КОНСПЕКТ ЛЕКЦІЙ
(чернетка)

“ІНЖЕНЕРІЯ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ”

Київ - 2014

Зміст

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

Програ́ммное обеспе́чение (**ПЗ**) — все или часть программ, процедур, правил и соответствующей документации системы обработки информации (*ISO/IEC 2382-1: 1993. Information technology — Vocabulary — Part 1: Fundamental terms*).

Другие определения из международных и отечественных стандартов:

- Компьютерные программы, процедуры и, возможно, соответствующая документация и данные, относящиеся к функционированию компьютерной системы (FCD ISO/IEC 24765. Systems and Software Engineering Vocabulary)[6].
- **Совокупность программ системы обработки информации и программных документов, необходимых для эксплуатации этих программ (ГОСТ 19781-90).**

Инженерия ПЗ — это системный подход к анализу, проектированию, оценке, реализации, тестированию, обслуживанию и модернизации программного обеспечения, то есть применение инженерии к разработке программного обеспечения.

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

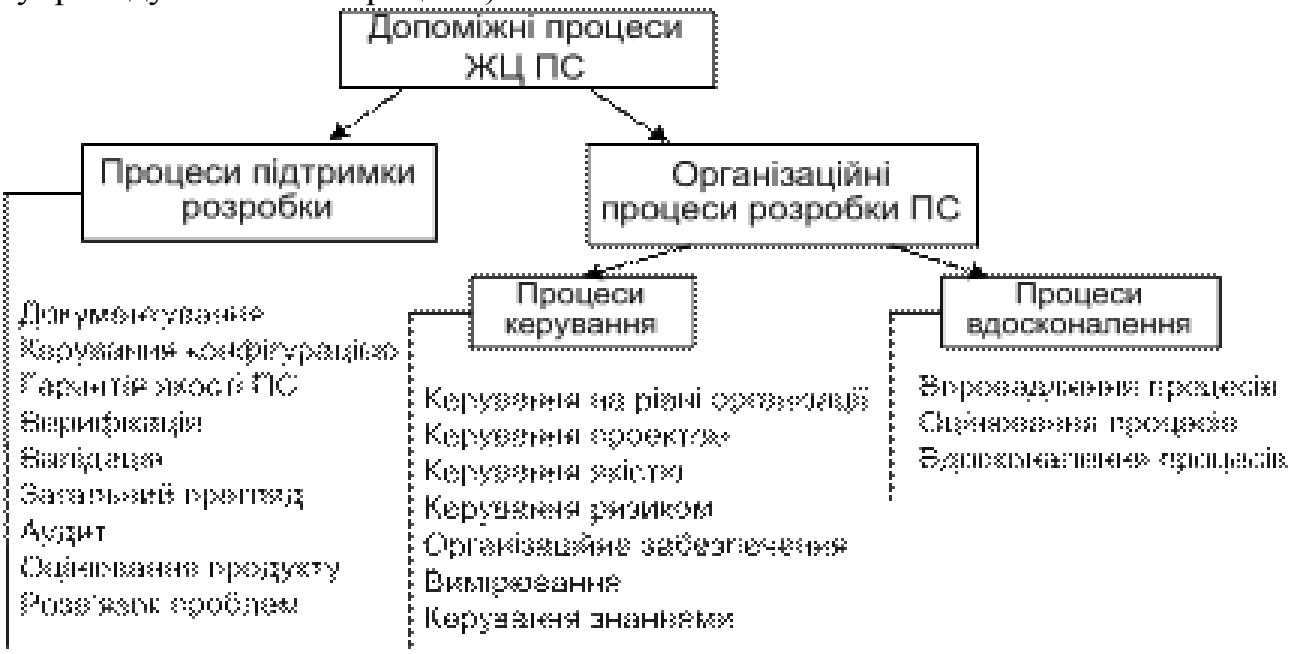
Стандарт ISO/IEC 12207:1995 "Information Technology - Software Life Cycle Processes" є основним нормативним документом, який регламентує склад процесів життєвого циклу ПЗ. Він визначає структуру життєвого циклу, що містить процеси, дії і завдання, які повинні бути виконані під час створення ПЗ.

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	• Рішення про участь в розробці • Комерційні пропозиції/ конкурсна заявка • Договір на поставку/ розробку • План управління проектом • Реалізація/корегування • Акт приймально-

		документація	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 тестування ПЗ • Інтеграція ІС і кваліфікаційне тестування ІС	• 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, ПЗ, документація на ІС, тести	• 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 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.**

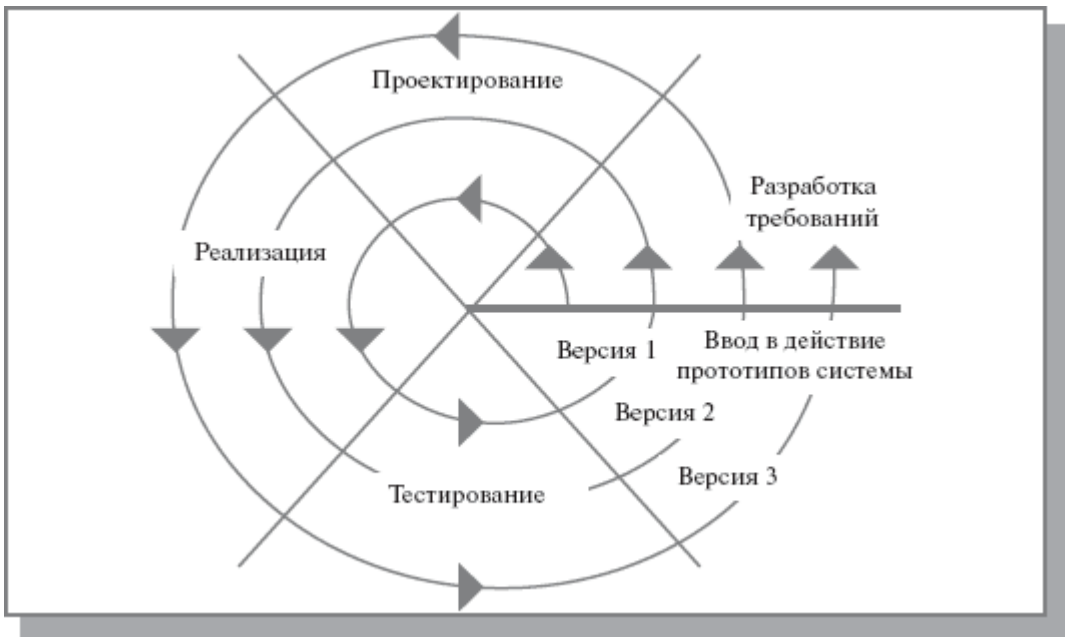


Рис. 2.3. Спіральна модель ЖЦ ІС

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

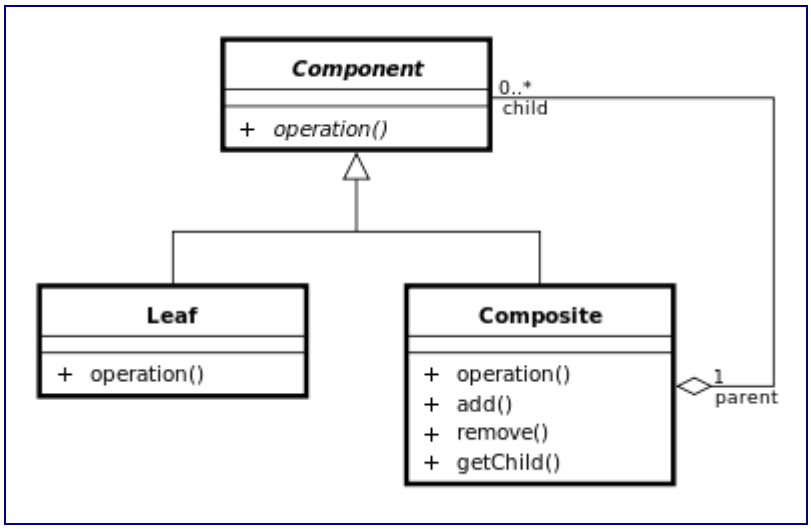


рис. 6.1

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.

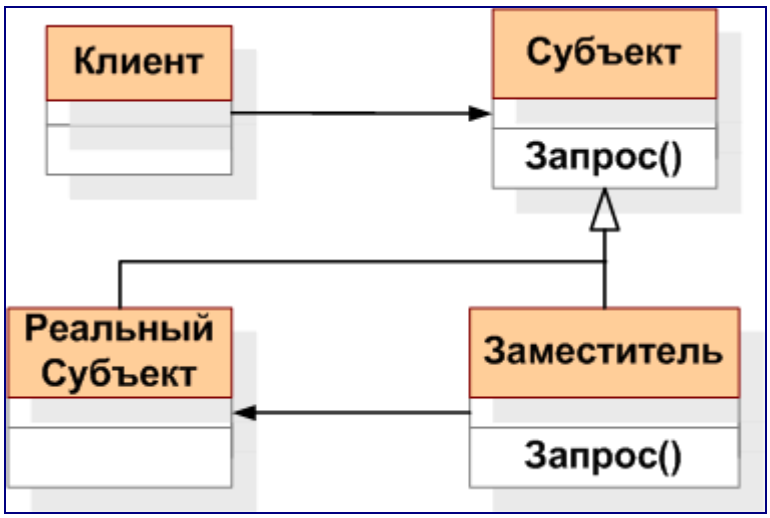


Рис. 7.1

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

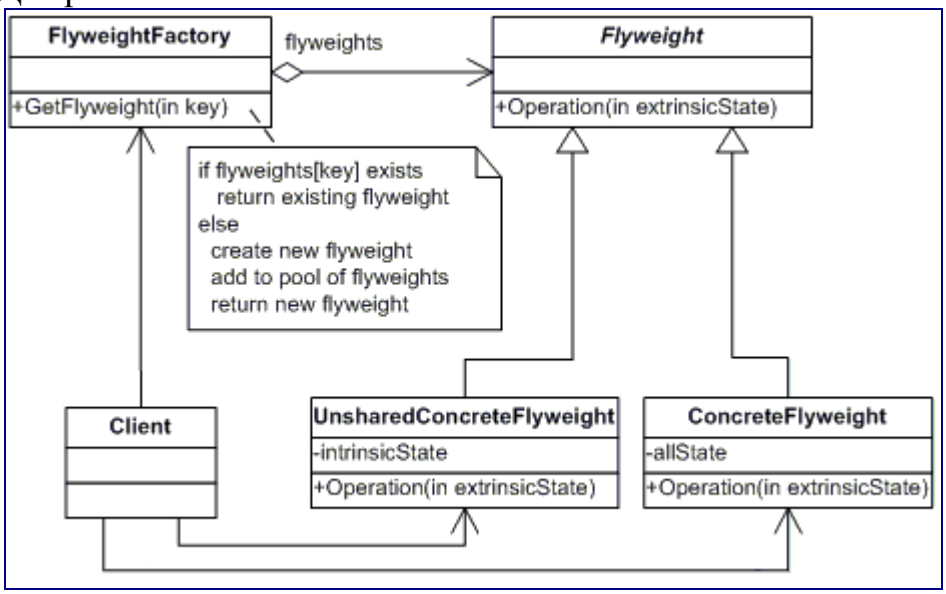


Рис. 7.2

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

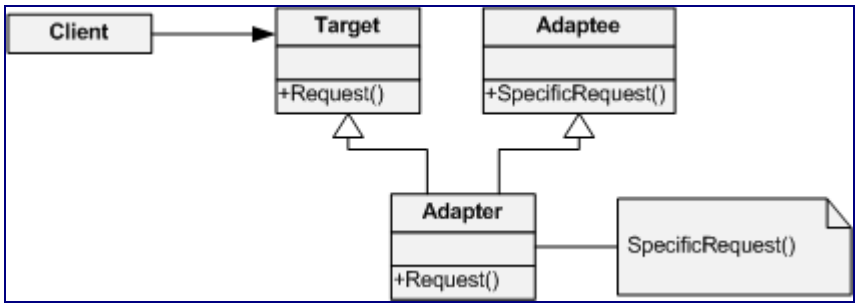


Рис. 8.1

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

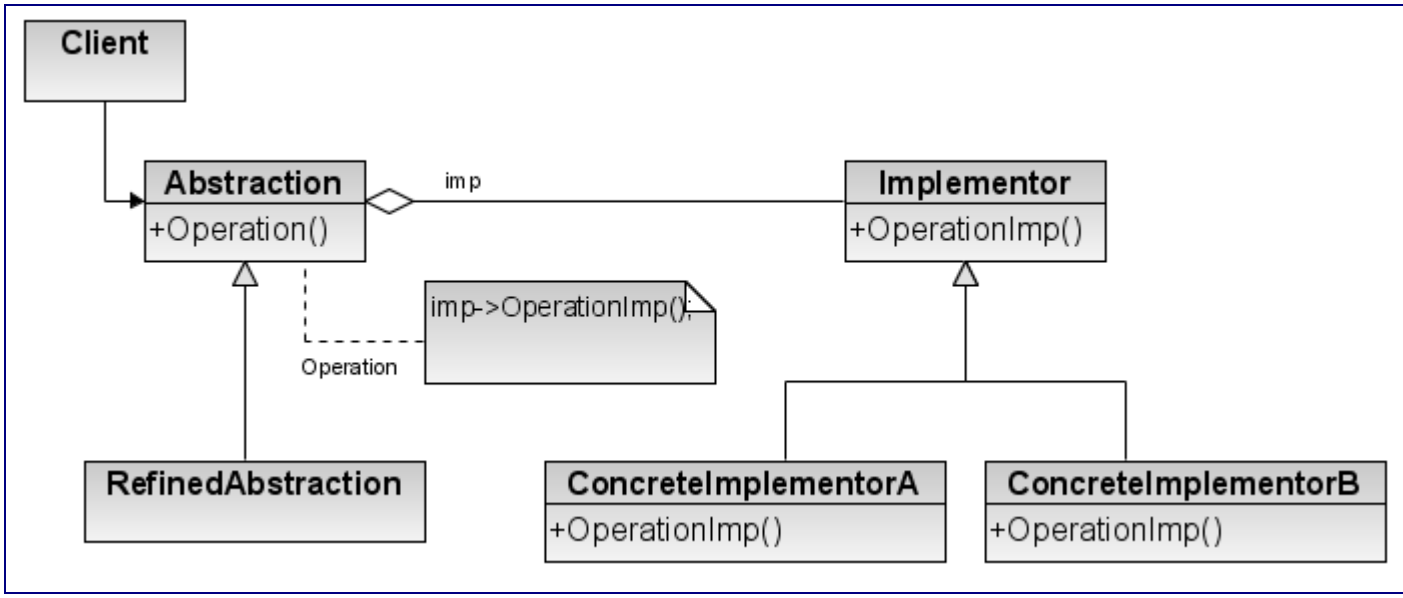


Рис. 8.2
UML діаграма, що описує структуру шаблону проектування *Micm*

- 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.

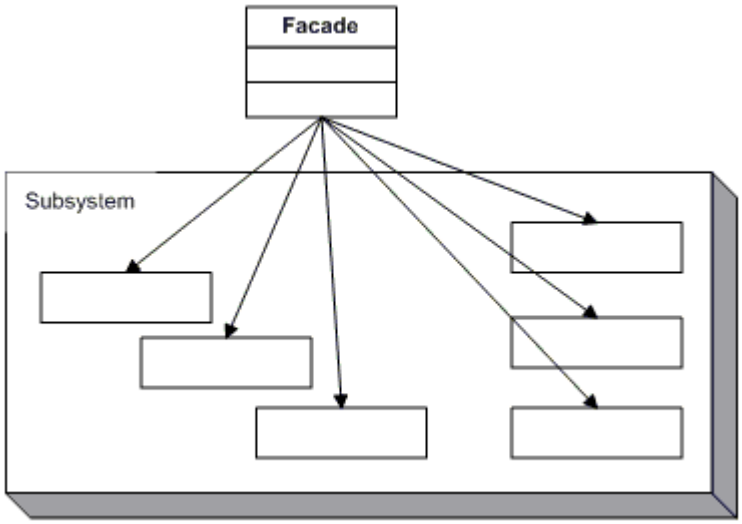


Рис. 8.3

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

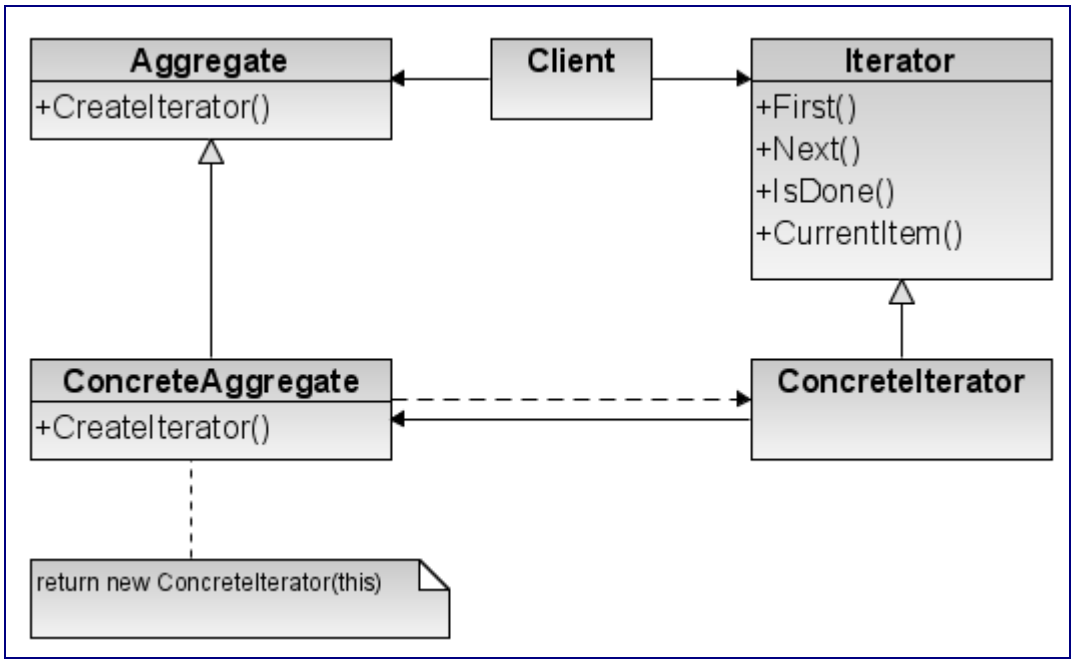


Рис. 9.1
UML діаграма, що описує структуру шаблону проектування Ітератор

- 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

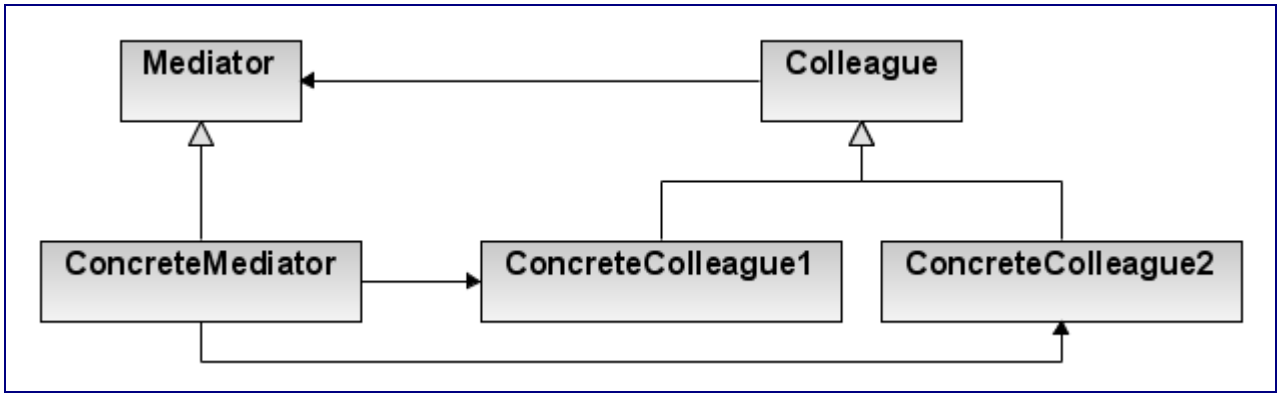


Рис. 9.2
UML діаграма, що описує структуру шаблону проектування *Посередник*

- 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

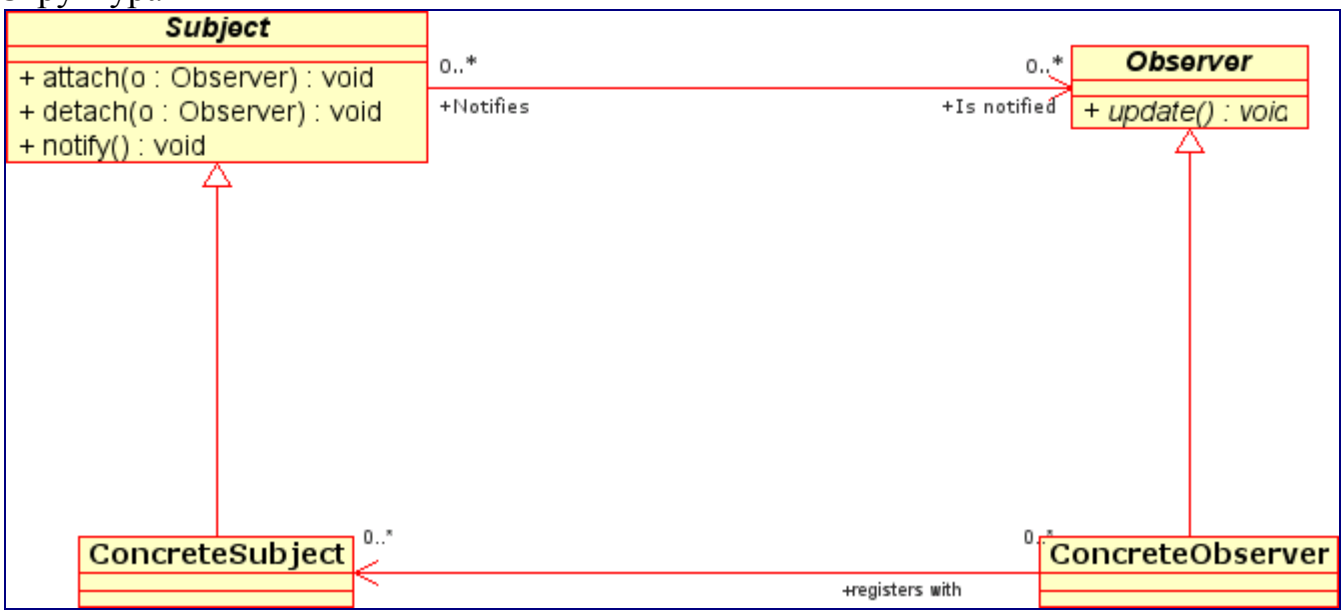


Рис. 10.1

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

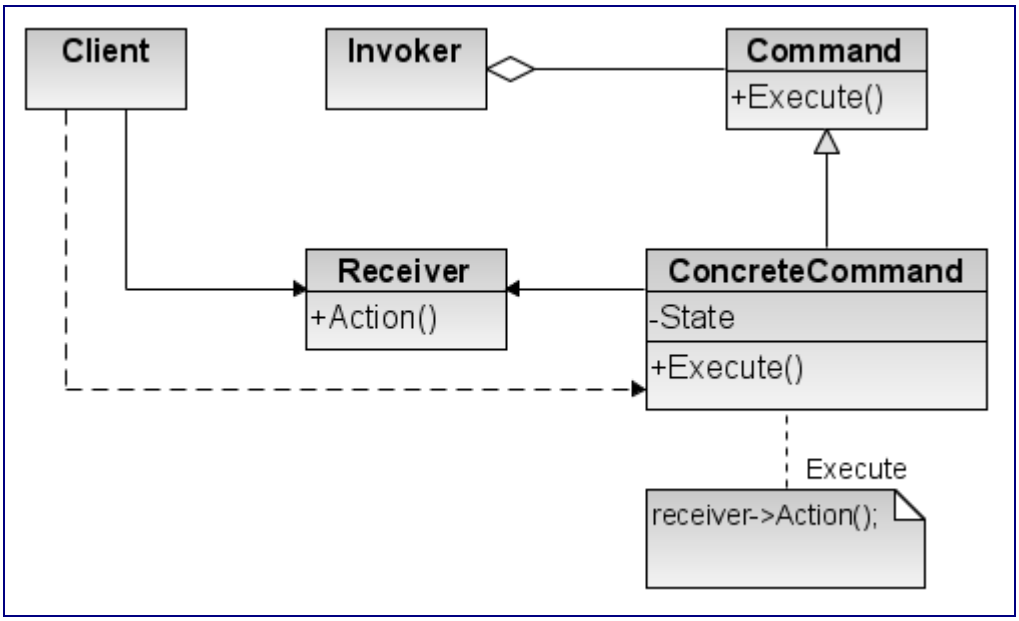


Рис. 11.1
UML діаграма, що описує структуру шаблону проектування *Команда*

- 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

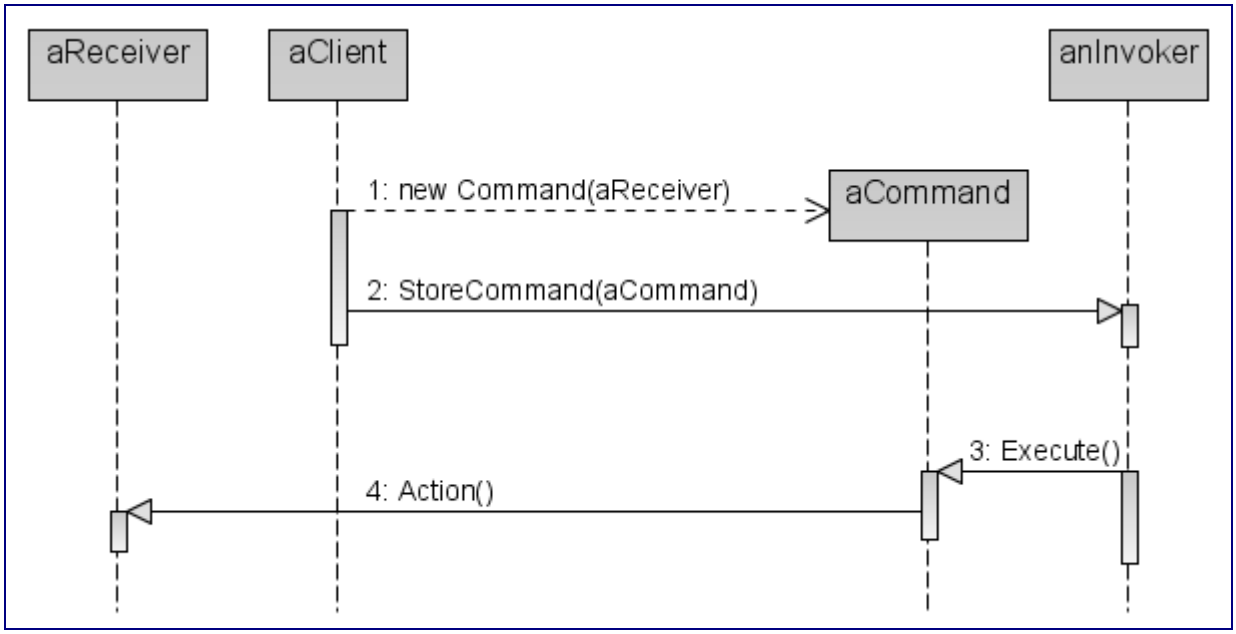


Рис. 11.2
UML діаграма, що описує взаємовідносини поміж об'єктів шаблону проектування *Команда*

- 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).

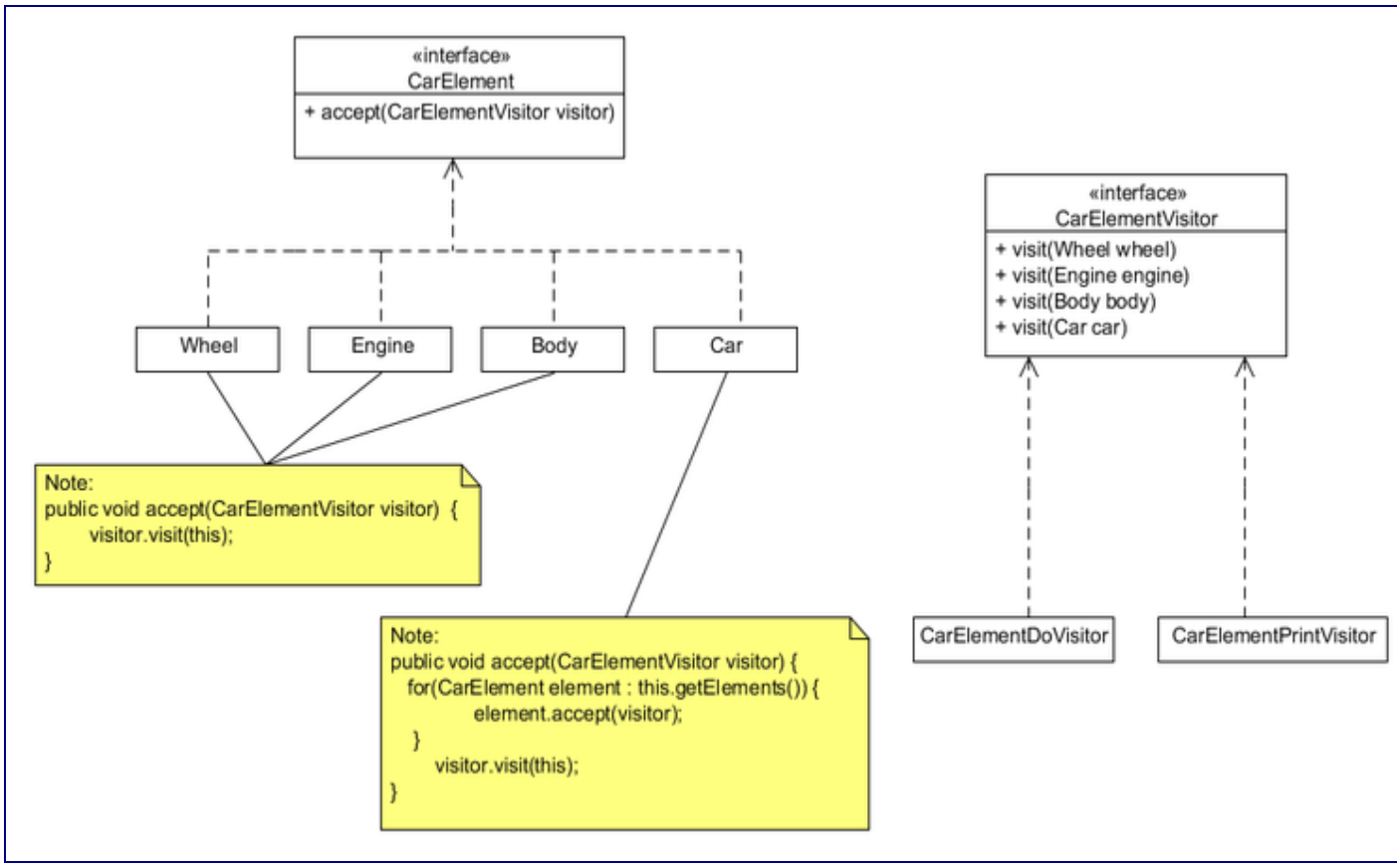


рис. 11.3

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

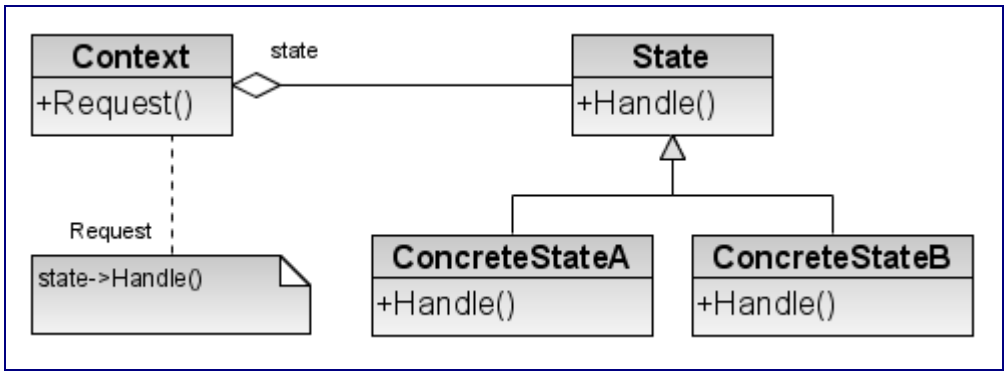


Рис. 12.1. UML діаграма, що описує структуру шаблону проектування *Стан*

- 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

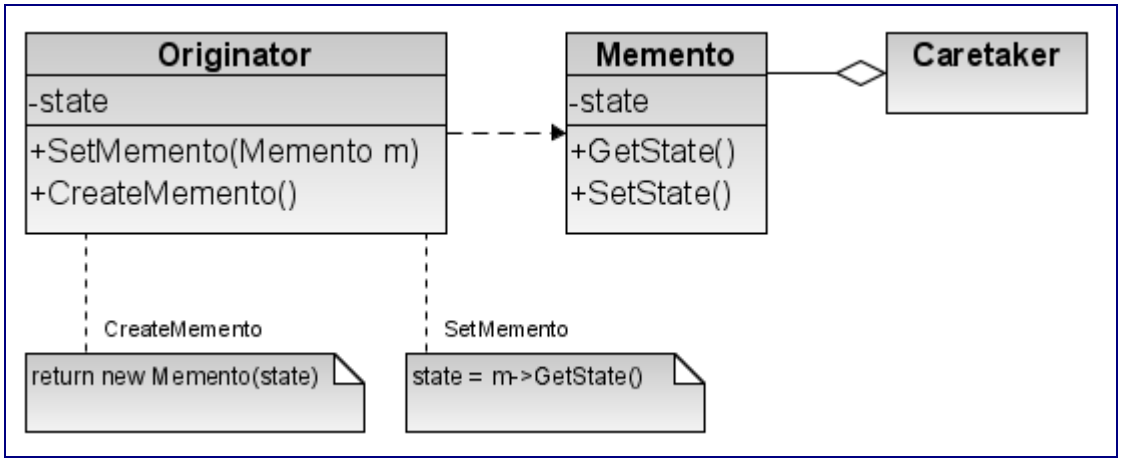


Рис.12.2. UML діаграма, що описує структуру шаблону проектування *Знімок*

- 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;

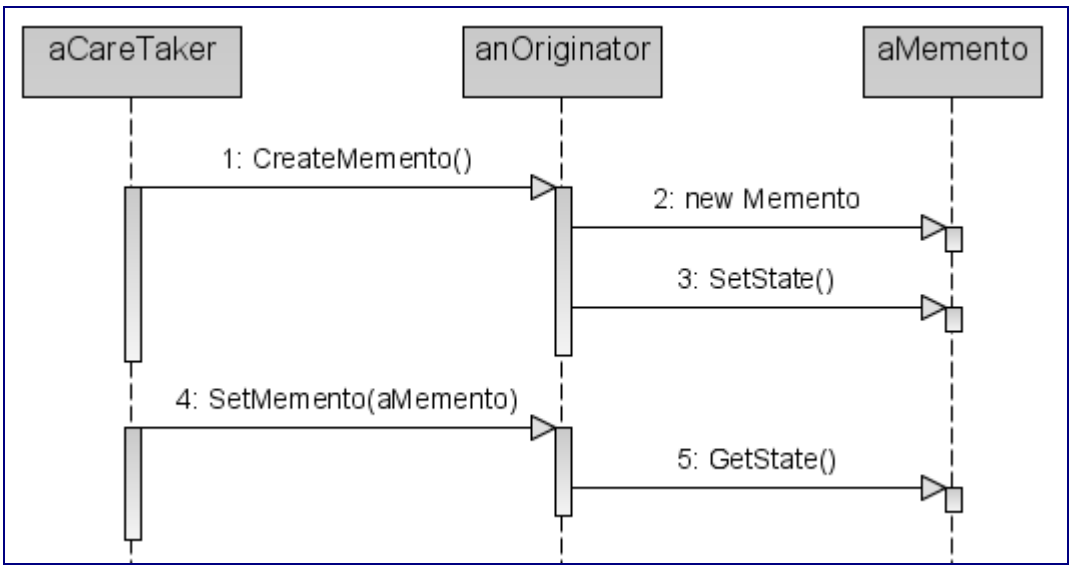


Рис. 12.3. UML діаграма, що описує відносини між об'єктами шаблону проектування *Знімок*

- 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

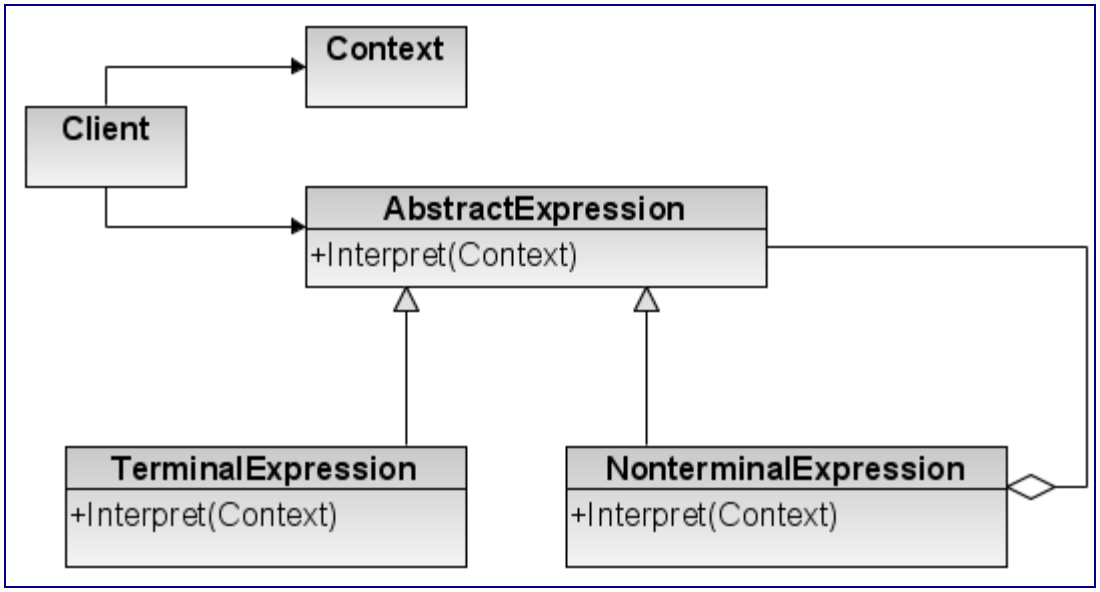


Рис.13.1. UML діаграма, що описує структуру шаблону проектування Інтерпретатор

- 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.

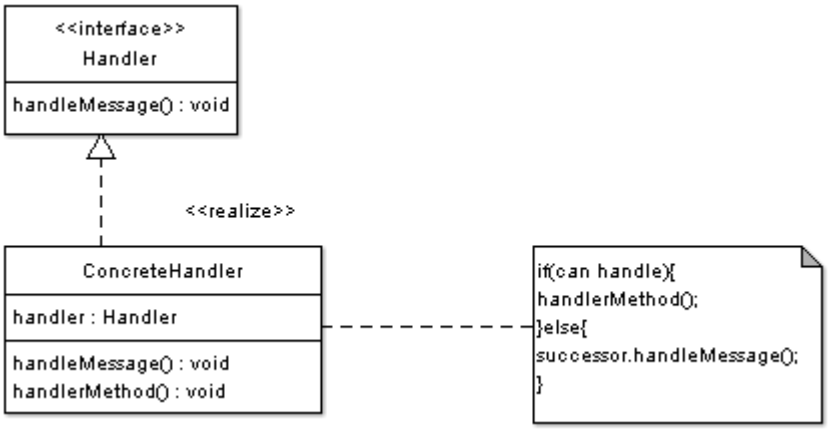


Рис. 13.2

```
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.

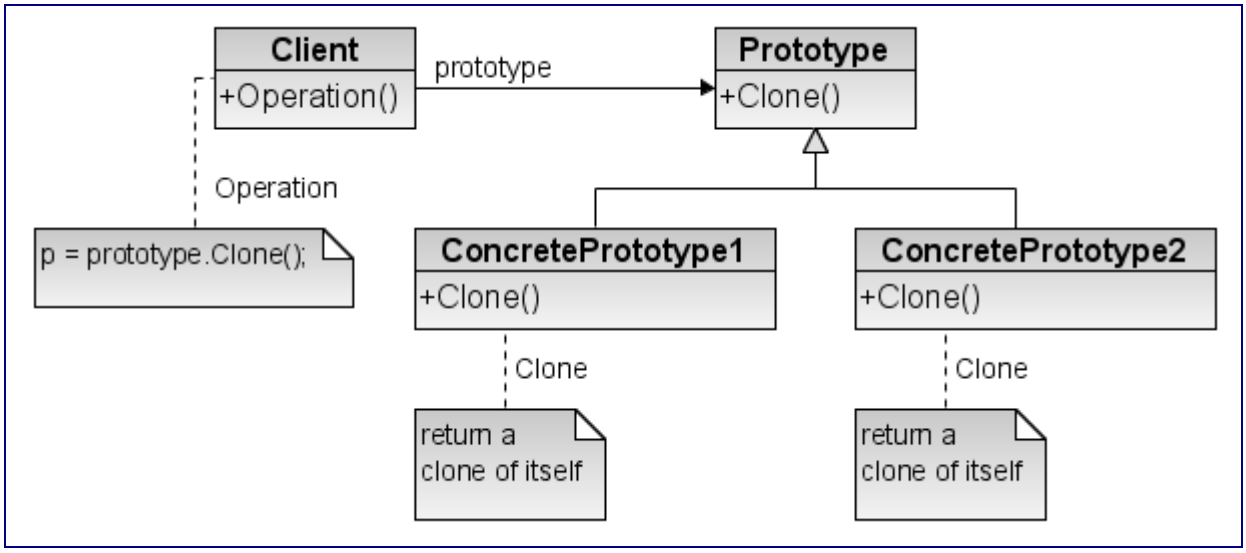


рис. 14.1

```
      /**
 * 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

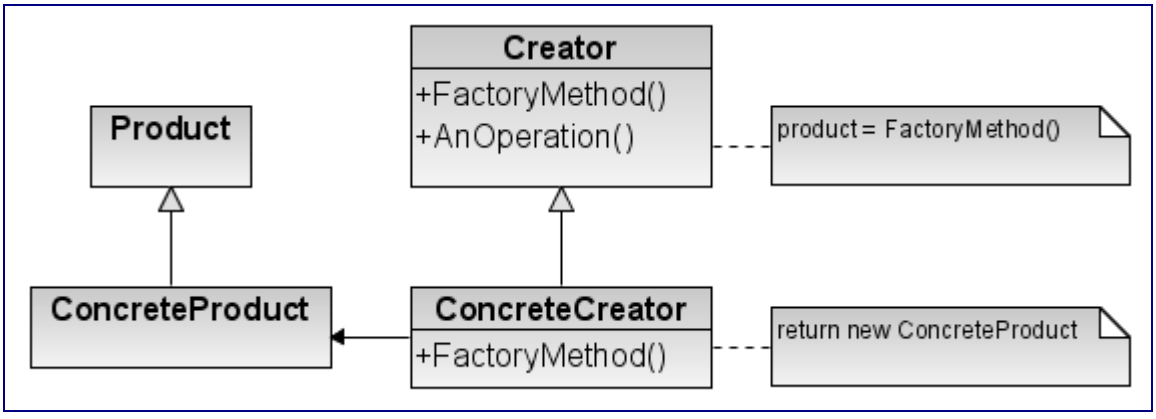


Рис.14.2. UML діаграма, що описує структуру шаблону проектування *Фабричний метод*

- 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

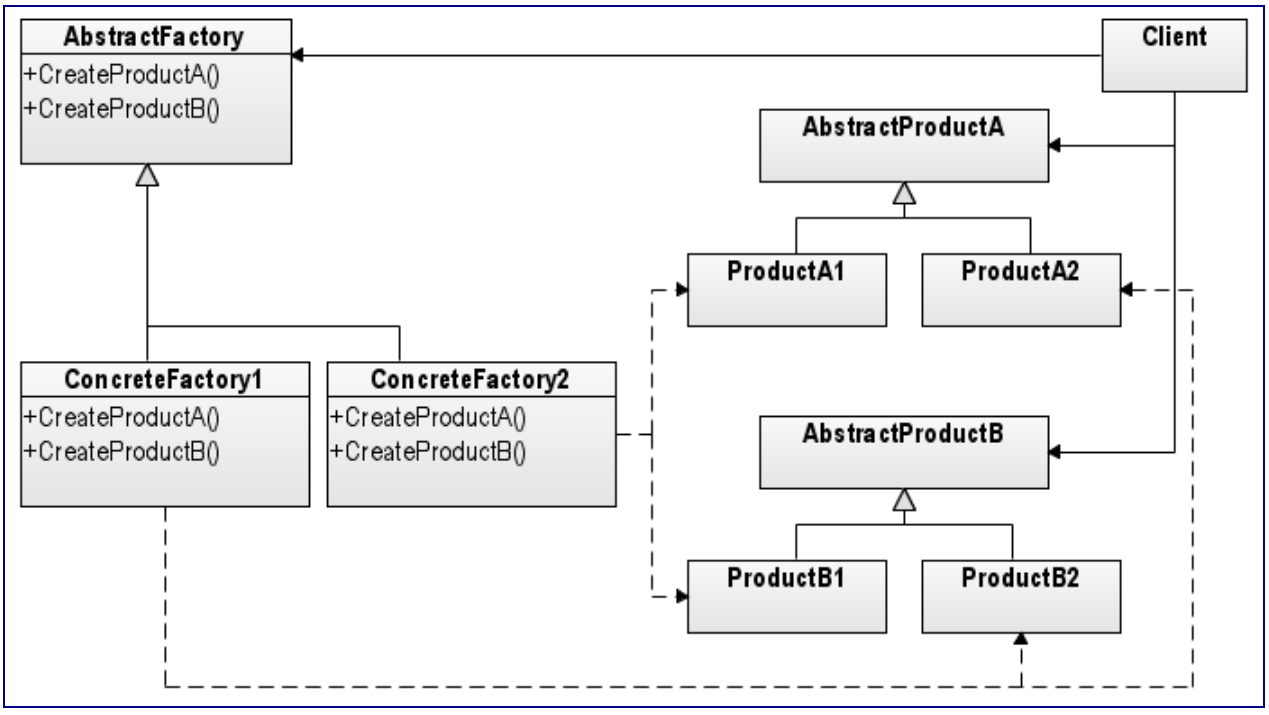


рис. 14.3 UML діаграма, що описує структуру шаблону проектування *Абстрактна фабрика*

- 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

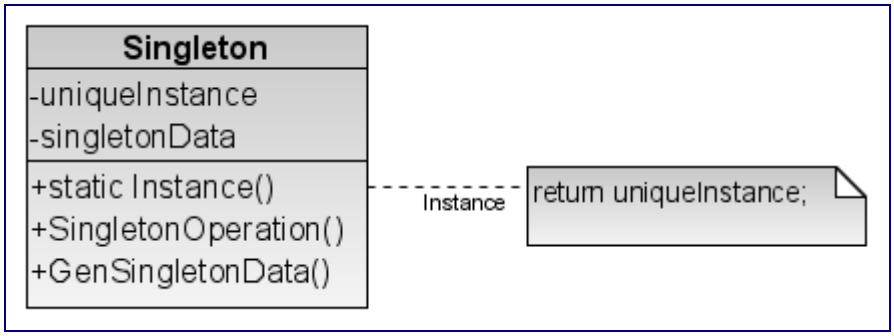


Рис. 15.1. Діаграма класів, що описує структуру шаблону проектування *Одинак*

- 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

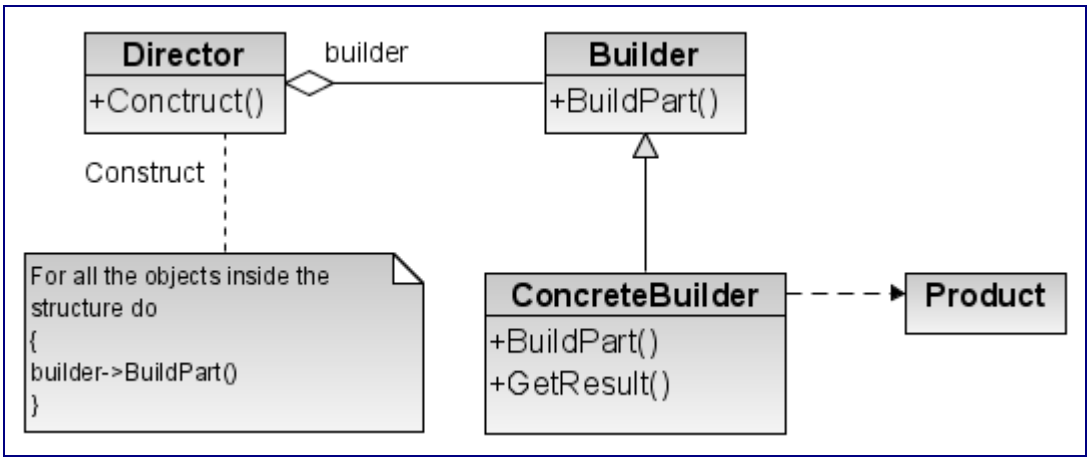


Рис. 15.2. UML діаграма, що описує структуру шаблону проектування *Будівник*

- 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*.

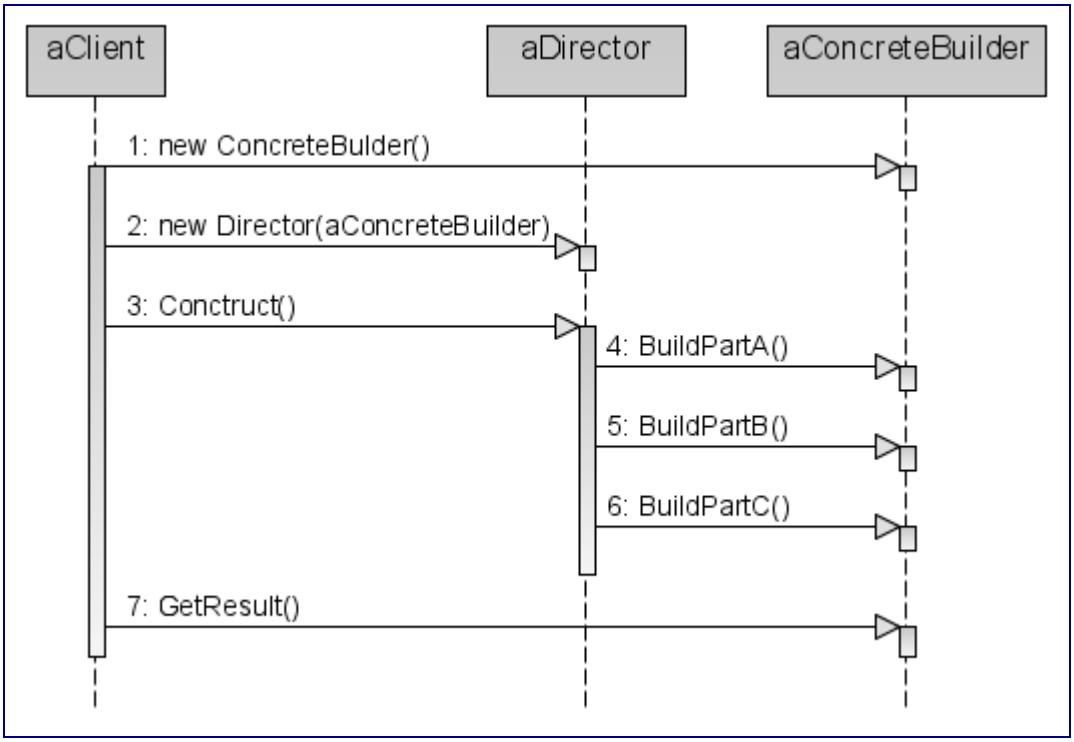


Рис. 15.3

```
      /** "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 неукоснітелно. 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 переваг послідовності 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);
        }
    });
  }
}
```

Лекція 17. Моделі подій та промальовування.

- Моделі подій.
- Моделі промальовування графічного інтерфейсу користувача.[3, с. 197-259]

Моделі обробки подій

Незважаючи на істотні зміни механізму обробки подій в AWT, Java1.1 підтримує зворотну сумісність з моделлю обробки подій, прийнятою в Java 1.0. Однак така сумісність відноситься до типу “все або нічого” ці дві моделі настільки відрізняються один від одного, що їх неможливо використати в одному додатку одночасно.

Модель обробки подій Java 1.0

Кожний компонент може обробляти події, замістивши певні методи, що викликаються реалізацією методу handleEvents класу, що використовується за умовчанням Component. Цей метод викликається з об'єктом класу Event, що описує всі можливі типи подій. Що найчастіше використовуються події, наприклад, ті, що пов'язані з мишею і клавіатурою, диспетчеризуються іншим методам класу Component.

Всі події, пов'язані з мишею, викликаються з копією оригінальної події, а також з координатами x і y, в яких ця подія сталася.

- mouseEnter викликається в тому випадку, коли миша входить в компонент.
- mouseExit викликається при виході миші з області компонента.
- mouseMove викликається при переміщенні миші в області компонента.
- mouseDown викликається при натисненні кнопки миші.
- mouseDrag викликається при переміщенні миші з натисненою кнопкою.
- mouseUp викликається при відпущенні кнопки миші.

Аналогічно, keyDown і keyUp викликаються при кожному натисненні і відпущенні клавіші. Подія передається методу разом з кодом натисненої клавіші. Подію можна перевірити, щоб подивитися, чи натиснені в даний момент які небудь клавіші-модифікатори, для цієї мети можна також користуватися методами shiftDown, controlDown і metaDown. У класі Event визначені десятки констант, що дозволяють використати символічні імена, наприклад, PGUP і HOME.

Нарешті, для роботи зі спеціальними подіями, наприклад, із зворотними викликами (callback) з компонентів Button, Scrollbar і Menu, вам доведеться заміняти метод action. Цей метод викликається з початковою подією і з другим параметром, який являє собою компонент призначеного для користувача інтерфейсу, що створив цю подію. Ви повинні перевірити цей об'єкт, розібратися, який з компонентів послав вам подію, після чого передати управління відповідному даному компоненту обробнику. Для того, щоб перед приведенням

типу перевірити, чи належить об'єкт до певного класу, наприклад, до класу Button, ви можете використати оператор instanceof.

Модель обробки подій Java 1.1

Нова модель обробки подій являє собою, по суті, модель зворотних викликів (callback). При створенні GUI-елемента йому повідомляється, який метод або методи він повинен викликати при виникненні в ньому певної події (натиснення кнопки, миші і т.п.). Цю модель дуже легко використати в C++, оскільки ця мова дозволяє оперувати покажчиками на методи (щоб визначити зворотний виклик, необхідно усього лише передати покажчик на функцію). Однак в Java це неприпустиме (методи не є об'єктами). Тому для реалізації нової моделі необхідно визначити клас, що реалізовує деякий спеціальний інтерфейс. Потім можна передати примірник такого класу GUI-елементу, забезпечуючи таким чином зворотний виклик. Коли наступить очікувана подія, GUI-елемент викличе відповідний метод об'єкта, визначеного раніше.

Модель обробки подій Java 1.1 використовується як в пакеті AWT, так і в JavaBeans API. У цій моделі різним типам подій відповідають різні класи Java. Кожна подія є підкласом класу java.util.EventObject. Події пакету AWT, які і розглядаються в даному розділі, є підкласом java.awt.AWTEvent. Для зручності події різних типів пакету AWT (наприклад, MouseEvent або ActionEvent) вміщені в новий пакет java.awt.event.

Для кожної події існує породжуючий його об'єкт, який можна отримати за допомогою методу getSource(), і кожній події пакету AWT відповідає певний ідентифікатор, який дозволяє отримати метод getid(). Це значення використовується для того, щоб відрізняти події різних типів, які можуть описуватися одним і тим же класом подій. Наприклад, для класу FocusEvent можливі два типи подій: FocusEvent.FOCUS_GAINED і FocusEvent.FOCUS_LOST. Підкласи подій містять інформацію, пов'язану з даним типом події. Наприклад, в класі MouseEvent існують методи getX(), getY() і getClickCount (). Цей клас наслідує, в числі інших, і методи getModifiers() і getWhen().

Модель обробки подій Java 1.1 базується на концепції слухача подій. *Слухачем події* є об'єкт, зацікавлений в отриманні даної події. У об'єкті, який породжує подію (в джерелі *події), міститься* список слухачів, зацікавлених в отриманні повідомлення про те, що дана подія сталася, а також методи, які дозволяють слухачам додавати або видаляти себе з цього списку. Коли джерело породжує подію (або коли об'єкт джерела зареєструє подію, пов'язану з введенням інформації користувачем), він оповіщає всі об'єкти слухачів подій про те, що дана подія сталася.

Джерело події оповіщає об'єкт слухача шляхом виклику спеціального методу і передачі йому об'єкта події (примірника підкласу EventObject). Для того щоб джерело міг викликати даний метод, він повинен бути реалізований для кожного слухача. Це пояснюється тим, що всі слухачі подій певного типу повинні реалізовувати відповідний інтерфейс. Наприклад, об'єкти слухачів подій ActionEvent повинні реалізовувати інтерфейс ActionListener. У пакеті

Java.awt.event визначені інтерфейси слухачів для кожного з певних в йому типів подій (наприклад, для подій MouseEvent тут визначено два інтерфейси слухачів: MouseListener і MouseMotionListener). Всі інтерфейси слухачів подій є розширеннями інтерфейсу java.util.EventListener. У цьому інтерфейсі не визначається жоден з методів, але він грає роль інтерфейсу-мітки, в якому однозначно визначені всі слухачі подій як такі.

У інтерфейсі слухача подій може визначатися декілька методів. Наприклад, клас подій, подібний MouseEvent, описує декілька подій, пов'язаних з мишею, таких як події натиснення і відпущення кнопки миші. Ці події викликають різні методи відповідного слухача. По встановленій угоді, методам слухачів подій може бути переданий один єдиний аргумент, що є об'єктом тієї події, яка відповідає даному слухачеві. У цьому об'єкті повинна міститися вся інформація, необхідна програмі для формування реакції на дану подію. У таблиці приведені певні в пакеті java.awt.event типи подій, відповідні ним слухачі, а також методи, визначені в кожному інтерфейсі слухача.

Типи подій, слухачі і методи слухачів в Java 1.1

Клас події	*Інтерфейс слухача*	*Методи слухача*
ActionEvent	ActionListener	actionPerformed()
AdjustmentEvent	AdjustmentListener	AdjustmentValueChanged()
ComponentEvent	ComponentListener	ComponentHidden() componentMoved() componentResized() Cor”componentShown()
ContainerEvent	ContainerListener	ComponentAdded() componentRemoved()
FocusEvent	FocusListener	focusGained() focusLost ()
ItemEvent	ItemListener	ItemStateChanged()
KeyEvent	KeyListener	keyPressed() keyReleased() keyTyped()
MouseEvent	MouseListener	mouseClicked() mouseEntered() mouseExited() mousePressed() mouseReleased()
	MouseMotionListener	mouseDragged() mouseMoved()
TextEvent	TextListener	TextValueChanged()

WindowEvent	WindowListener	WindowActivated() windowClosed() windowClosing() windowDeactivated() windowDeiconified() windowlconified() windowOpened()

Моделі промальовування графічного інтерфейсу користувача

Система малювання AWT

Перш ніж говорити про деталі системи малювання, яка використовується в бібліотеці Swing, має сенс побачити, як малювання відбувається в базовій бібліотеці AWT. Саме вона зв'язує абстрактні виклики Java і реальні дії операційної системи на екрані.

У всіх сучасних операційних системах процес малювання відбувається приблизно однаково. При цьому ваша програма відіграє пасивну роль і терпляче очікує, коли настане потрібний час потрібного моменту. Момент цей визначають механізми операційної системи, і настає він, коли необхідно перемалювати частину вікна, що належить вашому додатку, наприклад, коли вікно вперше з'являється на екрані або коли схована його частина відкривається очам користувача. Операційна система при цьому визначає, яка частина вікна має потребу в перемальовуванні, і викликає спеціальну частину вашої програми, відповідальну за малювання, або посилає вашому додатку повідомлення про перемальовування, що ви при необхідності обробляєте. Частиною програми в графічних компонентах AWT, що викликається системою для промальовування компонента, є метод paint().

Програмне, або примусове, перемальовування, також необхідне - вам необхідно мати можливість вручну вказувати системі, що пора заново намалювати той або інший фрагмент вашого екрана. Цього вимагає анімація або будь-які динамічні зміни в інтерфейсі, не чекати ж, поки операційна система вирішить перемалювати ваше вікно. Перемальовування дозволяє викликати ще один доступний всім компонентам AWT метод repaint().

Найпростіше оцінити схему малювання, яка використовується за замовчуванням в Java й AWT, на простій діаграмі, і ми відразу побачимо всю її таємницю:

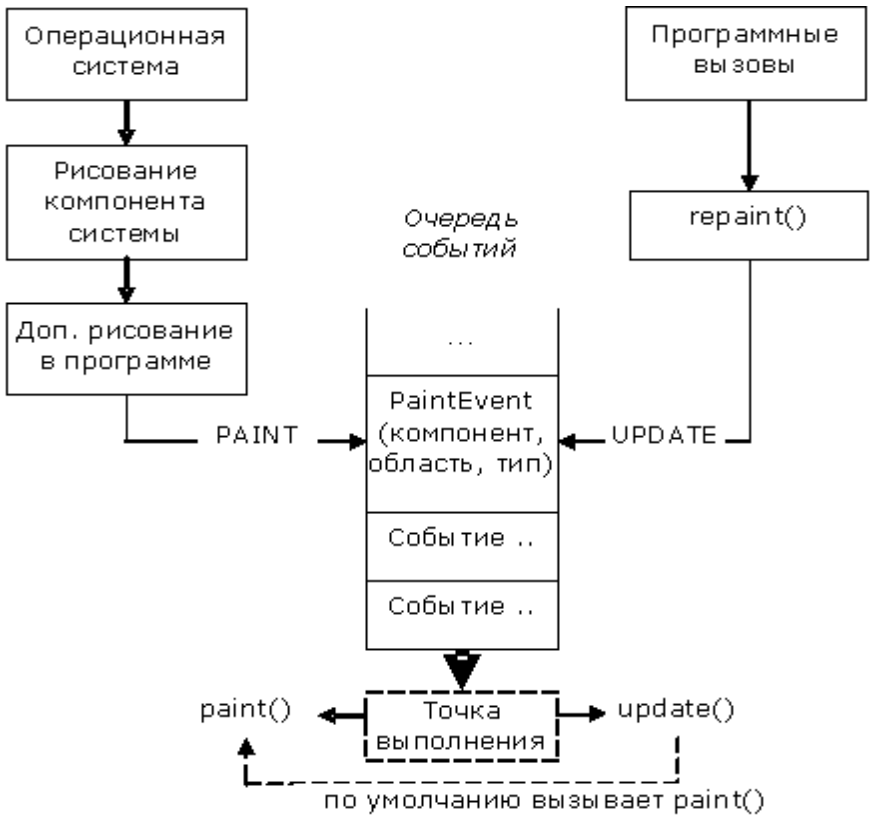


Рис. 17.1

Відштовхуватися доводиться від того, що графічна система зберігає всі свої події в черзі, щоб уникнути сміття на екрані, викликаного його непослідовним відновленням. Виклики про промальовування того або іншого фрагмента екрана також необхідно розмістити в черзі, і робиться це за допомогою спеціальної події PaintEvent.

У першому варіанті заклик намалювати фрагмент екрана надсилає операційна система, коли, на її думку, він був «ушкоджений», тобто згорнутий, закритий іншим вікном і т.п. Якщо в цей фрагмент входить системний компонент (той самий, що представлено компонентами AWT, такими як кнопки Button або списки List), він перемальовує себе сам, і виглядає саме так, як йому покладено в даній операційній системі. Після цього помічник (peer) компонент або системна частина Java створить подію PaintEvent з типом PAINT, укаже в ньому, яку область екрана необхідно перемалювати й у якому компоненті й помістить його в чергу подій EventQueue.

Програмне перемальовування багато простіше і не пов'язане з ніякими системними викликами. У методі repaint() просто створюється подія PaintEvent з типом UPDATE, вказується компонент (той самий для якого й був викликаний repaint()) і область перемальовування (її можна вказати вручну або буде використаний весь розмір компонента), і також поміщається в чергу подій.

Згадуючи архітектуру подій Swing, логічно було б подумати, що для одержання сигналів про промальовування можна приєднувати слухачів типу PaintEventListener, які потім сповіщаються при розсиланні подій з методів dispatchEvent() і processPaintEvent(). Однак, обробляти сигнали про

промальовування як звичайні події ми не можемо. Замість цієї події PaintEvent обробляються самою системою (у помічниках), коли потік розсилання подій «витягає» їх із черги й передає в метод dispatchEvent() компонента, до якого вони належать. Для подій типу PAINT викликається метод paint() компонента, у якому необхідно провести перемальовування. Для подій UPDATE викликається метод update(), що за замовчуванням також викликає метод paint(). Всі ці методи визначені в базовому класі будь-якого компонента Component.

Як засіб для малювання в кожний із цих «» методів, що малюють, передається графічний об'єкт Graphics (часто його називають графічним контекстом). Саме з його допомогою графічні примітиви виводяться на екран. Одержати його можна не тільки в цих методах, але й створити самому, викликавши метод getGraphics() (доступний у будь-якому компоненті). Це дозволяє намалювати щось миттєво, не чекаючи виклику «» методу, що малює, однак, це практично даремно. Будь який наступний виклик «» методу, що малює, однаково намалює на екрані те, що визначено в ньому, так що краще все зводити до методу paint(). До речі, об'єкт Graphics для методів, що малюють, системна частина Java також створює методом getGraphics().

Метод paint() - резюме

Отже, у методі paint() розміщається код, який промальовує компонент. Причому враховувати що саме в компоненті помінялося, не обов'язково, тому що в метод передається графічний контекст Graphics, якому вже заданий прямокутник відсікання (clip) (переданий або системою, або з методу repaint()). За межами цього прямокутника промальовування не здійснюється й час не затрачається.

Додавати до малювання системних компонентів AWT свої деталі не варто - занадто невизначена система взаємодії системної процедури промальовування й методу paint(). Як правило, системний компонент буде «прориватися» через намальоване вами, а те й взагалі не дасть нічого на собі намалювати. Спеціально для малювання в AWT передбачений компонент-«полотно» Canvas.

Якщо вам знадобиться змінити які-небудь глобальні параметри графічного об'єкта Graphics, наприклад включити згладжування, змінити прямокутник відсікання, а ваш компонент може містити інші компоненти, особливо легковагі, створюйте копію об'єкта, викликаючи метод create() класу Graphics. У противному випадку всі ваші настроювання перейдуть у спадщину всім компонентам, які можуть прорисовуватися після вашого компонента. При тому, є одна деталь - після завершення малювання для такого об'єкта прийде явно викликати метод dispose(), інакше ресурси системи малювання можуть швидко закінчитися.

Метод repaint() - пари доповнень

Як ми побачили, метод програмного перемальовування repaint() для стандартних компонентів AWT викликає спочатку метод update(). Колись творці AWT думали, що це допоможе реалізувати ефективну техніку інкрементального промальовування. Це значило що щораз коли ви викликали repaint(), у методі

update() до вже промальованого компонента можна було додати якісь деталі, а потім перейти до основної картини в методі paint() (встановивши попередньо необхідний прямокутник відсікання (clip), щоб не зникло те, що було тільки що намальоване).

Однак такий підхід рідко потрібно, а ефективне промальовування можна здійснити й просто обмеживши область малювання в методі paint() (застосовуючи той самий прямокутник відсікання). Так що задум творців AWT не набув широкої уваги мас.

Інтерес також можуть викликати деякі версії repaint(), яким можна вказати час (у мілісекундах), за яке перемальовування повинна б відбутися. Документація Sun серйозно стверджує, що цей параметр саме так і діє, однак насправді він поки не реалізований.

Головною рекомендацією залишається виклик repaint() з максимально звуженою областю перемальовування компонента. Це особливо вірно для складним, наповненим динамічним умістом й анімацією компонентів, тому що це приносить величезну економію за часом промальовування. Як ми зараз побачимо, Swing намагається взяти більшу частину цієї задачі на себе, однак де можливо, однаково варто відслідковувати мінімальну область промальовування самим.

Малювання легковагих компонентів

Як обробляється малювання системних компонентів, ми тільки що побачили. Однак вони настільки рідко застосовуються (і мерехтіння при промальовуванні ще раз доводить що не даремно), що їх можна розцінювати лише як приємне доповнення до легковагих компонентів. Як ми знаємо, легковагий компонент являє собою область екрана великовагового контейнера. Для того щоб він міг правильно відображатися на екрані, йому також необхідно одержувати від системи промальовування виклики свого методу paint(), однак операційна система, завідувачка цим процесом в AWT, нічого не знає про існування легковагих компонентів, вона бачить лише «рідні» великовагові компоненти й контейнери, яким і відправляє запити на перемальовування. Рішення тут очевидно - потрібно вмонтувати підтримку легковагих компонентів у великовагові контейнери, і саме так надійшли творці AWT.

Всі контейнери в AWT (а значить й в Swing) успадковані від свого базового класу Container. Саме там легковагі компоненти й стають повноправними учасниками процесу висновку на екран. Ніяких хитростей тут ні, потрібно лише чітко окреслити коло учасників цього процесу. Отже - легковагий компонент - це щось, успадковане від класу Component, і не пов'язане з операційною системою помічником (peer). Як він виглядає на екрані, визначає винятково його метод, що малює, paint(). Ці компоненти можна додавати в контейнер (будь-який, аби тільки він був успадкований від класу Container).

У контейнерах підтримується ієрархія компонентів, вибудувана «по осі Z» (z-order). Вони як би нанизуються на вісь Z, спрямовану *від нас*. Перший компонент має індекс 0, другий 1, і так далі. Малювання йде у зворотному порядку, так що перший доданий компонент завжди закриває інші, якщо вони перекривають один одного. Діаграма остаточно все прояснить:

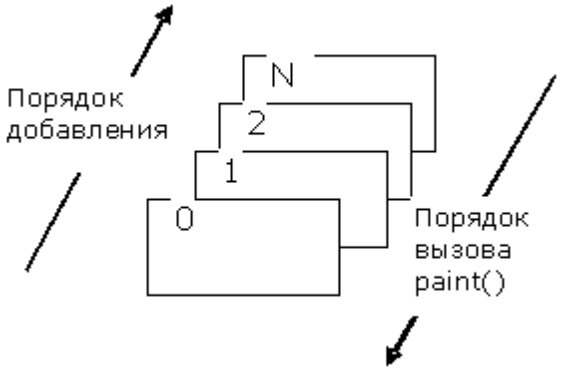


Рис. 17.2

У підсумку, коли операційна система запитує промальовування приналежного їй великовагового контейнера (у випадку додатка Swing це буде як правило вікно JFrame або діалог JDialog), викликається його метод paint. У ньому, відповідно до порядку додавання компонентів, від останнього до першого, відбуваються наступні кроки:

Встановлюється область відсікання для малювання (clip), що відповідає області, яку займає легковагий компонент у контейнері.

Для компонента, який промальовується, в об'єкті Graphics виставляються поточні кольори фону й шрифту (беруться з контейнера, втім використовувати їхній компонент зовсім не обов'язково).

Встановлюється поточний шрифт (з контейнера)

Настроєний об'єкт Graphics передається в метод paint() легковагого компонента, що і малює себе (а фактично, просто бере участь у процесі промальовування все того ж великовагового контейнера).

Ідея проста - кожен легковагий компонент вносить свою лепту в процес малювання контейнера, у той час як операційна система вважає що контейнер просто малює себе. Тому що встановлено область відсікання, легковагий компонент ніколи не зможе «залізти» в область, що йому не належить, і домалювати там щось зайве. Тому що малювання йде від останніх доданих компонентів до перших, перші завжди будуть перекривати останні у випадку перетинання.

Звідси ж виникає чудесна здатність легковагих компонентів бути прозорими й приймати будь-які форми - їх ніхто не змушує хоч щось малювати й не зобов'язує зафарбовувати фоновими кольорами займаним на екрані прямокутник, як це відбувається з великоваговими компонентами. Таким чином, усередині великовагового компонента руки в нас розв'язані й компонентам можна додати будь-які форми, хоча звичайно ж по суті вони залишаються прямокутними.

Із цієї ідилії випадають лише великовагові компоненти, якщо їх додавати в той же контейнер. Поза залежністю від їхньої позиції по осі Z, вони будуть *перекривати* легковагі компоненти. Пояснення отут простої – вони одержують команду paint() ззовні, від операційної системи, і якщо їх хтось перекриває, операційна система знову жадає від них перемальовування – і вони, природно, у свою чергу перемальовують ті легковагі компоненти, що по праву намагалися перекрити їх. Природа великовагових і легковагих компонентів занадто різна,

тому й рекомендується *не сполучати* їх в одному контейнері щоб уникнути проблем із промальовуванням і розташуванням на екрані.

Для повноти картини залишається додати, що для легковагих компонентів поводження методу repaint не дуже змінилося (). Суть його залишилася колишньою — він все також поміщає в чергу подій повідомлення про необхідність перемальовування, але відбувається це не прямо з методу repaint() легковагого компонента. Коли ви викликаєте repaint() для легковагого компонента, він переадресує запит контейнеру, у якому втримується, указавши як область перемальовування ту область, що він займає в контейнері. Так триває до тихнув пір, поки запит не досягне великовагового контейнера (легковагі компоненти можуть утримуватися в легковагих контейнерах, так що великоваговий контейнер можна шукати довго). У підсумку запит на перемальовування в чергу подій поміщає саме великоваговий контейнер, причому перемальовування «замовляється» лише для області потрібного легковагого компонента. Це дозволяє скоротити витрати, тому що легковагий компонент може займати лише малу частину контейнера. (прим. – *Згадуючи, що при виклику repaint() спочатку викликається метод update(), ми одержуємо що й отут метод update() викликається – але для великовагового контейнера. А от для легковагих компонентів update() не викликається, тому що їхній контейнер, що малює, прямо викликає метод paint()*).

Легковагові компоненти: резюме

Легковагові компоненти не доставляють проблем, якщо пам'ятати деякі деталі:

Легковагі компоненти зберігаються в контейнері в порядку додавання (по осі Z), і малюються навпаки - виходить, ті компоненти, що додано спочатку, мають на екрані пріоритет.

Легковагові компоненти можуть бути прозорі або малювати себе в будь-якій формі - через незачеплені області можуть «просвічувати» інші компоненти або фон контейнера.

Метод repaint() працює для легковагих компонентів - але за лаштунками перемальовується частина великовагового контейнера. Метод update() не викликається.

Якщо ви перевизначаєте контейнер і його метод paint(), і плануєте потім додавати в нього інші компоненти, не забудьте викликати базовий метод super.paint(), інакше про промальовування легковагих компонентів доведеться подбати самостійно.

Система малювання Swing

По великому рахунку, легковагі компоненти нічим не обмежені, і створити з їхньою допомогою, не мудруючи лукаво, навіть таку багату бібліотеку, як Swing, не являє собою технічних труднощів. Однак у часи зародження Swing користувальницькі інтерфейси Java, а особливо реалізація AWT, страждали від проблем із продуктивністю. З багатими можливостями, іноді навіть прозорі компоненти Swing просто зробили б з користувальницького інтерфейсу мало

легкотравного тихохода. Дійсно, всі легковагі компоненти малюються один за одним, поза залежністю від того, є чи серед них непрозорі області, і кожна операція із графікою - це довга операція по звертання до «рідного» коду операційної системи. Оптимізація була необхідна як повітря, і оптимізація серйозна.

Відмінність системи малювання Swing від стандартної й складається в оптимізації й підтримці UI-представників. Якщо говорити зовсім стисло, то бібліотека Swing у своїй системі малювання керується всього трьома принципами, які дозволяють нам позбутися від зайвої роботи й бути впевненими в тім, що малювання максимально оптимізовано:

Кешуй

Розподіляй і володарюй

З очей геть, із серця геть

Кешування

Стратегія кешування, яка застосовується в Swing, украй проста й стара як світ. Якщо пристрій виводить дані на екран занадто повільно (не то щоб зараз повільні відеокарти, але пам’ятайте, що в Java це виклики «рідного» коду), необхідно заздалегідь підготувати все у швидкому сховищі-буфері (у пам'яті), а потім одним махом (однією операцією) записати все в пристрій (на екран). Як буфер застосовується область у пам'яті у форматі екрана, по можливості - область у пам'яті відеокарти. Цей буфер зберігає допоміжний клас для малювання в Swing за назвою RepaintManager. Він же перевіряє що буфер коректний, заново створює його при необхідності й виконує тому подібну технічну роботу. Також цей клас дозволяє компонентам запросити промальовування в буфер з наступним висновком на екран.

Цікава особливість компонентів Swing полягає в тому, що вони «спілкуються» один з одним за допомогою системи прапорів. Вся ця система убудована в базовий клас JComponent. Таким чином, розраховувати на те, що компоненти Swing є автономними сутностями, кожен зі своїм життям і процедурою промальовування (як по великому рахунку це відбувається в AWT), не варто. Давайте подивимося як малюється компонент Swing, доданий у великоваговий контейнер:

Операційна система просить вікно заново намалювати вміст.

Вікно проходить за списком легковагих компонентів, що втримувалися в ньому, і знаходить там компонент Swing.

Викликається метод paint(). Щоб вмонтувати оптимізацію в усі компоненти Swing, метод paint() у них не перевизначається, і таким чином, роботу завжди виконує базовий клас JComponent.

Якщо (відповідно до прапорів), це перший викликаний для малювання компонент Swing, то він запитує малювання через буфер у класу RepaintManager. Той набудовує буфер і викликає метод paintToOffscreen() його компонента, що викликав, передаючи йому об'єкт Graphics для малювання в пам'яті. Виставляється прапор, що відтепер говорить, що малювання пішло в буфер.

Снову викликається метод paint(), але тому що прапор малювання в буфер уже виставлений, RepaintManager більше не застосовується. Викликаються «робітники» методи промальовування компонента, рамки й компоненти-нащадків (про ці методи трохи нижче).

Компоненти-нащадки малюють своїх нащадків (прямо, відповідно до прапорів), і так далі до повного промальовування всієї ієрархії даного компонента Swing.

Після закінчення цієї процедури керування (відповідно до кроку 4) повертається в RepaintManager, які копіює зображення компонентів з буфера на реальний екран.

Дана процедура вірна для будь-якого компонента Swing, і як видно, найбільш вдалим рішенням була б наявність одного компонента, що займав би всю площу великовагового контейнера, а виходить, саме цей компонент і був би єдиним учасником, що передає керування від AWT в Swing, і копіювати зображення з буфера на екран довелося б тільки один раз. Саме так і зроблено, а роль такого компонента в контейнерах вищого рівня Swing грає коренева панель JRootPane.

Кешування (буферизацію) графіки в Swing можна відключити, або методом RepaintManager.setDoubleBufferingEnabled(), або безпосередньо на компоненті методом setDoubleBuffered(). Правда, дещо змінюється зміст вимикання подвійної буферизації для окремих компонентів - ми вже бачили, що компонент малює за допомогою буфера, якщо буфер використовується його батьківським компонентом, незалежно від того, включена або виключена подвійна буферизація для нього самого. Якщо всі компоненти перебувають у кореневій панелі, виключати подвійну буферизацію має сенс тільки для неї (це буде рівносильно вимиканню подвійної буферизаціїи для всіх компонентів, що перебувають усередині цього контейнера вищого рівня). Урахуйте, що з появою в AWT починаючи з версій Java 1.4 системної підтримки буферизації система Swing може бути відключена щоб уникнути дублювання. З іншого боку, важко собі уявити, навіщо відключати оптимізацію самостійно, хіба що ваш додаток саме буде нею займатися.

Поділ обов'язків

Як видно, метод paint() у компонентах Swing зайнятий таки важливою справою - він кэширует висновок графіки компонентів на екран, користуючись допомогою класу RepaintManager. Перевизначати цей метод у кожному компоненті щоб намалювати його більше не можна - ми пам'ятаємо про систему прапорів і що метод paint() діє по різному залежно від ситуації. Тут творці Swing застосували гасло «розділяй і пануй» - малювання відтепер виробляється в інших методах, а метод paint() є носієм внутрішньої логіки бібліотеки й перевизначати його без справи не треба. Це значно спрощує відновлення компонентів і зміна їхнього зовнішнього вигляду, дозволяючи вам при створенні нового компонента не думати про те, як реалізувати для нього ефективний висновок графіки. Ви просто малюєте свій компонент, залишаючи низькорівневі деталі механізмам базового класу.

Малювання компонента в Swing, на відміну від простій до неможливості процедури в бібліотеці AWT, розбито на три етапи, і за кожний відповідає свій метод у класу JComponent. Саме ці методи ви будете перевизначати якщо вам захочеться намалювати що-небудь на компоненті Swing по своєму.

Метод paintComponent()

Метод paintComponent() викликається при промальовуванні компонента першим, і саме він малює сам компонент. Різниця між ним і класичним методом paint(), який виристовується в AWT, полягає в тому, що вам не потрібно піклуватися ні про оптимізацію малювання, ні про правильне промальовування своїх компонентів-нащадків. Про все це подбають механізми класу JComponent. Усе, що вам потрібно зробити, — намалювати в цьому методі компонентів і залишити всю чорнову роботу базовому класу.

Як ви пам'ятаєте, в Swing використовується трохи модифікована архітектура MVC, у якій відображення компонента і його керування виконуються одним елементом, називаним UI-представником. Виявляється, що промальовування компонента за допомогою UI-представника здійснюється саме з методу paintComponent(), певного в базовому класі JComponent. Діє метод дуже просто: він визначає, є чи в компонента UI-представник (чине дорівнює він порожньому посиланню null), і, якщо представник є, викликає його метод update(). Метод update() для всіх UI-представників працює однаково: по властивості непрозорості перевіряє, потрібно чи зафарбовувати всю свою область кольорами фону, і викликає метод paint(), визначений у базовому класі всіх UI-представників — класі ComponentUI. Останній метод і малює компонент. Залишається лише одне питання: що таке властивість непрозорості?

Ми відзначали, що одним із самих вражаючих властивостей легковагих компонентів є їхня здатність бути прозорими. Однак при написанні бібліотеки творці Swing виявили, що набір з декількох десятків легковагих компонентів, здатних «просвічувати» друг крізь друга, приводить до великого завантаження системи малювання й відповідному вповільненню роботи програми. Дійсно, перемальовування будь-якого компонента оберталося дійсною каторгою: крізь нього просвічували інші компоненти, які зачіпали ще одні компоненти, і так могло тривати довго. У підсумку перемальовування навіть невеликої частини одного компонента приводило до перемальовування доброго десятка компонентів, серед яких могли виявитися й дуже складні. З іншого боку, компоненти начебто текстових полів або кнопок рідко бувають прозорими, і зайва робота для них зовсім ні до чого. Так і з'явилася властивість непрозорості (opaque), наявне в будь-якого компонента Swing.

Якщо в AWT будь-який легковагий компонент автоматично вважається прозорим, то в Swing усе зроблено навпаки. Властивість непрозорості визначає, чи зобов'язується компонент зафарбовувати всю свою область, щоб позбавити Swing від додаткової роботи з пошуку й промальовування всього того, що перебуває під компонентом. Якщо властивість непрозорості дорівнює true (а за замовчуванням воно дорівнює true), то компонент зобов'язаний зафарбовувати всю свою область, інакше на екрані замість нього з'явиться сміття. Додаткової

роботи тут небагато: усього лише необхідно замалювати всю свою область, а полегшення для механізмів промальовування виходить значне. Ну а якщо ви все-таки вирішите створити компонент довільної форми або прозорий, викличте для нього метод setOpaque(false), і до вас знову повернуться всі чудесні можливості легковагих компонентів — система промальовування буде попереджена. Однак зловживати цим не варто: швидкість промальовування такого компонента значно падає. Багато в чому через це в Swing не так вуж і багато компонентів, що мають прозорі області.

Повернемося до методу paintComponent(). Тепер роль його цілком очевидна: він прорисовує компонент, за замовчуванням використовуючи для цього асоційованого з компонентом UI-представника. Якщо ви збираєтеся створити новий компонент із власним UI-представником, то він буде прекрасно вписуватися в цю схему. Успадкуйте свого UI-представника від базового класу ComponentUI і перевизначите метод paint() , у якому й малюйте компонент. Базовий клас подбає про властивість непрозорості. Якщо ж вам просто потрібно що-небудь намалювати, успадкуйте свій компонент від будь-якого підходящого вам класу (краще всіх для цього підходять безпосередньо класи JComponent або JPanel, тому що самі вони нічого не малюють) і перевизначите метод paintComponent(), у якому й малюйте. Правда, при такому підході потрібно подбати про властивість непрозорості (якщо воно дорівнює true) самостійно: буде потрібно зафарбовувати всю область промальовування або викликати перед малюванням базову версію методу super.paintComponent().

Метод paintBorder()

Завдяки методу paintBorder() в Swing є така чудова річ, як рамка (border). Для будь-якого компонента Swing ви можете встановити рамку, використовуючи метод setBorder(). Виявляється, що підтримка рамок цілком і повністю забезпечується методом paintBorder() класу JComponent. Він викликається другим, після методу paintComponent(), дивиться, чи встановлена для компонента яка-небудь рамка, і якщо рамка є, прорисовує її, викликаючи певен в інтерфейсі Border метод paintBorder(). Єдине питання, що при цьому виникає: де саме малюється рамка? Прямо на просторі компонента або для неї виділяється окреме місце? Відповідь проста - ніякого спеціального місця для рамки немає. Вона малюється безпосередньо поверх компонента після промальовування останнього. Так що при малюванні компонента, якщо ви не хочете несподіваного накладення рамки на зайняте місце, ураховуйте місце, що вона займає

Перевизначати метод paintBorder() навряд чи варто. Роботу він виконує нехитру, і поліпшити її або докорінно змінити не є можливим. Якщо вам потрібно створити для свого компонента фантасмагоричну рамку, краще скористатися послугами інтерфейсу Border або сполучити кілька стандартних рамок.

Метод paintChildren()

Заключну частину процесу малювання виконує метод paintChildren(). Як ви пам'ятаєте, під час обговорення легковагих компонентів в AWT ми відзначали, що для їхнього правильного відображення в контейнері, якщо ви перевизначили їхні

методи paint(), необхідно викликати базову версію paint() із класу Container, інакше легковагі компоненти на екрані не з'являться. Базовий клас JComponent бібліотеки Swing успадкований від класу Container і цілком міг би скористатися його послугами із промальовування нащадок^-нащадків-компонентів-нащадків, що втримуються в ньому. Однак творці Swing вирішили від послуг класу Container відмовитися й реалізували власний механізм промальовування нащадків. Причина проста - недостатня ефективність механізму промальовування AWT. Поліпшений оптимізований механізм і реалізується методом paintChildren(). Для додання йому максимальної швидкості компонента Swing використовуються дві властивості: уже відоме нам властивість непрозорості opaque, а також властивість isOptimizedDrawingEnabled.

Метод paintChildren() діє по алгоритму, що був злегка вільнодумний названий нами «з очей геть, із серця геть». Він одержує список нащадків, що втримуються в компоненті, і починає перебирати їх, використовуючи при цьому поточний прямокутник відсікання. Основною задачею його є знаходження «сліпих зон», тобто зон, де компонента закриваються один одним. Якщо компонент закритий або взагалі не попадає в область відсікання, то навіщо його малювати - ніхто праці не помітить. Можна намалювати тільки той компонент, що видно зверху.

На цьому етапі «вступають у бій» дві вищезгадані властивості. З першою властивістю усе більш-менш зрозуміло: якщо властивість непрозорості компонента дорівнює true, це означає, що, скільки б компонентів не перебувало під ним і не перетинало б його, всю свою область він зобов'язується зафарбувати, а виходить, продовжувати пошук нащадків у цій області не має змісту - їх однаково не буде видно. Тепер зрозуміло, чому так важливо виконувати вимога заповнення всієї області екрана при використанні властивості непрозорості: у противному випадку на екрані неминучий сміття, що за домовленістю повинен забирати сам компонент, а не система промальовування. Із властивістю isOptimizedDrawingEnabled картина дещоінша.

Дана властивість визначена в класі JComponent як призначена тільки для читання: ви не можете змінити її, крім як успадкувавши власний компонент і перевизначивши метод isOptimizedDrawingEnabled(). За замовчуванням для більшості компонентів властивість isOptimizedDrawingEnabled дорівнює true. Це дозволяє знизити завантаження системи промальовування. Простіше кажучи, ця властивість гарантує, що з під одних нащадків не «просвічують» інші й не зачіпають їх, при цьому на них не накладений додатковий прозорий компонент, і т.д. У випадку простих компонентів ця властивість приносить невеликі дивіденди, однак якщо у вас є складні які повільно малюються, це значно прискорить процес. Коли властивість isOptimizedDrawingEnabled дорівнює true, метод paintChildren() просто перебирає нащадків і перемальовує їхні ушкоджені частини, не розбираючись, що вони собою представляють і як один з одним співвідносяться. Дану властивість перевизначають лише три компоненти: це багатошарова панель JLayeredPane, робочий стіл JDesktopPane й область перегляду JViewport. У них компоненти-нащадки часто перекриваються й вимагають особливої уваги.

Загальна діаграма малювання в Swing

Тепер, коли всі деталі оптимізації й поділу промальовування в Swing нам відомі, ми можемо все об'єднати в нескладну діаграму, на яку можна дивитися якщо раптом щось на екрані перестане малюватися як потрібно:

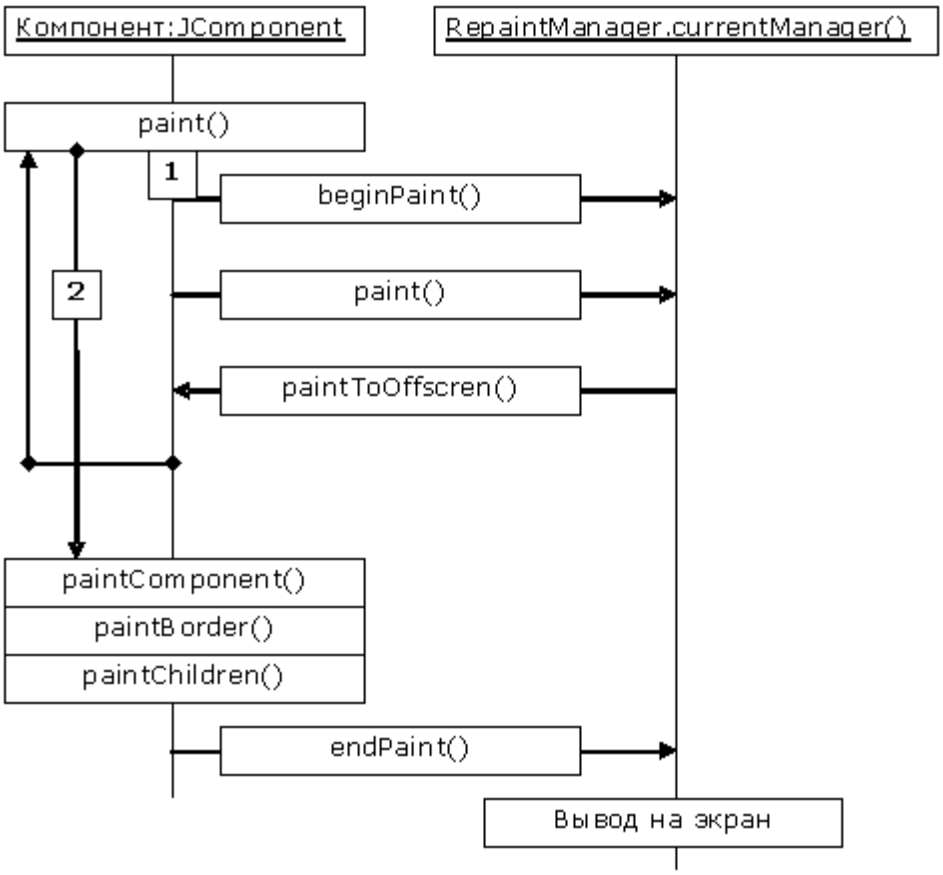


Рис. 17.2

Як ми з'ясували, у методі paint() компонентів Swing є два шляхи - перший працює в тому випадку, якщо компонент Swing малюэться не компонентом, що є Swing - у цьому випадку включаються процеси RepaintManager і готуються буфер-кеш. Якщо ж вони включені, слідує стандартний другий шлях, що малює всі відомими вже нам трьома методами. І той же самий другий спосіб викликається прямо, якщо буферизація в компоненті відключена (вручну або якщо буферизація включена на рівні операційної системи).

Програмне перемальовування в Swing

Якщо згадати як ми розглядали програмне перемальовування в AWT (метод repaint()), то легко бачити що це по суті постановка події на перемальовування (PaintEvent) у чергу подій інтерфейсу. Саме ця проста ідея підштовхнула творців Swing включити оптимізацію й тут, цього разу застосувавши принцип пакетної обробки.

Дивлячись на чергу, мабуть, що кожна подія із черги виконується якийсь час, і поки справа дійде до нашої події на перемальовування PaintEvent, у черзі можуть з'явиться нові події такого ж типу. Досить імовірно, що вони ставляться не лише вікну, у якому була запитана перемальовування в перше, але й навіть

тому самому компоненту, і можливо, наступні події роблять перші зовсім непотрібними, тому що повністю замалюють те, що намалюють перші. Уявіть активне прокручування великої таблиці - подібний процес буде генерувати величезна кількість викликів методу repaint(), і наступні виклики вже будуть перемальовувати області, які повинні б були намальовані першими викликами. Тут можна оптимізувати процес і позбутися від зайвого малювання.

Метод repaint() перевизначений у класі JComponent і замість прямої постановки події PaintEvent у чергу подій просить нашого старого знайомого RepaintManager додати область, яку потрібно перемалювати, у список «забруднених» (dirty region). «Забруднені» області потрібно перемалювати, як тільки до них дійдуть руки потоку розсилання подій.

RepaintManager поєднує всі прихожі запити на промальовування «забруднених» областей в один пакет, ставлячи в чергу подій особлива подія, що при спрацьовуванні разом перемалює всі що нагромадилися (за час очікування виконання цієї особливої події, адже черга подій може бути зайнятаі) «брудні» області. Таким чином, ми позбуваємося від надлишкових подій. На додаток до цього, у класі RepaintManager області для промальовування оптимізуються - вони поєднуються в одну область для компонента, області, які закриті іншими компонентами або більше не видні на екрані, викидаються зі списку промальовування й так далі.

Наступна схема прекрасно ілюструє що відбуваються і які класи беруть участь у процесі:

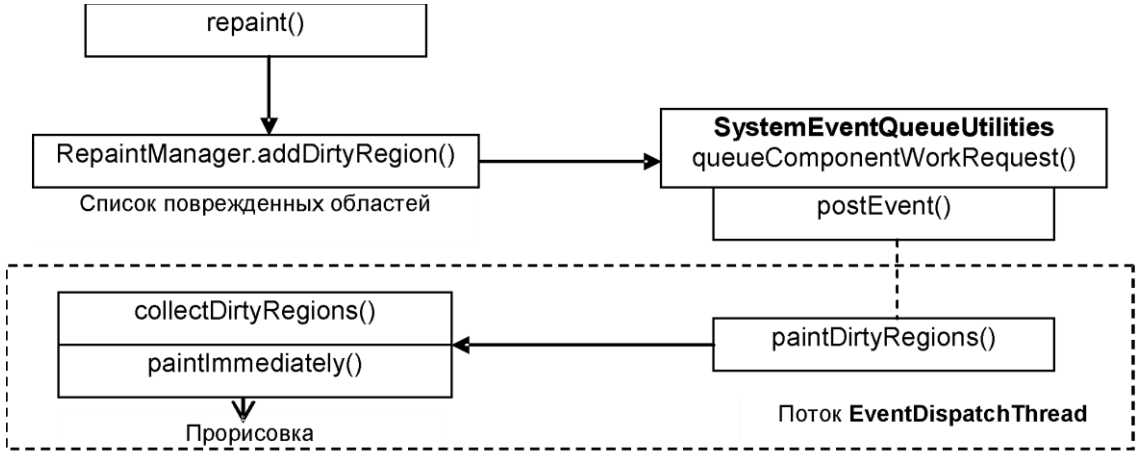


Рис. 17.3

Допоміжний клас SystemEventQueueUtilities ставить та сама особлива подія (його ім'я ComponentWorkRequest) у чергу, використовуючи метод для постановки postEvent(). Події PaintEvent в Swing взагалі не застосовуються. Як тільки потік розсилання доходить до даної події, воно виконується, і викликає метод все того ж RepaintManager з назвою paintDirtyRegions(). Опускаючи деталі реалізації, ми приходимо до того, що викликається метод, певний у базовому класі JComponent за назвою paintImmediately(). Ну а в ньому вже всі зовсім просто - у підсумку створюється графічний об'єкт методом getGraphics(), і викликається прекрасно знайомий нам метод paint(). Роботу цього методу в Swing ми вивчили в деталях ледве раніше, і все виконується по тій же самій діаграмі, що ми склали.

Підсумок - компонент Swing перемальовується, причому лише один раз за певний проміжок часу, і максимально оптимізовано.

Малювання в Swing: резюме

Підводячи підсумки всієї системи малювання Swing, можна визначити наступні найважливіші висновки:

Метод paint() в Swing є частиною внутрішньої системи й виконує важливу роботу. Перевизначати його для малювання компонента Swing не треба.

Для малювання компонента або просто графіки застосовується метод paintComponent().

При малюванні в Swing необхідно враховувати властивість непрозорості opaque. Якщо компонент непрозорий, необхідно зафарбовувати його тло або не залишати непромальованих областей. Робиться це або вручну, або викликом super.paintComponent(), щоб тло зафарбував UI-представник. Однак, другий спосіб не працює для компонентів, успадкованих прямо від JComponent. Якщо не виконати дану умову, сміття на екрані неминуче.

Перемальовування repaint() в Swing виконується пакетами й оптимізується. Якщо вас це не влаштовує, доведеться перевизначити або метод repaint(), або написати свій варіант класу RepaintManager. Втім, необхідності в цьому практично не виникає.

Розділ 5. Моделювання програмного забезпечення

Лекція 18. Інформаційне моделювання та моделювання бізнес-процесів

- Методології моделювання SADT, IDEF, DFD, ELM, OOAD.
- Мови моделювання.
- Інформаційне моделювання.
- Діаграми сутність-зв’язок, класів.
- Моделювання бізнес-процесів, організацій та цілей. [4, с. 506-537]

Методології моделювання

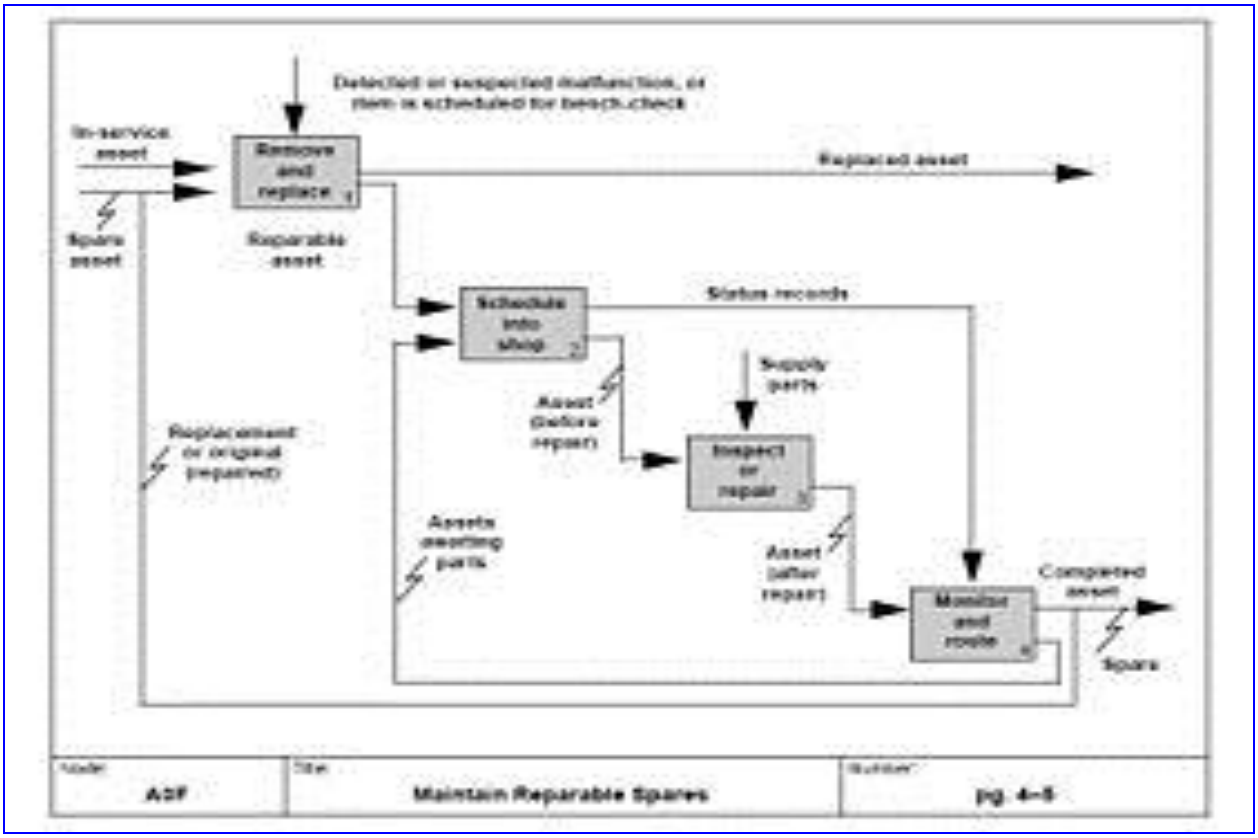
IDEF-методології сімейства ICAM (Integrated Computer-Aided Manufacturing) для рішення завдань моделювання складних систем, дозволяє відображати й аналізувати моделі діяльності широкого спектра складних систем у різних розрізах. При цьому широта й глибина обстеження процесів у системі визначається самим розроблювачем, що дозволяє не перевантажувати створювану модель зайвими даними.

IDEF-методології створювалися в рамках запропонованої ВВС США програми комп'ютеризації промисловості — ICAM, у ході реалізації якої виявилася потреба в розробці методів аналізу процесів взаємодії у виробничих (промислових) системах. Принциповою вимогою при розробці розглянутого сімейства методологій була можливість ефективного обміну інформацією між всіма фахівцями — учасниками програми ICAM (звідси назва: Icam DEFinition — IDEF; інший варіант — Integrated DEFinition). Після опублікування стандарту він був успішно застосований у всіляких областях бізнесу, показавши себе ефективним засобом аналізу, конструювання й відображення бізнесів-процесів. Більше того, із широким застосуванням IDEF (і попередньої методології — SADT) і пов'язане виникнення основних ідей популярного нині поняття — BPR (бізнес-процес реінжиніринг).

Сімейство стандартів

У даний момент до сімейства IDEF можна віднести наступні стандарти:

IDEF0



Example of an IDEF0 diagram: A function model of the process of "Maintain Reparable Spares".

Function Modeling — методологія функціонального моделювання. За допомогою наочної графічної мови IDEF0 досліджувана система з'являється перед розроблювачами й аналітиками у вигляді набору взаємозалежних функцій (функціональних блоків — у термінах IDEF0). Як правило, моделювання засобами IDEF0 є першим етапом вивчення будь-якої системи. Методологію IDEF0 можна вважати наступним етапом розвитку добре відомої графічної мови опису функціональних систем SADT (Structured Analysis and Design Technique);

IDEF1

Example of an IDEF1X Diagram.

Information Modeling — методологія моделювання інформаційних потоків усередині системи, що дозволяє відображати й аналізувати їхню структуру й взаємозв'язки. IDEF1X (IDEF1 Extended) — Data Modeling — методологія моделювання баз даних на основі моделі «сутність-зв'язок». Застосовується для побудови інформаційної моделі, що представляє структуру інформації, необхідну для підтримки функцій виробничої системи або середовища. Метод IDEF1, розроблений Т. Ремей (T. Ramey), також заснований на підході П. Чена й дозволяє побудувати модель даних, еквівалентну реляцыйный моделі в третій нормальній

формі. У цей час на основі вдосконалювання методології IDEF1 створена її нова версія - методологія IDEF1X. IDEF1X розроблена з урахуванням таких вимог, як простота вивчення й можливість автоматизації. IDEF1X-діаграми використовуються поряд з розповсюдженими CASE-засобами (зокрема, ERwin, Design/IDEF).

IDEF2

Example of an Enhanced Transition Schematic, modelled with IDEF3.

Simulation Model Design — методологія динамічного моделювання розвитку систем. У зв'язку з досить серйозними складностями аналізу динамічних систем від цього стандарту практично відмовилися, і його розвиток призупинився на самому початковому етапі. У цей час присутні алгоритми і їхні комп'ютерні реалізації, що дозволяють перетворювати набір статичних діаграм IDEF0 у динамічні моделі, побудовані на базі «розфарбованих мереж Петри» (CPN — Color Petri Nets);

IDEF3

Process Description Capture (Документування технологічних процесів) — методологія документування процесів, що відбуваються в системі (наприклад, на підприємстві), описуються сценарій і послідовність операцій для кожного процесу. IDEF3 має прямий взаємозв'язок з методологією IDEF0 — кожна функція (функціональний блок) може бути представлена у вигляді окремого процесу засобами IDEF3;

IDEF4

Example of the IDEF4: A Behavior Diagram for methods Implementing Louder.

Object-Oriented Design — методологія побудови об’єктно-орієнтованих систем, дозволяють відображати структуру об'єктів і закладені принципи їхньої взаємодії, тим самим дозволяючи аналізувати й оптимізувати складні об’єктно-орієнтовані системи. Докладніше - Технологія;

IDEF5

Example of an IDEF5 Composition Schematic for a Ballpoint Pen.

Ontology Description Capture — Стандарт онтологічного дослідження складних систем. За допомогою методології IDEF5 онтологія системи може бути описана за допомогою певного словника термінів і правил, на підставі яких можуть бути сформовані достовірні твердження про стан розглянутої системи в деякий момент часу. На основі цих тверджень формуються висновки про подальший розвиток системи й виробляється її оптимізація;

IDEF6

IDEF6 model of IDEF4 Design Activities

Design Rationale Capture — Обґрунтування проектних дій. Призначення IDEF6 складається в полегшенні одержання «знань про спосіб» моделювання, їхнього подання й використання при розробці систем керування підприємствами. Під «знаннями про спосіб» розуміються причини, обставини, сховані мотиви, які спричиняються обрані методи моделювання. Простіше кажучи, «знання про спосіб» інтерпретуються як відповідь на питання: «чому модель вийшла такою, якою вийшла?» Більшість методів моделювання фокусуються на характерних одержуваних моделях, а не на процесі їхнього створення. Метод IDEF6 акцентує увагу саме на процесі створення моделі;

IDEF7

Information System Auditing — Аудит інформаційних систем. Цей метод визначений як затребуваний, однак так і не був повністю розроблений;

IDEF8

User Interface Modeling — Метод розробки інтерфейсів взаємодії оператора й системи (користувальницьких інтерфейсів). Сучасні середовища розробки користувальницьких інтерфейсів більшою мірою створюють зовнішній вигляд інтерфейсу. IDEF8 фокусує увагу розроблювачів інтерфейсу на програмуванні бажаного взаємного поводження інтерфейсу й користувача на трьох рівнях: виконуваної операції (що це за операція); сценарії взаємодії, обумовленому специфічною роллю користувача (по якому сценарії вона повинна виконуватися тим або іншим користувачем); і, нарешті, на деталях інтерфейсу (які елементи керування пропонує інтерфейс для виконання операції);

IDEF9

Typical business systems.

Scenario-Driven IS Design (Business Constraint Discovery method) — Метод дослідження бізнес обмежень був розроблений для полегшення виявлення й аналізу обмежень в умовах яких діє підприємство. Звичайно, при побудові моделей опису обмежень, що роблять вплив на протікання процесів на підприємстві приділяється недостатня увага. Знання про основні обмеження й характер їхнього впливу, що закладають у моделі, у найкращому разі залишаються неповними, неузгодженими, розподіленими нераціонально, але часто їх зовсім немає. Це не обов'язково приводить до того, що побудовані моделі нежиттєздатні, просто їхня реалізація стикнеться з непередбаченими труднощами, у результаті чого їхній потенціал буде не реалізований. Проте у випадках, коли мова йде саме про вдосконалювання структур або адаптацію до передбачуваних змін, знання про існуючі обмеження мають критичне значення;

IDEF10 - IDEF14

1. IDEF10 — Implementation Architecture Modeling — Моделювання архітектури виконання. Цей метод визначений як затребуваний, однак так і не був повністю розроблений;
2. IDEF11 — Information Artifact Modeling. Цей метод визначений як затребуваний, однак так і не був повністю розроблений;
3. IDEF12 — Organization Modeling — Організаційне моделювання. Цей метод визначений як затребуваний, однак так і не був повністю розроблений;
4. IDEF13 — Three Schema Mapping Design — Трисхемне проектування перетворення даних. Цей метод визначений як затребуваний, однак так і не був повністю розроблений;
5. IDEF14 — Network Design — Метод проектування комп'ютерних мереж, заснований на аналізі вимог специфічних мережевих компонентів, існуючих конфігурацій мереж. Також він забезпечує підтримку рішень, пов'язаних з раціональним керуванням матеріальними ресурсами, що дозволяє досягти істотної економії.

Моделювання бізнес-процесів

Для моделювання бізнес-процесів використовується декілька різних методів, в основі яких лежить як структурний, так і об'єктно- орієнтований підходи до моделювання. Проте, класифікація самих методів на структурні та об'єктні є доволі умовною, оскільки найбільш розвинуті методи використовують елементи обох підходів. Стисло розглянемо характеристики найбільш поширених методів:

- метод функціонального моделювання SADT (IDEF0);
- метод моделювання процесів IDEF3;
- моделювання потоків даних DFD;
- метод ARIS;
- метод Ericsson-Penker;
- метод технології Rational Unified Process.

Метод SADT

Метод SADT (Structured Analysis and Design Technique) вважається класичним методом підходу до управління на основі процесів, базовим принципом якого є структуризація діяльності організації у відповідності з її бізнес-процесами. Бізнес-модель відповідає таким вимогам:

1. верхній рівень моделі відображає виключно контекст системи - взаємодію підприємства із зовнішнім середовищем;
2. другий рівень описує основні види діяльності підприємства - тематично згруповані бізнес-процеси;

3. подальша деталізація бізнес-процесів здійснюється за допомогою бізнес-функцій та елементарних бізнес-операцій, згрупованих за певними ознаками;
4. опис елементарної бізнес-операції відбувається шляхом визначення алгоритму її виконання.

Метод використовується для моделювання штучних систем середньої складності.

Метод моделювання IDEF3

Метод моделювання IDEF3 - частина сімейства стандартів IDEF; використовується для моделювання послідовності виконання дій і їх взаємозалежностей в рамках процесу. Метод отримав визнання серед системних аналітиків як доповнення до методу функціонального моделювання IDEF0.

Основою моделі IDEF3 служить сценарій процесу, який відокремлює послідовність дій і підпроцесів системи. Як і в методі IDEF0, основною одиницею моделі є діаграма. Іншим важливим компонентом є дія або "одиниця роботи" (Unit of Work), взаємодія яких зображається за допомогою зв' язків.

Діаграми потоків даних (Data Flow Diagrams - DFD)

Діаграми потоків даних (Data Flow Diagrams - DFD) представляють собою ієрархію функціональних процесів, що пов'язані потоками даних. Мета такого представлення полягає у демонстрації того, як кожен процес перетворює свої вхідні дані у вихідні і виявлення зв'язків між цими процесами.

Відповідно до методу, модель системи визначається як ієрархія діаграм потоків даних, основними компонентами яких є:

1. зовнішні об' єкти;
2. системи та підсистеми;
3. процеси;
4. накопичувачі даних;
5. потоки даних.

Перший компонент представляє собою матеріальний об' єкт або фізичну особу, яка є джерелом або приймачем інформації; наприклад: замовники, персонал, постачальники, склад.

Метод ARIS (Architecture of Integrated Information System)

Метод ARIS (Architecture of Integrated Information System), представляє собою комплекс засобів аналізу і моделювання діяльності підприємства. Його методичну основу складає сукупність різноманітних методів моделювання, що відображають різні погляди на системи. ARIS підтримує чотири типи моделей, які віддзеркалюють різні аспекти системи, що досліджується:

1. організаційні, що представляють структуру системи;
2. функціональні, які містять ієрархію цілей;

3. інформаційні - відображають структуру всієї інформації, необхідної для реалізації функцій системи;
4. моделі управління, що представляють комплексний підхід до реалізації бізнес-процесів в рамках системи.

Для побудови зазначених типів моделей використовуються як власні методи моделювання ARIS, так і різні відомі методи та мови моделювання, зокрема UML.

Метод Ericsson-Penker

Автори методу Ericsson-Penker створили свій профіль UML для моделювання бізнес-процесів - Ericsson-Penker Business Extensions, ввівши набір стереотипів, які описують основні категорії бізнес-моделі: процеси, ресурси, правила і цілі діяльності підприємства.

Метод Rational Unified Process

Мова UML використовується також в методі, який є частиною технології Rational Unified Process (фірми IBM). Цей метод спрямовано насамперед на створення основи для формування вимог до ПЗ. Передбачає побудову двох базових моделей:

1. моделі бізнес-процесів (Business Use Case Model);
2. моделі бізнес-аналізу (Business Analysis Model).

Модель бізнес-процесів

Модель бізнес-процесів представляє собою розширення моделі варіантів використання (Use Case) UML шляхом введення набору стереотипів - Business Actor (стереотип діючої особи) та Business Use Case (стереотип варіанту використання). Діючими особами можуть бути акціонери, замовники, постачальники, партнери, потенційні клієнти, місцеві органи влади, зовнішні системи, співробітники тих підрозділів організації, діяльність яких не враховується у моделі, тощо.

Business Use Case визначається як опис послідовності дій (потоку) в рамках певного бізнес-процесу, що дає результат для певної діючої особи.

Методи об'єктно-орієнтованого аналізу

Уніфікована мова моделювання (UML - Unified Modeling Language) з'явилась внаслідок розвитку методів об'єктно-орієнтованого аналізу і проектування (OOA&D), що виникли наприкінці 80-х років. Мова UML поєднує в собі методи Граді-Буча (Booch чи Booch'91), Джима Рамбо (Object Modeling Technique - OMT) і Айвара Джекобсона (Object-Oriented Software Engineering - OOSE), проте володіє розширеними можливостями. Мова моделювання пройшла процес стандартизації в рамках консорціуму OMG (Object Management Group) і на сьогоднішній день представляє собою фактичний стандарт OMG.

UML - це назва мови моделювання, але не методу, оскільки більшість методів містять щонайменше мову моделювання та процес. Мова моделювання - це нотація (як правило, графічна), яка використовується методами для опису проектів; процес - це рекомендація щодо етапів, які необхідно виконати при розробці проекту. Таким чином, мова моделювання є найважливішою частиною методу. Якщо проект обговорюється розробниками, всі вони повинні розуміти саме мову моделювання, а не процес, що використовується при розробці проекту. Розробниками мови UML було також створено і RUP (Rational Unified Process) - раціональний уніфікований процес. Причому, при застосуванні мови UML не висувається вимога одночасного використання RUP, оскільки вони є абсолютно незалежними. Процес RUP може використовуватись для розробки проекту в залежності від типу останнього та вимог замовника.

Розглянемо етапи розвитку галузі розробок програмного забезпечення для ілюстрації процесу становлення мови UML. Поняття об'єкту набуло практичного значення у проектуванні в 70-х - 80-х роках і стало основою методів об'єктно-орієнтованих розробок. В період з 1988 по 1992 роки з'явились наступні праці, присвячені методам об' єктно-орієнтованого аналізу і проектування:

- роботи Саллі Шлеєр (Sally Shlaer) та Стіва Меллора (Steve Mellor), що пізніше були втілені у рекурсивному проектуванні (Recursive Design);
- Пітер Коуд (Peter Coad) та Ед Йордон (Ed Yourdon) розробили неформальний та орієнтований на прототипи метод Коуда;
- Граді Буч (Grady Booch) з компанії Rational Software написав фундаментальну роботу по розробці систем на мові Ada;
- Джим Рамбо (James Rumbaugh) очолив групу у дослідній
- лабораторії General Electric і розробив популярний метод Object Modeling Technique - OMT;
- Айвар Джекобсон (Ivar Jacobson) виклав свій досвід роботи з телефонними системами фірми Ericsson і вперше ввів поняття варіанту використання (Use Case).

До 1995 року серед методів аналізу і проектування спостерігалось різноманіття підходів та жорстка конкуренція. На той час кожний з авторів методів (про яких згадувалось вище), був лідером групи розробників- практиків, що підтримували їх ідеї. У період 1989-1994 р. загальне число найбільш відомих мов моделювання зросло з десяти до більш ніж п'ятдесят. Багато користувачів відчували істотні труднощі при виборі мови моделювання, оскільки жодна з них не задовольняла всім вимогам, що висуваються до побудови моделей складних систем. Прийняття окремих методик і графічних нотацій як стандартів (IDEF0, IDEF1X) не змогло змінити сформовану ситуацію непримиренної конкуренції між ними на початку 90-х років, що одержала назву "війни методів".

Всі ці методи були подібні по суті, проте розрізнялись у вторинних деталях, що викликало плутанину у замовників. Наприклад, різне представлення одної і тої самої графічної нотації суттєво ускладнювало розуміння моделі, побудованої з

використанням різних методів. Виникла необхідність стандартизації усіх існуючих методів проектування.

Проте, у відповідь на пропозицію звести всі методи до єдиного стандарту, компанія OMG отримала відкритий лист із протестом від всіх авторів основних методологій. Тоді Джим Рамбо залишив General Electric і приєднався до Буча в Rational Software з наміром об'єднати їх методи та досягти стандартизації за прикладом компанії Microsoft. Всі, хто не погодився з таким рішенням, сформували анти-Бучівську коаліцію. На конференції OOPSLA'95 Буч та Рамбо представили перший опис об'єднаного методу у формі документації під робочою назвою "Unified Method 0.8". В цей самий час фірма Rational Software здійснила придбання компанії Objectory і до розробників уніфікованого методу приєднався Джекобсон. Розробка нового методу - вже під назвою UML - тривала до 1997 року, причому з відкритим несхваленням зі сторони голови OMG.

Тоді ж деякі компанії та організації побачили в мові UML стратегічний інтерес для свого бізнесу. Компанія Rational Software разом з декількома організаціями, що виявили бажання виділити ресурси для розробки строгого визначення версії 1.0 мови UML, заснувала консорціум партнерів UML, у який спочатку ввійшли такі фірми, як Digital Equipment Corp., HP, i-Logix, Intellicorp, IBM, ICON Computing, MCI Systemhouse, Microsoft, Oracle, Rational Software, TI і Unisys. Ці компанії забезпечили підтримку подальшої роботи з більш точного визначення нотації.

В січні 1997 року ряд організацій представив свої пропозиції по стандартизації методів обміну інформацією між різними моделями. Серед тих пропозицій була і документація на мову UML 1.0. Після внесення декількох істотних доповнень версія 1.1. мови UML була обрана в якості офіційного стандарту OMG.

На даний час усі питання подальшої розробки мови UML сконцентровані в рамках консорціуму OMG. При цьому статус мови UML визначений як відкритий для всіх пропозицій по його доробці та удосконаленню. Сама мова UML не є власністю і не запатентована, хоча зазначений вище документ є захищений законом про авторське право. У той же час абревіатура UML, як і деякі інші (OMG, CORBA, ORB), є торговою маркою їх законних власників.

На ринку CASE-засобів представлені десятки програмних інструментів, що підтримують нотацію мови UML і забезпечують інтеграцію, включаючи пряму і зворотну генерацію коду програм, з найбільш розповсюдженими мовами і середовищами програмування, такими як MS Visual C++, Java, Object Pascal/Delphi, Power Builder, MS Visual Basic, Forte, Ada, Smalltalk.

З кожним роком інтерес до мови UML з боку фахівців неухильно зростає. Мова UML поступово стає не тільки основою для розробки і реалізації в багатьох перспективних інструментальних засобах, але і у CASE-засобах візуального та імітаційного моделювання. Більш того, закладені в мові UML потенційні можливості широко використовуються як для об' єктно-орієнтованого моделювання систем, так і для документування бізнес-процесів, а в більш

широкому контексті - для представлення знань в інтелектуальних системах, якими в перспективі стануть складні програмно-технологічні комплекси.

Представлення про модель в рамках мови UML

У рамках мови UML усі представлення про модель складної системи фіксуються у вигляді спеціальних графічних конструкцій, що одержали назву діаграм:

Діаграма (diagram) - графічне представлення сукупності елементів моделі у формі зв'язного графа, вершинам і ребрам (дугам) якого приписується визначена семантика

Нотація канонічних діаграм є основним засобом розробки моделей мовою UML.

Нотація множини символів і правила їх застосування, що використовуються для представлення понять і зв'язків між ними.

У нотації мови UML визначені наступні види канонічних діаграм:

варіантів використання (use case diagram);
класів (class diagram);
кооперації (collaboration diagram);
послідовності (sequence diagram);
станів (statechart diagram);
діяльності (activity diagram);
компонентів (component diagram);
розгортки (deployment diagram);

Перелік цих діаграм і їх назв є канонічними в тому сенсі, що являють собою невід'ємну частину графічної нотації мови UML. Більше того, процес об'єктно-орієнтованого проектування нерозривно пов'язаний із процесом побудови цих діаграм. Сукупність побудованих у такий спосіб діаграм є самодостатньою в тому сенсі, що в них міститься вся інформація, яка необхідна для реалізації проекту складної системи (рис.18.1).

Кожна з цих діаграм деталізує і конкретизує різні представлення про модель складної системи в термінах мови UML. При цьому діаграма варіантів використання являє собою найбільш загальну концептуальну модель складної системи, що є вихідною для побудови всіх інших діаграм. Діаграма класів, по своїй суті - логічна модель, що відбиває статичні аспекти структурної побудови складної системи.

Діаграми кооперації і послідовностей являють собою різновид логічної моделі, що відображають динамічні аспекти функціонування складної системи. Діаграми станів і діяльності призначені для моделювання поведінки системи. Діаграми компонентів і розгортання служать для представлення фізичних компонентів складної системи і тому відносяться до її фізичної моделі. Крім графічних елементів, що визначені для кожної канонічної діаграми, на них може бути зображена текстова інформація, що розширює семантику базових елементів.

Нотація насамперед є синтаксисом мови моделювання. Наприклад, нотація діаграми класів показує, як саме визначаються такі елементи і поняття, як клас, асоціація та кратність (рис.18.2-18.5).

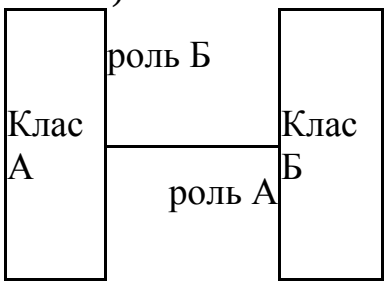


Рис.18.2. Асоціація у визначенні нотації діаграми класів

Відповідно, виникає потреба у точному визначенні самих елементів та понять, оскільки, як правило, розробники моделей використовують неформальні визначення.

Ідея строгих мов для специфікації і проектування є найбільш поширеною в області формальних методів. В них відповідні означення є математично строгими і виключають неоднозначність. Проте, такі визначення не є універсальними: навіть якщо програмна реалізація відповідає математичній специфікації, не існує способу довести, чи дійсно ця специфікація відповідає реальним вимогам системи.

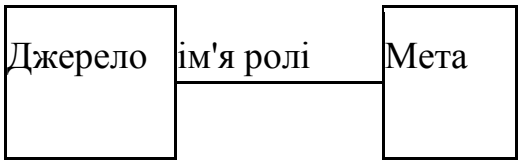


Рис.18.4. Навігація у визначенні нотації UML

Проектування повинно базуватись на всесторонньому аналізі всіх ключових питань розробки. Використання формальних методів зазвичай призводить до того, що проект містить масу другорядних деталей. Крім того, формальні методи проектування є складнішими для розуміння, ніж мови програмування. До того ж, формальні методи не можуть виконуватись, як програмні продукти.

Клас Б

Рис.18.5. Поняття залежності у визначенні нотації діаграми класів

Більшість об' єктно-орієнтованих методів не характеризуються строгістю: їх нотації радше спрямовані на інтуїтивне розуміння, аніж на формальне визначення. Розробники об' єктно-орієнтованих методів шукають способи досягти більшої строгості методів без втрати практичності моделі.

Клас А

Один з таких способів полягає у визначенні деякої метамоделі: діаграми (як правило, діаграми класів), яка визначає нотацію. Метамодель допомагає

визначити, чи побудована модель є синтаксично правильною, проте для практичного застосування нотації UML глибоке розуміння метамоделей не є обов'язковим. Вимога вільного оперування метамоделями висувається лише для спеціалістів, що мають високу кваліфікацію в області моделювання.

Лекція 19. Поведінкове, структурне та функціональне моделювання

- Поведінкове моделювання.
- Діаграми станів, діяльності, взаємодії, послідовності, часові.
- Структурне моделювання.
- Функціональне моделювання. [4, с. 506-537]

Поведінкове моделювання

Головне завдання поведінкового моделювання – це моделювання процесів, які протікають в моделюючому середовищі. При використанні цього моделювання поведінка системи розглядається як обмін повідомленнями між об’єктами для досягнення певної мети. Різні сторони взаємодії об’єктів відображаються у діаграмах різного типу:

- діаграми станів (state diagrams);
- діаграми діяльності (активності, activity diagrams);
- діаграми взаємодії (collaboration diagrams);
- діаграми послідовності (sequence diagram);
- часові діаграми (синхронізації, timing diagrams).

Діаграми станів

Діаграми станів демонструють динаміку зміни станів об’єктів під впливом перебігу подій. Тобто вони застосовуються для того, щоб пояснити,яким чином працюють складні об’єкти. Класичне визначення поняття “стан” (state)– це ситуація в життєвому циклі об’єкта, під час якої він задовольняє деякі умови, виконує певну діяльність або очікує якусь подію. Стан визначається значеннями деяких атрибутіві присутністю чи відсутністюзв’язків з іншими об’єктами.

Діаграма станів показує, як об’єкт переходить з одного стану в інший. Очевидно, що вони служать для моделювання динамічних аспектів системи (як і діаграми послідовностей, взаємодії, прецедентів, діяльності). Діаграма станів корисна при моделюванні життєвого циклу. Від інших діаграм вона відрізняється тим, що описує процес зміни станів тільки одного примірника певного класу – одного об’єкта, причому об’єкта реактивного, тобто такого, поведінка якого характеризується його реакцією на зовнішні події. Поняття життєвого циклу застосовується саме до реактивних об’єктів, даний стан (і поведінку) яких обумовлено їх минулим станом. Але діаграми станів важливі не тільки для опису динаміки окремого об’єкта. Вони можуть використовуватися для конструювання виконуваних систем шляхом прямого і зворотного проектування.

Модель станів UML базується на використанні розширеної моделі скінченного автомата. Нею визначаються:

- умови переходів (застереження –guards on transitions);
- переходи, зумовлені певними подіями;

- дії при переході;
- дії при вході в стан;
- діяльність, яка триває доти, доки стан є активним;
- дії при виході зі стану;
- вставка станів;
- паралельно діючі стани.

Приклад використання діаграми станів на рис. 19:

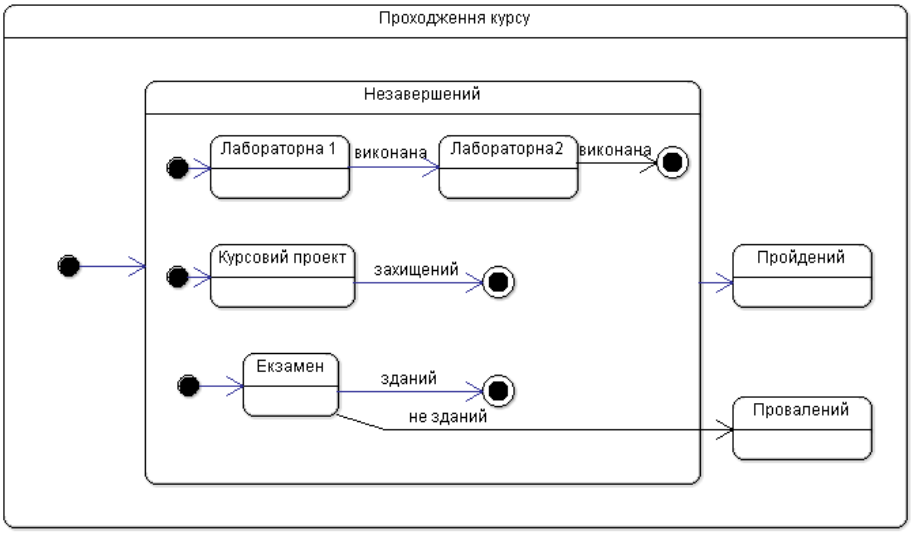


Рис.19 Діаграма станів

Діаграми діяльності

Діаграми діяльності демонструють потоки керування при взаємодії об’єктів та являються окремим випадком діаграм станів. Їх зручно застосовувати для візуалізації алгоритмів, по яких працюють операції класів. Модель діяльності в UML являє собою поведінку системи як певні роботи, котрі можуть виконувати як система, так і актор, причому послідовність робіт може залежати від прийняття певних рішень залежно від умов, що склалися. Окрема діяльність зображується на діаграмі прямокутником із закругленими кутами. Потоки керування між роботами показуються стрілками. Якщо мова йде про прийняття рішення, то з відповідного прямокутника виходять дві стрілки, на кожній може позначатися текст умови, якій вона відповідає. Діаграма діяльності нагадує відомі блок-схеми алгоритмів та програм, зокрема передбачено відображення можливості виконувати паралельно кілька діяльностей і точки синхронізації завершення їх.

Приклад діаграми діяльності на рис. 20:

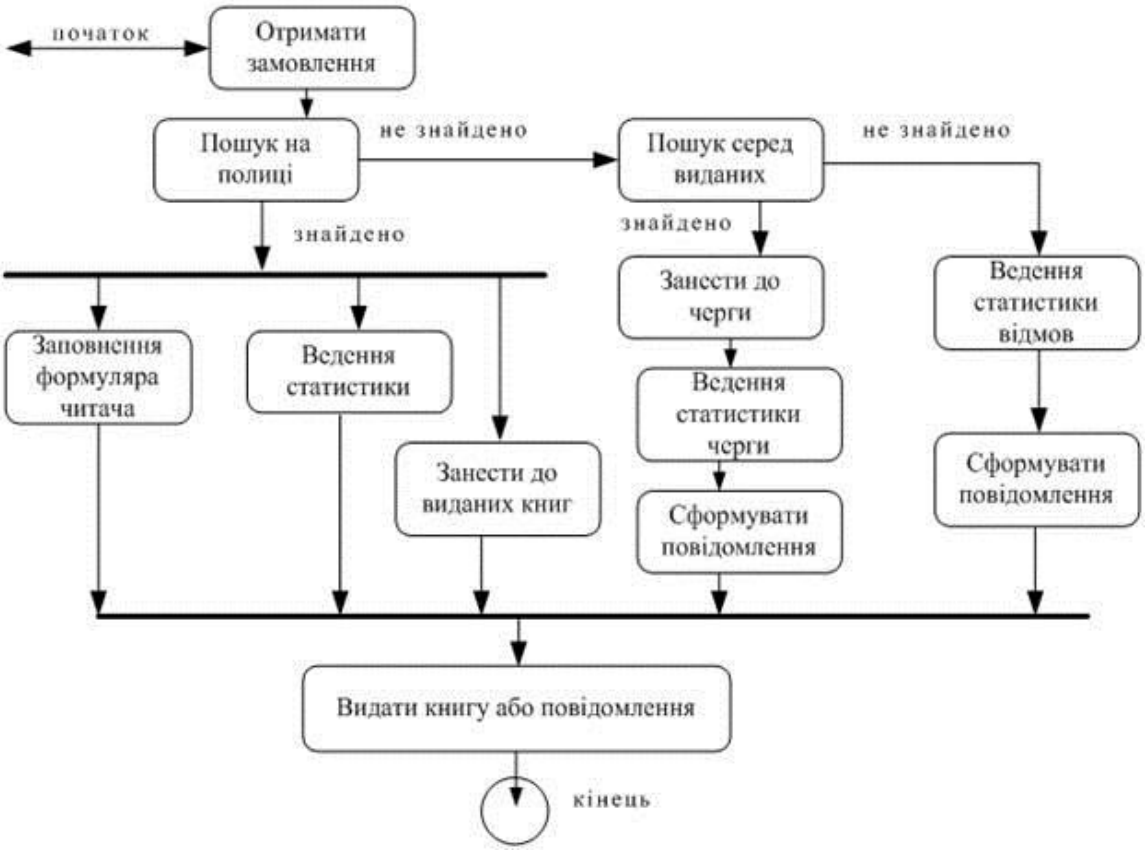


Рис. 20 Діаграма діяльності

Нехай пошук замовленої книги в бібліотеці має супроводжуватися кількома супутніми процедурами, а саме збиранням даних про виконані та невиконані замовлення. Горизонтальні лінії позначають розпаралелювання та синхронізацію окремих робіт.

Діаграма може відображати той факт, що певна діяльність виконується для кожного існуючого екземпляра об’єкта, наприклад, для кожного рядка замовлення (якщо замовляється кілька книжок одразу). Тоді стрілка, яка веде до такої діяльності, позначається зірочкою.

Діаграма послідовності

Діаграма послідовності показує послідовність, в якій об’єкти в процесі взаємодії обмінюються повідомленнями. Варто зазначити, що повідомлення – специфікація передачі інформації від одного об’єкта до другого. Сам об’єкт позначається прямокутником, усередині якого вказані підкреслені його ім’я і назва класу (не обов’язково), розділені двокрапкою. Об’єкти розташовуються у верхній частині діаграми один за одним. Вниз від кожного з них тягнеться пунктирна лінія – лінія життя, яка зображує існування об’єкта протягом певного проміжку часу. Повідомлення, якими обмінюються об’єкти, зображуються у вигляді стрілок, спрямованих від лінії життя одного з них до лінії життя іншого. Ці лінії грають роль шкали часу, таким чином, послідовність повідомлень легко

читається “зверху вниз”. Об’єкт відправляє повідомлення в розрахунку на те, що воно викличе якусь реакцію і за ним відбудеться деяка діяльність.

Довгими переривчастими смугами на лініях життяпозначаються періоди часу, коли об’єкт має фокус управління, тобто виконує деяку дію. Проте фокус часто не зображують, оскільки він легко може бути відновленим.

Якщо об’єкт в процесі взаємодії руйнується, цей факт позначають на його лінії життя хрестиком, який її і закінчує.

Приклад зображений на рис. 21а:

Рис. 21 Діаграма послідовності

Самі по собі повідомлення бувають синхронними і асинхронними. Синхронні припиняють потік виконання до тих пір, поки не буде отримана відповідь. Банк виносить рішення про надання кредиту і повідомляє його вам, касир видає підтвердження платежу – чек. Інший вид повідомлень – асинхронні. Вони не чекають відповіді, не припиняють потік виконання, відразу після їх відсилання відбувається негайний перехід до наступного кроку і послідовність триває. Рефлексивне повідомлення малюють, якщо потрібно показати дію, що виконується самим об’єктом (або всередині нього), або те, що об’єкт сам себе вводить в деякий стан. Повідомлення-відповіді зазвичай зображують пунктирною лінією зі стрілкою, хоча часто вони мають такий же вигляд, як і звичайні, тільки направлені в протилежний бік. Стрілки для зображення різних типів повідомлень на рис. 4:

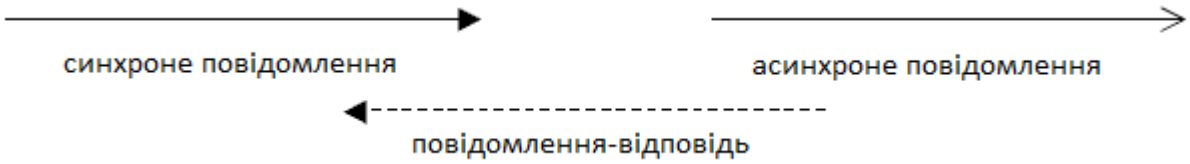


Рис. 21а Типи повідомлень

Можливі випадки, коли нам відомий адресат повідомлення, але невідомий його відправник – такі називають знайденими. Або навпаки: відправник відомий,

а одержувач – ні. Це – записки в пляшках, які колись кидали в море жертви корабельних аварій. Такі повідомлення називають втраченими.

На діаграмах послідовностей, так само як і на діаграмах взаємодії, показуються ролі класів. Але на першій вони показані з точки зору тимчасового аспекту, а на другій – з точки зору відносин взаємодіючих частин. Можна сказати, що діаграма послідовностей є двійником діаграм взаємодії.

Рекомендації з побудови діаграм послідовності:

- виділити лише ті класи, об’єкти яких беруть участь в модельований вами взаємодії. Потім всі об’єкти нанести на діаграму. Слід визначити ті з них, які будуть існувати постійно, і ті, які будуть існувати тільки під час виконання ними дій;
- зобразити повідомлення і стереотипи. Для знищення об’єктів потрібно передбачити спеціальні повідомлення;
- дотримуйтесь балансу між деталізацією і складністю: краще кожен альтернативний потік управління показувати на окремій діаграмі.

Часові діаграми

Часові діаграми є різновидом діаграм послідовностей і призначені для наочного зображення потоку зміни станів декількох ролей (класів, компонент). Останні зображуються не вертикально, а горизонтально, і основна увага робиться на те, як стани змінюються в часі. Така можливість корисна, наприклад, при моделюванні вбудованих систем.

На рис. 5 показаний фрагмент роботи системи AccessControl, яка управляє відкриттям/блокуванням дверей при пред’явленні людиною електронного ключа. Показано три компоненти цієї системи.Перша, panel, є пристроєм з дисплеєм для відображення поточного стану всієї системи і пристроєм зчитування електронного ключа. Спочатку panel знаходиться в стані locked (відповідний напис відображається на дисплеї). Після того як людина приклала електронний ключ і було зчитано інформацію, panel посилає її у вигляді повідомлення verify другому компоненту – процесору (access_processor)і переходить у стан waiting. Процесор до отримання повідомлення verify знаходиться в стані idle, а після його отримання переходить до стануverifying.

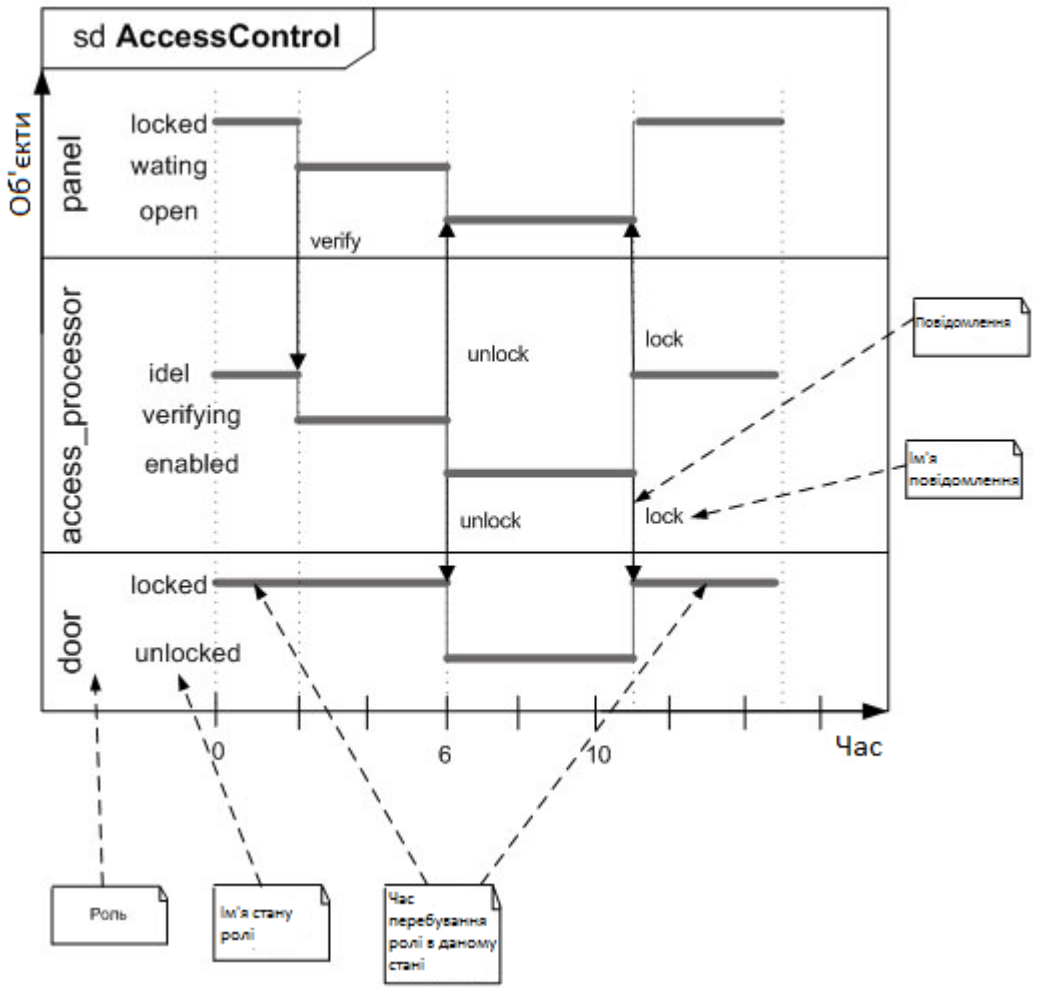


Рис. 22 Діаграма опису роботи системи контролю доступу

Після успішного закінчення перевірки процесор посилає компонентам panel і door повідомлення unlock і переходить до стану enable. Компонента panel переходить в стан open. Третя компонента, door (власне, самі двері), до цього перебувала в стані locked і, отримавши повідомлення unlock, відкривається (переходить в стан unlock). Відкритою вона залишається рівно 5 секунд, після чого процесор надсилає їй команду lock і вона закривається – знову переходить до стану locked. Одночасно процесор посилає команду lock також і компоненті panel, яка переходить у свій вихідний стан locked і відображає слово “locked” на дисплеї.

Діаграма взаємодії

Діаграма взаємодії призначена для опису поведінки системи на рівні окремих об’єктів, які обмінюються між собою повідомленнями, щоб досягти потрібної мети або реалізувати певний варіант використання. Власне, слово “взаємодія” означає “кооперація”, “співпраця”. Такі діаграми показують, як об’єкти працюють разом, акцентуючись на їх ролях. З точки зору аналітика або

архітектора системи в проекті важливо представити структурні зв’язки окремих об’єктів між собою. Таке уявлення і забезпечує діаграма взаємодії.

Зазвичай вони застосовуються для того, щоб:

- показати набір взаємодіючих об’єктів “з висоти пташиного польоту”;
- розподілити функціональність між класами, ґрунтуючись на результатах вивчення динамічних аспектів системи;
- описати логіку виконання складних операцій, особливо в тих випадках, коли один об’єкт взаємодіє ще з кількома;
- вивчити ролі, а також відносини між об’єктами, в які вони залучаються, виконуючи ці ролі.

На діаграмі взаємодії розміщуються об’єкти, що представляють собою екземпляри класів. Далі, як і на діаграмі класів, показуються структурні відносини між об’єктами у вигляді різних сполучних ліній. Зв’язки можуть доповнюватися іменами ролей, які відіграють об’єкти. І, нарешті, зображуються динамічні взаємозв’язки – потоки повідомлень у формі стрілок з зазначенням напрямку поруч з сполучними лініями між об’єктами, при цьому задаються імена повідомлень та їх порядкові номери в загальній послідовності повідомлень.

Одна і та ж сукупність об’єктів може брати участь в реалізації різних кооперацій. В залежності від даної кооперації, можуть змінюватися як зв’язки між окремими об’єктами, так і потік повідомлень між ними. Саме це відрізняє діаграму взаємодії від діаграми класів, на якій повинні бути вказані всі без винятку класи, їх атрибути і операції, а також всі асоціації та інші структурні відносини між елементами моделі.

Об’єкти – сутність з добре визначеними кордонами і індивідуальністю, яка інкапсулює стан і поведінку.Для діаграм взаємодії повне ім’я об’єкта в цілому являє собою рядок тексту, розділений двокрапкою і записаний в форматі:

<Власне ім’я об’єкта>‘/’<Ім’я ролі класу>: <Ім’я класу>.

Важливо відзначити, що весь запис імені об’єкта підкреслюється.Зв’язок на діаграмі – будь-яке семантичне відношення між деякою сукупністю об’єктів.

На відміну від діаграми послідовності, на діаграмі взаємодії явно вказуються відносини між об’єктами. Вони мають вільний формат упорядкування об’єктів і зв’язків. Щоб підтримувати порядок повідомлень при такому форматі, їх хронологічно нумерують. Читання діаграми починається з повідомлення 1.0 і продовжується у напрямку пересилання повідомлень.

Приклад діаграми взаємодії зображено на рис. 23:

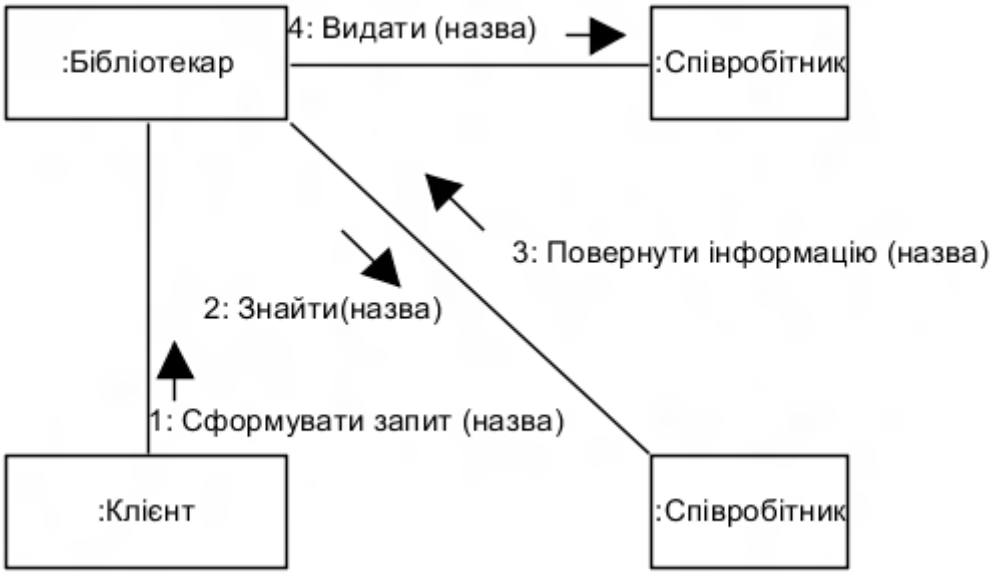


Рис. 23 Діаграма взаємодії

Діаграма взаємодії показує багато в чому схожу інформацію, що і діаграма послідовності, але через відмінність способу подання інформації деякі речі на одній з них бачити простіше, ніж на іншій. Ця діаграма наочніше показує, з якими елементами взаємодіє інший елемент, а діаграма послідовності ясніше відображає в якому порядку відбуваються взаємодії.

Рекомендації з побудови діаграм взаємодії:

1. побудова діаграми починається відразу після побудови діаграми класів. Спочатку зображуються об’єкти і зв’язки між ними. Далі необхідно нанести всі повідомлення, вказавши їх порядок і інші семантичні особливості. Діаграма взаємодії може містити тільки ті об’єкти та зв’язки, які вже визначені на побудованій раніше діаграмі класів. В іншому випадку, якщо необхідно додати об’єкти, які створюються на основі відсутніх класів, то діаграма класів повинна бути модифікована.
2. побудова діаграми взаємодії повинна бути узгоджена з побудовою діаграми послідовності. Не допускається різний порядок проходження повідомлень для моделювання однієї і тієї ж взаємодії на діаграмах.

Структурне моделювання

Моделювання класів

Модель класів описує статичну структуру системи: об'єкти і відношення між ними, атрибути та операції для кожного класу об'єктів. Модель класів - найважливіша з трьох основних моделей. Ми вважаємо, що в основі системи повинні бути об'єкти, тому що об'єктно-орієнтована система краще відповідає

реальному світу і опиняються більш життєздатною при можливих змінах. Моделі класів являються інтуїтивним графічним представленням системи і тому особливо корисні для спілкування із замовниками.

UML. Концепції об'єкта і класу

1. Об'єкти

Об'єкт (object) - це концепція, абстракція або сутність, що володіє індивідуальністю і має сенс в рамках програми. Об'єкти часто бувають іменами власними або конкретними посиланнями, які використовуються в описі задач або при спілкуванні з користувачами. Деякі об'єкти існують або існували в реальному світі (наприклад, Альберт Ейнштейн чи компанія General Electric), тоді як інші є суто концептуальними сутностями. Об'єкти третього типу додаються в модель в процесі реалізації і не мають ніякоговідношення до фізичної реальності. Вибір об'єктів залежить від природи задачі і від уподобань розробника. Коректне рішення в даному випадку не є єдиним. Всі об'єкти мають індивідуальність і тому відрізняються один від одного.Два яблука однакового кольору, форми і текстури все одно є різними яблуками. Ви можете з'їсти спочатку одне з них, а потім друге. Схожі один на одного близнюки теж є незалежними індивідуальностями. Індивідуальність означає, що об'єкти відрізняються один від одного внутрішньо, а не за зовнішніми властивостями.

2. Класи

Об'єкт є екземпляром класу. Клас (class) описує групу об'єктів з однаковими властивостями (атрибутами), однаковим поведінкою (операціями), типами відносин і семантикою. Як приклади класів можна навести такі: людина, компанія, процес і вікно. Кожна людина має ім'я і дату народження, а також може будь-де працювати. Кожен процес має власника, пріоритет і список необхідних ресурсів. Класи часто бувають іменами загальними та іменними групами, які використовуються в описі задач або при спілкуванні з користувачами. Об'єкти одного класу мають однакові атрибути і форми поведінки. Більшість об'єктів відрізняються один від одного значеннями своїх атрибутів і відносинами з іншими об'єктами. Однак можливе існування різних об'єктів з однаковими значеннями атрибутів, що знаходяться в однакових відносинах з усіма іншими об'єктами. Вибір класів залежить від природи і області застосування програми і часто є суб'єктивним.

Кожен об'єкт «знає» свій власний клас. Більшість об'єктно-орієнтованих мов програмування дозволяють визначати клас об'єкта під час виконання програми. Клас об'єкта - це його неявна властивість. Якщо предметом моделювання є об'єкти, то чому взагалі заходить мова про класи? Вся справа в необхідності абстрагування. Групуючи об'єкти в класи, ми виробляємо абстрагування в рамках завдання. Саме завдяки абстракцій моделювання виявляється настільки потужним інструментом, дозволяючи проводити узагальнення від кількох конкретних випадків до безлічі подібних альтернатив.

Загальні визначення (такі як назва класу і назви атрибутів) зберігаються окремо для кожного класу, а не для кожного екземпляра.

3. Діаграми класів

Ми обговорили деякі основні концепції моделювання, а саме об'єкт і клас. Ми розповідали про них і приводили приклади. Такий підхід не дає достатньої чіткості і непридатний для опису складних додатків. Нам потрібно засіб опису моделей, який був би узгодженим, точним і простим у використанні. Моделі структури бувають двох типів: діаграми класів і діаграми об'єктів. Діаграми класів дозволяють описати модель класів і їх за допомогою графічної системи позначень. Діаграми класів корисні як для абстрактного моделювання, так і для проектування конкретних програм. Вони точні, прості для розуміння і добре зарекомендували себе на практиці

Асоціації класів. Узагальнення

Наслідування є однією з фундаментальних основ об’єктно-орієнтованого програмування, у якому клас “отримує” всі атрибути і операції класу, нащадком якого він є, і може визначати заново або змінювати деякі з них, а також додавати власні атрибути і операції.

Асоціації

Асоціація означає взаємозв’язок між класами, вона є базовим семантичним елементом і структурою для багатьох типів “з’єднань” між об’єктами.

Асоціації є тим механізмом, який надає об’єктам змогу обмінюватися даними між собою. Асоціація описує з’єднання між різними класами (з’єднання між дійсними об’єктами називається об’єктним з’єднанням, або зв’язком).

Агрегація

Агрегації є особливим типом асоціацій, за якого два класи, які беруть участь у зв’язку не є рівнозначними, вони мають зв’язок типу “ціле-частина”. За допомогою агрегації можна описати, яким чином клас, який грає роль цілого, складається з інших класів, які грають роль частин. У агрегаціях клас, який грає роль цілого, завжди має численність рівну одиниці.

Композиція

Композиції — це асоціації, які відповідають дуже сильній агрегації. Це означає, що у композиціях ми також маємо справу з співвідношеннями ціле-частина, але тут зв’язок є настільки сильним, що частини не можуть існувати без цілого. Вони існують лише у межах цілого, після знищення цілого буде знищено і його частини.

IDEF4

IDEF4 є об'єктно-орієнтоване метод проектування для розробки компонентів на основі клієнт / сервісних системи. Він був розроблений для підтримки плавного переходу від предметної області і моделі аналізу вимог до проектування і до створення вихідного коду. Він визначає дизайн об'єктів з досить докладними деталями, щоб згенерувати код.

Структура IDEF4

IDEF4 користувачі дизайн в трьох різних шарів: (1) проектування системи, (2) розробки програм, і (3) дизайну найнижчого рівня. Ця тришарова організація знижує складність конструкції. Рівень проектування системи забезпечує зв'язок зіншими системамив контексті дизайну. На прикладному рівні зображує взаємодію міжкомпонентами системи, що розробляється. Ці компоненти включають в себе комерційні додатки, які були раніше розроблені та впровадженні програми. Рівень низького дизайну представляє основу об'єктів системи.

Об'єкти

Об'єкти можуть бути екземплярами класів та розділів. Наприклад працівник з іменем “Євген” і автомобіль з номером "BYJ14T" є прикладами об'єктів.

Класи

Узагальнене поняття про об'єкт і використовується для управління системою об’єктів, використовується для групування, скориставшись схожістю об'єктів. Набагато ефективніше скласти загальну характеристику та інформацію про схожі об’єкти, чим описувати кожний об’єкт окремо.

Підклас.Суперклас

Суперклас - це клас, відякогобуде відбуватися наслідування, а підклас – цеклас, який успадкував якості суперкласу. Коли наслідування здійснено, підклас властивості суперкласу плюс його власні властивості.

Діаграма класів

На діаграмах класів буде показано різноманітні класи, які утворюють систему і їх взаємозв’язки. Діаграми класів називають “статичними діаграмами”, оскільки на них показано класи разом з методами і атрибутами, а також статичний взаємозв’язок між ними.

Рис. 24 Діаграма класів

Функціональне моделювання за допомогою IDEF0

Методологія функціонального моделювання IDEF0

Методологія функціонального моделювання IDEF0 – це технологія опису системи в цілому як множини взаємопов'язаних дій або функцій. Важливо зазначити функціональну направленість: IDEF0-функції системи досліджуються незалежно від об'єктів, які забезпечують їх виконання. «Функціональна» точка зору дозволяє легко відділити аспекти призначення системи від аспектів її фізичної реалізації. На рисунку 25 приведено приклад типічної діаграми IDEF0.

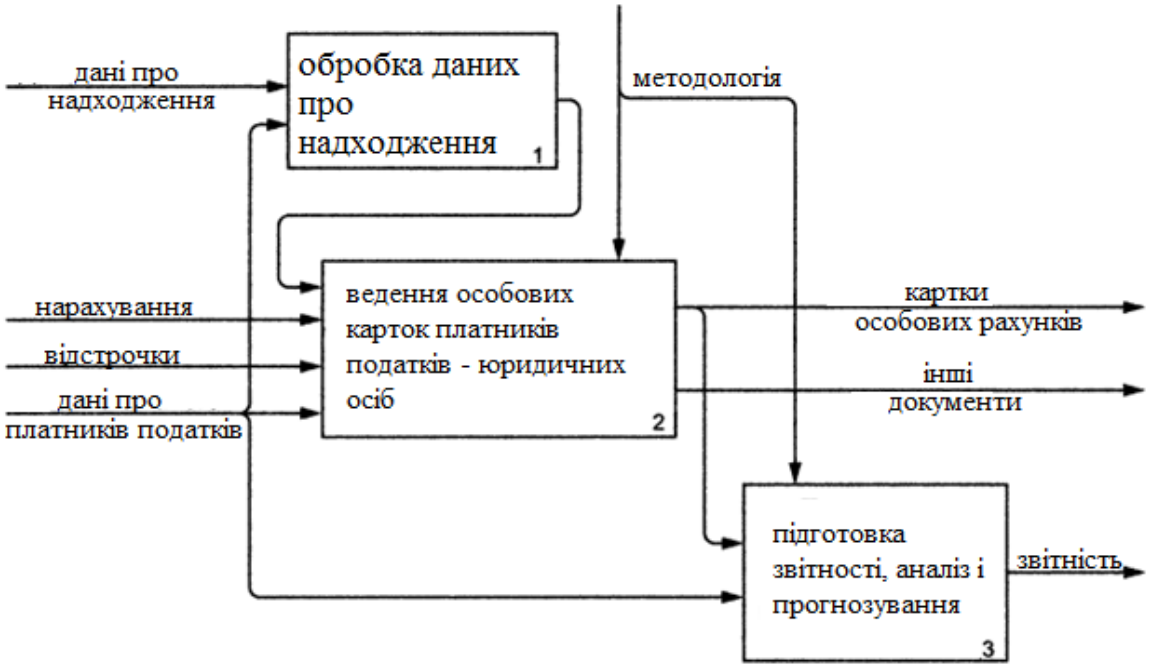


Рис. 25 Приклад діаграми IDEF0

IDEF0 поєднує в собі невелику по об'єму нотацію (вона містить в собі тільки два позначення: блоки та стрілки) зі строгими і чітко визначеними

рекомендаціями, призначеними дял побудування якісної і зрозумілої моделі системи.

Методологія IDEF0 в деякій мірі нагадує рекомендації, існуючі в видавничій справі: часто набір надрукованих IDEF0-моделей організовується в брошуру (в термінах IDEF0-*комплект*), яка має зміст, глосарій, та інші елементи, характерні для завершеної книги.

Кроки при побудові моделі IDEF0:

- Визначити *призначення* моделі – набір питань, на які повинна відповісти модель (можна порівняти з передмовою книги).
- Визначити *границі моделювання* – для позначення ширини охоплення предметної області і глибини деталізації. Вони є логічним продовженням вже визначеного призначення моделі.
- Визначення *цільової аудиторії* – для якої створюється модель. Часто від неї залежить рівень деталізації, з якою повинна створюватись модель.
- Визначення точки зору – під якою розуміється перспектива, з якої спостерігається система при побудові моделі. Обирається з врахуванням вже обраних границь моделі.

Дія, в IDEF0 звичайно називається функцією, оброблює або переводить вхідні параметри і виводить вихідні. Так як моделі IDEF0 моделюють систему як множину вкладених функції, в першу чергу повинна бути визначена функція, як описує систему в цілому – *контекстна функція*. Функції зображуються як іменовані прямокутники або функціональні блоки. Імена підбираються таким чином, щоб характеризувати модель з обраної точки зору.

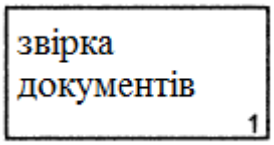


Рис. 26. Функціональний блок

Опис будь якого блоку повинен містити опис як мінімум двох об’єктів – вхідного і вихідного. В IDEF0 також моделюються *управління* та *механізми виконання* .Під управлінням розуміють об’єкти, які впливають на спосіб, яким блок перетворює вхідні у вихідні дані. Механізм виконання – об’єкти, які виконують перетворення вхідних даних у вихідні, але самі не змінюються. Для типізації категорій інформації в IDEF0 діаграмах використовується абревіатура ICOM:

I(Input) – вхід – вхідні дані.

C(Control) – управління – обмеження та інструкції, які впливають на виконання процесу.

O(Output) – вихід – те, що являє собою результат виконання процесу.

M(Mechanism) – виконуючий механізм – те, що використовується для процесу, але не змінюється.

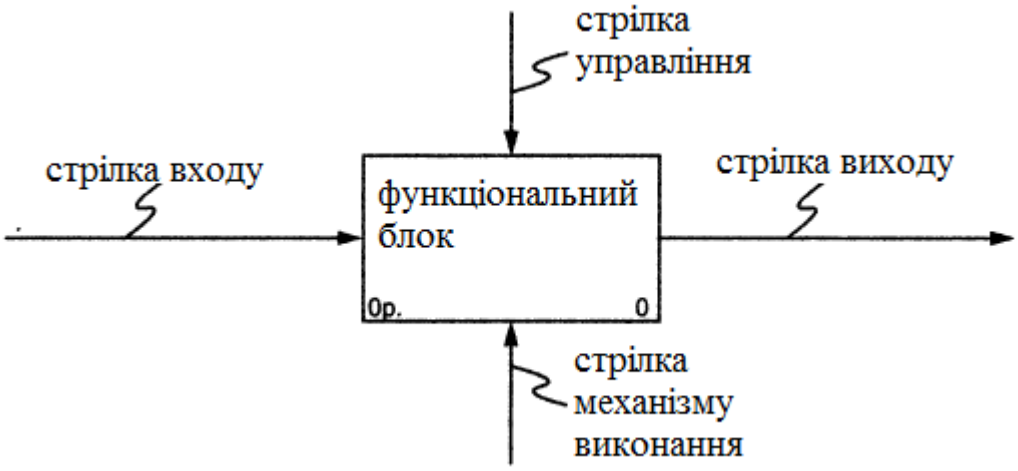


Рис. 27 Типи стрілок і їх розміщення

Стрілки *входу* можуть бути відсутніми на діаграмі, так як рішення може прийматись на основі деяких факторів, які ніяким чином не перетворюються в інші види даних. Стрілки *управління* відповідають за те, як і коли буде виконуватись певний блок. Кожний функціональний блок повинен мати як мінімум одну стрілку управління. Управління можна розглядати як специфічний вид входу. Стрілки *виходу* – продукція, отримана в результаті роботи блоку. Кожний блок обов'язково повинен мати як мінімум один вихід. Дія, яка не має чітко визначеного виходу краще не моделювати взагалі. Стрілки *механізму виконання* є ресурсом, який безпосередньо виконує завдання. За допомогою механізмів можуть моделюватись: ключовий персонал, техніка та/або обладнання. Стрілки механізму можуть бути відсутніми.

Різні стрілки можуть комбінуватись в IDEF0 діаграмах: вихід-вхід, вихід-управління, вихід-механізм виконання, вихід-зворотний зв'язок на управління, вихід-зворотній зв'язок на вхід.

Стрілки можуть бути роз'єднані. Від'єднанні частини стрілок можуть мати власні назви. Початкова і роз'єднана стрілки мають назву *зв'язані*. Аналогічно стрілки можуть бути об'єднані. Це використовується для того, що б відділити частину певну частину даних.

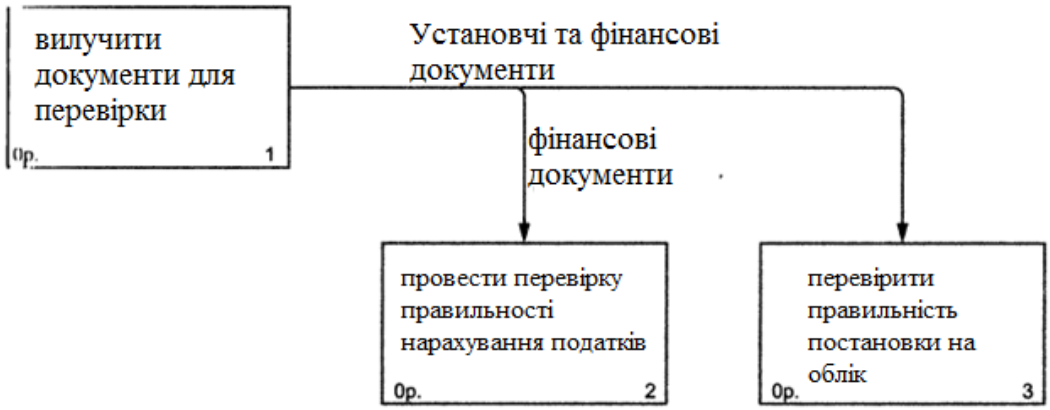
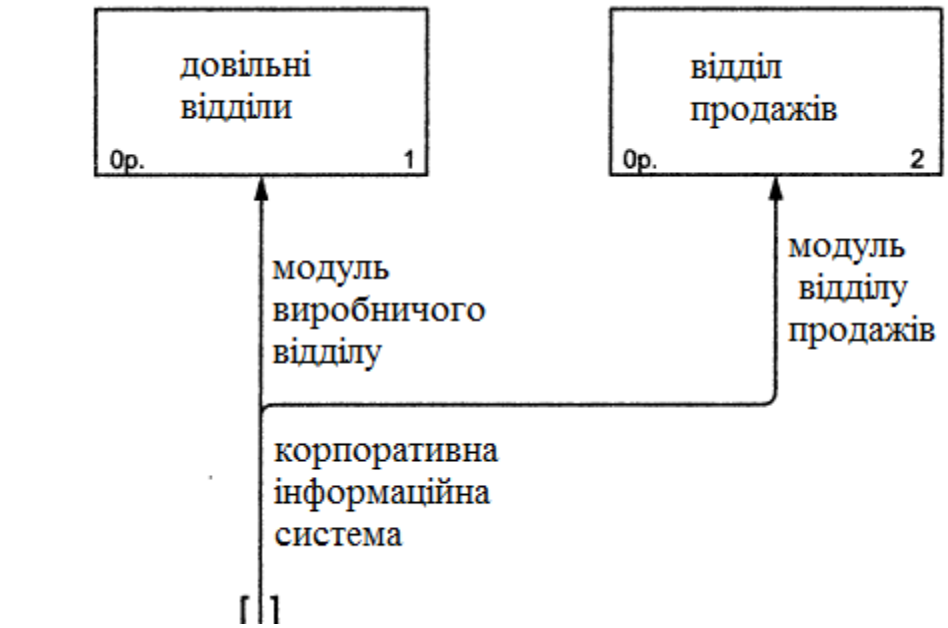


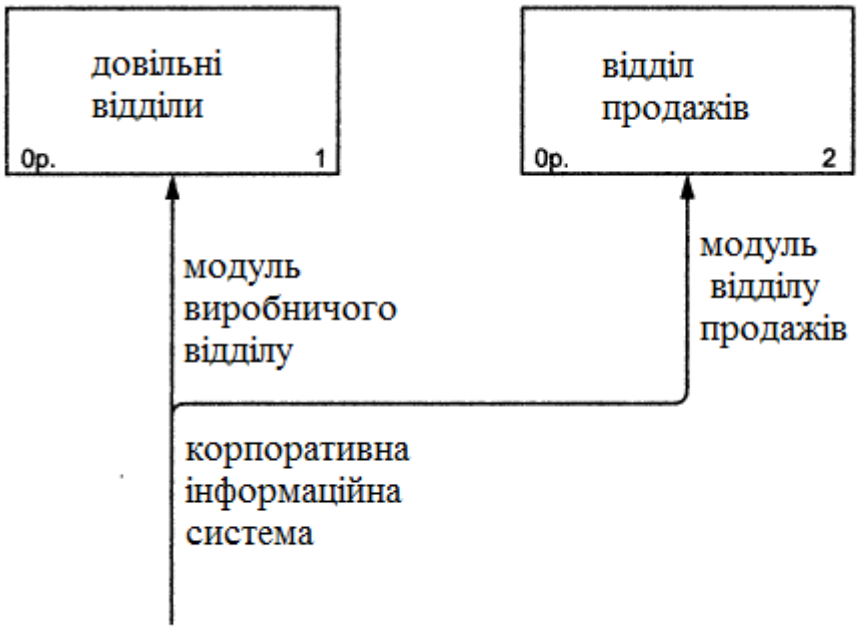
Рис. 28 Роз’єднана і перейменована стрілка

Поняття *зв’язаних* стрілок використовується для управлінням рівнем деталізації діаграми. Якщо одна зі стрілок відсутня на батьківській діаграмі (наприклад, при не великій важливості для батьківського рівня) і не пов’язана з іншими стрілками тієї же діаграми, точка входу або виходу позначається *тунелем.*

Також тунелі використовуються в ситуаціях, коли стрілка, присутня на батьківській діаграмі, відсутня в діаграмі декомпозиції відповідного блока. Приклади для першого в другого випадків використання тунелів можна розглянути на рисунках 29а та 29.б відповідно.



29а



29б

Рис. 29 Приклад тунелів

На рисунку 30 ми бачимо типову IDEF0-діаграму, показану разом з розташованої на її полях службової інформації, яка складається з добре виділених верхнього та нижнього колонтитулів(заголовка та «підвала»). Елементи заголовка використовується для відстеження процесу створення моделі. Елементи «підвала» відображають найменування моделі, до якої відноситься діаграма, та показує її розташування відносно інших діаграм моделі.

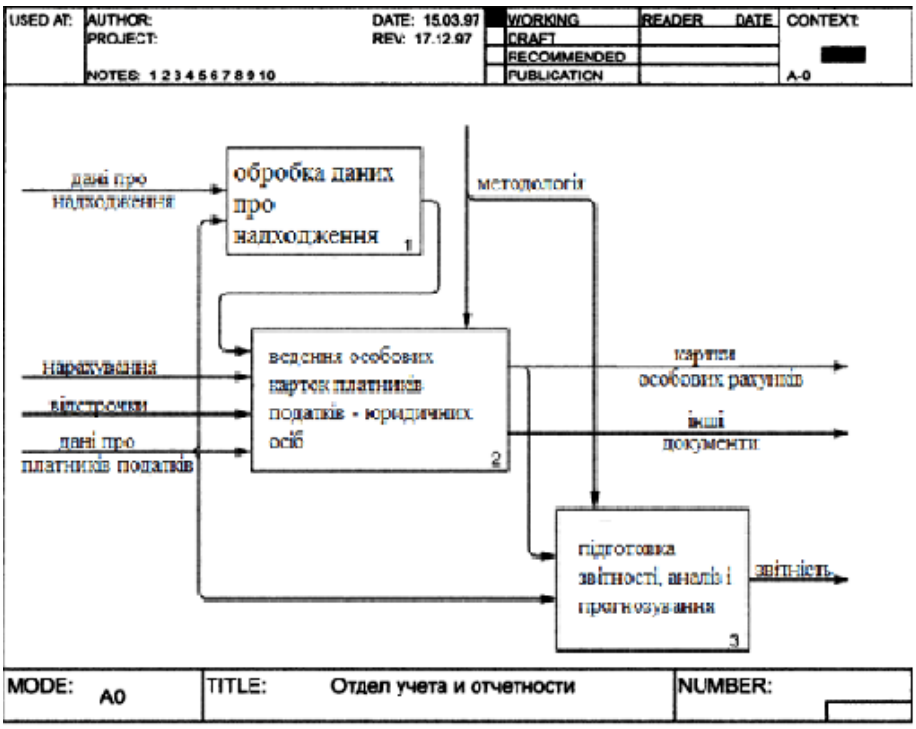


Рис. 30 типова IDEF0- діаграма

Поле	Назначення
Used AT	Використовуються для відображення внішніх посилань на дану діаграму (заповнюється на друкованому документі вручну)
Author, date, project	Містить ПІБ автора діаграми, дату створення, датц послденюю внесення змін, Найменування проекту, в рамках якого вони створювалася
Notes 1 … 10	При ручному редагуванні Діаграма пользватель можуть закреслювати Цифа кожен раз, Когад вони вносять чергове виправлення
Status:	Статус відображає стан розробки або затвердження даний діарамми. Це поле використовується для реалізації фольмлаьного процесу ітерації перегляду та затвердження
Working	Нова діаграма, глобальне зміни або новий активатор для існуючої діаграми
Draft	Діаграма досягла деякого прийнятного для читачів рівня і готова для подання на затвердження
Recommended	Діаграма схвалено і затверджено. Будь-які зміни не передбачаються
Publication	Діаграма готова для остаточної друку та публікації
Reader	ПІБ читача
Date	Дата знайомства читача з діаграмою
Context	Схематичне зображення функціональних блогів на батьківській діаграмі, на якому підсвічений декомпозіруемий даний діаграмою блок. Для діаграми самого верхнього рівня (коттекстной діаграми) в поле поміщається контекст TOP

Табл. 2 Елементи заголовка діаграми

Поле	Назначення
Mode	Номер діаграми, що співпадає з номером батьківського функціонального блоку
Title	Ім'я батьківського функціонального блоку
Number (C-Number)	Унікльний індефікатор данна версії данна діаграми. Таким образоа, кожна нова версія діаграми буде мати нова значення в цьому полі. Як правило, C-Number складається з ініціалів автора і послідовного унікального ідентифікатора, наприклад SDO005. При публікації ці номери сторінок. Якщо діаграма заміщає іншу, номер замінної діаграми може бути укладений в дужки - SDO005 (SDO004). Це забезпечує зберігання історії всіх діарграмм моделі.

Табл. 3 Елементи «підвала» діаграми

Функціональне моделювання за допомогою UML діаграм діяльності

Діаграма діяльності призначена для моделювання складного життєвого циклу об’єкту. Діаграма діяльності описує перехід від однієї діяльності до іншої на відміну від діаграм взаємодії де акцент робиться на перехід управління від об’єкта до об’єкта. Таким чином ці діаграми можуть використовуватись для функціювального моделювання на рівні з діаграмами IDEF0. Наведемо простий приклад діаграми діяльності на рисунку 31

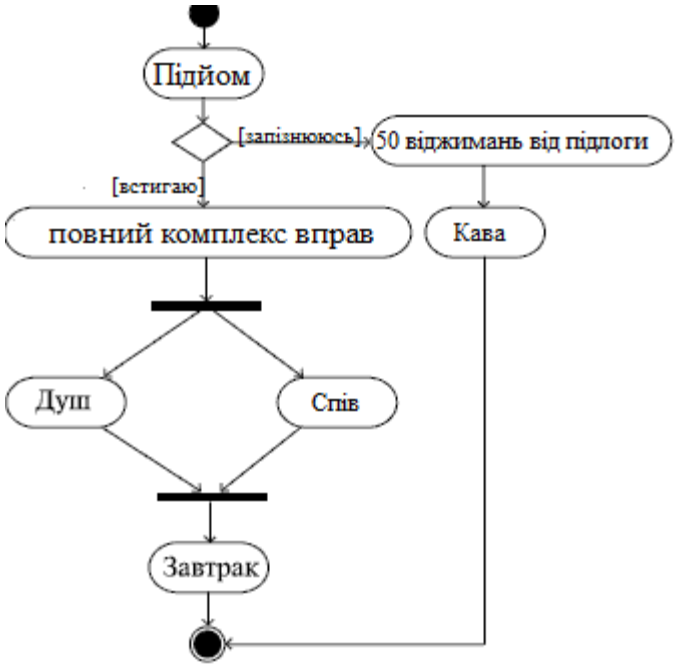


Рис. 31 Діаграма діяльності

Фігури зображені на рисунку 32 позначають початок и кінець нашої діяльності.

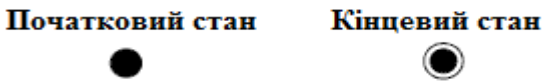


Рис. 32 Початок і кінець діяльності

Символи зображені на рисунку 33 позначають розпаралелення (синхронне виконання деяких дій) та синхронізацію(об’єднання деяких паралельних дій). Також в цих діаграмах можуть знаходитись блоки прийняття рішень, що буде аналогом управляючої стрілки діаграми IDEF0.

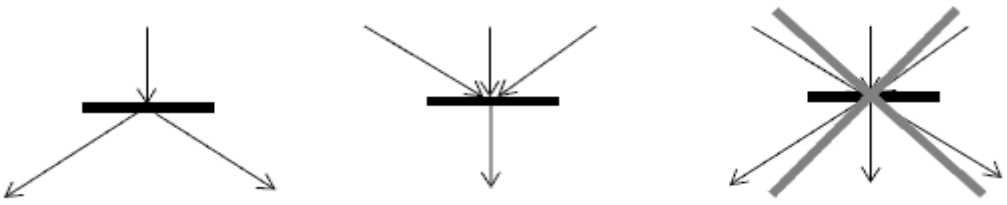


Рис. 33 Приклад розпаралелення і синхронізації

Приклад блоку прийняття рішень показано на рисунку 34

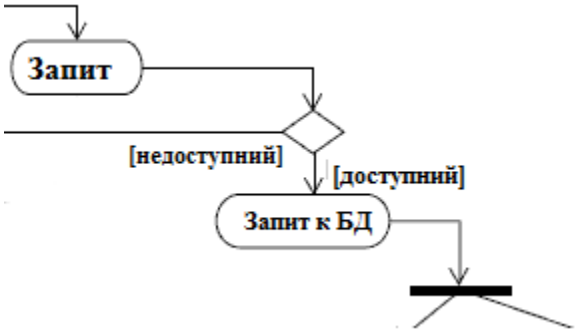


Рис. 34 Приклад блоку прийняття рішень

Рішення однієї задачі за допомогою UML діаграм та IDEF0.

Представимо один і той самий процес за допомогою нотації IDEF0 и діаграми діяльності. Для прикладу, розглянемо вирішення квадратного рівняння.

В нотації IDEF0:

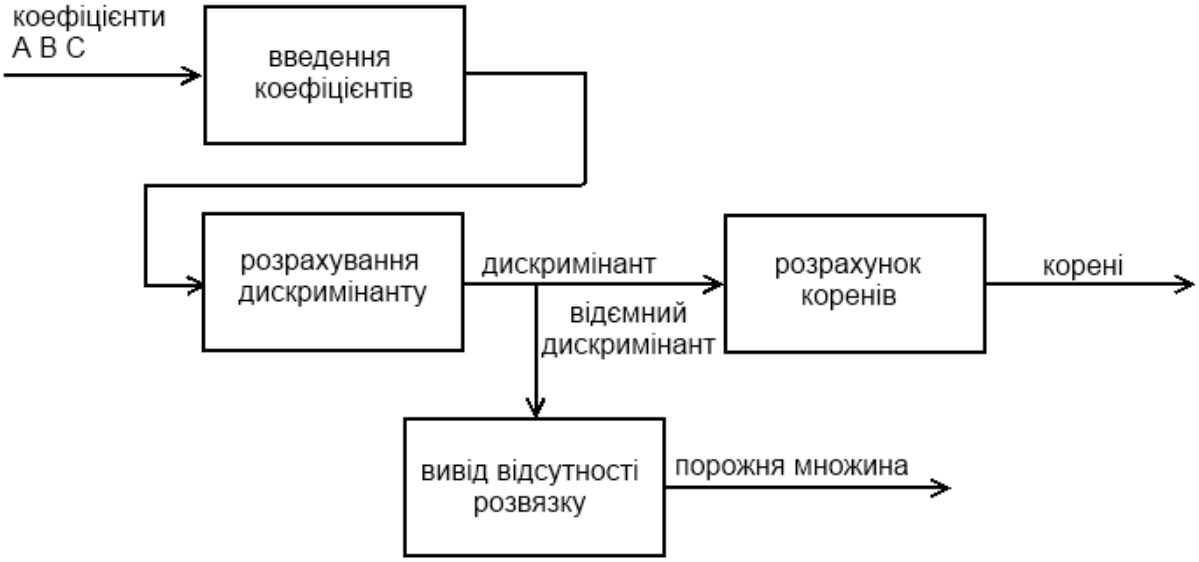


Рис. 35 Представлення процесу за допомогою IDEF0 нотації

Діаграма діяльності:

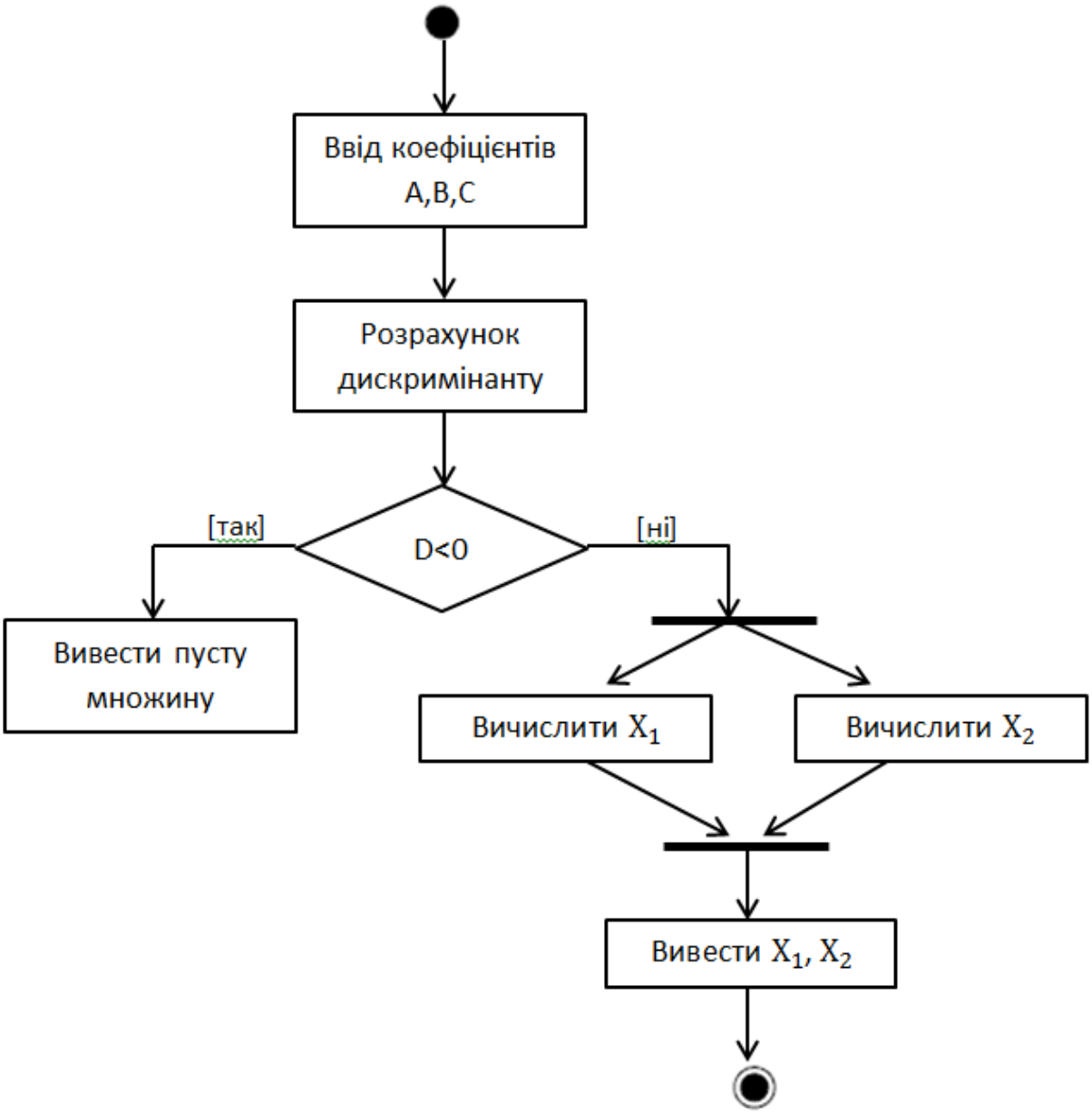


Рис. 36 Представлення процесу за допомогою UMLдіаграми

Лекція 20. Моделювання потоків даних

- Моделювання потоків даних.
- Засоби автоматизації моделювання (ERWin,BPWin, Enterprise Architect та інші). [4, с. 506-537]

Моделювання потоків даних.

Діаграми потоків даних

Діаграми потоків даних (DataFlowDiagramming) є основним засобом моделювання функціональних вимог до проектованої системи і використовуються для опису документообігу та обробки інформації. Детальніше IDEF0, DFD представляє систему ,яка моделюється як мережа зв‘язаних між собою робіт. . Їх можна використовувати як доповнення до моделі IDEF0 для більш наочного відображення поточних операцій документообігу в корпоративних системах обробки інформації.

Головна мета DFD – показати, як кожна робота перетворює свої вхідні дані у вихідні, а також виявити відношення між цими роботами. Вимоги представляються у вигляді ієрархії процесів, пов'язаних потоками даних. Діаграми потоків даних показують, що кожний процес перетворює свої вхідні дані у вихідні, і виявляють відносини між цими процесами.

Будь-яка DFD - діаграма може містити роботи, зовнішні сутності, стрілки (потоки даних) та сховища даних.

Роботи.

Роботи зображаються прямокутниками із закругленими кутами (рис. 37), сенс їх співпадає із сенсом робіт IDEF0 та IDEF3. Також як роботи IDEF3, вони мають входи та виходи, але не підтримують управління та механізми, як IDEF0. Всі сторони роботи рівнозначні. В кожну роботу може входити і виходити по декілька стрілок.

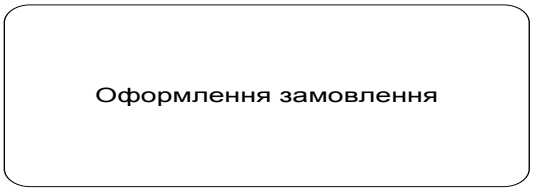


Рис. 37. Робота в DFD

Зовнішні сутності.

Зовнішні сутності зображають входи в систему і/або виходи з неї. Одна зовнішня сутність може одночасно надавати входи (функціонуючи як постачальник) та приймати виходи (функціонуючи як одержувач). Зовнішня сутність представляє собою матеріальний об‘єкт, наприклад замовники, персонал, постачальники, клієнти, склад. Визначення деякого об‘єкту або системи в якості

зовнішньої сутності вказує на те , що вони знаходяться за межами границь системи, яка аналізується. Зовнішні сутності зображуються у вигляді прямокутника з тінню та зазвичай розміщуються по краям діаграми (рис. 38).

Рис. 38 Зовнішня сутність в DFD

Стрілки (потоки даних).

Стрілки описують рух об‘єктів з однієї частини системи в другу (звідси слідує, що діаграма DFD не може мати граничних стрілок). Оскільки всі сторонни роботи в DFD рівнозначні, стрілки можуть починатися та закінчуватися в будь-якій стороні прямокутника. Стрілки можуть бути двонаправлені. Стрілки в DFD показують ,як об’єкти (дані) фактично взаємодіють між собою. Це представлення, об’єднуючи збережені в системі дані і зовнішні для системи об’єкти, дає DFD-моделям велику гнучкість для відображення фізичних характеристик системи, таких як обмін даними, розробка схем їх зберігання та обробки.

Сховище даних.

На відміну від стрілок, які описують об‘єкти в русі, сховища даних зображають об‘єкти в спокої (рис. 39). Сховище даних – це абстрактний пристрій для зберігання інформації, котру можна в будь-який момент помістити в накопичувач і через деякий час витягти, причому способи поміщення та витягання можуть бути будь-якими. Він (пристрій) в загальному випадку являється прообразом майбутньої бази даних, та опис даних, які в ньому зберігаються, повинно відповідати інформаційній моделі (Entity-RelationshipDiagram).

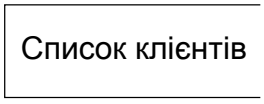


Рис. 39. Сховище даних в DFD

Діаграми потоків даних як взаємно зв’язаний набір дій

Діаграми потоків даних (DFD) моделюють системи як взаємно зв’язаний набір дій, котрі обробляють дані в “сховищі” як всередині ,так і за межами границь модельованої системи. Діаграми потоків даних зазвичай застосовуються при моделюванні інформаційних систем.

Подібно IDEF0, DFD представляє модельовану систему як мережу пов'язаних робіт. Основні компоненти DFD (як було сказано вище) - процеси або роботи, зовнішні сутності, потоки даних, накопичувачі даних (сховища).

На відміну від IDEF0, де система розглядається як взаємозалежні роботи, DFD розглядає систему як сукупність предметів. Контекстна діаграма часто включає роботи і зовнішні посилання. Роботи звичайно іменуються за назвою системи, наприклад "Система обробки інформації". Включення зовнішніх посилань у контекстну діаграму не скасовує вимоги методології чітко визначити мету і точку зору на модельовану систему.

У DFD роботи (процеси) представляють собою функції системи, що перетворюють входи у виходи. Потоки робіт зображуються стрілками і описують рух об'єктів з однієї частини системи в іншу. Оскільки в DFDкожна сторона роботи(процесу) не має чіткого призначення, як в IDEF0, стрілки можуть підходити і виходити з будь-якої межі прямокутника роботи(процесу). У DFD також застосовуються двонаправлені стрілки для опису діалогів типу "команда-відповідь" між роботами, між роботою і зовнішньої сутністю і між зовнішніми сутностями.

На відміну від стрілок, що описують об’єкти в русі , сховища даних зображають об’єкти в спокої (рис. 40)

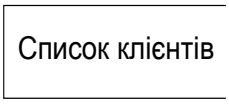


Рис. 40 Сховище даних

У матеріальних системах, сховища даних зображуються там, де об'єкти очікують обробки, наприклад в черзі. У системах обробки інформації сховища даних є механізмом, який дозволяє зберегти дані для наступних процесів.

У DFD стрілки можуть зливатися і розгалужуватися, що дозволяє описати декомпозицію стрілок. Кожен новий сегмент, який зливається або розгалужується стрілками, може мати власне ім'я.

Діаграми DFD можуть бути побудовані з використанням традиційного структурного аналізу, подібно до того, як будуються діаграми IDEF0.

У DFD номер кожної роботи може включати префікс (A), номер батьківської роботи і номер об'єкта. Номер об'єкта - це унікальний номер роботи на діаграмі.Наприклад, робота може мати номер A.12.4. Унікальний номер мають сховища даних і зовнішні сутності незалежно від їхнього розташування на діаграмі. Кожне сховище даних має префікс D і унікальний номер, наприклад D5. Кожна зовнішня сутність має префікс E і унікальний номер, наприклад E5.

Коли і які методології застосовувати

IDEF0 найкраще застосовувати як засіб аналізу та логічного моделювання систем, що, як правило, виконується на ранніх стадіях роботи над проектом. Дані аналізу, отримані з використанням IDEF0-моделювання, зазвичай використовується на стадії розробки моделей IDEF3 і діаграм потоків даних (DFD) - рис. 41.

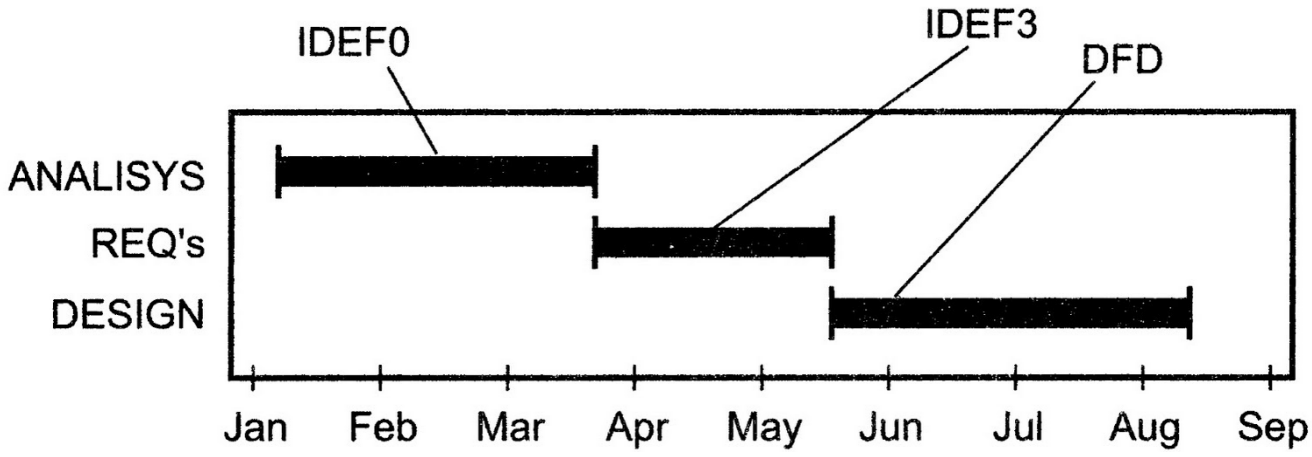


Рис. 41 Часова шкала різних методологій моделювання

В UML моделювання потоків даних представлено у вигляді **діаграми кооперації (collaborationdiagram).**

Головна особливість діаграми кооперації полягає в можливості графічно представити не тільки послідовність взаємодії, але і всі структурні відносини між об'єктами, які беруть участь в цій взаємодії.

Перш за все, на діаграмі кооперації у вигляді прямокутників зображуються об'єкти,які беруть участь у взаємодії, що містять ім'я об'єкта, його клас і, можливо, значення атрибутів. Далі, як і на діаграмі класів, вказуються асоціації між об'єктами у вигляді різних сполучних ліній. При цьому можна явно вказати імена асоціації і ролей, які відіграють об'єкти в даній асоціації. Додатково можуть бути зображені динамічні зв'язки - потоки повідомлень. Вони представляються також у вигляді сполучних ліній між об'єктами, над якими розташовується стрілка зі вказівкою напряму, імені повідомлення і порядкового номера в загальній послідовності ініціалізації повідомлень.

На відміну від діаграми послідовності, на діаграмі кооперації зображаються тільки відносини між об'єктами, що грають певні ролі у взаємодії. На цій діаграмі не вказується час у вигляді окремого вимірювання. Тому послідовність взаємодій і паралельних потоків може бути визначена за допомогою порядкових номерів. Отже, якщо необхідно явно специфікувати взаємозв'язки між об'єктами в реальному часі, краще це робити на діаграмі послідовності.

Кооперація

Поняття кооперації (collaboration) є одним з фундаментальних понять у мові UML. Воно служить для позначення множини взаємодіючих з певною метою об'єктів в загальному контексті модельованої системи. Мета самої кооперації полягає в тому, щоб специфікувати особливості реалізації окремих найбільшзначущих операцій у системі. Кооперація визначає структуру поведінки системи в термінах взаємодії учасників цієї кооперації.

Кооперація може бути представлена на двох рівнях:

рівень специфікації - показує ролі класифікаторів і ролі асоціацій в розглянутій взаємодії;

рівень прикладів - вказує екземпляри і зв'язку, що утворюють окремі ролі в кооперації.

Діаграма кооперації рівня специфікації показує ролі, які грають беруть участь у взаємодії елементи. Елементами кооперації на цьому рівні є класи і асоціації, які позначають окремі ролі класифікаторів і асоціації між учасниками кооперації.

Діаграма кооперації рівня прикладів представляється сукупністю об'єктів (екземпляри класів) і зв'язків (екземпляри асоціацій). При цьому зв'язки доповнюються стрілками повідомлень. На даному рівні показуються тільки об'єкти, що мають безпосереднє відношення до реалізації операції або класифікатора. При цьому зовсім не обов'язково зображувати всі властивості або всі асоціації, оскільки на діаграмі кооперації присутні тільки ролі класифікаторів, але не самі класифікатори. Таким чином, у той час як класифікатор вимагає повного опису всіх своїх екземплярів, роль класифікатора вимагає опису тільки тих властивостей і асоціацій, які необхідні для участі в окремій кооперації.

Звідси випливає важливий наслідок. Одна і та ж сукупність об'єктів може брати участь в різних кооперациях. В залежності від даної кооперації, можуть змінюватися як властивості окремих об'єктів, так і зв'язку між ними. Саме це відрізняє діаграму кооперації від діаграми класів, на якій повинні бути вказані всі властивості і асоціації між елементами діаграми.

Кооперація на рівні специфікації зображається на діаграмі пунктирним еліпсом, усередині якого записується ім'я цієї кооперації. Таке уявлення кооперації відноситься до окремого варіанту використання та деталізує особливості його подальшої реалізації. Символ еліпса кооперації з'єднується відрізками пунктирної лінії з кожним з учасників цієї кооперації, в якості яких можуть виступати об'єкти або класи. Кожна з цих пунктирних ліній позначається роллю (role) учасника. Ролі відповідають іменам елементів в контексті всієї кооперації. Ці імена трактуються як параметри, які обмежуютьспецифікацію елементів при будь-якому їх появу в окремих уявленнях моделі.

Простий клас на діаграмі кооперації позначається прямокутником класу, всередині якого записується рядок тексту. Цей рядок тексту називається роллю класифікатора (classifierrole). Роль класифікатора показує особливість використання об'єктів даного класу. Зазвичай в прямокутникупоказується тільки секція імені класу, хоча не виключається можливість вказівки секцій атрибутів і операцій.

Рядок тексту в прямокутнику повинен мати такий формат:

'/' <Ім'я ролі класифікатора> ':' <Ім'я класифікатора>
[':' <Ім'я класифікатора>] *

Тут ім'я класифікатора, якщо це необхідно, може включати повний шлях всіх вкладених пакетів. При цьому, один пакет від іншого відділяється подвійною двокрапкою «::». В окремих випадках можна обмежитися вказівкою тільки

найближчого з пакетів, якому належить дана кооперація. Символ «*» застосовується для вказівки можливості ітеративного повторення імені класифікатора.

Якщо кооперація допускає узагальнене уявлення, то на діаграмах можуть бути вказані відносини узагальнення відповідних елементів. Цей спосіб може бути використаний для визначення окремих кооперацій, які є окремим випадком або спеціалізацією іншої кооперації. Така ситуація зображаєтьсязвичайною стрілкою узагальнення, спрямованою від символу дочірньої кооперації до символу кооперації-предка. Ролі дочірніх кооперацій можуть бути спеціалізаціями ролей кооперацій-предків.

В окремих випадках виникає необхідність явно вказати той факт, що кооперація є реалізацією деякої операції або класифікатора. Це можна представити одним з двох способів.

По-перше, можна з'єднати символ кооперації пунктирною лінією зі стрілкою узагальнення з символом класу, реалізацію операції якого специфікує дана кооперація.

По-друге, можна просто зобразити символ кооперації, всередині якого вказати всю необхідну інформацію, записану за певними правилами. Ці правила визначають формат запису імені кооперації, після якого записують двокрапку і ім'я класу. За ім'ям класу слід подвійне двокрапка і ім'я операції.

Подібне загальне уявлення кооперації на рівні специфікації використовуєтьсяна початкових етапах проектування. В подальшому кожна з кооперацій підлягає деталізації на рівні прикладів, на якому розкривається зміст і структура взаємозв'язків її елементів на окремій діаграмі кооперації. В цьому випадку в якості елементів діаграми кооперації виступають об'єкти і зв'язки, доповнені повідомленнями.

Об'єкти

Об'єкт як окремим екземпляром класу

Як зазначалося вище, об'єкт (object) є окремим екземпляром класу, який створюється на етапі виконання програми. Він може мати своє власне ім'я і конкретні значення атрибутів. Стосовно об'єктів формат рядка класифікатора доповнюється ім'ям об'єкту і набуває наступний вигляд (при цьому весь запис підкреслюється):

<Ім'я об'єкта> '/' <Ім'я ролі класифікатора> ':' <Ім'я класифікатора>
[':' <Ім'я класифікатора>] *

Ім'я ролі може бути опущено, якщо існує тільки одна роль в кооперації, яку можуть відігравати об'єкти, створені на базі цього класу.

Таким чином, для позначення ролі класифікатора достатньо вказати або ім'я класу (разом з двокрапкою), або ім'я ролі (разом з похилою рискою). В іншомувипадку прямокутник відповідатиме звичайному класу. Якщо роль, яку

має відігравати об'єкт, успадковується від декількох класів, то всі вони повинні бути вказані явно і розділятися комою і двокрапкою.

Рис. 42 Об’єкт.

Нижче наводяться можливі варіанти запису рядка тексту в прямокутнику об'єкту:

·: C — анонімний об’єкт, утворюваний на основі класу C;
·/ R — анонімний об’єкт, виконуючи роль R;
·/ R : C — анонімний об’єкт, утворюваний на основі класу C и виконуючи роль R;
·O / R — об’єкт з ім’ям O, виконуючий роль R;
·O : C — об’єкт з ім’ям O, утворюваний на основі класу C;
·O / R : C — об’єкт з іменем O, утворюваний на основі класу C і виконуючи роль R;
·O або — об’єкт з іменем O;
·O : — «об’єкт-сирота» з іменем O;
·/ R — роль з іменем R;
·: C — анонімна роль на базі класу C;
·/ R : C — роль з іменем R на основі класу C.

Мультиоб'єкт

Мультиоб'єкт (multiobject) являє собою безліч об'єктів на одному з кінців асоціації. На діаграмі кооперації мультиоб'єкт використовується для того, щоб показати операції і сигнали, які адресовані всій множині об'єктів, а не тільки одному. Мультиоб'єкт зображується двома прямокутниками, один з яких виступає із правої верхньої вершини іншого. Стрілка повідомлення відноситься до всього безлічі об'єктів, які позначають даний мультиоб'єкт. На діаграмі кооперації може бути явно вказано відношення композиції міжмультиоб'єктом і окремим об'єктом з його множини. Приклад на рис. 43.

Рис. 43 Мультиоб’єкт.

Об'єкти пасивні та активні.

У контексті мови UML всі об'єкти діляться на дві категорії: **пасивні** та **активні**.

Пасивний об'єкт оперує тільки даними і не може ініціювати діяльність з управління іншими об'єктами. Проте пасивні об'єкти можуть посилати сигналив процесі виконання запитів, які вони отримують.

Активний об'єкт (activeobject) має свою власну нитку (thread) управління і може ініціювати діяльність з управління іншими об'єктами. Під ниткою тутрозуміється деякий полегшений потік управління, який може виконуватисяпаралельно з іншими обчислювальними нитками або нитками управління вмежах одного обчислювального процесу або процесу управління.

Активні об'єкти на канонічних діаграмах позначаються прямокутником з більшширокими межами. Іноді може бути явно вказано ключове слово {active}, щобвиділити активний об'єкт на діаграмі. Кожен активний об'єкт може ініціювати єдину нитку або процес управління і представляти вихідну точку потоку керування.

Складовий об'єкт (compositeobject) або об'єкт-контейнер призначений для подання об'єкта, що має власну структуру і внутрішні потоки (нитки) управління. Складовий об'єкт є екземпляром складового класу (класу-контейнера), який пов'язаний відношенням агрегації або композиції зі своїми частинами. Аналогічні відносини пов'язують між собою і відповідні об'єкти.

На діаграмах кооперації складений об'єкт зображується як звичайний об'єкт, що складається з двох секцій: верхньої і нижньої. У верхній секції записується ім'я складеного об'єкта, а в нижній його складові частини замість списку атрибутів. Також допускається мати в якості складових частин інші складові об'єкти.

Зв'язок

Зв'язок (link) є екземпляром або прикладом довільної асоціації. Зв'язок як елемент мови UML може мати місце між двома і більше об'єктами. Бінарний зв'язок на діаграмі кооперації зображується відрізком прямої лінії, що з'єднує два прямокутники об'єктів. На кожному з кінців цієї лінії можуть бути явно вказані імена ролей даної асоціації. Поруч з лінією в її середній частині може записуватися ім'я відповідної асоціації.

Зв'язки не мають власних імен, оскільки повністю ідентичні як екземпляри асоціації. Іншими словами, всі зв'язки на діаграмі кооперації можуть бути тільки анонімними і записуються без двокрапки перед ім'ям асоціації. Для зв'язків не вказується також і кратність. Проте інші позначення спеціальних випадків асоціації (агрегація, композиція) можуть бути присутніми на окремих кінцях зв'язків. Приклад на рис. 9.

Рис. 9. Зв’язок.

Зв'язок може мати деякі стереотипи, які записуються поряд з одним з її кінців і вказують на особливість реалізації даного зв'язку. У мові UML для цієї мети можуть використовуватися наступні стереотипи:

«Association» - асоціація (передбачається за замовчуванням, тому цей стереотип можна не вказувати);

«Parameter» - параметр методу. Відповідний об'єкт може бути тільки параметром деякого методу;

«Local» - локальна змінна методу. Її область видимості обмежена тільки сусіднім об'єктом;

«Global» - глобальна змінна. Її область видимості поширюється на всю діаграму кооперації;

«Self» - рефлексивна зв'язок об'єкта з самим собою, яка допускає передачуоб'єктом повідомлення самому собі. На діаграмі кооперації рефлексивний зв'язок зображується петлею у верхній частині прямокутника об'єкту.

Повідомлення

Повідомлення, як елементи мови UML, вже розглядалися раніше при вивченні діаграми послідовності. При побудові діаграми кооперації вони мають деякі додаткові семантичні особливості. Повідомлення на діаграмі кооперації специфікує комунікацію між двома об'єктами, один з яких передає іншому деяку інформацію. При цьому, перший об'єкт чекає, що після отримання повідомлення другим об'єктом послідує виконання певної дії. Таким чином, саме повідомлення є причиною або стимулом для початку виконання операцій, відправки сигналів,

створення та знищення окремих об'єктів.Зв'язок забезпечує канал для спрямованої передачі повідомлень між об'єктами від об'єкта-джерела до об'єкта-одержувача.

Повідомлення в мові UML також специфікують ролі, які грають об'єкти відправник та одержувач повідомлення. Повідомлення на діаграмі кооперації зображаються поміченими стрілками поряд (вище або нижче) з відповідним зв'язком або роллю асоціації. Напрямок стрілки вказує на одержувача повідомлення. Зовнішній вигляд стрілки повідомлення має певний сенс. На діаграмах кооперації може використовуватися один з чотирьох типів стрілок для позначення повідомлень:

1) суцільна лінія з трикутною стрілкою позначає виклик процедури або іншого вкладеного потоку керування. Може бути також використана спільно з паралельно активними об'єктами, коли один з них передає сигнал і чекає, покине закінчиться деяка вкладена послідовність дій. Зазвичай всі такі повідомлення є синхронними, тобто ініційованих по завершенні певної діяльності або при виконанні деякої умови;
2) суцільна лінія з V-подібною стрілкою позначає простий потік управління.Кожна така стрілка зображає один етап в послідовності потоку керування. Зазвичай всі такі повідомлення є асинхронними;
3) суцільна лінія з напівстрілкою використовується для позначення асинхронного потоку керування. Відповідні повідомлення формуються в довільні,заздалегідь не відомі моменти часу, як правило, активними об'єктами. Зазвичай повідомлення цього типу є початковими в послідовності потоку керування і частіше за все ініціюються акторами;
4) пунктирна лінія з V-подібною стрілкою позначає повернення з виклику процедури. Стрілки цього типу часто відсутні на діаграмах кооперації,оскільки неявно передбачається їх існування після закінчення процесу активізації деякої діяльності.

Кожне повідомлення може бути позначено рядком тексту, який має такий формат:

<Попередні повідомлення><[Охоронна умова]>
<Вираз послідовності>
<Повертане значення-ім'я повідомлення><Список аргументів>

Попередні повідомлення - це розділені комами номери повідомлень, записані перед похилою рискою:

<Номер повідомлення ','><Номер повідомлення, '>' / '

Якщо список номерів повідомлень порожній, то весь запис, включаючи похилуриску, опускається. Кожен номер повідомлення може бути вираженнямпослідовності без рекурсивних символів. Вираз повинен визначати номер іншого повідомлення у цій же послідовності.

Сенс вказівки попередніх повідомлень полягає в тому, що дане повідомлення не може бути передано, поки не будуть передані своїм адресатам всі повідомлення, номери яких записані в даному списку.

Приклад запису попередніх повідомлень:

A3, B4 / C5: помилка запису (сектор).

Безпечна умова є звичайним булевих виразом і призначена для синхронізації окремих ниток потоку керування. Безпечна умова записується у квадратних дужках і може бути опущена, якщо воно відсутнє у даного повідомлення. Семантика безпечної умови забезпечує передачу повідомлення тільки в тому випадку, якщо ця умова приймає значення «істина».

Приклад запису безпечних умов без номерів попередніх повідомлень:

[(x> = 0) & (x <= 255)] 1.2: зобразити на екрані колір (x);
[кількість цифр номера = 7] 3.1: набрати телефонний номер ();

Вираз послідовності - це розділений крапками список окремих термів послідовностей, після якого записується двокрапка:

<Терм послідовності '.'><Терм послідовності '.'> ':'

Кожен з термів представляє окремий рівень процедурної вкладеності у формі закінченої ітерації. Найбільш верхній рівень відповідає самому лівому терму послідовності. Якщо всі потоки управління паралельні, то вкладеність відсутня. Кожен з термів послідовності має наступний синтаксис:

'*' '[' Пропозиція-ітерація ']' для запису ітеративного виконання відповідного виразу. Ітерація представляє послідовність повідомлень одного рівня вкладеності. Пропозиція-ітерація може бути опущена, якщо умови ітерації ніяк не специфікуються. Найбільш часто пропозиція-ітерація записується на деякому псевдокоді або мові програмування. У мові UML формат запису цієї пропозиції не визначений. Наприклад, "* [/: = / .. n]", що означає послідовну передачу повідомлення з параметром /, який змінюється від 1 до деякого цілого числа n з кроком 1;

'[' Пропозиція-умова У ']' для запису розгалуження. Ця умова являє таке повідомлення, передача якого з даної гілки можлива тільки при істинності цієї умови. Найчастіше пропозицію-умова записують на деякому псевдокоді або мові програмування, оскільки в мові UML формат запису цієї пропозиції не визначений. Наприклад, [x> y] означає, що повідомлення за деякою гілки буде передано тільки в тому випадку, якщо значення x більше значення y.

Значення, що повертається представляється у формі списку імен значень, що повертаються після закінчення комунікації або взаємодії в повній ітерації даної процедурної послідовності. Ці ідентифікатори можуть виступати в якості аргументів в наступних повідомленнях. Якщо повідомлення не повертає ніякого значення, то ні значення, ні оператор присвоювання на діаграмі кооперації не вказуються.

Наприклад, повідомлення

1.2.3: p: = знайти_документ (специфікація_документа)

означає передачу вкладеного повідомлення із запитом пошуку в базі данихпотрібного документа по його специфікації, причому джерела повідомлення повинен бути повернений знайдений документ.

Ім'я повідомлення, записане в сигнатурі після значення, що повертається, означає ім'я події, яка ініціюється об'єктом-одержувачем повідомлення після його прийому. Найбільш часто такою подією є виклик операції об'єкту. Це може бути реалізовано різними способами, один з яких - виклик операції. Тоді відповідна операція повинна бути визначена в тому класі, якому належить об'єкт-одержувач.

Список аргументів є розділені комами і укладені в круглі дужки дійсні параметри тієї операції, виклик якої ініціюється даним повідомленням. Список аргументів може бути порожнім, проте дужки все одно записуються. Для запису аргументів також може бути використаний деякий псевдокод або мова програмування.

Так, у наведеному вище прикладі повідомлення

1.2.3: р: = знайти_документ (специфікація_документа)

аргумент знайти_документ є ім'ям повідомлення,
а специфікація_документа - списком аргументів, що складається з єдиного дійсного параметра операції.

При цьому ім'я повідомлення означає звернення до операції знайти_документ, що має бути визначена у відповідному класі об'єкту-одержувача.

Приклад діаграми кооперації зображено на рис. 44.

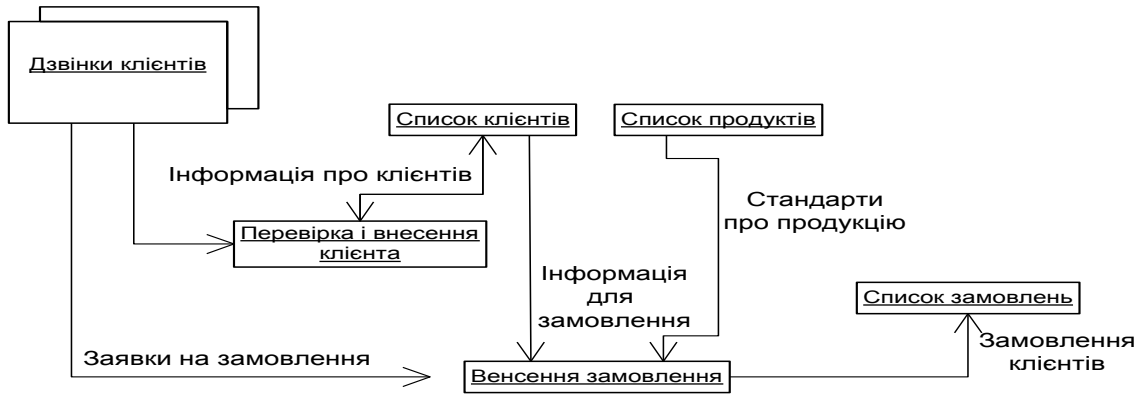


Рис. 44 Приклад діаграми кооперцій

Розділ 6. Методи забезпечення та контролю якості ПЗ

Лекція 21. Тестування ПЗ

- Якість ПЗ;
- метрики і стандарти якості ПЗ.
- Верифікація та валідація ПЗ.
- Тестування ПЗ.
- Інструменти автоматизації процесів тестування (JUnit, JMeter). [4, с. 368-402]

JUnit - Бібліотека для тестування програмного забезпечення для мови Java

Створення

JUnit — бібліотека для тестування програмного забезпечення для мови Java, створена Кентом Беком і Еріком Ґаммою. JUnit є представником родини фреймворків xUnit для різних мов програмування, яка бере початок у SUnit Кента Бека для Smalltalk. JUnit породив екосистему розширень — JMock, EasyMock, DbUnit, HttpUnit, Selenium тощо. JUnit підключається як JAR файл підчас компілювання . Для версій JUnit 3.8 та нижче бібліотека знаходиться в пакеті junit.framework, а для версій 4 і вище – у org.junit.

Функції JUnit:

1. Перевірка очікуваних результатів
2. Випробування пристосувань для обміну загальними даними тесту
3. Випробування тестувальників для виконання тестів

JUnit тести не потребують людського втручання для інтерпретації, також можна запускати кілька тестів одночасно. Для того щоб протестувати код необхідно:

1. Додати до методу анотацію @org.junit.Test
2. Для перевірки значення необхідно статично імпортувати org.junit.Assert.* і викликати assertTrue() і передати булеве значення true, якщо тест пройдено.

JUnit 3

Для установки тесту потрібно унаслідувати тест-клас TestCase, перевизначити методи setUp і tearDown якщо потрібно і створити тестові методи (повинні починатися з test). При запуску тесту спочатку створюється екземпляр тест-класу (для кожного тесту в класі окремий екземпляр класу), потім виконується метод setUp, запускається сам тест, і по завершенню виконується метод tearDown. Якщо якийсь метод викидає виключення (Exception) тест рахується проваленим.

Примітка : тестові методи мають бути publicvoid і можуть бути static.

Самі тести складаються з виконання певного коду і перевірок. Перевірка часто виконується за допомогою класу Assert хоча інколи використовують ключове слово assert.

JUnit 4

Головні особливості

У JUnit 4 була додана підтримка нових можливостей з Java 5, тести тепер можуть бути оголошені за допомогою анотацій. При цьому існує зворотня сумісність з попередньою версією фреймворку, і всі вищерозглянуті приклади будуть працювати і тут (крім RepeatedTest - його нема в новій версії).

Основні анотації:

1. для спрощення роботи наслідуємо клас Assert (це не обов'язково)
2. анотація @Before позначає методи, які будуть викликані до виконання тесту, методи повинні бути public void. Тут зазвичай розміщуються ініціалізація тесту, в цьому випадку це генерація тестових даних (метод setUpToHexStringData).
3. анотація @BeforeClass позначає методи, які будуть викликані до створення екземпляру тест-класу, методи мають бути public static void. Є сенс розмістити ініціалізацію тесту у випадку, коли клас містить декілька тестів, що використовують різні попередні налаштування, або коли декілька тестів використовують одні і ті самі дані, щоб не затрачати час на їх створення для кожного тесту.
4. анотація @After позначає методи, які будуть викликані після виконання тесту. Методи мають бути public void. Тут розміщуються операції вивільнення ресурсів після тесту, в цьому випадку очистка тестових даних (метод tearDownToHexStringData).
5. анотація @AfterClass аналогічна @BeforeClass, але виконує методи після тесту, як і у випадку з @BeforeClass, методи мають бути public static void.
6. анотація @Test позначає тестові методи. Як і раніше, ці методи мають бути public void. Тут розміщуються самі перевірки. Крім того, в цієї анотації є два параметри, expects — задає очікувані виключення (Exception) і timeout — задає час, по закінчення якого тест рахується проваленим.

JUnit38ClassRunner

JUnit38ClassRunner призначений для запуску тестів написаних з використанням JUNIT3. SuiteMethod або AllTests також призначені для запуску JUNIT3 тестів. На відміну від попереднього варіанту, тут передається клас з статичним методом suite який повертає тест (послідовність всіх тестів).

Suite — еквівалент попереднього тільки для JUNIT4 тестів. Для настройки запуску тестів використовується анотація @SuiteClasses

Enclosed — те саме що і попередній варіант але замість налаштування з допомогою анотації використовуються всі внутрішні класи.

Categories — спроба організувати тести по категоріях (групах). Для цього тестам задається категорія з допомогою @Category, потім настроюються запускаючі категорії тестів у юніті.

Parameterized — дозволяє писати параметризовані тести. Для цього в тест класі оголошується статичний метод який повертає список даних, які пізніше будуть використані в якості аргументів конструктора класу

Theories — параметризує тестовий метод а не конструктор. Дані помічаються з допомогою @DataPoints і @DataPoint, тестовий метод — допомогою @Theory.

Пристосування

Якщо тести оперують над схожими або однаковими наборами даних тести необхідно запустити на тлі відомого набору об'єктів. Такий набір називають тестувальним пристосуванням . При написанні тестів часто більше часу витрачається на написання коду для підготовки пристосування ніж для власне тестування значень.

Певною мірою написання коду пристосування можна спростити уважно слідкуючи за конструкторами. Спільне використання коду пристосувань ще більше спрощує написання. Часто одні і ті ж пристосування можна використати для різних тестів. Кожна версія буде передавати різні повідомлення до одного пристосування і буде перевіряти різні результати.

Для використання спільного пристосування необхідно:

1. Додати поле для кожної частини пристосування
2. Анотувати метод @org.junit.Before і ініціалізувати змінні у ньому
3. Анотувати метод @org.junit.After щоб звільнити зайняті ресурси

Очікуванні виключення

Перевірка правильної роботи коду є частиною програмування. Переконатись, що код поводить себе у виключних ситуаціях як очікувалось – теж. Наприклад код:

newArrayList<Object>().get(0);

Має викликати виключення IndexOutOfBoundsException. Анотація @Test має необов'язковий параметр «expected» що приймає підкласи Throwable. Для перевірки, що ArrayList викликає правильне виключення можна написати:

@Test(expected= IndexOutOfBoundsException.class) publicvoidempty() {

new ArrayList<Object>().get(0);

}

Запуск тестів

Засоби для визначення набору тестів

JUnit надає засоби для визначення набору тестів для запуску і відображення їх результату. Для цього з Java програми необхідно запустити

org.junit.runner.JUnitCore.runClasses(TestClass1.class, ...).

Або

javaorg.junit.runner.JUnitCoreTestClass1 [...інші класи тестування...]

з консолі.

Щоб зробити тестові класи JUnit 4 доступними TestRunner, що працює з попередніми версіями JUnit визначте статичний метод suite(), що повертає тест:

```
public static junit.framework.Test suite() {
   return new JUnit4TestAdapter(Example.class);
}
```

Завдання

1. Вивчити структуру JUnit. Вільно володіти основними поняттями.
2. Ознайомитись з призначенням і структурою. Знати основні функції.
3. Вміти розрізняти версії JUnit 3 та JUnit 4. Знати список основних анотацій.
4. Створити типовий проект у робочому просторі (workspace) середовища Eclipse. В пакеті com.labs.labwork2 запустити на виконання (run) клас TestMain.
5. За допомогою автоматизованих засобів виконати повне документування розроблених класів (також методів і полів), при цьому документація має в достатній мірі висвітлювати роль певного класу в загальній структурі та особливості конкретної реалізації.

ApacheJMeter

Інструмент позиціонує себе як універсальний

ApacheJMeter - це напевно єдиний інструмент для навантажувального тестування, який з одного боку безкоштовний (opensource), а з іншого боку досить розвинений і має можливість створення тестового навантаження одночасно з декількох комп'ютерів. Тести для JMeter створюються візуально і мають деревовидну структуру в вікні редагування тесту. Запуск тестів можна робити як з вікна програми, так і з командного рядка, що в свою чергу корисно, якщо ви їх запускаєте за розкладом, наприклад, вночі. JMeter призначений для тестування не тільки веб-серверів і їх окремих компонентів (сервлети, CGI-сценарії), але також і FTP-серверів і баз даних (з використанням драйвера JDBC). Передбачені механізми авторизації віртуальних користувачів, підтримуються настроювані сеанси.

Інструмент позиціонує себе як універсальний, що дозволяє працювати як з http запитами, так і FTP, JDBC запити, SOAP, Web Services, TCP, LDAP, JMS,

Mail testers. Але, однозначно, він краще за все "заточений" для навантажувального тестування веб-додатків.

Якщо ви створюєте багатокористувацький веб-додаток, то перед його випуском у вас однозначно виникнуть питання:

1. чи стабільно працює додаток під великим навантаженням;
2. яке максимально можливе число користувачів витримає додаток на певній конфігурації;
3. наскільки швидше стало працювати додаток після поліпшення архітектури коду.

На всі ці питання, з відносно невеликими витратами часу, вам відповість JMeter.

З чого складається тест?

При створенні тесту JMeter пропонує кілька типів компонент:

Samplers- основні елементи, які безпосередньо спілкуються з тестованим додатком, наприклад httpsampler для звернення до веб-додатком

Logiccontrollers - елементи, що дозволяють групувати інші елементи в цикли, групи паралельного запуску, і т.д.

Assertions- елементи, що виконують контроль. З їх допомогою ви можете перевірити текст, який ви очікуєте на веб-сторінці або, наприклад, вказати що ви очікуєте відповідь від сервера не більш ніж 2 секунди. Якщо який-небудь Assert не буде задоволений, тест буде мати негативний результат.

Чи можна розширювати JMeter?

Можна і потрібно. JMeter сам по собі являється зручним інструментом для запуску та моніторингу тестів, а також для перегляду звітів. Тому якщо вам потрібні особливі види тестів, які ви можете написати на Java, то все що вам потрібно, це написати код самого тесту. Далі його потрібно оформити як Sampler, після чого JMeter допоможе вам робити всю іншу роботу з організації, конфігурації і виконання тестів (в тому числі і з декількох комп'ютерів).

Запуск програми

Для роботи JMeter потрібно попередньо встановлений на комп'ютер пакет JRE /JDK. Для запуску JMeter у віконному режимі необхідно виконати файл jmeter.bat (Для Windows) або jmeter (Для Linux), який знаходиться в папці з встановленим JMeter в директорії bin.

Для запуску JMeter в консольному режимі необхідно виконати jmeter.bat з наступними опціями:

-n [Визначає, що запуск JMeter необхідно зробити в консольному режимі]
-t [Ім'я JMX-файла плану тестування (Test Plan)]
-l [Ім'я JTL-файла для логування результатів виконання завдання]
-H [Адреса проксі]
-P [Порт проксі]

Приклад:

jmeter.bat-n-t my_test.jmx-l log.jtl-H my.proxy.server-P 8000

Візуалізація і ведення журналу результатів

Для візуалізації та ведення журналу результатів тестування в JMeter реалізовані Слухачі (Listeners). Слухачі можуть бути декількох типів:

-- *Graph Full Results* - Відображення результатів тесту у вигляді графіка

-- *Graph Results* - Відображення основних результатівтесту (відхилення, середні величини) у вигляді графіка

-- *Spline Visualizer* - Відображення результатів тесту у вигляді графіка з усередненими (згладженими) значеннями

-- *Assertion Results* - Відображення результатів дії затверджених

-- *View Results Tree* - Відображення результатів тесту у вигляді дерева

-- *Aggregate Report* - Відображення результатів дії кожного запиту тесту у вигляді таблиці

-- *View Results in Table* - Відображення сумарних результатів тесту у вигляді таблиці

-- *Simple Data Writer* - Запис результатів тесту в файл

-- *Monitor Results* - Моніторинг завантаження сервера (тільки для веб-серверів з Tomcat 5)

-- *Distribution Graph (alpha)* - Відображає час виконання кожного запиту в графічному вигляді

-- *Aggregate Graph* - Сумарні результати тесту у вигляді таблиці і графіка

-- *Mailer Visualizer* - Відправка звіту на email (необхідна наявність smtp.jar і pop3.jar)

-- *BeanShell Listener* - Самостійна конфігурація звіту про тестування на вбудованій мові BeanShell

-- *Summary Report* - Відображення результатів дії кожного запиту тесту у вигляді таблиці (аналог Assertion Results, але використовує менше ресурсів комп'ютера)

На сторінці налаштувань Відповідності необхідно задати наступні параметри:

-- Назва відповідності (Name)

-- Яку відповідь сервера необхідно обробляти (повернувся текст, url, код відповіді,повідомлення відповіді)

-- Встановити правила обробки знайдених входження (Patterns) - містить / не містить

-- Визначити Входження, які треба шукати в результатах запиту (PatternstoTest). Входження підтримують perl-сумісні регулярні вирази (PCRE) *(наприклад, "<ahref =index.phtml \? Rid = [0-9] + &d = [0-9] {4,4} - [0-9] {2,2} - [0-*

9] {2,2}> [^ <] + </ a> "), але також можна вказати і просто шуканий текст *(наприклад," Новини за тиждень ").*

Розробка плану тестування програми

Тестовий план описує серію кроків, які JMeter буде виконувати при запуску. Закінчений тестовий план складається з одного чи більше Груп Потоків,типових контролерів, контролерів логіки, слухачів, відповідностей, таймерів, конфігураційних елементів.

Додавання та видалення елементу

Додати чи загрузити елемент з файлу можна в меню add. Спочатку треба додати Групу Потоків (ThreadGroup). В налаштуваннях групи потоків треба вказати назв, кількість потоків, що будуть запускатись, час затримки між початками потоків, кількість циклів виконання завдання.

Далі в створену групу треба додати Зразок запиту (Sampler). Для вирішення багатьох задач (навантажувальне тестування, перевірка працездатності серверів)

В налаштуваннях треба задати такі параметри :

1) Назва
2) Адреса веб-серверу(URL | IP)
3) Порт веб-серверу (default - 80)
4) Протокол (default - HTTP)
5) Метод передачі
6) Шлях до файлу, що запускається на сервері
7) Параметри та їх значення

Крім цього присутній параметр перенаправлення (redirection).

Параметри «За замовчуванням»

Для оптимізації вводу даних, що повторюються, існує можливість задати параметри «за замовчуванням» для цього в групу потоків треба додати елемент HTTPRequestDefault (Add->ConfigElement ->HTTPRequestDefault).

В налаштуваннях «За замовчуванням» можна задавати всі ті ж параметри , що і в простому запиті (див. вище)

Обробка результатів запитів

За замовчуванням успішно виконаним запитом, вважається запит, на який було отримано відповідь сервера з 200-м або304-м кодом відповіді, а помилковим – з кодами помилок 404, 500 і т.д. Тим не менш, не завжди код 200 говорить про те, що всі компоненти програми були виконані належним чином. Наприклад, сервер може повернути 200-й код, тоді як при запиті веб-сторінки неможливо було встановити з'єднання з сервером баз даних - додаток вивело на екран помилку підключення, але код повернуло 200. Для класифікації подібних випадків логічно було б використовувати систему визначення помилок веб-

додатків на основі тексту, що видається браузером, а не тільки коду відповіді, що повертається.

Лекція 22. Постійна інтеграція ПЗ

- Принципи постійної інтеграції ПЗ.
- Оптимізація коду та рефакторинг.
- Аспекти продуктивності ПЗ.
- Сервера постійної інтеграції (Hudson, CruiseControl). [4, с. 368-402]

Принципи постійної інтеграції ПЗ

Безперервна інтеграція - це практика розробки програмного забезпечення, яка полягає у виконанні частих автоматизованих збірок проекту для якнайшвидшого виявлення та вирішення інтеграційних проблем . У звичайному проекті, де над різними частинами системи розробники трудяться незалежно, стадія інтеграції є завершальною. Вона може непередбачувано затримати закінчення робіт. Перехід до безперервної інтеграції дозволяє знизити трудомісткість інтеграції і зробити її більш передбачуваною за рахунок найбільш раннього виявлення та усунення помилок і протиріч.

Вимоги до проекту

- Вихідні коди і все, що необхідно для побудови та тестування проекту, зберігається в репозиторії системи керування версіями;
- Операції копіювання з репозиторію, збірки і тестування всього проекту автоматизовані і легко викликаються із зовнішньої програми.

Організація

На виділеному сервері організовується служба, до завдань якої входять:

- Отримання вихідного коду з репозиторію;
- Складання проекту;
- Виконання тестів;
- Розгортання готового проекту;
- Відправлення звітів.

Локальна збірка може здійснюватися:

- За зовнішнім запитом,
- За розкладом,
- За фактом оновлення репозиторію і за іншими критеріями.

Збірка за розкладом

У разі складання за розкладом, вони, як правило, проводяться кожної ночі в автоматичному режимі -*нічні збірки*(щоб до початку робочого дня були готові результати тестування). Для розрізнення додатково вводиться система нумерації збірок - зазвичай, кожна збірка нумерується натуральним числом, яке збільшується з кожною новою збіркою. Вихідні тексти та інші вихідні дані при взятті їх з репозиторію системи контролю версій позначаються номером збірки. Завдяки цьому, точно така ж збірка може бути точно відтворена в майбутньому - достатньо взяти вихідні дані по потрібній мітці і знову запустити процес. Це дає можливість повторно випускати навіть дуже старі версії програми з невеликими виправленнями.

Переваги

1. Проблеми інтеграції виявляються і виправляються швидко, що виявляється дешевше;
2. Негайний прогін модульних тестів для свіжих змін;
3. Постійна наявність поточної стабільної версії разом з продуктами збірок - для тестування, демонстрації, тощо п.
4. Негайний ефект від неповного або непрацюючого коду привчає розробників до роботи в ітеративному режимі з більш коротким циклом.

Недоліки

1. Витрати на підтримку роботи безперервної інтеграції;
2. Потенційна необхідність у виділеному сервері під потреби безперервної інтеграції;
3. Негайний ефект від неповного або непрацюючого коду відучує розробників від виконання періодичних резервних включень коду в репозиторій.
    - У разі використання системи управління версіями вихідного коду з підтримкою розгалуження, ця проблема може вирішуватися створенням окремої "гілки" (англ. відділення) проекту для внесення великих змін (код, розробка якого до працездатного варіанту займе кілька днів, але бажано частіше резервне копіювання в репозиторій). Після закінчення розробки та індивідуального тестування такої гілки, вона може бути об'єднана (англ. злиття) з основним кодом або "стволом" (англ. стовбур) проекту.

Безперервна інтеграція - Continuous Integration (CI)

У розробці програмного забезпечення, безперервна інтеграція - практика частого складання й тестування проекту з метою виявлення помилок на ранній стадії. Безперервна інтеграція - автоматизований процес, у якому, як правило, використовується спеціальне серверне ПЗ, відповідальне за пошук змін у коді в системі контролю версій, складання, розгортання й тестування додатка.

Безперервна інтеграція в Custis

У проектах Custis, реалізованих на Java, як сервер CI використовується **Jenkins**.

Безперервна інтеграція «Continuous Integration» - це ліки від страху. Допомагає при програмуванні. Dr. Zoidberg ©

Згідно **Wikipedia** термін Continuous Integration введений Мартіном Фаулером (Martin Fowler) і Кентом Беком (Kent Beck). Даний термін був придуманий ними для позначення практики частого складання (інтеграції) проекту. Максимально часте складання є логічним продовженням ланцюжка ітераційні складання -> нічні складання -> безперервне складання.

У наш час Continuous Integration (безперервна інтеграція) одна із практик застосовуваних у сімействі гнучких (Agile) методологій. У подібних методологіях вона вдало поєднується з іншими практиками, такими як модульне(unit) тестування, рефакторинг, стандарт кодування. Але навіть без них можна одержати користь від безперервної інтеграції.

Основні принципи

Кожна зміна повинна інтегруватися

Слово continuous у терміні **Continuous Integration** означає «безперервний». Це означає, що в ідеалі складання вашого проекту повинна йти буквально увесь час. Кожна зміна в системі контролю версій (наприклад **CVS**) повинна інтегруватися без пропусків або затримок. Організація нічних складань - це гарна практика, але це не continuous integration. Адже результати такого нічного складання будуть доступні лише наступного дня, коли їхня актуальність для розроблювачів уже значно знижена. На практиці досить часто реалізують обидва процеси: і безперервну інтеграцію, і нічні складання - більш рідку інтеграцію. У дуже великих проектах цю вимогу іноді неможливо дотримати, але інтеграція щодоби - це межа, за яку не варто виходити. Принцип безперервної інтеграції не виконаємо без іншої умови - «Складання повинно йти швидко».

Швидке складання

«Складання повинно йти швидко» - точніше не більше 10 хвилин. Якщо після одного невеликого комміта ваш інтеграційний сервер піде в 2-х годинне пихтіння на складання, тестування й розгортання, від цього буде мало користі. Розроблювачі будуть уже думати над рішеннями інших проблем, їм буде складно повернутися й зрозуміти причини збою, якщо такий був. Адже суть безперервної інтеграції в одержанні швидкого feedback. До того ж, пізня відповідь із сервера може відволікти їх від іншої справи.

У випадку якщо всі етапи процесу ніяк не вдається втиснути в прийнятні тимчасові рамки, його можна розділити на кілька частин. При кожному комміті робити лише саме складання й мінімальний набір тестів (smoke tests), щоб зменшити час. А вночі проводити повний цикл інтеграції, результати якого

команда буде аналізувати з ранку. Але це скоріше змушена міра, а не приклад для наслідування.

Зробіть тести

Тести просто необхідно включати в continuous integration процес, у противному випадку ви не можете бути впевнені в якості й працездатності свого проекту. Чим тестів більше, тим краще, у розумних межах звичайно. Основними двома обмежувачами на кількість тестів буде:

1. час інтеграції - складання як і раніше повинен залишатися швидким, основне тестування можна перенести «на ніч»,
2. наявність автоматизованих тестів - не всі тести вимагають автоматизації, нема рації робити автоматизовані тести тільки для самих тестів, вони повинні бути доцільні.

Чим краще ваші тести, тим раніше знаходяться помилки й раніше виправляються. Як відомо, чим раніше помилка виправлена, тим дешевше її виправлення. Це одне з основних переваг практики безперервної інтеграції - зниження вартості виправлення помилок (не всіх звичайно). Попутно наявність гарного набору тестів у процесі інтеграції дає більше впевненості в тім, що проект працює правильно.

Саме присутність тестів - одне з відмінностей інтеграції від натискання кнопки Build у вашої улюбленої IDE.

Інтеграція на спеціальній машині

Організовувати процес необхідно на спеціально виділеній машині. Така машина за своєї конфігурацією і набором прикладних програм повинна максимально відповідати оточенню в якому проект буде розгорнутий (production enviroment). Очевидно, що повного збігу досягти практично неможливо - малоймовірно, що експлуатуватися програма буде на машині зі установленими засобами складання, тестування й т.п. Але точний збіг версій операційних систем (і сервіс паків) є обов’язковим.

При цьому, це не повинна бути машина розроблювача або когось ще, це повинна бути виділена машина (можна віртуальна). Адже найчастіше проект, зібраний на машині одного розроблювача, не збирається на машині іншого. Виділення машини для цілей інтеграції дозволяє зменшити ризик пов'язаний з конфігурацією програмного й апаратного забезпечення.

Методи

Continuous Integration сервер

Хоча в принципі практика continuous integration не вимагає ніякого технічного й програмного забезпечення, набагато зручніше, простіше й дешевше налагодити процес із використанням таких засобів. Такі засоби називаються сервера інтеграції (continuous integration server)- спеціалізовані додатки для автоматизації даного процесу.

Найбільш відомий із серверів інтеграції мабуть CruiseControl. CruiseControl це сервер для інтеграції додатків на java, написаний на java. Так само широко розповсюджений його побратим (точніше портована версія) під .NET — CruiseControl.NET.

Нижче наведена схема організації такого сервера інтеграції:

Ручний процес

Хоча рішення з виділеним сервером для continuous integration здається простим і дешевим, у нього є супротивники. Точніше, прихильники ручного процесу. Один з таких Джеймс Шор (James Shore) у своїй статті Continuous Integration on a Dollar a Day пише, як правильно організувати continuous integration процес без спеціалізованих додатків, начебто CruiseControl. У цій статті ми не звертаємося до даного питання.

Процес інтеграції

Continuous integration процес складається з декількох етапів, деякі з яких обов'язкові, інші немає:

- **Trigger** — обов'язковий
- **Update** — обов'язковий
- **Analyse** — не обов'язковий
- **Build** — а як без нього?
- **UnitTest** — вкрай бажаний
- **Deploy** — потрібний по обставинах
- **Test** — не обов'язковий, але вкрай бажаний
- **Archive** — бажаний
- **Report** — обов'язковий

Trigger

Цикл інтеграції починається зі спрацьовування тригера. Це може бути одне з наступних подій:

- зміна в системі контролю версій
- зміна у файловій системі
- певний момент часу
- складання іншого проекту
- натиснута «червона» кнопка
- зміна на веб сервері

Варто відзначити, що не всі CI сервера підтримують всі можливі варіанти тригерів, але основні (система контролю версій і файлова система) підтримуються більшістю.

Характерним прикладом буде випадок, коли один з розроблювачів робить комміт у систему контролю версій. Для інтеграційного сервера це означає, що у вихідному коді проекту відбулися зміни й необхідно провести складання для

перевірки того, що ці зміни нічого не зіпсували й погодяться з раніше зробленими. Після цього йде наступний етап.

Update

На даному етапі CI сервер робить update своєї локальної копії вихідного коду проекту. У процесі update з'ясовуються зміни в коді (і не тільки), які відбулися з останньої інтеграції. З'ясування змін необхідно для того, щоб у випадку збою можна було легко з'ясувати причину й знайти відповідального.

Analyse

Після того, як свіжа версія проекту витягнена із системи контролю версій, але складання ще не почате, можна провести статичний аналіз коду. Існує безліч автоматичних засобів, для різних мов програмування, що дозволяють провести такий аналіз. Звичайно виміряються наступні характеристики коду:

- наявність типових помилок
- статичні характеристики коду: складність, розмір, інше
- відповідність прийнятим стандартам кодування
- інше

Даний етап є необов'язковим для процесу continuous integration, але у випадку його наявності можна одержати додатковий переваги від введення практики у вигляді метрик по коду. Даний етап має на увазі не тільки одержання статичних характеристик коду, але і їхнє включення у звіти створювані сервером інтеграції.

Build

Один з основних етапів процесу це складання проекту. Тут відбувається компіляція (трансляція) вихідних кодів у здійсненні файли або якийсь інший результат. Оскільки сервер інтеграції являє собою спеціально виділену машину з чіткою конфігурацією, результат тільки цього складання можна вважати кінцевим. Більше ніяких «Проект збирається на моїй машині!». Є тільки одне місце, де проект може збиратися - це інтеграційний сервер.

Природно складання є обов'язковим етапом інтеграції.

UnitTest

У методології Extreme Programming модульне (unit) тестування є невід'ємною частиною розробки додатка. Модульні тести споконвічно автоматизовані, їхнє включення в процес інтеграції вкрай бажано. Оскільки часто розроблювачі не мають часу або бажання запускати такі тести перше ніж зміни відправлені в систему контролю версій, додаткове їхнє виконання ніколи не буде зайвим. Додаткову інформацію можна витягти, вимірюючи покриття модульних тестів. Ця метрика допоможе краще контролювати якість продукту, що випускає.

Природно, при відсутності самих тестів у проекті цей етап не виконаємо. Хоча наявність поставленого процесу безперервної інтеграції без модульних тестів змушує задуматися про їхню необхідність.

Deploy

Після того як ми переконалися в деякій працездатності проекту — він збирається (етап Build) і всі модульні тести проходять (UnitTest) проект необхідно «розгорнути». У випадку веб-додатка- це викладення на веб-сервер (сервер додатків) і запуск. Для GUI додатків це (пере)встановка в системі.

Етап розгортання повинен проходити як можна більше «чисто». При цьому для наступного тестування часто необхідно привести додаток у якесь «стандартне» стан:

- «залити» дамп бази
- настроїти в стандартному режимі
- забрати сліди попередньої діяльності додатка

Test

Після того, як додаток «розгорнуто» необхідно його протестувати. Можна через автоматичні функціональні тести, інакше кажучи на даному етапі проводиться регресійне тестування.

Після проходження регресійних тестів можна вважати, що інтеграція пройшла успішно й у проект не внесено виправлень, які можуть привести до його непрацездатності (тут все залежить від вашого набору тестів модульних і функціональних). У противному випадку інтеграція не успішна - код містить помилки й потрібно його виправлення/доробка.

Тестування це один з «фатальних» етапів процесу, помилка на якому означає збій складання. Усього є кілька таких «фатальних» етапів:

- **Build** — проект не збирається
- **UnitTest** — модульні тести не пройшли або покриття впало нижче заданого рівня
- **Test** — регресійні тести не пройшли або покриття впало нижче заданого рівня

Іноді до них приєднують етап **Analyse** — якщо в коді виявлена невідповідність стандартам кодування, то це є помилкою.

Archive

Після того як досягнута максимальна впевненість як вихідний код необхідно зберегти його. Це можна зробити, наприклад, за допомогою міток у системі контролю версій. Так само необхідно зберегти бінарні файли проекту. Вони можуть знадобитися, якщо потрібно буде відтворити помилку в конкретній версії й для ручного тестування.

Continuous integration процес можна використати як формалізацію процесу передачі версії проекту на тестування. Наприклад можна настроїти сервер

публікувати свіжу версію кожні два тижні й сповіщати про це тестувальникам по електронній пошті. Тестувальники завжди будуть знати звідки брати свіжу й «правильну» версію. А наявність регресійних і модульних тестів є свого роду первинним (smoke) тестуванням і гарантує (до деякої міри звичайно) працездатність даної версії. У такий спосіб на тестування не потрапить версія, що має істотні недоліки перешкоджаючому тестуванню.

Report

Наприкінці йде важливий етап генерації й публікації звітів. Звіти містять у собі наступне:

- причина складання - наприклад зміни в репозиторію
- зміни у вихідних кодах - тут можливі два варіанти зміни від останнього складання або від останнього успішного складання
- звіти по статичному аналізі коду - всі результати які є
- лог складання
- лог модульних тестів - які тести пройшли і, що важливіше, які не пройшли
- лог регресійних тестів - аналогічно модульним тестам
- статистика складань проекту:
  - загальне число вдалих/провалених складань
  - розподіл удалих/провалених складань у часі
  - статистика результатів статичного аналізу коду
- всі інші метрики використовувані й збирають у проекті - це допоможе менеджерові проекту бачити все й відразу

Механізм публікації звіту може бути різний і навіть не один. Це може бути IRC або jabber бот, розсилання по електронній пошті, публікація на web або ftp сервері, спеціалізовані клієнти що дозволяють довідатися статус складання.

Найбільш ефективна публікація результатів декількома різними методами відразу. Наприклад розсилання короткого листа команді, тільки у випадку провалу складання, і публікація повного звіту на веб сервері.

Для правильної організації даного етапу важливо розуміти кого і як необхідно сповіщати про результати інтеграції. Тут треба вибрати між двома крайностями - сповіщати завжди або ніколи. Зразкове рішення цього завдання буде таким:

- розроблювачі - мінімум при збої інтеграції, у противному випадку, як розроблювач довідається, що внесені їм зміни зламали код? Звичайно, можна сповіщати й завжди, це залежить від частоти складань.
- тестувальники - якщо вони входять у команду, то сповіщати тоді ж, коли й розроблювачів, адже іноді помилки можуть бути й у тестах. Якщо практикується незалежне тестування взагалі не сповіщати їх або сповіщати при закінченні інтеграції.
- менеджер проекту – суто по бажанню

Профіти

Так яку користь можна одержати об впровадження безперервної інтеграції у своєму проекті?

У першу чергу це безболісна інтеграція всього проекту. Інтеграція різних модулів і виправлень різних програмістів перестає бути справою в принципі, вона відбувається «сама» без участі людей й якщо щось не так, ви про це довідаєтеся. Звичайно, зараз рідкість, що проект має особливу стадію інтеграції, коли з купи різних модулів намагаються зробити додаток, але все-таки не треба недооцінювати користь від безперервної інтеграції.

Більше ніяких «Це працює на моїй машині!». Якщо щось не працює на складальному сервері - значить воно не працює взагалі. Аргументи програміста, що в нього все працює в цьому випадку не допоможуть. Сервер інтеграції ставати суддею в таких питаннях і цьому судді безсторонній.

Всі аналізатори коду й тести, які ви використаєте й написали, обов'язково запускаються над кожним складанням. Якщо в систему контролю версій потрапив «поганий» код - ви про це довідаєтеся. І не важливо, чи порушений один зі стандартів кодування, або статичний аналізатор коду показує, що в код потрапила потенційна помилка або тести не пройшли, а може просто покриття коду модульними тестами впало нижче необхідного мінімуму. Ви про це довідаєтеся й зможете вжити заходів.

Більше того, запуск всіх цих аналізаторів корисний не тільки для визначення стану в сучасний момент часу, але й для аналізу тенденцій. Можна побачити, коли ваш код став сильно більше, складніше, у яких модулях ця складність сконцентрована. Так, це вимагає наявності й використання відповідного інструментарію.

Чим більше й серйозніше проведена робота з настроювання сервера інтеграції, тим більше користі можна одержати. Якщо ваш сервер просто збирає проект після кожної зміни в коді, то користь від нього не так велика, але й зусиль на нього майже не витрачено.

Continuous Improvement

Після того як ви налагодили процес безперервної інтеграції вам може здатися, що справа зроблена: сервер працює, білди збираються, пошта йде й всі добре, поки немає ніяких НП (начебто поломки сервера). Але це не так. Сам процес вимагає постійного налагодження, підстроювання. Якщо спочатку у вас не було ніяких тестів, то їх потрібно зробити. Після цього ви захочете збирати інформацію про покриття вашого додатка тестами, потім зміну такого покриття в часі, можливо якісь ще специфічні метрики. Ну, а якщо виявилося, що більше поліпшувати нема чого, почекайте якийсь час і таку необхідність з'явиться.

Посилання та Публікації

- **Continuous Integration** — стаття Мартіна Фаулера (Martin Fowler) по Continuous Integration
- **Continuous Integration on a Dollar a Day** — стаття Джеймса Шора (James Shore) про ручний метод інтеграції
- **Automated Continuous Integration and the Ambient Orb** — стаття Майка Свонсона (Mike Swanson)
- **«Введение в непрерывную интеграцию или каша из топора»** — стаття Андрія Сатарина, представлена на конференції **SEC(R) 2008**. Так само доступна **презентация** і **презентация-handout**.
- **c2.com** — стаття по Continuous Integration в wiki на c2.com
- **en.wikipedia.org** — стаття по Continuous Integration в Wikipedia
- **ru.wikipedia.org** — стаття по безперервній інтеграції у Википедии
- **www.pragprog.com** — книга «Pragmatic Project Automation». **Глава**, яка присвячена безперервної інтеграції доступна безкоштовно.
- **www.amazon.com** — книга Підлоги Дюваля (Paul Duvall) про безперервну інтеграцію, що одержала Jolt Award в 2008 році
- **www.ozon.ru** — ця ж книга видана російською мовою

Інструменти

Сервера інтеграції

- **CruiseControl** — сервер інтеграції для Java (див. так само **CruiseControl**).
- **ThoughtWorks Cruise** — комерційний сервер інтеграції від компанії ThoughtWorks (є безкоштовна версія).
- **CruiseControl.NET** — сервер інтеграції для .NET (див. так само **CruiseControl.NET**)
- **CruiseControl.rb** — сервер інтеграції для Ruby.
- **Hudson** — open-source сервер інтеграції, створена як альтернатива **CruiseControl**. Функціональність розширюється плагінами.

- **Bitten** — open-source сервер інтеграції написаний на **Python**, інтегрується з **Trac**.
- **TeamCity** — комерційний сервер інтеграції від компанії JetBrains для java й .NET (є безкоштовна версія).

Інструменти складання

- **Ant** — засіб складання для Java
- **Maven** — засіб складання для Java
- **NAnt** — аналог Ant під .NET

Статичний аналіз

- **PMD** — аналіз коду Java
- **Findbugs** — аналіз коду на типові помилки
- **Simian** — пошук повторів (copy+paste) у коді Java
- **FXCop** — аналіз коду .NET
- **QALab** — об'єднання логів декількох інструментів аналізу коду, збір статистики
- **Panopticode** — об'єднання логів декількох інструментів аналізу коду для Java, графічне подання результатів

Модульне тестування й покриття

- **JUnit** — де-факто стандарт модульного тестування Java
- **TestNG** — інструмент нового покоління для модульного тестування Java
- **NUnit** — модульні тести для додатків .NET
- **Cobertura** — вимірювання покриття коду модульними тестами для Java
- **Clover** — аналіз покриття коду тестами для Java
- **Clover.NET** — аналіз покриття коду тестами під .NET

Розділ 7. Менеджмент програмних проектів

Лекція 23. Управління проектами

- Задачі управління проектами.
- Трикутник обмежень.
- Управління змістом та якістю проекту. [4, с. 562-597]

Поняття проекту

Традиційне (радянське) розуміння проекту - сукупність документації по створенню будь-яких об'єктів. На Заході це називається дизайном або інжинірингом.

У сучасній західній практиці та літературі ПРОЕКТ - процес цілеспрямованої зміни технічної або соціально-економічної системи, що переводить її з одного стану в інший. Основа західної. концепції проекту - погляд на проект як на щось суцільне протягом усього його життєвого циклу.

ПРОЕКТ – це деяка задача з певними. початковими даними і бажаними результатами/цілями, що обумовлюють способи її рішення.

ПРОЕКТ: одноразова сукупність взаємопов’язаних дій, які здійснюються з певною метою, протягом певного часу, при встановлених ресурсних обмеженнях. Це найбільш повне визначення.

Проект, як і будь-яка діяльність, має низку властивих йому рис, знання яких допоможе здійснити ефективну реалізацію проекту.

До таких *рис* можна віднести наступні:

1. Виникнення, існування та закінчення проекту у певному оточенні;
2. Зміна структури проекту протягом життєвого циклу;
3. Наявність певних зв’язків між елементами проекту як системи;
4. Можливість відміни вхідних ресурсів проекту.

Виходячи з визначення проекту можна виділити наступні характеристики (ознаки) проекту:

1) спрямованість на досягнення конкретної цілі/цілей, які можуть бути досягнуті з одночасним виконанням низки технічних, економічних та інших вимог;

2) координоване виконання взаємопов’язаних дій (внутрішні та зовнішні взаємозв'язки операцій, задач і ресурсів включно);

3) обмежена протяжність у часі (будь-який проект має чітко визначений термін початку і термін завершення);

4) обмеженість ресурсів (будь-який проект має свій обсяг матеріальних, людських та фінансових ресурсів, які використовуються за встановленим і лімітованим бюджетом);

5) певна міра неповторності та унікальності (як мети, так і умов його здійснення);

6) неминучість різних конфліктів (ризик).

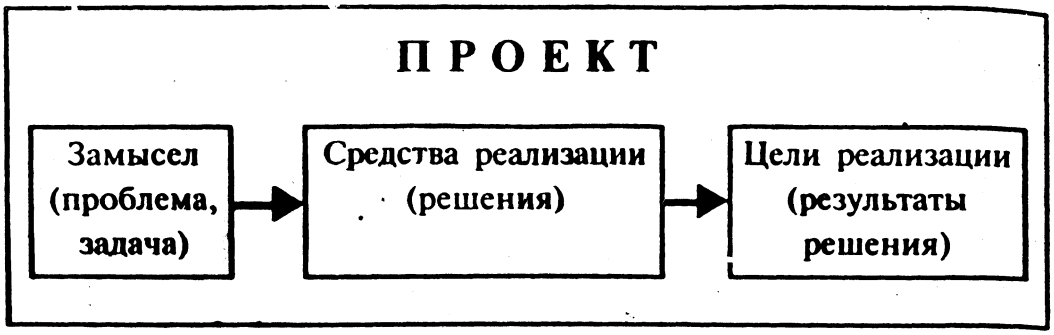


Рис. 2.1.1. Основные элементы проекта

Причини виникнення і сутність управління проектами

Будь-який проект проходить ряд етапів/фаз/стадій. Для проведення проекту через всі фази ним треба управляти.

Необхідність методології управління проектами, усвідомлена в середині 50-х років в розвинених країнах світу, викликана масовим зростанням складності і кількості проектів, а також посиленням впливу наступних чинників:

1) вимоги замовників і збільшення їх компетентності
2) складність кінцевих продуктів проектів
3) взаємозв'язок і взаємовплив зовнішнього оточення проекту
4) міра невизначеності і ризику
5) організаційні перебудови
6) частота зміни технологій
7) помилки планування і ціноутворення

Вплив цих чинників призводив до порушення термінів здійснення проекту, перевитрат коштів, невиконання вимог до характеристик кінцевої продукції, що вело до зменшення прибутку, а нерідко -- і до збитків.

Методи управління проектами дозволяють:

1) визначити цілі проекту і провести його обгрунтування
2) виявити структуру проекту: підцілі, основні етапи, роботи
3) визначити необхідні об'єми і джерела фінансування
4) підібрати виконавців, зокрема через торги, тендери, конкурси
5) підготувати і укласти контракти
6) визначити терміни виконання проекту
7) скласти графік реалізації проекту
8) розрахувати необхідні ресурси
9) розрахувати кошторис і бюджет
10) спланувати і врахувати ризики
11) забезпечити контроль за ходом виконання проекту

Для методології управління проектами характерно зосередження прав і відповідальності за досягнення цілей на одній людині або невеликій групі.

Інститут управління проектами США дає визначення:

"Управління проектами - мистецтво керування, і координації людських і матеріальних ресурсів протягом життєвого циклу проекту шляхом застосування системи сучасних методів і техніки управління для досягнення визначених в проекті результатів за складом і об'ємом робіт, вартістю, якістю і задоволенням потреб учасників проекту."

Управління процесом виконання проекту

- Декомпозиція функцій в управлінні проектами
- Чим управляє проект?
- Процесний підхід до виділення функцій в управлінні проектами

Декомпозиція функцій в управлінні проектами

Концепція управління проектом може розглядатися в різних аспектах. Найбільш поширеними є:

1. функціональний;
2. динамічний;
3. предметний.

Функціональний - найбільш універсальний, передбачає розгляд основних функцій управлінської діяльності: аналіз, планування, організація, контроль.

Динамічний - дозволяє визначити конкретний зміст функцій на кожному етапі здійснення проекту; передбачає розгляд у часі всіх процесів, пов'язаних з основною діяльністю по виконанню проекту. Цей процес пов'язаний з логікою розвитку робіт і визначає так зване спеціальне управління реалізації проекту, яке включає аналіз проблеми, розробку концепції проекту, базове і детальне проектування, будівельно-монтажні і пусково-налагоджувальні роботи, експлуатацію і демонтаж.

Предметний підхід визначає об'єкти проекту, на які направлене управління.

Крім названих аспектів в управлінні проектами з метою декомпозиції функцій використовується такий аспект, як *рівень діяльності*. Виділяються 2 види такого розділення: організаційний рівень і масштаби діяльності по управлінню.

Організаційний рівень: проект загалом; міжфірмові утворення; організації-учасники; окремі колективи розробників.

Масштабність діяльності: політика; стратегія; тактика; функції; процедури; операції.

Чим управляє проект

Базові функції управління проектом:

1. управління предметною областю;
2. управління якістю;

3. управління часом;
4. управління вартістю.

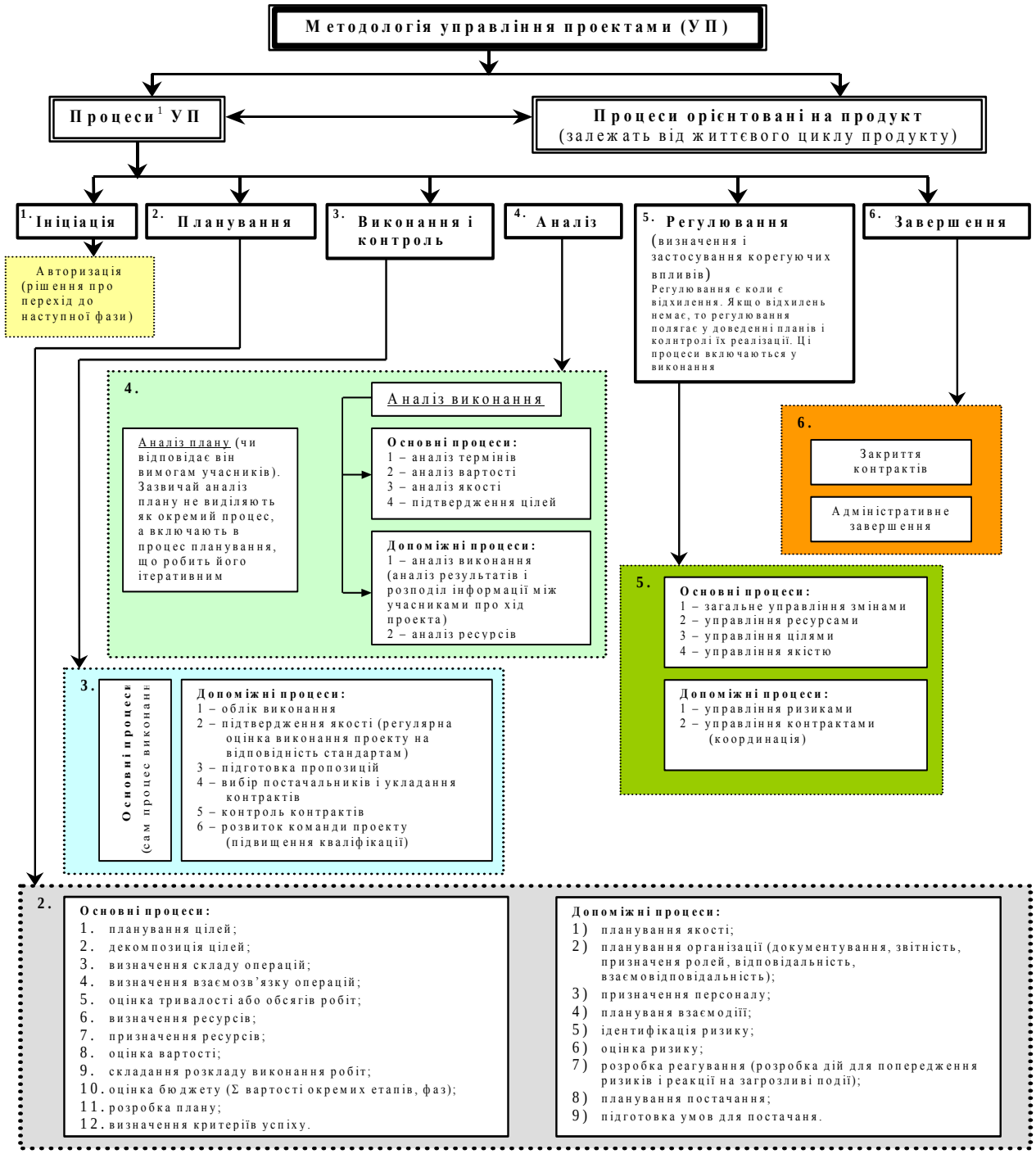
Виділяються 4 інтегруючих функції:

1. управління ризиком;
2. управління персоналом;
3. управління контрактами і забезпеченням проекту;
4. управління взаємодією і інформаційними зв'язками.

Виділення вказаних функцій обумовлюється такими критеріями оцінки:

1. технічна здійсненність, що визначається предметною областю і якістю;
2. конкурентоздатність, що визначається якістю, часом і вартістю;
3. трудомісткість, що визначається часом і вартістю;
4. життєздатність;
5. ефективність здійснення проекту, що визначається персоналом, що бере участь, засобами комунікації, системою матеріально-технічного забезпечення.

Процесний підхід до виділення функцій в управлінні проектами



[1] **Процеси** – дії та процедури, пов'язані з реалізацією функцій управління

Управління якістю

1. Сутність управління якістю та його основні функції
2. Інтеграція функцій забезпечення якості
3. Способи і техніка управління якістю проектів ІС
4. Розробка системи управління і забезпечення якості програмних продуктів у відповідності з ISO-9000

1. Сутність управління якістю та його основні функції

Складові якості проекту:

1. матеріали, що використовуються, обладнання, сировина;
2. якість робіт, що виконуються;
3. якість отриманих результатів.

Управління якістю реалізовується через встановлення вимог і стандартів до якості результатів проекту; забезпечення виконання вимог до якості через систему контролю і підтримки.

Управління якістю (слайд)

2. Інтеграція функцій забезпечення якості

Опрацювати самостійно з літературних джерел!

3. Способи і техніка управління якістю проектів ІС

Опрацювати самостійно з літературних джерел!

4. Розробка системи управління і забезпечення якості програмних продуктів у відповідності з ISO-9000

1. Загальна характеристика стандартів ISO-9000 із забезпечення якості продукції чи послуг
2. Необхідність розробки системи управління по забезпеченню якості ПП
3. Відповідальність керівництва
4. Система якості
5. Аналіз контракту
6. Управління проектуванням
7. Управління документацією і даними
8. Закупки
9. Ідентифікація та відслідковування продукції
10. Управління процесами
11. Контроль та випробування
12. Управління продукцією, що не відповідає вимогам
13. Коригуючі та запобіжні дії
14. Погрузочно-розгрузочні роботи, зберігання, упаковка, консервація та поставка
15. Управління реєстрацією даних про якість
16. Внутрішній аудит якості
17. Підготовка кадрів
18. Обслуговування

4.1. Загальна характеристика стандартів ISO-9000 із забезпечення якості продукції чи послуг

ISO-9000 - стандарт на якість проектування, розробку, виготовлення та післяпродажного обслуговування. ISO-9000 визначає базовий набір заходів з контролю якості та містить схему функціонування бізнес-процесів підприємства, яке забезпечує якість його роботи.

ISO-9000, не є стандартом якості власне для товарів та послуг, що виробляє підприємство. Схема "покриває" всі етапи виробничого циклу випуску товарів/послуг:

1. закупку сировини і матеріалів
2. проектування
3. створення і доставка товарів
4. обслуговування клієнтів
5. навчання персоналу

Власне ISO-9000 регламентує два ключових момента:

- наявність і документування відповідного бізнес-процесу
- вимірювання його якості

Насправді ISO -9000 це серія стандартів з управління якістю і забезпечення якості.

Є чотири частини ISO -9000:

1. 9000-1 - настанови щодо вибору і застосування
2. 9000-2 - настанови щодо застосування ISO-9001, ISO-9002 та ISO-9003
3. 9000-3 - настанови щодо застосування ISO-9001 до розроблення, поставляння та супроводження ПЗ.
4. 9000-4 - настанови щодо управління програмною надійністю.

ISO -9001- системи якості - модель забезпечення якості в процесі проектування розроблення, виробництва, монтажу та обслуговування. Цей стандарт є найбільш повним. Він специфікує модель забезпечення якості на всіх етапах життєвого циклу товару/послуги.

ISO -9002 - системи якості - модель забезпечення якості в процесі виробництва, монтажу та обслуговування.

ISO-9003 - системи якості - модель забезпечення якості в процесі контролю готової продукції та її випробування.

ISO-9004-1 - управління якістю та елементи системи якості: Частина 1: настанови

ISO-9004-2 - управління якістю та елементи системи якості: Частина 2: настанови щодо послуг.

ISO-9004-3 - управління якістю та елементи системи якості: Частина 3: настанови щодо перероблюваних матеріалів.

ISO-9004-4 - управління якістю та елементи системи якості: Частина 4: настанови щодо поліпшення якості.

Сертифікація підприємства за стандартом ISO-9000 включає наступні три етапи:

- Застосування стандартів на підприємстві, яке полягає в розробці і введення в дію низки засобів пропонованих стандартами;
- Проведення власної сертифікації акредитованими ISO –органами;
- Періодичні (два рази на рік) перевірки підприємства на наслідування (відповідність) стандартів;

Сертифікація за ISO-9000 є добровільною справою кожного підприємства. Основною причиною сертифікації є те що закордонні компанії вимагають наявності сертифіката від своїх постачальників. Більш того, наявність сертифіката може бути обов'язковою умовою участі підприємства в міжнародних тендерах, держзамовленнях, а також отримання пільгових кредитів та страховок.

4. 2. Необхідність розробки системи управління по забезпеченню якості ПП

В середині 80-х років за постановою ДКНТ СРСР, програми стали продукцією технічного призначення. Але і до тепер, більшість програмістських фірм не спромоглися довести організацію процесу розробки програмних систем до рівня "індустріального". Для налагодження такого процесу недостатньо мати висококваліфікованих, талановитих фахівців, необхідними є:

- Чітка організація спільної роботи команди розробників;
- Загальне управління проектом, яке включає не тільки розподіл обов"язків, а і розподіл відповідальності.

Система якості є організаційним стрижнем створення оптимальних умов для продуктивної праці фахівців. Вона дозволяє перейти від кустарного рівня створення програм до наукового , організованого масового виробництва програмного продукту, завдяки застосуванню особливих методів управління якістю. Ці методи варіюються від компанії до компанії, але основні їх положення єдині для всіх і визначаються стандартом ISO-9003.

Цей стандарт включає усі положення загального стандарту ISO-9001, а а також необхідні доповнення до них, що стосуються розробки, поставки та обслуговування ПЗ.

ISO-9001 встановлює вимоги до системи якості постачальника і дозволяє оцінити його можливості з проектування та постачання продукції , що відповідає цим вимогам.

Вимоги стандарту мають на меті задоволення запитів користувача через попередження появи будь-яких невідповідностей продукції на усіх стадіях її життєвого циклу від проектування до обслуговування ПЗ.

Основні поняття, якими оперує стандарт:

1. Продукт – результат дії або процесів;
2. Програмний продукт- набір комп"ютерних програм, процедур і , можливо, пов'язаних з ними документів та даних;

3. Елемент програмного забезпечення- (software item)-будь-яка частина програмного продукту, що ідентифікована;
4. Основа (baseline)- формально узгоджена версія елемента конфігурації , зафіксована у певний момент часу на протязі життєвого циклу елемента конфігурації;
5. Розробка(development)- процес життєвого циклу ПП, що охоплює аналіз вимог, проектування , кодування, інтеграцію, тестування,установку та підтримку;
6. Модель життєвого циклу- (life cycle model)- , базова модель, яка включає процеси, дії та задачі, що мають місце у розробці, функціонуванні та супроводі ПП і охоплюють увесь життєвий цикл системи від визначення вимог до завершення
7. Етап- певний сегмент роботи ;
8. Регресійне тестування- (regression testing)- тестування , що дозволяє упевнитися в тому, що зміни, внесені з метою виправлення виявлених помилок , не породили нових;
9. Реплікація- (replication)- копіювання програмного продукту з одного носія на інший.

Стандарт впроваджується на добровільній основі і зобов”язує підприємства суворо регламентувати певні аспекти виробничої діяльності. Їх перелік наведено в плані лекції п. 4.1-4.18.

4.3. Відповідальність керівництва

На керівництво компанії –постачальника накладається задача визначення політики та своїх зобов”язань з якості.

Ця політика повинна бути узгодженою як з цілями самої компанії , так і з очікуваннями і запитами самого користувача.

Постачальник повинен забезпечити розуміння цієї політики службовцями компанії ,її проведення та підтримку на усіх рівнях.

Керівництво визначає і документально оформляє відповідальність, повноваження та механізми взаємодії персоналу, який виконує та перевіряє роботу, що впливає на якість.

Керівництво призначає менеджера з якості з відповідними повноваженнями для забезпечення розробки, впровадження та підтримки в робочому стані системи якості та надання звітів про її функціонування , які дозволяють проаналізувати та удосконалити систему якості.

4.4.Система якості

Компанія-постачальник повинен розробити, документально оформити та підтримати в робочому стані систему якості як засіб, що забезпечує відповідність продукції вимогам стандарту ISO-9001.

Система якості передбачає наявність:

- Інструкції з якості, яка включає методики системи якості компанії;
- Опис структури документації , що використовується в системі якості.

Масштаб та ступінь деталізації методик системи якості залежать від складності роботи, методів, що використовуються, необхідних навичок та роботи персоналу.

Постачальник також визначає та документально оформляє дії, пов”язані з реалізацією вимог до якості, тобто планує якість – формулює вимоги до якості, описує модель ЖЦ, задає критерії початку і кінця кожного етапу проекту, визначає типи аналізу , тестування та інших дій з перевірки та затвердження ПП, визначає процедури управління конфігурацією. Планування якості “настроює” систему якості на певний проект.

4.5. Аналіз контракту

Система якості передбачає певні дії з аналізу контракту.

ПП може розроблятись:

1. За контрактом із замовником;
2. Як комерційний продукт для певного сектора ринку;
3. Як система, вмонтована в апаратне забезпечення;
4. Як система підтримки бізнес-процесів постачальника.

Загальні положення аналізу контрактів, визначені в ISO 9000-3, прийнятні для будь-якої з цих ситуацій.

Постачальник повинен попередньо проаналізувати заявку на тендер, контракт чи замовлення (до надання заявки або укладання контракту). Це дозволить гарантувати адекватне визначення вимог до проекту і можливість виконати контракт.

У процесі аналізу контрактів на ПЗ можуть враховуватись різноманітні аспекти взаємодії з замовником, технічні міркування,аспекти управління та фактори забезпечення захисту і конфіденційності.

Наприклад, можуть аналізуватися:

1. узгоджені із замовником критерії прийняття або відмови від готової системи;
2. термінологія;
3. ступінь участі замовника у спільній роботі;
4. відповідальність замовника за надання певних даних або обладнання;
5. зобов'язання замовника перед постачальником із заміни на нові версії системи або зобов”язання постачальника підтримувати старі версії , тощо.

Тут можуть розглядатись стандарти та процедури , які будуть використовуватись при розробці програмної системи , операційна та апаратна платформи, вимоги до реплікації(тиражування) і розподілення системи, вимоги до установки, супроводу та підтримки і низка інших.

4.6.Управління проектуванням

Це найбільший розділ стандарту, оскільки торкається базової складової загального процесу створення ПП , яка найбільше впливає на його якість.

Постачальник розробляє та документує методики управління та верифікації проекту з метою забезпечення виконання встановлених вимог.

Цей розділ стандарту IS0 9000-3 містить вказівки з таких основних дій в процесі розробки:

1. аналіз вимог до проекту;
2. проектування архітектури систем;
3. детальне проектування та кодування;
4. планування розробки.

Проект розробки ПП організується за певною моделлю ЖЦ. ISO 9000-3 не визначає, якою повинна бути модель ЖЦ, це залежить від специфіки задачі. Стандарт наводить лише визначення моделі ЖЦ як сукупності процесів. Модель показує, коли ці процеси і як підключаються до реалізації проекту.

Відомо,що розробка системи – це процес початкових вимог до кінцевого продукту. Стандарт оговорює , що цей процес повинен проводитись у чітко визначеному порядку, що дозволить попередити появу помилок та знищить залежність від процесів перебірки та затвердження як єдиних методів виявлення проблемних ситуацій. Вимога строго дотримуватись дисципліни процесу розробки означає наявність та підтримку в робочому стані документованих процедур, які і стануть гарантією того ,що ПП створюється у відповідності з заданими вимогами та планами розробки і забезпечення якості.

Управління проектом повинно враховувати такі аспекти як:

1. метод проектування та його відповідність конкретній задачі;
2. досвід попередніх проектів;
3. вимоги наступних процесів;
4. тестування, установки, супроводу та використання;
5. вимоги до захисту та безпеки;
6. спеціальні вимоги до стійкості систем або до її зворотніх дій на потенційні аварійні ситуації.

Для процесів кодування необхідно задавати правила використання мов програмування, принципи кодування та правила розробки відповідних коментарів.

Інструментальні засоби і методи, що використовуються при розробці ПП (системи аналізу та проектування, компілятори) повинні попередньо затверджуватись і контролюватися системою конфігураційного управління.

Область застосування інструментарію повинна бути задокументована,а його використання періодично аналізуватися, з метою встановлення необхідності удосконалення інструментальних систем або заміни на нові продукти.

Проектування та розробка повинні ретельно плануватись. План розробки ПП повинен формулювати задокументовані дії з:

1. аналізу вимог до систем;
2. проектування;
3. кодування;
4. інтеграції;
5. тестування ;
6. встановленню
7. підтримки.

План повинен включати також плани, пов”язані з основним процесом:

1. забезпечення якості;
2. управління ризиками;
3. управління конфігурацією;
4. плани інтеграції;
5. плани установки;
6. плани навчання співробітників та ін.

Повинні бути визначені і задокументовані принципи організаційно технічної взаємодії між різними групами, які приймають участь у розробці.

Необхідно чітко встановити межі відповідальності кожного учасника і те яким чином технічна інформація буде передаватися між учасниками. Тут же обмовляється відповідальність замовника проекту, якщо він бере участь в розробці: необхідність брати участь в проекті, зобов'язання по своєчасному наданню потрібної інформації. У разі обопільної домовленості між постачальником і замовником може бути запланований спільний аналіз ведення проекту, регулярно або на певних його етапах. Цей аналіз торкається такі чинники, як хід розробки з боку постачальника, участь в розробці з боку замовника, відповідність системи вимогам замовника, результати перевірок, результати тестування.

Вхідні проектні вимоги до продукції. Вимоги формулює замовник, а постачальник аналізує, наскільки вони адекватні. Неповні, двозначні або суперечливі вимоги є предметом урегулювання з особами, відповідальними за їх пред'явлення. У певних ситуаціях, за обопільній згодою, специфікацію вимог може провести постачальник.

Вихідні проектні дані також оформляються документально, причому таким чином, щоб їх можна було перевірити і підтвердити відносно вхідних проектних вимог. Вихідні дані проекту програмної системи можуть включати: специфікацію архітектури проекту, детальну специфікацію проекту, початкові коди, керівництво користувача.

Постачальник програмного продукту повинен планувати і провести офіційний, документально оформлений аналіз результатів проектування. Міра формальності і суворість процесів аналізу відповідають складності системи, що розробляється і мірам ризику, пов'язаного з її використанням. Аналіз проектування торкається таких аспектів, як можливість виконати проект, задоволення вимогам захисту і безпеки системи, виконання правил програмування і можливість тестування.

На певних стадіях проектування проводиться перевірка відповідності вихідних даних вхідним вимогам. Така верифікація проекту може включати аналіз вихідних даних, демонстрації, в тому числі за допомогою прототипів і моделювання, або тестування. Тільки перевірені вихідні проектні дані затверджуються для остаточного прийому і подальшого використання. Всі виявлені в процесі перевірки проблемні ситуації повинні адекватно дозволятися.

Перш ніж система буде передана замовнику, постачальник повинен затвердити систему на відповідність заданому призначенню. Замовнику може бути переданий тільки затверджений програмний продукт.

Всі зміни і модифікації проекту повинні бути ідентифіковані, документально оформлені, проаналізовані і затверджені до їх реалізації. Постачальник встановлює і підтримує в робочому стані процедури управління змінами в проекті, які можуть виникнути на будь-якій стадії життєвого циклу системи.

4.7. Управління документацією і даними

Постачальник повинен розробити і підтримувати в робочому стані документовані процедури управління всіма специфікаціями і даними, що відносяться до вимог стандарту ISO 9001, включаючи і документи зовнішнього походження (стандарти і креслення споживача). Документи і дані можуть розміщуватися на будь-якому паперовому або електронному носії.

Для реалізації контролю за документами і даними можуть використовуватися процедури управління конфігурацією. У область дії цього розділу стандарту попадають контракти, що специфікують вимоги до продукту; документи, що описують систему якості, що містять різні плани постачальника і взаємодію постачальника з споживачем; документи і дані, що описують конкретний програмний продукт.

4.8. Закупівля

Постачальнику ставиться в обов'язок розробити і підтримувати в робочому стані документовані процедури, що гарантують відповідність закупленої продукції встановленим вимогам. При розробці, постачанні, інсталяції і супроводі програмних продуктів як закупівля може фігурувати: комерційне ПЗ; розробка по субконтракту; комп'ютерне і комунікаційне апаратне забезпечення; кошти розробки; персонал, що наймається за контрактом; сервіс підтримки і супроводу; повчальні курси і матеріали.

4.9. Ідентифікація та відслідковування продукції

Постачальник повинен встановити і підтримувати в робочому стані процедури ідентифікація елементів програмного забезпечення на всіх етапах, від специфікації вимог до розробки, реплікації і доставок. Протягом життєвого циклу системи повинні працювати процедури відстеження компонентів складових частин програмного забезпечення.

Процедури ідентифікації і відстеження реалізовуються в системі конфігураційний управління, що визначається як дисципліна управління, відповідно до якої здійснюється технічне і адміністративне керівництво розробкою і підтримкою життєвого циклу елементів конфігурації, включаючи елементи програмного забезпечення. Ця дисципліна застосовна також до документації і апаратного забезпечення, що стосується програмного продукту, що розробляється. Міра використання конфігураційний управління залежить від розміру і складності проекту, а також від рівня пов'язаного з ним ризику.

Одна з цілей конфігураційний управління задокументувати і забезпечити повну видимість поточної конфігурації продукту і міри відповідності початковим вимогам. Інша мета надати кожному, хто працює над проектом, точну інформацію на будь-якому етапі життєвого циклу. Система конфігураційний управління дає, зокрема, можливість ідентифікувати версію кожного елемента програмного забезпечення, управляти одночасною модифікацією даного елемента двома або більш незалежними розробниками, координувати модифікацію безлічі продуктів і т.д.

У визначенні конфігурації беруть участь: структура продукту і вибір елементів конфігурації, друкарська документація і комп'ютерні файли, угоди по привласненню імен, завдання конфігураційний основ.

4.10. Управління процесами

Постачальник повинен визначити і спланувати процеси розробки, установки і обслуговування продукту. Процес розробки програмної системи організований як безліч процесів, що перетворюють початкові вимоги в програмний продукт (див. розділ "Управління проектуванням"). Розділ стандарту оговорює вимоги до реплікації, доставки і інсталяції програмних продуктів.

4.11. Контроль і випробування

Постачальник повинен розробити і підтримувати в робочому стані документовані процедури контролю і випробувань для перевірки виконання встановлених вимог до продукції.

Контроль і випробування, простіше кажучи, тестування програмного продукту може зажадатися на декількох рівнях, від окремих елементів ПЗ до закінченої системи. Існує декілька підходів до тестування, які даним стандартом не обговорюються. Вибір підходу залежить від постачальника. Об'єм тестування, міра контролю за середою випробувань, вхідний і вихідні дані тестів можуть варіюватися в залежності від вибраного підходу, складності системи і пов'язаних з нею ризиків.

Постачальник повинен визначити, задокументувати і періодично аналізувати план тестування модулів, інтеграційних процесів, системи загалом і тестування для остаточного приймання.

У процесі розробки постачальник повинен контролювати і випробовувати продукцію відповідно до програми якості. Всі види остаточного контролю і тестування також потрібно провести відповідно до цієї програми, щоб

пересвідчитися, що готовий продукт відповідає встановленим вимогам. Для виробника програмних систем це означає, що перш ніж продукт буде переданий замовнику, постачальник повинен офіційно підтвердити, що продукт працює відповідно до заданого призначення, в умовах, схожих з тими, в яких буде застосовуватися система. Будь-які відмінності між середою затвердження системи і фактичною середою її використання, а також ризик, пов'язаний з цими відмінностями, повинні бути виявлені і зареєстровані на можливо більш ранніх етапах життєвого циклу.

Крім тестування для остаточного затвердження, тестування може провести замовник при прийманні вже затвердженого продукту. При цьому він буде визначати, відповідає чи ні продукт раніше узгодженим критеріям. Замовник може передати повноваження на проведення таких тестів постачальнику або третій особі. Постачальник і замовник повинні погодити між собою методи розв'язання проблем, виявлених в ході процедури приймання.

4.12. Управління невідповідною продукцією

Постачальник повинен розробити і підтримувати в робочому стані документовані процедури, що гарантують, що продукт, не відповідний встановленим вимогам, не буде ненавмисно використовуватися або встановлюватися. При розробці програмного забезпечення для ізоляції його невідповідних елементів може зажадатися вивести їх з середи розробки або тестування в окрему середу. У разі розробки вбудованого ПЗ можлива ізоляція апаратного компонента, що містить невідповідний прикладний елемент.

4.13. Коригуючі та попереджаючі дії

Постачальник повинен розробити і підтримувати в робочому стані документовані процедури по виконанню коректуючих і попереджаючих дій. Будь-яка така дія, зроблена для усунення причин фактичних або потенційних невідповідностей, повинна бути адекватно проблемам і враховувати міру ризику.

Якщо коректуючі дії безпосередньо впливають на програмний продукт, може бути підключений процес конфігураційний управління для контролю за змінами. Початкові дані для попереджаючих дій може дати аналіз базових причин невідповідностей, що виникли.

4.14. Вантажно-розвантажувальні роботи, зберігання, упаковка, консервація і постачання

Цей розділ стандарту ISO 9000-3 конкретизує специфіку можливих пошкоджень програмного продукту, засобу зберігання, методи упаковки, консервації і постачання ПЗ.

Пошкодження програмного забезпечення означає зміну його інформаційного вмісту. Стандарт оговорює, що зараження програмного продукту комп'ютерним вірусом вважається пошкодженням ПЗ. Заходи захисту від вірусів описуються в керівництві по реплікації.

Термін "знесення" не застосуємо до тієї інформації, яку містить програмна система. Однак зноситися може носій, на якому зберігається система, і постачальник повинен вжити відповідні заходів обережності.

Постачальник повинен визначити систему зберігання елементів програмного забезпечення, управління доступом до цих елементів і підтримка версій продукту. Для того щоб підтримати цілісність системи і забезпечити базис для управління змінами елементи програмного забезпечення потрібно зберігати в середовищі, яке здатне захистити їх від несанкціонованих змін і забезпечує доступ, що контролюється і вибірку основної і будь-яких інших списів. Крім того, повинні бути враховані умови зберігання носіїв, особливо можливий електромагнітний і електростатичний вплив.

Можливості консервації елементів програмного продукту також зажадають від постачальника створення певної системи, яка буде підтримувати, наприклад, регулярне резервування, гарантоване копіювання ПЗ на змінний носій через певні проміжки часу, зберігання носіїв в захищеному та стійкому до відмов середовищі, що дозволяє гарантувати відновлення у разі катастроф.

4.15. Управління реєстрацією даних про якість

Постачальник розробляє і підтримує в робочому стані документовані процедури ідентифікації, збору, індексування, доступу, складання картотеки, зберігання, ведення і вилучення зареєстрованих даних про якість. Приклади таких даних документовані результати тестування, повідомлення про проблеми, запити про зміни, анотовані документи, результати аналізу, протоколи різних засідань, аудиторські повідомлення.

У тому випадку, якщо дані про якість зберігаються в електронному вигляді, при визначенні часу збереження і доступу до даних необхідно враховувати міру можливої деградації електронних списів і доступність пристроїв і програмних компонентів для доступу до даних.

4.16. Внутрішній аудит якості

Постачальник розробляє і підтримує в робочому стані документовані процедури планування і проведення внутрішніх перевірок якості, які дозволять пересвідчитися, що діяльність в області якості і її результати відповідають запланованим заходам, а також підтвердити ефективність системи якості.

Внутрішні перевірки якості плануються на основі статусу і важливості діяльності, що перевіряється. Внутрішній аудит повинен провестися персоналом, не залежним від осіб, які несуть відповідальність за діяльність, що перевіряється. Внутрішній аудитор аналізує узгодженість програми якості для конкретного проекту із загальною системою якості, прийнятою в організації.

4.17. Підготовка кадрів

Постачальник повинен розробити і підтримувати в робочому стані документовані процедури визначення потреб в підготовці кадрів, а також забезпечувати підготовку всього персоналу, що виконує роботи, які впливають на

якість. Персонал повинен бути кваліфікований на основі відповідного утворення, підготовки і досвіду. Визначення задач навчання повинно ув'язуватися з тими інструментальними засобами, методами і комп'ютерними ресурсами, які будуть використовуватися в розробці програмного продукту і управлінні ім. Можливо, буде також потрібне спеціальне навчання в тій області, з якою пов'язаний програмний продукт, що розробляється.

4.18. Обслуговування

Що стосується програмного забезпечення, то обслуговування тут розуміється як супровід системи (maintenance) і підтримка замовників (customer support). Підтримка замовників обговорюється в стандарті ISO 9000-2.

Супровід системи, як правило, включає в себе виявлення і аналіз невідповідностей в програмній системі, що викликають збої в її роботі; корекцію програмних помилок; модифікацію інтерфейсів, що необхідно у разі внесення додатків або змін в апаратуру; функціональне розширення або поліпшення продуктивності

Всі дії по супроводу повинні провестися і контролюватися відповідно до плану супроводу, який зазделегідь визначається і узгоджується постачальником і замовником.

На закінчення нам залишається лише додати, що технологія розробки програмного забезпечення це ціла наука, якої в Росії, леле, майже не вчать. Звідси явний дефіцит хороших менеджерів і фахівців з комплексних проектів. Загальні положення стандарту по забезпеченню якості лише верхівка айсберга. За межами нашої статті залишилися деталі тих процесів, які реально забезпечують якість кінцевого продукту. Але це, як правило, "ноу-хау" компанії.

Лекція 24. Управління ресурсами, ризиками та конфігураціями

- Управління ресурсами.
- Планування графіку виконання проекту.
- Управління ризиками програмного проекту.
- Управління конфігураціями та змінами [4, с. 562-597]

Планування в УП

1. Необхідність планування в УП
2. Цілі, призначення та види планів
3. Розвиток методів планування
4. Сітьові моделі планування проектів
5. Управління плануванням
7. Процес планування
8. План реалізації проекту

1. Необхідність планування в УП

В управлінні проектом планування займає основне місце, втілюючи в собі організаційні засади всього процесу реалізації проекту. Сутність планування полягає в обгрунтуванні цілей та способів їх задоволення на основі виявлення детального комплексу робіт, визначення ефективних методів та способів, ресурсів усіх видів, необхідних для їх виконання, встановлення взаємодії між організаціями-учасниками проекту. Діяльність по розробці планів охоплює всі етапи проектного циклу. Вона починається з участі проект-менеджеру у процесі розробки концепції проекту, продовжується при виборі стратегічних рішень виконання проекту та розробці його деталей, включаючи складання контрактних пропозицій, укладання контрактів, виконання робіт, і закінчується при завершенні проекту. На етапі планування визначаються всі необхідні параметри реалізації проекту - тривалість (в цілому, окремих етапів та робіт), потреби у трудових, матеріально-технічних та фінансових ресурсах, терміни поставки сировини, матеріалів, комплектуючих і технологічного обладнання, терміни та об'єми залучення проектних, будівельних та інших організацій. Прийняті рішення повинні забезпечити можливість реалізації проекту у заданий термін з мінімальною вартістю та затратами ресурсів та при високій якості виконання робіт.

2. Цілі, призначення та види планів

Основна ціль планування - інтеграція всіх учасників проекту для виконання комплексу робіт, що забезпечують досягнення кінцевих результатів проекту.

Планування являє собою сукупність дій, що передбачають визначення цілей та параметрів взаємодії між роботами та організаціями-учасниками, розподіл ресурсів і вибір інших організаційних, технологічних та економічних рішень, що

забезпечують досягнення поставлених в проекті цілей. Традиційно склалася наступна система планів.

На передінвестиційній стадії у складі так званого обгрунтування інвестицій та ТЕО - укрупнений (попередній) план реалізації проекту, включаючи потреби в основних видах ресурсів.

Види планів

У методології управління проектами сформована наступна система планів.

Проект має чотири фундаментальних рівні управління:

1. концептуальний;
2. стратегічний;
3. тактичний, який в свою чергу включає:
4. поточний;
5. оперативний.

Для кожного рівня повинен бути розроблений відповідний план.

На концептуальному рівні

На концептуальному рівні визначаються цілі, задачі проекту, розглядаються альтернативні варіанти дій по досягненню намічених результатів з оцінкою негативних і позитивних аспектів кожного варіанту, встановлюються концептуальні напрямки реалізації проекту, включаючи опис предметної області, укрупненої структури робіт, логіки їх розвитку, основні віхи, попередню оцінку тривалості, вартості та потреби у ресурсах.

Стратегічний план визначає:

1. цільові етапи та основні віхи, які характеризуються строками введення об'єктів, виробничих потужностей, об'ємами випуску продукції;
2. етапи проекту, які характеризуються термінами завершення комплексів робіт (нульовий цикл, монтаж каркасу та ін.), термінами поставки продукції (обладнання), термінами підготовки фронту робіт;
3. кооперацію організацій виконавців;
4. потреби у матеріальних, технічних і фінансових ресурсах з розподілом по рокам, кварталам.

Основне призначення плану на цьому рівні показати як проміжні етапи реалізації логічно вишиковуються у напрямку до кінцевих цілей проекту. Стратегічний план встановлює стабільне зовнішнє та внутрішнє оточення, фіксовані цілі для проектної команди та забезпечує загальне бачення проекту.

Проект-менеджер ув'язує окремі віхи у єдиній стратегії з інвестором і знайомить з цим планом проектну команду. Також на цьому рівні фокусується увага на проміжних етапах, які допомагають розподілити роботу по підрозділам команди. Підрозділи команди отримують завдання по виконанню проміжного етапу і планують свою власну роботу незалежно від інших членів проектної

команди. Вони знають, що повинні виконати свій етап до певної дати для того, щоб забезпечити подальше виконання проекту.

На тактичному рівні:

1. поточний план - уточнює строки виконання комплексів робіт, потреби у ресурсах, встановлює чіткі межі між ділянками робіт, за виконання яких відповідають різні організації-виконавці, у розрізі року і кварталу;
2. оперативний план - деталізує завдання учасникам на місяць, тиждень, добу по комплексам робіт.

Плани можуть деталізуватися по функціям управління.

Функціональний план розроблюється на кожний комплекс робіт (підготовчі роботи, проектно-дослідницькі роботи, поставка матеріалів та обладнання, будівництво, пусковий період і освоєння виробничих потужностей) або на комплекс робіт, які виконуються однією організацією.

Також слід розрізняти плани за ступенем охоплення робіт проекту:

1. зведений,комплексний, головний (на всі роботи проекту);
2. детальний (частковий) по організаціям учасникам;
3. детальний (частковий) по видам робіт.

Типи календарних планів вибираються в залежності від цілей планування, особливостей проекту і організації управління.

3. Розвиток методів планування

Спочатку використовувалися досить прості засоби і методи, зокрема лінійні діаграми. Іншими додатковими до лінійної діаграми засобами, що використовувалися для УП є: таблиці трудових витрат, таблиці витрат ресурсів, в яких вказані терміни підготовки робочої документації, початок експлуатації, терміни по інших етапах і ресурси, необхідні для їх здійснення; таблиці обладнання, що містять дані про типи обладнання, терміни постачання; фінансові таблиці, що відображають прибутки і витрати.

Потім почали використовувати методи дослідження операцій, сітьові методи (метод критичного шляху).

Проект «Поляріс» - використовувався метод аналізу і оцінки програм - PERT. Він має перевагу тоді, коли досягнення цілей проекту пов'язане з фактором невизначеності. Для кожної операції визначається три оцінки (оптимальна, песимістична, найбільш вірогідна). Дозволяє керівництву проекту точно знати, що необхідно в даний момент зробити, і хто це повинен робити, а також імовірність виконання деяких операцій.

Сіть передування - дає найпростіше уявлення про операції, що паралельно виконуються. Метод критичного шляху і сіті передування краще використовувати тоді, коли операції проекту мають певну тривалість, а якщо оцінки тривалості носять ймовірносний характер, то більш ефективний метод PERT.

Загальна умова - для реалізації події повинні бути завершені усі попередні операції.

Метод аналізу і графічної оцінки – GERT - доцільно застосовувати у випадку, коли для завершення планування не обов'язкове виконання всіх операцій. Він дає оцінки імовірності реалізації подій, на основі даних, визначених за допомогою моделювання ситуацій.

4. Сітьові моделі

Сітьовою моделлю комплексу робіт називається орієнтований граф, який використовується для опису залежностей між роботами і етапами проекту. Сіткові моделі доцільно використовувати тільки для складних проектів.

Види сіток.

Існує три типи сіток:

1. сітки типу "вершини-роботи";
2. сітки "вершини-події";
3. змішані сітки.

Сітки типу "вершини-роботи".

У сітках типу "вершини-роботи" елементи роботи подані у вигляді прямокутників, зв'язаних логічними залежностями, які ідуть один за одним.

Існують чотири типи логічних взаємозалежностей між роботами (рис. 3):

1. закінчення-початок: B не може початися, поки не закінчиться A;
2. закінчення-закінчення: D не може закінчитися поки не закінчиться C;
3. початок-початок: D не може початися поки не почнеться C;
4. початок-закінчення: F не може закінчитися поки не почнеться E.

Сітки типу "вершини-події".

Сітки такого виду часто називаються IJ сітками, така як кожна робота визначається номером IJ (початок/кінець). У сітках цього типу робота зображується стрілкою між двома вузлами і визначається номерами вузлів, які вона зв'язує. Рис. 4 - це рис. 2, виконаний у вигляді IJ сітки. Робота A стала роботою 1-2. Так як роботи повинні бути унікальними, то дві роботи B і C не можуть зв'язувати той самий вузол. Таким чином, B і C закінчуються у вузлах 3 і 4 відповідно і ці вузли пов'язуються фіктивною роботою. Так як роботи пов'язані через вузли, використовується логічна залежність виду закінчення-початок. Можливе введення фіктивних робіт для зображення трьох інших логічних зв'язків.

Змішані сітки.

Робота зображується у вигляді прямокутника (вузла) або лінії (стрілки). Крім того, існують прямокутники і лінії, які можуть не зображувати роботу: одночасні події та логічні залежності. Лінії використовуються не для об'єднання

прямокутників по початкам і закінченням, а для відображення моменту часу до, під час або після виконання роботи. У останніх модифікаціях змішаних сіток зникає різниця між вузлами та лініями. Математичний апарат змішаних сіток - це принципово нова область.

Методи побудови сіткових моделей

Всі види сіткових моделей забезпечують розрахунок раннього та пізнього початку і закінчення, резервів часу для кожної роботи проекту, у припущенні, що задані тривалості робіт і логічні залежності між ними. Основа цього є настільки потужною, що дозволяє досліджувати різні варіанти і за формулою "ЩО-ЯКЩО", яка передбачує варіювання тривалостями і логічними залежностями між роботами.

Зображення сіток.

У сітках типу "вершини-роботи" кожна робота зображується прямокутником, поділеним на 7 частин. У верхніх сегментах цього прямокутника подані дані про ранній початок, тривалість роботи і її раннє закінчення. У нижніх - пізній початок, резерв часу і пізнє закінчення. Середня частина містить опис роботи.

У сітках типу "вершини-події" вузол має 4 сегменти: ідентифікатор, значення ранніх і пізніх моментів часу та резерв часу. Час - це початок наступної роботи і закінчення попередньої.

Розрахунок сіткової моделі.

Ранні початок і закінчення розраховуються на етапі прямого проходу по сітці. Ранній початок першої роботи дорівнює 0, раннє закінчення розраховується додаванням значення тривалості роботи. Раннє закінчення перетворюється у наступній роботі у ранній початок додаванням випередження або відніманням запізнення, які передбачують залежність закінчення-початок. Для залежності "початок" час початку перетворюється у закінчення. Якщо робота має дві чи більше попередніх робіт, то перетворюється робота із максимальним значенням раннього закінчення. Процес повторюється по всій сітці.

Дати пізнього початку, закінчення, резерв часу підраховуються при виконанні зворотного проходу. Ранній початок останньої роботи приймається рівним її пізньому закінченню. Шляхом віднімання тривалості роботи підраховується пізній початок. пізній початок перетворюється у пізнє закінчення попередньої роботи. Перетворена дата початку або закінчення приймається у якості нового часу початку або закінчення у відповідності з типом залежності. Коли робота має дві чи більше попередніх роботи, вибирається робота із найменшим значенням часу початку (після додавання запізнення і віднімання випередження). Процес повторюється по всій сітці. Резерв часу у першої і останньої роботи повинен дорівнювати 0. На рис. (Слайд) показана сітка після зворотного проходу.

Визначення критичного шляху.

Критичний шлях - це послідовність робіт з нульовими резервами часу.

Управління плануванням

У наш час застосовуються методи:

1. *Директивний* передбачає інформування виконавців робіт тільки про короткострокові плани. Персональна відповідальність за розробку планів, але коректування і уточнення несе розробник планів, тобто учасники проекту практично не задіяні в процесі планування.
2. Формування планів тільки на випадок кризового становища. Такі плани - комплекси заходів щодо ліквідації такого становища.
3. Короткострокове планування методом «Плани комітетів» - плани складаються виконавцями робіт в ході періодичних нарад по аналізу ходу проекту.
4. Метод семінарів - менеджер проекту призначає представників організації - учасників проекту відповідальними за успішне виконання проекту.

Діяльність групи по плануванню організується у вигляді семінарів, які дозволяють швидко встановити взаємодію між учасниками проекту і створити реалістичні плани, що враховують вимоги і зацікавленість всіх учасників.

Семінари звичайно проводяться в три етапи:

1. Взаємодія генерального підрядчика і робочої групи планування.
2. Взаємодія членів робочої групи по плануванню.
3. Взаємодію замовників і членів робочої групи.

Обов'язковою умовою цього методу є надання всім членам робочої групи необхідної інформації по проекту.

Перший етап

семінару має на меті домовлятися про умови передачі управління проектом від генерального підрядника (замовника) до менеджера проекту і робочої групи по плануванню. Генеральний підрядник повинен поінформувати групу про цілі проекту, умови контракту, корпоративні цілі і задачі.

Другий етап є основним.

Члени робочої групи безпосередньо займаються процесом планування; менеджер проекту здійснює загальну організацію процесу планування і контроль розроблених планів на їх відповідність концепції проекту.

Третій етап.

Представник замовника повинен поінформувати членів робочої групи відносно цілей проекту, корпоративних цілей і задач замовника.

Важливість термінів і якості проекту. Всі ці аспекти повинні бути враховані в проектах, що розробляються. Замовник аналізує ці документи, а у разі

схвалення погодить їх. Остаточний варіант всіх планів необхідно направляти замовнику офіційно, а після твердження всім учасникам проекту.

Процес планування

Основні етапи процесу планування включають:

1. цілі, задачі та основні техніко-економічні показники проекту, тривалість та ресурси, специфікацію виконуваних робіт, етапів та віх проекту;
2. структуризацію проекту;
3. організаційно-технологічні рішення;
4. сіткові моделі пакетів робіт;
5. оцінку можливості реалізації, оптимізацію по строкам та критеріям якості використання ресурсів та іншим критеріям;
6. потреби у ресурсах;
7. документи по пакету планів;
8. затвердження планів та бюджету;
9. доведення планових завдань до виконавців;
10. підготовку та затвердження звітної документації для контролю планів.

Номенклатура і глибина розробки окремих етапів може змінюватись в залежності від масштабу, вартості і типу проекту.

План реалізації проекту

Глибина деталізування

планування визначається розмірами і складністю проекту, характером проекту, типом об'єктів, що створюються. Типова структура плана проекту по створенню виробничого об'єкта включає:

1. Основні цілі проекту.
2. Фінансовий план проекту.
3. План виконання субконтрактів.
4. Функціональний план.
5. Аналіз чинників виконання проекту.
6. Додатки до плана проекту.
7. При формулюванні цілей потрібно уникати спрощених і практично нездійсненних цілей. Цілі потрібно викладати якомога більш чітко і конкретно, враховуючи відносне значення кожної мети і її вплив на прийняття альтернативних управлінських рішень.
8. Визначається мінімальні рівні інвестицій кожного інвестора, методи і умови фінансування. План фінансування, як правило, впливає на графік капітальних витрат, погашення заборгованості і при угоді між учасниками проекту і інвесторами. Фінансовому плануванню передує кошторисна робота. Кошториси складаються на кожний вид, етап робіт. Крім того

фінансове планування передбачає оцінку фінансово-економічних показників проекту (дисконтований прибуток, термін окупності).

9. Представляє спільну стратегію замовника, генерального підрядника, субпідрядників і постачальників. Є зв'язуючим для всіх учасників проекту. Основою для його складання служить чітке визначення і розподіл задач і обов'язків всіх учасників проекту. При складанні цього плана:
    а) оцінюються альтернативи укладання субконтрактів; критеріями такої оцінки є можливість виконання технічних завдань субпідрядниками в необхідні терміни;
    б) вибір найбільш відповідного типу контрактів для кожного субпідрядника і визначаються терміни, відповідальні за підготовку і укладання цих контрактів;
    в) розробляється концептуальний календарний план проекту як складова частина контрактної документації.
10. Визначає структуру функціональних комплексів робіт, терміни і особливості їх виконання. До функціональних комплексів можуть бути віднесені: проектні роботи, матеріально-технічне постачання (МТП), будівництво, контроль якості, здавання в експлуатацію. Складається з наступних планових документів:
11. головного календарного плана проекту;
12. календарних планів робіт (короткострокових);
13. функціональних планів робіт.

Календарні плани

Головний календарний план (ГКП) формується шляхом уточнення і деталізування концептуального плана проекту. Він визначає етапи проекту, які поділяються на цільові етапи і етапи проекту, пов'язані з початком і завершенням функціональних комплекс робіт. Головний календарний план, як правило, представляється в формі гістограми, оскільки на цій фазі проекту декомпозиція на окремі роботи відсутня, а конкретні обмеження не визначені.

Короткострокові календарні плани формуються на основі ГКП і містять переліки робіт, терміни, прізвища осіб, відповідальних за їх здійснення.

Функціональні плани робіт (ФПР)

Це - системи планових документів, що містять заходи, а також конкретні технічні і проектні рішення з окремих особливостей виконання проекту. До таких аспектів відносяться:

1. ***За планом практичних робіт:***
    a) початкові дані для проектування;
    b) пріоритети виконання робіт;
    c) потреба в персоналі;
    d) процедури затвердження проектних розробок;

2. ***за планом матеріально-технічного постачання:***
    a) терміни постачання обладнання;
    b) терміни і контроль за установкою обладнання;

3. ***за планом будівництва:***
    a) вимоги до приміщень для розміщення обладнання і персоналу;
    b) організація робіт з підготовки і будівництва приміщень;

4. ***за планом контролю якості:***
    a) загальні критерії якості;
    b) контроль обладнання, що поставляється;

5. ***за планом введення в експлуатацію:***
    a) підготовчі роботи;
    b) доексплуатаційний контроль;
    c) введення в експлуатацію.

Чинник виконання проекту

Це - ключові параметри або процеси проекту, які можуть вплинути на досягнення його цілей.

Мета цього розділу - розробка системи заходів, що компенсують недоліки планових рішень попередніх етапів планування і спрямованих на запобігання впливу протидіючих чинників, на отримання вигоди від впливаючих чинників, на збільшення суми економічного ефекту від взаємодії всіх чинників.

Документальною формою цього розділу плана є угода про заходи, яка затверджується зацікавленими учасниками і узгоджується з відповідальними виконавцями.

Дані, що використовуються при складанні плана:

1. дані з концепції проекту;
2. матеріали обстеження;
3. вимоги і обмежень до проекту і інш.
4. Календарні плани

Центральне місце у плануванні проекту займають задачі календарного планування - складання і коригування розкладу, в якому роботи, виконані різними організаціями, ув'язуються в часі між собою і з можливостями їх забезпечення різними видами матеріально-технічних ресурсів. При ув'язці повинно бути забезпечено дотримання заданих обмежень (строки пакетів робіт, макети ресурсів, фіксування та ін.), оптимальне (по прийнятому критерію) розподілення ресурсів.

У найпростішому випадку параметри календарного плану складають дати початку і закінчення кожної роботи, їх тривалості та необхідні ресурси. При аналізі календарних планів визначають також резерв часу (величина можливого відхилення тривалості для кожної роботи, яка не вплине на завершення проекту вчасно). У більшості складних календарних планів існує до 6 варіантів моментів

початку, закінчення, тривалості робіт і резервів часу. Це - ранні, пізні, базові, планові і фактичні дати, реальний і вільний резерв часу.

Методи розрахунку сіткових моделей дозволяють обчислювати тільки ранні і пізні дати. Базові і поточні планові дати необхідно вибирати із врахуванням інших факторів. Існують три варіанти вибору:

* календарний план по раннім початкам (жорстко зліва): використовується для стимулювання виконавці проекту;

* календарний план по пізнім закінченням (жорстко справа): використовується для подання виконання проекту у кращому вигляді для споживача;

* календарний план між ними: робиться або для згладжування використовуваних ресурсів або для показу замовнику найбільш вірогідного закінчення.

Тривалість - це час виконання роботи.

Ранні і пізні дати

Ці дати можуть бути визначені на основі оціночних тривалостей всіх робіт. Початок і закінчення однієї роботи можуть залежати від закінчення іншої. Таким чином існує сама рання дата, коли робота може бути розпочата - дата раннього початку. Дата раннього початку плюс оціночна тривалість роботи складають дату раннього закінчення. Якщо дата пізнього початку відрізняється від дати раннього початку, то проміжок, під час якого робота може бути розпочата, називається резервом часу.

Роботи з нульовим резервом часу називаються *критичними*; їх тривалість визначає тривалість проекту в цілому.

Критична тривалість - мінімальна тривалість,на протязі якої може бути виконаний весь комплекс робіт проекту.

Критичний шлях - шлях у сітковій моделі, тривалість якого дорівнює критичній.

Роботи, які лежать на критичному шляху, називаються критичними роботами.

Метод критичного шляху

Цей метод є основним математичним засобом для обчислення ранніх і пізніх початків і закінчень робіт і резервів часу.

Календарний план більш наочно можна представити у вигляді лінійних діаграм (які іноді називають діаграмами Гантта.

Тривалість роботи - це головний параметр планування не лише у відношенні початку і закінчення даної роботи, але і в обчисленні для нього раннього початку із врахуванням узагальненої тривалості передуючих робіт і пізнього закінчення, яке враховує узагальнену тривалість наступних робіт.

Тривалість залежить від:

* сумарної трудомісткості, яка затрачується на виконання елементів роботи, і числа робітників, які можуть її виконати;

* часу чекання поставки деяких виробів, який не залежить від кількості робітників, які виконують роботу.

Тривалість деяких робіт залежить від часу очікування фронту робіт або поставки матеріалів, або додаткової інформації, або від очікування здійснення яких-небудь змін.

Вони можуть включати:

* поставку матеріалів та обладнання;
* підготовку фронту робіт;
* підготовку звітів;
* переговори з клієнтами або підрядчиками;
* отримання планових дозволів і фінансових затверджень.

Ресурсні гістограми і згладжування ресурсів

При призначенні базових або поточних планових дат необхідно враховувати ресурсні обмеження. Якщо потреби в ресурсах для всіх робіт проекту відомі і встановлені дати початку і закінчення, то можна обчислити функцію зміни потреб для кожного ресурсу проекту, яка представляє таблицю рівнів ресурсів або ресурсну гістограму.

Існує три основних види залежності потреби у ресурсах від ходу роботи (тривалості):

* постійний - протягом всієї роботи завантаження (фронт робіт) ресурсу не змінюється;

* східчастий - протягом роботи завантаження ресурсу змінюється стрибкоподібно (сходинками);

* трикутний - завантаження ресурсу лінійно наростає від початку роботи до максимального значення, а потім спадає до закінчення роботи.

Аналіз можливості реалізації проекту

Календарний план, отриманий у результаті розрахунку сіткової моделі, перевіряється, уточнюється, при необхідності деталізується, і коли є повна впевненість, що у план включені всі роботи, є повна інформація щодо наявних і потрібних ресурсів, переходять до аналізу можливості реалізації.

Управління ризиком

- Сутність управління ризиком та його функції
- Класифікація ризиків за різними ознаками

Сутність управління ризиком та його функції

Формальні методи визначення, аналізу, оцінки, попередження виникнення, прийняття заходів щодо зниження ступеню ризику

Ризик - вплив на проект і його елементи непередбачених подій, які можуть нанести певні збитки і перешкоджати досягненню цілей проекту.

Класифікація ризиків за різними ознаками

Ризик характеризується трьома чинниками:

- подіями, що негативно впливають на проект;
- імовірністю появи таких подій;
- оцінка збитків, нанесених такими подіями.

Підфункції управління ризиком:

- виявлення ризику;
- зниження міри ризику.

Виявлення ризику:

- виявлення зовнішніх непередбачуваних подій;
- виявлення зовнішніх передбачуваних, але невизначених подій;
- виявлення внутрішніх нетехнічних подій;
- виявлення внутрішніх технічних чинників;
- юридичні і правові чинники.

Зниження міри ризику:

- страхування ризику;
- аналіз впливу;
- планування реагування на ризикові події;
- система реагування на ризикові події;
- використання інформації по ризиках.

Ризику схильні передусім фінансовий, технічний, організаційний і соціально-політичний аспекти проекту. Управління ризиком застосовується, якщо міра ризику є досить високою.

Лекція 25. Контроль та моніторинг стану проекту

- Контроль та моніторинг стану проекту.
- Метрики контролю.
- Організація роботи проектної команди.
- Ролі та зони відповідальності учасників команди. [4, с. 562-597]

Контроль в управлінні проектами

- Сутність та види контролю в управлінні проектами.
- Загальні принципи побудови системи контролю.
- Моніторинг робіт по проекту.
- Вимірювання прогресу і аналіз результатів.
- Прийняття рішень

Сутність та види контролю в управлінні проектами

Сутність

Контроль - це процес в якому керівник проекту встановлює чи досягаються поставлені цілі, виявляє причини, які дестабілізують хід роботи і обгрунтовує прийняття управлінських рішень, що коригують виконання робіт по проекту, перш ніж будуть завдані збитки проекту.

Мета контролю – забезпечення виконання планових показників і підвищення загальної ефективності функцій планування та контролю проекту.

Основні задачі контролю:

- отримання фактичних даних про хід виконання робіт по проекту,
- їх зіставлення з плановими;
- виявлення відхилень.

Для розв’язання цих задач контроль повинен забезпечити:

- Моніторинг - систематичне і планове спостереження за всіма операціями.
- Виявлення відхилень від цілей реалізації проекту за допомогою ряду критеріїв і обмежень, які фіксуються в календарних планах, бюджетах і інш. документації.
- Прогнозування наслідків ситуації, що склалась.
- Обгрунтування необхідності прийняття коригуючих впливів.

Предмет контролю - факти і події, перевірка виконання конкретних рішень, з'ясування причин відхилення, оцінка ситуації, прогнозування наслідків.

За часовою ознакою та метою проведення

За часовою ознакою та метою проведення розрізняють три види контролю:

1 - попередній,

2 - поточний,
3 – заключний.

- *Попередній* контроль здійснюється до фактичного початку виконання робіт і направлений на дотримання певних правил і процедур, здебільшого він стосується ресурсного забезпечення робіт.
- *Поточний* контроль здійснюється при реалізації проекту, він включає контроль часових характеристик, контроль досягнення проміжних цілей проекту, контроль виконання заданих обсягів робіт, контроль бюджету, контроль ресурсів, контроль якості. Основна мета - оперативне регулювання ходу реалізації проекту. Такий підхід базується на порівнянні досягнутих результатів з встановленими в плані проекту вартісними, часовими, ресурсними характеристиками.
- *Заключний* контроль проводиться на стадії завершення проекту з метою інтегральної оцінки реалізації проекту. Основне призначення - узагальнення отриманого досвіду для подальшої розробки і реалізації проектів-аналогів і з метою вдосконалення процедур управління.

Іноді контроль ототожнюють з оцінкою.

Дійсно, між контролем і оцінкою існує тісний зв’язок. Оцінка так само, як і контроль, є важливою функцією зворотного зв'язку однак вона базується на попередньому підведенні проміжних підсумків і оцінюванні загальної картини і зазвичай проводиться особою або групою осіб, які не працюють безпосередньо над проектом.

Контроль передбачає постійне стеження за просуванням проекту, він сфокусований на деталях проекту. Первинний план може виявитися неспроможним через різні чинники, наприклад через зміну термінів початку проекту, перегляд умов фінансування, зміни потреб, неточного планування залежностей між роботами, часових оцінок і ресурсних обмежень для робіт, затримки в передачі робочої документації або відсутності необхідного обладнання у підрядчиків, непередбачених технічних ускладнень або зміни зовнішніх умов. За контрольні дії несе відповідальність керівник проекту.

Основні методи аналізу стану робіт, що використовуються менеджером, передбачають збір фактичних даних про досягнуті результати і оцінку фактичних витрат, оцінку обсягу робіт, що залишився, аналіз фактичного вироблення на поточну дату.

Керівництво повинно встановити послідовність збору даних через певні інтервали часу, аналізувати отримані дані, аналізувати поточні розходження фактичних і планових показників і прогнозувати вплив поточного стану справ на витрати по обсягу робіт, що залишився. Іншими словами, керівництво повинно організувати процеси контролю проекту.

Процеси контролю проекту

Процеси контролю проекту поділяються на основні і допоміжні :

Основні:

• *загальний контроль змін* - координація змін по проекту загалом;

• *ведення звітності по проекту* - збір і передача звітної інформації про хід реалізації проекту, включаючи звіти про виконані роботи, про виконання планових показників, прогноз з урахуванням результатів.

Допоміжні:

• *контроль змін змісту* - контроль за змінами змісту проекту;

• *контроль розкладу* - контроль за змінами в розкладі проекту;

• *контроль витрат* - контроль витрат по роботах і змін бюджету проекту;

• *контроль якості* - відстеження конкретних результатів проекту для визначення їх відповідності встановленим стандартам і прийняття необхідних заходів по усуненню причин, що приводять до порушення якості;

• *контроль ризику* - реагування на зміну рівня ризику в ході реалізації проекту.

Процеси контролю проекту тісно взаємопов'язані

Процеси контролю проекту тісно взаємопов'язані і можуть бути представлені за необхідності як один інтегрований процес, що складається з вибраних процесів. Наприклад, спільна реалізація процесів ведення звітності, контролю змін змісту, контролю розкладу і контролю витрат може бути представлена у вигляді трьохетапного процесу відстеження фактичного стану робіт, аналізу результатів і вимірювання прогресу, і проведення коректуючих дій для досягнення цілей проекту:

• *відстеження:* збір і документування фактичних даних; визначення в офіційних і неофіційних звітах міри відповідності фактичного виконання запланованим показникам;

• *аналіз:* оцінка поточного стану робіт і порівняння досягнутих результатів із запланованими; визначення причини і шляхів впливу на відхилення від виконання плану;

• *коригування:* планування і здійснення дій, спрямованих на виконання робіт відповідно до плану, мінімізацію несприятливих відхилень або отримання переваг від виникнення сприятливих відхилень.

Загальні принципи побудови системи контролю

Система контролю проекту – частина загальної системи УП між елементами якої є зворотні зв’язки і можливість коригування прийнятих раніше показників.

Вимоги до системи контролю виробляються до початку реалізації проекту за участю всіх зацікавлених сторін і визначають склад інформації, що аналізується, структуру звітів і відповідальність за збирання даних, аналіз інформації і прийняття рішень. Система управління проектом повинна забезпечувати коригуючі впливи там і тоді, де і коли вони необхідні. Наприклад, якщо відбувається затримка закінчення окремих робіт, то, наприклад, прискорити

їх виконання можна за рахунок перерозподілу трудових ресурсів і обладнання. Якщо ж затримується постачання проектної документації, збільшуються витрати на матеріали і обладнання, субпідрядники зривають директивні терміни, то необхідно переглянути план проекту. Корекція плану може бути обмежена переглядом параметрів робіт, а може вимагати розробки абсолютно нової сітьової моделі, починаючи з поточного стану і до момента закінчення проекту.

Принципи побудови ефективної системи контролю застосовуються для ефективного управління в рамках оперативного циклу проекту, який вимагає проектування, розробки і впровадження добре організованої системи контролю, необхідного для досягнення безпосереднього зворотного зв'язку. Існує декілька основних принципів побудови ефективної системи контролю:

1 - *Наявність конкретних планів.* Плани повинні бути змістовні, чітко структуровані і фіксовані, з тим щоб забезпечувати основу для контролю. Якщо плани оновлюються дуже часто і без застосування процедур контролю за змінами, контроль над проектом може бути втрачений.

2. - *Наявність інформативної системи звітності.* Звіти повинні відображати стан проекту відносно початкових планів на основі єдиних підходів і критеріїв. Для забезпечення цього повинні бути чітко визначені і досить прості процедури підготовки і отримання звітів, а також встановлені для всіх видів звітів чіткі часові інтервали. Результати, представлені в звітах, повинні обговорюватися на нарадах.

3. - *Наявність ефективної системи аналізу фактичних показників і тенденцій.* Визначення базової траєкторії, нормативів, стандартів для порівняння з ними поточних значень елементів, що контролюються. В наслідок аналізу зібраних даних керівництво проекту повинно визначити, чи відповідає поточна ситуація запланованій, а якщо ні, то розрахувати розмір і серйозність наслідків відхилень. Двома основними показниками для аналізу є час і вартість. Для аналізу тенденцій у вартісних і часових оцінках робіт проекту необхідно використати спеціальні звіти. Прогноз, наприклад, може показати збільшення вартості проекту або затримки по термінах. Однак часто відхилення у часових і вартісних показниках впливають також на зміст майбутніх робіт і якість результатів.

4 - *Регулярне спостереження* за ходом робіт і зіставлення поточного стану проекту з базовою траєкторією і стандартами з метою раннього виявлення проблем, що виникають і вживання чітких дійових заходів для їх розв'язання.

5 - *Наявність ефективної системи реагування.* Завершальним кроком процесу контролю є дії керівництва і направлені на подолання відхилень у ході робіт проекту. Ці дії можуть бути направлені на усунення виявлених недоліків і подолання негативних тенденцій в рамках проекту. Однак в ряді випадків може бути потрібен перегляд плану. Перепланування вимагає проведення аналізу "що, коли...", який забезпечує прогноз і розрахунок наслідків від дій, що плануються. Від менеджера залежить також переконання і мотивація команди проекту в необхідності тих або інших дій.

У рамках функції контролю і оперативного управління реалізацією проекту вирішуються задачі вимірювання, прогнозування і оцінки оперативної ситуації,

що складається по досягненню результатів, витратах часу, ресурсів і фінансів, аналізу і усуненню причин відхилення від затвердженого плану, а також корекція плану. Звичайно при управлінні проектом контролюються три основні кількісні характеристики час, обсяг робіт і вартість. Крім того, керівництво відповідає за управління змістом робіт (змінами), якістю і організаційною структурою.

Важливим для аналізу ходу робіт параметром є поточна (порогова) дата, яка є моментом часу, відносно якого проводиться аналіз. Стан робіт по проекту оцінюється відносно порогової дати.

Таким чином для створення ефективної системи контролю необхідно:

- ретельно спланувати всі роботи, виконання яких необхідне для завершення проекту;
- точно оцінити час, ресурси і витрати;
- визначити склад і рівень деталізування робіт, що підлягають контролю.
- визначити склад показників, що контролюються.
- встановити форми надання і терміни представлення первинної інформації і аналітичних звітів.
- визначити склад, методи і технології надання аналітичних і графічних звітів.
- призначити відповідальних за повноту, достовірність і своєчасність надання фактичних даних.
- визначити комплекс програмно-інформаційних засобів, що будуть використовуватися (якщо будуть використовуватися).

Моніторинг робіт по проекту

Моніторинг - контроль, стеження, облік, аналіз і складання звітів про фактичне виконання проекту порівняно з планом.

Перший крок в процесі контролю - збір і обробка даних про фактичний стан робіт. Керівництво зобов'язане безперервно стежити за ходом виконання проекту, визначати міру завершеності робіт і, виходячи з поточного стану, оцінювати параметри виконання майбутніх робіт. Для цього необхідно мати ефективні зворотні зв'язки, що дають інформацію про досягнуті результати і витрати.

Інформація що відображає стан і хід робіт по проекту поступає від членів проектної команди, організацій-виконавців, незалежних контролерів або з планових і звітних документів.

Формальні джерела інформації необхідні для контролю включають: форми обліку часу експлуатації машин і обладнання, звіти про виконання заданих обсягів робіт, табелі використання робочої сили, наряди, різні види повідомлень про хід робіт і інші документи.

Можна скласти спеціальні звіти за різними формами:

а) безпосередньо при особистих контактах

б) у табличній формі

у) в графічній формі надання

Незалежно від форми надання, звітна інформація повинна включати такі основні елементи:

кошторисну вартість
фактичні результати
результати, що прогнозуються
відхилення та причини, які пояснюють ці відхилення

Інформація про фактичний хід робіт і прогнозована інформація, представляється в проектну команду відповідальним виконавцем. Ця інформація використовується для розрахунку, перерахунку і аналізу сітьової моделі, а також коригуванню нормативно-довідкової бази. Перерахунок моделі включає в себе фіксацію на сітьовому графіку стану поточних і закінчених робіт, подій, що здійснилися, внесення нових робіт і подій, виключення анульованих, уточнення формулювань, опис робіт. Всілякі зміни, що стосуються граничних робіт і подій, погоджують з суміжними відповідальними виконавцями.

При розробці системи збору інформації менеджер проекту повинен насамперед визначити склад даних, що збираються і періодичність збору. Рішення з цих питань залежать від задач аналізу параметрів проекту, періодичності проведення нарад і видачі завдань. Деталізація аналізу в кожному конкретному випадку визначається виходячи з цілей і критеріїв контролю проекту. Наприклад, якщо основним пріоритетом є своєчасність виконання робіт, то методи контролю використання ресурсів і витрат можуть бути задіяні обмежено.

Обов'язкові вимоги до системи збору інформації для системи контролю:

- точність
- своєчасність
- повнота інформації
- забезпечення єдності інформації для всіх учасників проекту

Розрізнюють три ступеня деталізування інформації для досягнення мети контролю:

- керівники підрозділів і відповідальні виконавці отримують найбільш деталізовану інформацію, що дозволяє оцінити стан кожної із закріплених за ним робіт і її положення в комплексній сітьовій моделі.
- керівник організації і виконавці при підкомплектованих роботах повинні мати інформацію, що дозволяє дати загальну оцінку стану ходу робіт, закріплених за даною організацією і що містять найдокладніші відомості по граничних подіях, якими визначаються зв'язки з іншими організаціями або підкомплексами а також зведення про роботи даної організації або підкомлексу, що попали в критичну зону.
- керівник проекту отримує деталізовану інформацію по роботах критичної зони, які дозволяють йому укрупнено оцінити загальний стан робіт по проекту загалом, його окремих найважливіших елементів і етапів а також проконтролювати планові терміни настання граничних подій,

що визначають зв'язки між організаціями-виконавцями і структурними підрозділами всередині головної організації.

Методи контролю фактичного виконання проекту поділяються на:

- *метод простого контролю (контроль в момент закінчення робіт),* який також називають методом "0 - 100", оскільки він відстежує тільки моменти завершення детальних робіт (існують тільки дві міри завершеності роботи: 0% і 100%). Іншими словами, вважається, що робота виконана тільки тоді, коли досягнуто її кінцевого результату;
- *метод детального контролю,* який передбачає виконання оцінок проміжних станів виконання роботи (наприклад, завершеність детальної роботи на 50% означає, що, за оцінками виконавців і керівництва, цілі роботи досягнуті наполовину). Цей метод складніший, оскільки вимагає від менеджера оцінювати процент завершеності для робіт, що знаходяться в процесі виконання. Для цього необхідно розробити спеціальні шкали для оцінювання міри виконання робіт.

Іноді зустрічаються дещо модифіковані варіанти методу детального контролю:

- *метод 50/50,* в якому є можливість обліку деякого проміжного результату для незавершених робіт. Міра завершеності роботи визначається в момент, коли на роботу витрачено 50% бюджету;
- *метод по віхах (контроль у заздалегідь встановлених точках проекту),* який застосовується для тривалих робіт. Робота ділиться на частини віхами, кожна з яких означає певну міру завершеності роботи.
- *регулярний оперативний контроль*, який здійснюється на рівні відповідальних виконавців.
- *експертна оцінка* ступеню виконання робіт і готовності проекту - виконується зазвичай для надання інформації на вимогу когось з учасників проекту, наприклад, інвестора.

Використовуючи один з цих методів, менеджер може розробити інтегровану систему контролю, яка дозволяє зосередити увагу на мірі завершеності робіт, а не тільки на часових і об'ємних параметрах проекту і задовольняє критеріям обгрунтування фінансування.

Критерії і дані, необхідні для контролю основних параметрів проекту, наведено в табл. 1.

Таблиця 1. Критерії для контролю і необхідні дані

Критерій контролю	Кількісні дані	Якісні дані
Час і вартість	Планова дата початку/закінчення Фактична дата початку/закінчення Обсяг виконаних робіт	

	Обсяг майбутніх робіт Інші фактичні витрати Інші майбутні витрати	
Якість		Проблеми якості
Організація		Зовнішні затримки Проблеми внутрішньої координації ресурсів
Зміст робіт		Зміни в обсязі робіт Технічні проблеми

Звичайно кількісні показники збираються на рівні робіт або пакетів робіт і потім узагальнюються для верхніх рівнів контролю відповідно до структури поділу робіт (ієрархії робіт). Оскільки оцінки виконання проекту загалом і окремих його етапів розраховуються на основі даних про виконання детальних робіт, важливо на етапі розробки системи контролю вибрати відповідні вагові коефіцієнти для визначення узагальнених показників.

Наприклад, використання тривалості робіт в якості вагових коефіцієнтів призводить до того, що основний внесок в процент виконання сумарної (укрупненої, складової) роботи будуть вносити найтриваліші елементарні (дочірні) роботи.

Вага роботи може встановлюватися й відповідно до її планової вартості. Як правило, планова вартість є досить надійним показником значущості роботи. Іноді витрати і обсяги робіт не пов'язані безпосередньо, наприклад у разі використання в процесі реалізації робіт дорогих матеріалів і обладнання. Можливо, більш вдалим у цьому випадку буде визначати питомі ваги робіт на основі витрат, пов'язаних тільки з використанням ресурсів або планового обсягу.

Вимірювання прогресу й аналіз робіт

Прогрес в реалізації проекту

У західній науці управління проектами в системах контролю ходу реалізації проекту для позначення критеріїв оцінки стану проекту використовувався термін - прогрес в реалізації проекту. Прогрес може бути виражений різними способами, наприклад, повне завершення окремих етапів робіт, часткова реалізація робіт, там де для оцінки стану справ використовувався - процент виконання; незавершеність проекту, якщо вона планується.

Виконання або невиконання яких-небудь контрольних етапів називаються *якісним прогресом*.

Кількісний прогрес

Кількісним прогресом називають прогрес, який можна оцінити показниками, вираженими в одиницях вимірювання робіт. Конкретні фізичні показники прогресу можуть інтегруватись в єдиний показник грошових витрат, це дозволяє порівняти фактичні витрати з плановими.

Для вимірювання прогресу можуть використовуватися різні шкали в залежності від специфіки роботи, що виконується, наприклад:

Вимірні роботи, для яких можуть визначатися дискретні прирости відповідно до графіка виконання, завершення таких робіт має конкретні матеріальні результати. Тобто це роботи для яких можна визначати кількісний прогрес.

Роботи впливу

Роботи впливу, які не можна розбити на дискретні заплановані прирости, роботи типу підтримки і керівництва проектом, лобіювання у владних структурах тощо. Для таких робіт визначають якісний прогрес.

Контроль прогресу в реалізації проекту - це порівняння запланованих і реалізованих до відповідного терміна проміжних або кінцевих результатів.

Фактична інформація по виконанню робіт не впливає на базовий (директивний) план; за визначенням, базовий план є основою для вимірювання прогресу. Базовий план повинен бути незмінним і використовуватися для порівняння з поточним станом в звітах.

Варіанти оцінки прогресу

Одним з варіантів оцінки прогресу - реєстрація його по таких критеріях:

- досягнення контрольних точок (етапів) у виконанні календарного плану проекту (контроль і аналіз термінів закінчення робіт та ін. часових характеристик);
- витрати фінансових коштів;
- витрати ресурсів і ефективність їх використання (блок показників на кожний вид ресурсів);
- величина отриманих прибутків або обсяги виконаних робіт;
- якість (блок якісних характеристик проекту);
- зміст робіт.

Очевидно, що найголовнішим показником для контролю і аналізу є *терміни закінчення робіт*. Якщо були виявлені затримки в роботах критичного шляху або в досягненні ключових віх проекту, то швидше усього весь проект буде затриманий на відповідний термін.

Фактична інформація використовується для складання нових графіків, що базуються на реальних даних. Для кожної роботи оцінюється її стан (початок, закінчення, час, що вже витрачено і час (тривалість), що залишився), обчислюються нові тривалості для робіт, що виконуються. Ці нові тривалості, які можуть бути довшими або коротшими за тривалості у базовому плані, переміщують всі послідовні роботи по графіку, що призводить до зміни дат робіт, які ще не розпочаті. Цей процес звичайно веде до необхідності коригування дати завершення проекту в цілому.

Після отримання звіту з фактичними даними отримуємо два графіки робіт: базовий графік і поточний графік, який відображає вплив останніх фактичних даних. Визначення стану проекту означає порівняння цих двох планів.

Слід зазначити, що звіт про процент завершення часто не дає розробнику корисної інформації про прогрес у роботі (наприклад, можлива ситуація, коли

роботи, які досягли 80% завершеності, у подальшому можуть виконуватися протягом 50% або більше часу від загального (встановленого) часу їх виконання). З іншого боку, звіт по виконаній тривалості дає можливість оцінити час, затрачений на виконання роботи, але не розглядає, скільки додаткових зусиль, що треба докласти для її завершення. Для забезпечення повноцінної підтримки прийняття рішень розробник повинен використати комплекс методів і набір стандартних звітів, що забезпечують його значущою інформацією.

Загальна тривалість роботи завжди дорівнює сумі тривалостей вже минулих робочих періодів (до даної дати) і оціночної тривалості майбутніх робочих періодів, необхідних для виконання роботи. Це вірно як для часових оцінок, так і для ресурсних і вартісних оцінок.

Використання методів планування часових параметрів проекту дозволяє легко перерахувати дати закінчення всіх робіт.

Виконання і витрачений час є досить інформативними показниками, оскільки часто існує суттєва невідповідність між кількістю часу, яку проект або робота використали до поточної дати, і дійсними результатами, ступенем завершеності роботи.

У процесі виконання проекту проводиться аналіз фактичного стану проекту, враховуючи повністю закінчені роботи, досягнуті проміжні результати, а також такі, що піддаються вимірюванню й оцінці ступеня завершеності роботи.

Оцінки по виконаних і майбутніх обсягах робіт також можуть бути корисні для:

- перегляду оцінок тривалостей робіт;
- визначення причин затримок;
- вартісного аналізу фактичного стану.

Перегляд оцінок тривалостей робіт проводиться тоді, коли на стадії планування зроблені помилки в оцінюванні тривалостей робіт на основі їх обсягу, що неминуче виявиться в звітах про фактичне виконання. У цьому випадку оцінки тривалості робіт повинні бути переглянуті.

Визначення причин затримок проводиться на основі спільного аналізу відхилень від плану за часом і виконаними обсягами робіт, що може дати менеджеру початкову інформацію про причини затримок.

Прийняття рішень

Визначивши відхилення проекту від плану, менеджер повинен зробити відповідні дії. Чим раніше коригуючі дії зроблені, тим краще. Дії по відновленню контролю над проектом рекомендується також ретельно планувати.

П'ять основних можливих варіантів дій найчастіше використовуються у разі відхилення проекту від плану:

Знайти альтернативне рішення.

Насамперед необхідно розглянути можливості, пов'язані з підвищенням ефективності робіт за рахунок нових технологічних або організаційних рішень.

Нове рішення, наприклад, може полягати в зміні послідовності виконання ряду робіт;

Перегляд вартості.

Даний підхід означає збільшення обсягів робіт і призначення додаткових ресурсів. Рішення може полягати в збільшенні навантаження на існуючі ресурси або залученні додаткових людей, обладнання, матеріалів. Цей підхід звичайно застосовується у разі необхідності усунення тимчасових затримок проекту;

Перегляд термінів.

Даний підхід означає, що терміни виконання робіт будуть перенесені. Керівництво проекту може піти на таке рішення у разі жорстких обмежень по вартості;

Перегляд змісту робіт.

Даний підхід передбачає, що обсяг робіт по проекту може бути зменшений і відповідно лише частина запланованих результатів проекту буде досягнута, за умови, що перегляд якісних характеристик результатів проекту не передбачається;

Припинення проекту.

Це приймається тоді, коли прогнозовані витрати по проекту перевищують очікувані вигоди. Рішення, пов'язане з припиненням проекту, крім суто економічних аспектів, пов'язане з подоланням проблем психологічного характеру, пов'язаних з інтересами різних учасників проекту.