

   public void constructPizza() {
      pizzaBuilder.createNewPizzaProduct();
      pizzaBuilder.buildDough();
      pizzaBuilder.buildSauce();
      pizzaBuilder.buildTopping();
   }
}


/** A customer ordering a pizza. */
class BuilderExample {
   public static void main(String[] args) {
       Waiter waiter = new Waiter();
       PizzaBuilder hawaiianPizzaBuilder = new HawaiianPizzaBuilder();
       waiter.setPizzaBuilder(hawaiianPizzaBuilder);
       waiter.constructPizza();

       Pizza pizza = waiter.getPizza();
   }
}
```

Розділ 4. Проектування інтерфейсу користувача.

Лекція 16. Графічний інтерфейс користувача.

- Призначення та типи інтерфейсів користувача.
- Принципи побудови графічного інтерфейсу користувача. [3, с. 46-88]

Призначення та типи інтерфейсів користувача

Призначення інтерфейсів користувача

Інтерфе́йс користувача́ (англ. *User Interface, UI*, дружній інтерфейс) — засіб зручної взаємодії користувача з інформаційною системою.

Інтерфейс користувача — сукупність засобів для обробки та відображення інформації, максимально пристосованих для зручності користувача; у графічних системах інтерфейс користувача реалізовується багатовіконним режимом, змінами кольору, розміру, видимості (прозорість, напівпрозорість, невидимість) вікон, їхнім розташуванням, сортуванням елементів вікон, гнучкими налаштовуваннями як самих вікон, так і окремих їхніх елементів (файли, папки, ярлики, шрифти тощо), доступністю багатокористувацьких налаштувань.

Графі́чний інтерфе́йс кори́стувача (ГІК, англ. *GUI, Graphical user interface*) — інтерфейс між комп'ютером і його користувачем, що використовує піктограми, меню, і вказівний засіб для вибору функцій та виконання команд. Зазвичай, можливе відкриття більше, ніж одного вікна на одному екрані.

ГІК — система засобів для взаємодії користувача з комп'ютером, заснована на представленні всіх доступних користувачеві системних об'єктів і функцій у вигляді графічних компонентів екрану (вікон, значків, меню, кнопок, списків і т. п.). При цьому, на відміну від інтерфейса командного рядка, користувач має довільний доступ (за допомогою клавіатури або пристрою координатного введення типу «миша») до всіх видимих екранних об'єктів.

Вперше концепція ГІК була запропонована вченими з дослідницької лабораторії Xerox PARC в 1970-х, але отримала комерційне втілення лише в продуктах корпорації Apple Computer. У операційній системі AMIGAOS ГІК з багатозадачністю був використаний в 1985 р. В наш час[*Коли?*] ГІК є стандартний складовий більшості доступних на ринку операційних систем і застосунків.

Приклади операційних систем, що використовують ГІК: Mac OS, Ubuntu, Microsoft Windows, NEXTSTEP, OS/2.

Типи інтерфейсів користувача

Для реалізації графічного інтерфейсу (GUI) в Java існують два основні пакети класів[9]:

- Abstract Window Toolkit (AWT)

- Swing

Перевагами першого є простота використання, інтерфейс подібний до інтерфейсу операційної системи та дещо краща швидкодія, оскільки базується на засобах ОС, щоправда має обмежений набір графічних елементів. Другий пакет Swing реалізує власний Java інтерфейс. Цей пакет створювався на основі AWT, і має набагато більше можливостей та більшу кількість графічних елементів. [10]. Swing-компоненти ще називають *полегшеними* (англ. *lightweight*), оскільки вони написані повністю на Java і, через це, платформонезалежні.

Існують також сторонні пакети, найпопулярнішим є Standard Widget Toolkit (SWT, вимовляється «ес-дабл-ю-ті») — Стандартний інструментарій віджетів. Розроблений підрозділом Rational фірми IBM і компанією Object Technology International (OTI), зараз розвивається фондом Eclipse.

Abstract Window Toolkit (AWT — абстрактний віконний інтерфейс) — це оригінальний пакет класів мови програмування Java, що слугує для створення графічного інтерфейсу користувача (GUI). AWT є частиною Java Foundation Classes (JFC) — стандартного API для реалізації графічного інтерфейсу для Java-програми. Пакет містить платформо-незалежні елементи графічного інтерфейсу, щоправда їхній вигляд залежить від конкретної системи.

AWT визначає базовий набір елементів керування, вікон та діалогів, які підтримують придатний, простий до використання, але обмежений у можливостях графічний інтерфейс. Однією з причин обмеженості AWT є те, що AWT перетворює свої візуальні компоненти у відповідні їм еквіваленти, платформи на якій встановлена віртуальна машина Java. Це означає, що зовнішній вигляд компонентів визначається платформою, а не закладається в Java. Оскільки компоненти AWT використовують «рідні» ресурси коду, вони називаються ваговитими (англ. *highweigh*).

Використання «рідних» рівноправних комонентів породжує деякі проблеми. По-перше, у зв'язку із різницею, що існує між операційними системами, компонент може виглядати або навіть вести себе по-різному на різноманітних платформах. Така мінливість суперечила філософії Java: «написане один раз, працює скрізь». По-друге, зовнішній вигляд кожного компонента був фіксованим (оскільки усе залежало від платформи), і це неможливо було змінити (принаймні, це важко було зробити). У зв'язку з цим в AWT на різних платформах виникали різні помилки і програмісту доводились перевіряти пряцездатність програм на кожній платформі окремо[1].

Незабаром після появи початкової версії Java, стало очевидним, що обмеження, властиві AWT, були настільки незручними, що потрібно було знайти кращий підхід. в результаті з'явились класи Swing як частина бібліотеки базових класів Java (JFC). В 1997 році вони були включені до Java 1.1 у вигляді окремої бібліотеки. А починаючи з версії Java 1.2, класи Swing (а також усі останні, що входили до JFC) стали повністю інтегрованимими у Java. Щоправда графічні класи AWT до сих пір використовується при написанні невеликих програм та аплетів. Крім того Swing хоч і надає більше можливостей з роботою з графікою,

проте не заміняє їх повністю. Так, наприклад, обробка подій залишилась та ж сама[1].

Swing — інструментарій для створення графічного інтерфейсу користувача (GUI) мовою програмування Java. Це частина бібліотеки базових класів Java (*JFC*, Java Foundation Classes).

Swing розробляли для забезпечення функціональнішого набору програмних компонентів для створення графічного інтерфейсу користувача, ніж у ранішого інструментарію AWT. Компоненти Swing підтримують специфічні *look-and-feel* модулі, що динамічно підключаються. Завдяки ним можлива емуляція графічного інтерфейсу платформи (тобто до компоненту можна динамічно підключити інші, специфічні для даної операційної системи вигляд і поведінку). Основним недоліком таких компонентів є відносно повільна робота, хоча останнім часом це не вдалося підтвердити через зростання потужності персональних комп'ютерів. Позитивна сторона —— універсальність інтерфейсу створених програм на всіх платформах.

Історія графі́чного інтерфе́йса кори́стувача

На початку існування Java класів Swing не було взагалі. Через слабкі місця в AWT (початковій GUI системі Java) було створено Swing. AWT визначає базовий набір елементів керування, вікон та діалогів, які підтримують придатний до використання, але обмежений у можливостях графічний інтерфейс. Однією з причин обмеженості AWT є те, що AWT перетворює свої візуальні компоненти у відповідні їм еквіваленти, що не залежать від платформи, які називаються рівноправними компонентами. Це означає, що зовнішній вигляд компонентів визначається платформою, а не закладається в Java. Оскільки компоненти AWT використовують «рідні» ресурси коду, вони називаються ваговитими (англ. *highweigh*).

Використання «рідних» рівноправних компонентів породжує деякі проблеми. По-перше, у зв'язку із різницею, що існує між операційними системами, компонент може виглядати або навіть вести себе по-різному на різноманітних платформах. Така мінливість суперечила філософії Java: «написане один раз, працює скрізь». По-друге, зовнішній вигляд кожного компонента був фіксованим (оскільки усе залежало від платформи), і це неможливо було змінити (принаймні, це важко було зробити). По-третє, використання ваговитих компонентів тягнуло за собою появу нових обмежень. Наприклад, ваговитий компонент завжди має прямокутну форму і є непрозорим.

Незабаром після появи початкової версії Java, стало очевидним, що обмеження, властиві AWT, були настільки незручними, що потрібно було знайти кращий підхід. в результаті з'явились класи Swing як частина бібліотеки базових класів Java (JFC). В 1997 році вони були включені до Java 1.1 у вигляді окремої бібліотеки. А починаючи з версії Java 1.2, класи Swing (а також усі останні, що входили до JFC) стали повністю інтегрованимими у Java.

Архітектура графі́чного інтерфе́йса кори́стувача

- Незалежність від платформи: Swing — платформо-незалежна бібліотека, що означає, що програму з використанням Swing можна запустити на всіх платформах, які підтримують JVM.
- Можливість для розширення: Swing — дуже розподілена архітектура, яка дозволяє «підключати» реалізації користувача вказаної інфраструктури інтерфейсів: користувачі можуть створити свою власну реалізацію цих компонентів, щоб замінити компоненти без обумовлення (*за замовчуванням*). Взагалі, користувачі Swing можуть розширити структуру, продовжуючи (з допомогою *extends*) існуючі класи і/або створюючи альтернативні реалізації основних компонентів.

Принципи розробки графічного інтерфейсу користувача

Програмне забезпечення має розроблятися з урахуванням вимог і побажань користувача - система повинна підлаштовуватися до користувача. Користувачу комп'ютера потрібно надати вдалий досвід, який вселить їм впевненість у своїх силах і зміцнить високу самооцінку при роботі з комп'ютером. Їх дії з комп'ютером можуть бути охарактеризовані як "успіх породжує успіх. Добре продуманий інтерфейс, подібно доброму вчителю, забезпечує плідну взаємодію користувача і комп'ютера. Вдалі інтерфейси навіть здатні допомогти людині вийти зі звичного кола програм, якими він користується, і відкрити нові, поглибити розуміння роботи інтерфейсів і комп'ютерів.

Три принципи розробки користувальницького інтерфейсу формулюються так:

1) контроль користувачем інтерфейсу;
2) зменшення завантаження пам'яті користувача;
3) послідовність користувальницького інтерфейсу.

Правило 1: дати контроль користувачеві

Досвідчені проектувальники дозволяють користувачам вирішувати деякі завдання на власний розсуд. Досвідчені архітектори по завершенні будівництва складного комплексу будівель повинні прокласти між ними доріжки для пішоходів. Поки вони не знають, в якому саме місці люди будуть перетинати майданчика. Тому доріжки ніколи не прокладають одночасно із зведенням будівель. Через деякий час будівельники повертаються і тільки тепер, згідно з "волевиявлення" населення, заливають протоптані доріжки асфальтом.

Принципи, які дають користувачеві контроль над системою:

1) використовувати режими розсудливо;
2) надати користувачеві можливість вибирати: працювати або мишею, або клавіатурою, або їх комбінацією;
3) дозволити користувачеві сфокусувати увагу;
4) демонструвати повідомлення, які допоможуть йому в роботі;
5) створити умови для негайних і оборотних дій, а також зворотного зв'язку;
6) забезпечити відповідні шляхи і виходи;
7) пристосовуєте систему до користувачів з різним рівнем підготовки;

8) зробити користувальницький інтерфейс більш зрозумілим;

9) дати користувачеві можливість налаштовувати інтерфейс за своїм смаком;

10) дозволити користувачеві безпосередньо маніпулювати об'єктами інтерфейсу;

Використовувати режими розсудливо

Треба дозволити людині самій вибирати потрібні йому режими. Інтерфейс повинен бути настільки природним, щоб користувачеві було комфортно працювати з ними. Користувач не думає про перемикання в режим вставка або перезапису при роботі в текстовому процесорі - це цілком раціонально і природно.

Дозволити людині використовувати мишу і клавіатуру

Можливість роботи з клавіатурою використання клавіатури замість миші. Це значить, що користувачеві буде легше працювати, просто він або не може нею користуватися, або її у нього немає. Конфіденційність створені, щоб прискорити роботу при використанні миші. Однак при роботі з клавіатурою до них не можна добратися - для подібних випадків передбачені "випадають" меню.

Дозволити користувачеві сфокусувати увагу

Не змушувати користувачів закінчувати започаткованих послідовностей дій. Дати їм вибір - анулювати або зберегти дані і повернуться туди, де вони урвалися. Нехай у користувачів залишиться можливість контролювати процес роботи в програмі.

Показувати пояснювальні повідомлення і тексти

У всьому інтерфейсі використовувати зрозумілі для користувача терміни. Вони не зобов'язані знати про бітах і байтах!

Слід вибрати правильний тон у повідомленнях і запрошеннях. не менш важливо застрахуватися від проблем і помилок. Невдала термінологія і не правильний тон приведуть до того, що користувачі будуть звинувачувати себе в виникаючі помилки.

Забезпечити негайні і оборотні дії і зворотний зв'язок

Кожен програмний продукт повинен включати в себе функції UNDO і REDO. Необхідно інформувати користувача про те, що дана дія не може бути скасовано, і по можливості дозволити йому альтернативне дію. Постійно тримати людину в курсі, що відбувається в даний момент.

Надавати зрозумілі шляхи і виходи

Користувачі повинні отримувати задоволення при роботі з інтерфейсом будь-якого програмного продукту. Навіть інтерфейси, застосовувані в індустрії, не повинні лякати користувача, він не повинен бояться натискатися натискати кнопки або переходити на інший екран. Вторгнення Internet показало, що навігація - основна інтерактивна техніка в Internet. Якщо користувач розуміє, як зайти на потрібну

сторінку в WWW, то існує 80-відсоткова ймовірність, що він розбереться в інтерфейсі. Люди опановують методи роботи з браузером дуже швидко.

Пристосовуватися до користувачів з різними рівнями навичок

Не "жертвувати" досвідченими користувачами на благо звичайних. Треба передбачити для них швидкий доступ до функцій програми. Не втомлювати їх проходженням численних кроків для виконання будь-які дії, якщо вони звикли користуватися однією макрокоманд.

Зробити користувальницький інтерфейс "прозорим"

Інтерфейс користувача - "міфічна" частина програмного продукту. При хорошому проекті користувачі навіть не відчувають його "присутності". Якщо він розроблений невдало, користувачам доведеться докласти навіть чимало зусиль для ефективного використання програмного продукту. Завдання інтерфейсу - допомогти людям відчути себе відданих як би "всередині" комп'ютера, вільно маніпулювати і працювати з об'єктами. Це і називається "прозорим" інтерфейсом!

"Прозорість" інтерфейсу забезпечується тим, що людині буде надана можливість користуватися об'єктами, відмінними від системних команд.

Дати користувачу можливість налаштувати інтерфейс на свій смак

Секрет "прозорого" інтерфейсу - в прямому зв'язку з ментальною моделлю. Користувач повинен бути зосереджений безпосередньо на виконанні завдань, що стоять перед ним, а не розбиратися у функціях програми.

Дозволити користувачеві пряме маніпулювання об'єктами інтерфейсу

Користувач починає сумніватися у власних силах, якщо прямі маніпуляції з об'єктами не відповідають їх ментальної моделі і системі уявлень про взаємодію з реальним світом. Просте правило: збільшувати метафоричність, але не ламати її. Іноді система прямих маніпуляцій терпить крах, якщо користувач не знає, що треба взяти і куди це помістити. Об'єкти повинні "кричати" людині користувачу: "схопи мене, відпусти мене, звертайся зі мною, як з предметом, який я представляю!". Інакше людина не зрозуміє, як працювати з цим об'єктом. Користувачі повинні відчувати себе комфортно при виробництві даної операції і знати про передбачуване результаті. Крім того, необхідно, щоб інтерфейс можна було легко вивчити.

Дозволити користувачеві думати, що він контролює ситуацію

Добре розроблений інтерфейс повинен бути зручний для користувачів і розважати їх, поки комп'ютер знаходиться в стані завантаження.Людям не подобається сидіти біля комп'ютера, нічого не роблячи, поки комп'ютер зайнятий "своїми справами". Якщо не можна дати користувачеві контроль, то необхідно створити його ілюзію!

Правило 2: зменшити навантаження на користувача

Заснована на знанні того, як люди зберігають і запам'ятовують інформацію, сила комп'ютерного інтерфейсу повинна захистити пам'ять людей від надмірної завантаженості.

Принципи, що дозволяють знизити навантаження на пам'ять користувача:

1) не завантажувати короткочасну пам'ять;
2) покладатися на розпізнавання, а не на повторення;
3) представити візуальні заставки;
4) передбачити установки за замовчуванням, команди Undo і Rendo;
5) передбачити "швидкі" шляху;
6) активувати синтаксис дій з об'єктами;
7) використовувати метафори з реального світу;
8) застосовувати розкриття і пояснення понять і дій;
9) збільшити візуальну ясність.

Не навантажувати короткочасну пам'ять

Не змушувати користувачів запам'ятовувати і повторювати те, що може (і повинен) робити комп'ютер. Наприклад, коли необхідно заповнити анкету, буде потрібно ввести деякі дані - ім'я, адресу, телефонний номер, які фіксуються системою для подальшого використання, при повторному вході користувача в систему або відкриття запису. Система повинна "запам'ятовувати" введену інформацію і забезпечити безперешкодний доступ до неї в будь-який час.

Покладатися на розпізнавання, а не на повторення

Передбачити списки і меню, що містять об'єкти або документи, які можна вибрати, не змушуючи користувачів вводити інформацію вручну без підтримки системи. Чому люди повинні запам'ятовувати, наприклад, абревіатуру з двох літер для кожного штату США, коли вони заповнюють яку-небудь анкету або форму? Не треба змушувати їх запам'ятовувати коди для подальшого використання. Передбачити списки найбільш популярних об'єктів і документів, які можна просто вибрати без заповнення командних рядків і ін

Забезпечити візуальні підказки

Коли користувачі знаходяться в якомусь режимі або працюють мишею, це повинно вплинути на екрані. Індикація повинна повідомляти користувачу про режим, в якому він знаходиться. Форма курсора може змінюватися для вказівки поточного режиму або дії, а індикатор - включатися або відключатися. Тест на візуальну інформативність продукту: відійти від комп'ютера під час виконання завдання і пізніше повернуться до роботи. Звернути увагу на візуальні підказки інтерфейсу, які повинні інформувати про те, з чим ви працювали, де знаходилися і що робили.

Передбачити функції скасування останньої дії, його повтору, а також установки за замовчуванням

Використовувати здатність комп'ютера зберігати і відшукувати інформацію про вибір користувача, а також про властивості системи.Передбачити багаторівневі системи відміни і повтору команд, що забезпечують впевнену і спокійну роботу з програмою.

Забезпечити ярлики для інтерфейсу

Як тільки користувачі достатньо добре освоять програмний продукт, вони починають відчувати потребу в прискорювачах. Не ігнорувати цю необхідність, однак при розробці слідувати стандартам.

Активізувати синтаксис дій з об'єктами

Об'єктно-орієнтований синтаксис дозволяє людині зрозуміти взаємозв'язок між об'єктами та діями в програмному продукті. Користувачі можуть вивчати і "перегортати" інтерфейс, вибираючи об'єкти і переглядаючи доступні дії.

Використовувати метафори реального світу

Бути обережним при виборі і використанні метафор для інтерфейсу. Вибравши метафору, зафіксувати її і слідувати їй неукоснітельно. Якщо вийде, що метафора не відповідає своєму призначенню в усьому інтерфейсі, то вибрати нову. Продовжувати метафору, не перериваючи її.

Пояснювати поняття і дії

Ніколи не забувати про легкий доступ до часто використовуваних функцій і дій. Приховати непопулярні властивості та функції і дозволити користувачеві викликати їх у міру необхідності. Не треба намагатися відобразити всю інформацію в головному вікні. Використовувати вторинні вікна.

Збільшити візуальну ясність

Комп'ютерні графіки і оформлювачі книг добре освоїли мистецтво подання інформації. Цей навик повинні мати і розробники користувальницького інтерфейсу.

Правило 3: зробити інтерфейс сумісним

Сумісність - ключовий аспект для використання інтерфейсу. Однак не слід будь-що-будь прагнути до неї. Одним з основних перевагпослідовності є те, що користувачі можуть перенести свої знання та навички з старої програми, якою вони користувалися раніше, в нову.

Принципи створення сумісності інтерфейсу:

1) проектування послідовного інтерфейсу;
2) загальна сумісність всіх програм;
3) збереження результатів взаємодії;
4) естетична привабливість і цілісність;
5) заохочення вивчення;

Проектування послідовного інтерфейсу

Користувачі повинні мати опорні точки при переміщенні в інтерфейсі. Це заголовки вікон, навігаційні карти і деревоподібні структури.Інша візуальна допомогу надає негайний, динамічний огляд місця розташування. Користувач також повинен мати можливість завершити поставлену задачу без зміни умов роботи або перемикання між стилями введення інформації. Якщо спочатку він використовував клавіатуру, то повинна бути забезпечена можливість завершити роботу теж з нею як з головним інструментом для взаємодії.

Загальна сумісність всіх програм

Вивчення однієї програми не має кардинально відрізнятися від вивчення подібної програми. Коли схожі об'єкти не працюють однаково в різних ситуаціях, у користувачів відбувається негативне. Це гальмує вивчення програми і призводить до того, що користувач втрачає впевненість у своїх силах.

Поліпшення інтерфейсу і послідовності

Проектувальники програм повинні бути обізнані у застосуванні отриманих навичок і обережні при введенні нових. Якщо поліпшується інтерфейс, то користувач повинен вивчити лише кілька нових прийомів взаємодії. Не змушувати його переучуватися і забувати багаторічні навички. "Придушити" наявні навички набагато важче, ніж придбати нові.

Збереження результатів взаємодії

Якщо результати можуть бути відмінні від того, що очікує користувач, то інформувати його перед виконанням дії. Дати йому опції виконання дії, можливість скасувати дію або зробити інше.

Естетична привабливість і цілісність

Прийнятий для погляду інтерфейс не повинен приховувати недолік функціональності програмного продукту. Користувачі не повинні бачити "губної помади на бульдога", вони повинні отримати красивий інтерфейс, який допоможе їм у роботі.

Заохочення вивчення

Інтерфейси сьогоднішнього і завтрашнього дня - більш інтуїтивні, передбачувані, дружні, привабливі. Нашестя CD/DVD-ROM продуктів і браузерів Internet, домашніх сторінок і прикладних програм відкрило цілий світ для користувачів комп'ютера. Настав час перетворення дружніх інтерфейсів в приємні у використанні і заманюють інтерфейси майже у всіх програмах.

Керівні принципи

Для чого потрібні керівні принципи

Інструкції - це правила та пояснення, призначені для того, щоб дотримуватися їх при створенні елементів інтерфейсу, їх поведінки і зовнішнього вигляду. Інструкції відносяться до елементів подання інформації та взаємодії.

Дотримання інструкцій з проектування без урахування побажань користувача звичайно призводить до появи невдалого інтерфейсу. Зручний і послідовний інтерфейс не буде створено, якщо сліпо слідувати інструкції без розуміння механізму взаємодії між собою. "Багато інструкції занадто багато уваги приділяють" розташуванню кнопок "і мало - розумінню і навчанню" [Гулд 13]. Вивчення посібників і інструкцій не є єдиним критерієм успіху. Вихід з цього положення є дотримання керівних принципів при створенні елементів інтерфейсу.

Керівні принципи – це принципи, які адресовані представникам всього "айсберга" проектування. Керівні принципи створення інтерфейсу, відображені в інструкціях, повинні не знижувати і обмежувати творчу активність, а дозволяти користувачеві застосовувати до інтерфейсу своє знання реального світу (наприклад, якщо користувач бачить на екрані групу кнопок, схожих на кнопки на панелі радіоприймача, він може і повинен застосувати своє знання функцій кнопок у реальному світі до комп'ютера).

Керівні принципи побудови інтерфейсу розраховані на сьогоднішні системи виведення та введення інформації. Вони зачіпають такі технології, як використання пера, писання від руки і голосове введення. Одна з проблем розробки інструкцій, що відповідають новим технологіям, - це розшифровка способів взаємодії користувача з системою, так як ступінь цієї взаємодії ще точно не визначена. Інструкції повинні базуватися на тому, як користувачі реагують на нововведення і створюватиметься за подію деякого часу, необхідного, щоб користувачі освоїли інтерфейс і склали певну думку про нього.

Нормативи

Керівні принципи містять характеристики стандартів презентацій, поведінки та взаємодії з елементами управління інтерфейсом.

У керівництві за елементами інтерфейсу і його органам управління сказано, коли їх потрібно виправити, як "подати" і якою має бути техніка роботи з ними (наприклад, клавіатурна або за допомогою миші). Повний набір посібників розкриває сутність кожного об'єкта і елемента інтерфейсу в термінах і способах подання на екрані, їхня поведінка, механізм взаємодії з ними користувачів.

Розвиток існуючих керівних принципів проектування інтерфейсу

Багато програмних продуктів створених для роботи на різних платформах. З тих пір, як ці платформи мають різні операційні системи, інструменти та стилі інтерфейсу, дуже складно розробляти інтерфейс, що задовольняє всі платформи або що працює на кожній з платформ.

Доповнення - добірка індустріальних посібників з проектування - було розроблено Беллкором [12]. Воно містить опис і керівні принципи для основних компаній і операційних систем, як IBM CUA, OSF, Microsoft Windows і ін.

Завдання керівних принципів з проектування однозначні: надати користувачам можливість доступу до інформації з будь-якого місця системи, в будь-якій формі,

створити такий інтерфейс, який допомагав би людям працювати і подобається їм. Добре розроблений інтерфейс дозволяє користувачам сфокусуватися на виконанні завдань, а не на особливостях програмного й апаратного забезпечення.

Застосування керівних принципів

Цілі і керівні принципи розробки інтерфейсу повинні бути реалістичними і доступними для користувачів. Специфіка того чи іншого бізнесу накладає обмеження на дане середовище. Дані керівні принципи по розробці інтерфейсу також повинні проходити тестування. Щоб продукт відповідав керівним принципам, необхідно мати підтримку з боку розробників. Відповідальність за розробку сумісного інтерфейсу лежить на проектувальниках та розробників.

Керівні принципи по розробці інтерфейсу на макро- і мікрорівні

При розробці керівних принципів по призначеному для користувача інтерфейсу необхідно переконатися, що й розробку користувальницького інтерфейсу, і зручність застосування аналізуються з двох точок зору - мікро-та макрорівня.

Керівні принципи на мікрорівні розглядають подання користувачам індивідуальних елементів інтерфейсу (керуючих - кнопок, полів для галочки, полів для тексту, лінійок для прокручування і т. Д.), а так само способи інтерактивного їх взаємодії.

Розробка інтерфейсу на макрорівні представляє собою шаблон для користувача інтерфейсу - продукт збирається весь цілком і його концепція стає зрозуміла користувачам у міру взаємодії з ним.

"Кількість керівних принципів збільшується пропорційно збільшенню кількості людей, залучених у створення і розробку, а також використання комп'ютерних систем. Постійно збільшується обсяг робіт з виробництва все більш кращих інструкцій показує, як важко створювати систему відповідно до інструкції. Проектування - це певна кількість компромісів, низка конфліктів між непоганими принципами. Все це важко вмістити в інструкції. " (Джон Гулд).

Приклад графі́чного інтерфе́йса кори́стувача

Наступний код демонструє основи використання Swing. Ця програма зображує вікно (JFrame), у вікно буде міститиметься кнопка з написом «Натисніть сюди» на ній та написом праворуч «Ця кнопка не робить нічого:».

```
package com.example;

// Імпортує swing і AWT класи
import java.awt.EventQueue;
import java.awt.FlowLayout;

import javax.swing.JButton;
import javax.swing.JFrame;
import javax.swing.JLabel;
import javax.swing.WindowConstants;

/**
```

```
 * Простий приклад використання Swing
 */
public class SwingExample {
    public static void main(String[] args) {

        // Упевнюємося, що всі виклики Swing/AWT виконуються Event Dispatch Thread ("EDT")
        EventQueue.invokeLater(new Runnable() {
            @Override
            public void run() {
                // Створюємо JFrame, що має вигляд вікна з "декораціями",
                // наприклад заголовком і кнопкою закриття
                JFrame f = new JFrame("Приклад вікна Swing");

                // Установлюємо простий менеджер розмітки, що впорядковує всі компоненти
                f.setLayout(new FlowLayout());

                // Додаємо компоненти
                f.add(new JLabel("Ця кнопка не робить нічого:"));
                f.add(new JButton("Натисніть сюди!"));

                // "Пакує" вікно, тобто робить його величину відповідну до її компонентів
                f.pack();

                // Встановлюємо стандартну операцію закриття для вікна,
                // без цього вікно не закриється після активування кнопки закриття
                //  (Стандартно HIDE_ON_CLOSE, що просто приховує вікно)
                f.setDefaultCloseOperation(WindowConstants.DISPOSE_ON_CLOSE);

                // Уснановлюємо видимість=істина, тим самим показуючи вікно на екрані
                f.setVisible(true);
            }
        });
    }
}
```