МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ
“ДНІПРОВСЬКА ПОЛІТЕХНІКА”

ФАКУЛЬТЕТ ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ
Кафедра інформаційних технологій та
комп’ютерної інженерії

Гаркуша І.М.

Конспект лекцій
з дисципліни
“Проектування інформаційних систем”
для студентів галузі знань 12 “Інформаційні технології”
спеціальності 126 “Інформаційні системи та технології”

Дніпро
НТУ “ДП”
2020

УДК 004.01+004.4'2+004.43
Г20

Гаркуша І.М. Конспект лекцій з дисципліни “Проектування інформаційних систем” для студентів галузі знань 12 “Інформаційні технології” спеціальності 126 “Інформаційні системи та технології”. – Д.: НТУ «ДП», 2020. – 75 с.

В конспекті лекцій з дисципліни “Проектування інформаційних систем” для студентів спеціальності 126 “Інформаційні системи та технології” розглянуті класифікація інформаційних систем, стандарти та методології проектування програмного забезпечення, а також основні різновиди діаграм мови UML, яка використовується при описі систем, що проектуються.

В курсі також розглядаються засади проектування графічних інтерфейсів користувача на базі бібліотеки Java Swing, а також показані реалізації відомих шаблонів проектування на мові програмування Java.

Погоджено рішенням науково-методичної комісії спеціальності 126 Інформаційні системи та технології (протокол № 7 від 27.08.2020).

ЗМІСТ

ВСТУП

Метою дисципліни “Проектування інформаційних систем” для студентів спеціальності 126 “Інформаційні системи та технології” є формування компетентностей щодо проектування інформаційних систем з використанням мови моделювання UML та технологій Java.

При складанні лекцій значну увагу приділено розгляду певних різновидів діаграм мови UML, їх реалізації в середовищі BoUML.

Також приділено багато уваги реалізації компонентів програмного забезпечення, зокрема елементів графічного інтерфейсу користувача, мовою програмування Java. При використанні коду на Java студент може використовувати будь-які зручні середовища розробки. Рекомендацією є використання середовищ NetBeans або IntelliJ IDEA.

Лекційний курс є базою для подальшого вивчання процесу проектування інформаційних систем протягом проектно-технологічної практики, спецкурсу, який пов’язаний з детальним використанням шаблонів проектування, а також виконанням кваліфікаційної роботи.

Основними дисциплінарними результатами навчання, після завершення лекційного курсу “Проектування інформаційних систем”, є:

– володіння знаннями щодо методології створення інформаційних систем, поняття життєвого циклу, сучасних стандартів та технологій розробки програмного забезпечення, як компоненти інформаційних систем;

– надання обґрунтованого опису структури програмного забезпечення, яке входить до складу інформаційних систем, що проектуються;

– застосування універсальної мови UML в процесі опису проектів, структур та архітектур інформаційних систем та технологій;

– надання опису складових інформаційних систем за допомогою діаграм розгортання на мові UML;

– використання певних передових технологій для створення програмного забезпечення як складової інформаційних систем, зокрема Java-технології;

– обґрунтованого використання шаблонів проектування для розробки програмного забезпечення, що входить до складу інформаційних систем та технологій.

Лекція 1. Класифікація інформаційних систем

Різноманітність завдань, що вирішуються за допомогою інформаційних систем (ІС), призвело до появи безлічі різнотипових систем, що відрізняються принципами побудови та закладеними в них правилами обробки інформації. ІС класифікують за цілою низкою різних ознак. В основу представленої класифікації покладені найбільш істотні ознаки, що визначають функціональні можливості та особливості побудови сучасних систем. Залежно від обсягу вирішуваних завдань, технічних засобів, що використовують та організації функціонування, ІС діляться на ряд груп (класів).

1. За типом даних, що зберігаються

ІС діляться на фактографічні та документальні. Фактографічні системи призначені для зберігання та обробки структурованих даних у вигляді чисел і текстів. Над такими даними можна виконувати різні операції. У документальних системах інформація представлена у вигляді документів, що складаються з найменувань, описів, рефератів і текстів. Пошук по неструктурованим даним здійснюється з використанням семантичних ознак. Відібрані документи надаються користувачеві, а обробка даних в таких системах практично не проводиться.

2. За ступенем автоматизації інформаційних процесів

ІС діляться на ручні, автоматичні та автоматизовані.

Ручні ІС характеризуються відсутністю сучасних технічних засобів переробки інформації та виконанням всіх операцій людиною.

В автоматичних ІС всі операції з переробки інформації виконуються без участі людини.

Автоматизовані ІС припускають участь в процесі обробки інформації і людини, і технічних засобів, причому головна роль у виконанні рутинних операцій обробки даних відводиться комп'ютеру. Саме цей клас систем відповідає сучасному уявленню поняття "інформаційна система".

3. За характером обробки даних

ІС діляться на інформаційно-пошукові та інформаційно-вирішальні.

Інформаційно-пошукові системи роблять введення, систематизацію, зберігання, видачу інформації за запитом користувача без складних перетворень даних. Наприклад, ІС бібліотечного обслуговування, резервування та продажу квитків на транспорті, бронювання місць в готелях та ін.

Інформаційно-вирішальні системи здійснюють, крім того, операції переробки інформації за певним алгоритмом. Інформаційно-вирішальні ІС за характером використання вихідної інформації діляться на керуючі і такі, що

радять (рос.: советующие) що робити.

Результуюча інформація керуючих ІС безпосередньо трансформується в прийняття людиною рішення. Для цих систем характерні завдання розрахункового характеру та обробка великих обсягів даних. Наприклад, ІС планування виробництва або замовлень, бухгалтерського обліку.

ІС, які радять, виробляють інформацію, що приймається людиною до відома та враховується при формуванні управлінських рішень, а не ініціює конкретні дії. Ці системи імітують інтелектуальні процеси обробки знань, а не даних. Наприклад, експертні системи.

4. За сферою застосування

а) ІС організаційного управління – призначені для автоматизації функцій управлінського персоналу як промислових підприємств, так і непромислових об'єктів (готелів, банків, магазинів та ін.). Основними функціями подібних систем є: оперативний контроль та регулювання, оперативний облік й аналіз, перспективне і оперативне планування, бухгалтерський облік, управління збутом, постачанням та інші економічні й організаційні завдання.

б) ІС управління технологічними процесами (ТП) – служать для автоматизації функцій виробничого персоналу по контролю та управлінню виробничими операціями. У таких системах зазвичай передбачається наявність розвинених засобів вимірювання параметрів технологічних процесів (температури, тиску, хімічного складу і т.п.), процедур контролю допустимості значень параметрів і регулювання технологічних процесів.

в) ІС автоматизованого проектування (САПР) – призначені для автоматизації функцій інженерів-проектувальників, конструкторів, архітекторів, дизайнерів при створенні нової техніки або технології. Основними функціями подібних систем є: інженерні розрахунки, створення графічної документації (креслень, схем, планів), створення проектної документації, моделювання проектованих об'єктів.

г) інтегровані (корпоративні) ІС (КІС) – використовуються для автоматизації всіх функцій організації і охоплюють весь цикл робіт від планування діяльності до збуту продукції. Вони включають в себе ряд модулів (підсистем), що працюють в єдиному інформаційному просторі та виконують функції підтримки відповідних напрямів діяльності.

5. В залежності від рівня управління, на якому використовується ІС

а) інформаційна система оперативного рівня – підтримує виконавців, обробляючи дані про угоди і події (рахунки, накладні, зарплата, кредити, потік сировини та матеріалів). Така система є зв’язуючею ланкою між організацією та зовнішнім середовищем. Завдання, цілі, джерела інформації та алгоритми обробки на оперативному рівні заздалегідь визначені і у високому ступені структуровані.

б) ІС фахівців – підтримують роботу з даними та знаннями, підвищують

продуктивність й продуктивність роботи інженерів та проектувальників. Завдання подібних ІС – інтеграція нових відомостей в організацію та допомога в обробці паперових документів.

в) ІС рівня менеджменту – використовуються працівниками середньої управлінської ланки для моніторингу, контролю, прийняття рішень та адміністрування. Основні функції таких ІС:

– порівняння поточних показників з минулими;

– складання періодичних звітів за певний час, а не видача звітів щодо поточних подій, як на оперативному рівні;

– забезпечення доступу до архівної інформації і т.ін.

г) стратегічні ІС – забезпечують підтримку прийняття рішень по реалізації стратегічних перспективних цілей розвитку організації. ІС стратегічного рівня допомагають вищій ланці управлінців вирішувати неструктуровані завдання, здійснювати довгострокове планування. Основне завдання – порівняння змін, що відбуваються в зовнішньому оточенні, до існуючого потенціала організації. Вони покликані створити загальну середу комп'ютерної телекомунікаційної підтримки рішень в несподівано виникаючих ситуаціях. Використовуючи найдосконаліші програми, ці системи здатні в будь-який момент надати інформацію з багатьох джерел. Деякі стратегічні системи мають обмежені аналітичні можливості.

6. За програмно-апаратною реалізацією

а) традиційні архітектурні рішення, засновані на використанні виділених файл-серверів або серверів баз даних;

б) архітектури КІС, що базуються на технології Internet (Intranet-додатків);

в) архітектури ІС, що ґрунтуються на концепції "сховища даних" (Data Warehouse) – інтегрованого інформаційного середовища, що включає різнорідні інформаційні ресурси;

г) архітектура інтеграції інформаційно-обчислювальних компонентів на основі об'єктно-орієнтованого підходу (для побудови глобальних розподілених інформаційних додатків);

д) архітектури на основі хмарних сервісів.

Лекція 2. Методологія проектування корпоративних ІС. Основні положення

Мета методології полягає в регламентації процесу проектування ІС та забезпеченні управління цим процесом з тим, щоб гарантувати виконання вимог як до самої ІС, так і до характеристик процесу розробки.

Основними завданнями, вирішення яких має сприяти методологія проектування корпоративних ІС, є наступні.

1. Забезпечувати створення ІС, що відповідають цілям і задачам організації, а також вимогам, що пред'являються по автоматизації ділових процесів замовника.

2. Гарантувати створення системи із заданою якістю в задані терміни і в рамках встановленого бюджету проекту.

3. Підтримувати зручну дисципліну супроводу, модифікації та нарощування системи.

4. Забезпечувати спадкоємність розробки, тобто використання в ІС що розробляється, існуючої інформаційної інфраструктури організації.

Впровадження методології повинно призводити до зниження складності процесу створення ІС за рахунок повного і точного опису цього процесу, а також застосування сучасних методів і технологій створення ІС на всьому життєвому циклі ІС – від задуму до реалізації.

Проектування ІС охоплює три основні області.

1. Проектування об'єктів даних, що реалізуються в базі даних.

2. Проектування програм, екранних форм, звітів, які будуть забезпечувати виконання запитів до даних.

3. Облік конкретного середовища або технології, а саме: топології мережі, конфігурації апаратних засобів, використовуваної архітектури (файл-сервер або клієнт-сервер), паралельної обробки, розподіленої обробки даних і т.п.

Проектування ІС завжди починається **з визначення мети проекту**. У загальному вигляді мету проекту можна визначити як рішення ряду взаємопов'язаних завдань, що включають в себе забезпечення на момент запуску системи і протягом всього часу її експлуатації:

– необхідної функціональності системи і рівня її адаптивності до постійно змінюваних умов функціонування;

– необхідної пропускної спроможності системи;

– необхідного часу реакції системи на запит;

– безвідмовної роботи системи;

– необхідного рівня безпеки;

– простоти експлуатації та підтримки системи.

Відповідно до сучасної методології, процес створення ІС являє собою процес побудови і послідовного перетворення ряду узгоджених моделей на всіх етапах життєвого циклу (ЖЦ) ІС. На кожному етапі ЖЦ створюються специфічні для нього моделі – організації вимог до ІС, проекту ІС, вимог до додатків і т.д. Моделі формуються робочими групами команди проекту, зберігаються і накопичуються в репозиторії проекту. Створення моделей, їх контроль, перетворення і надання в колективне користування здійснюється з використанням спеціальних програмних інструментів – CASE-засобів.

Процес створення ІС ділиться на ряд етапів (стадій), обмежених деякими тимчасовими рамками і закінчуються випуском конкретного продукту (моделей, програмних продуктів, документації тощо.).

Зазвичай виділяють наступні етапи створення ІС.

1. Формування вимог до системи.
2. Проектування.
3. Реалізація.
4. Тестування.
5. Введення в дію.
6. Експлуатація та супровід.

Модель ЖЦ відображає різні стани системи. Це структура, яка містить процеси, дії і завдання, які здійснюються в ході розробки, функціонування та супроводу програмного продукту протягом усього життя системи, від визначення вимог до завершення її використання.

На даний час відомі і використовуються наступні моделі ЖЦ.

1. Каскадна модель (рис. 2.1) передбачає послідовне виконання всіх етапів проекту в строго фіксованому порядку. Перехід на наступний етап означає повне завершення робіт на попередньому етапі.

2. Поетапна модель з проміжним контролем (рис. 2.2). Розробка ІС ведеться ітераціями з циклами зворотного зв'язку між етапами. Міжетапні коригування дозволяють враховувати реально існуючий взаємовплив результатів розробки на різних етапах; час життя кожного з етапів розтягується на весь період розробки.

3. Спіральна модель (рис. 2.3). На кожному витку спіралі виконується створення чергової версії продукту, уточнюються вимоги проекту, визначається його якість і плануються роботи наступного витка. Особлива увага приділяється початковим етапам розробки – аналізу і проектування, де реалізація тих чи інших технічних рішень перевіряється та обґрунтовується за допомогою створення прототипів (макетування).

Рис. 2.1. Каскадна модель ЖЦ ІС

Рис. 2.2. Поетапна модель з проміжним контролем

Рис. 2.3. Спіральна модель ЖЦ ІС

На практиці найбільшого поширення набули наступні дві основні моделі:
– каскадна модель (характерна для періоду 1970-1985 рр.);
– спіральна модель (характерна для періоду після 1986 року).

Формування вимог до системи

Початковим етапом процесу створення ІС є моделювання бізнес-процесів, що протікають в організації і реалізують її цілі та завдання. Модель організації, описана в термінах бізнес-процесів та бізнес-функцій, дозволяє сформулювати основні вимоги до ІС. Це фундаментальне положення методології забезпечує об'єктивність у виробленні вимог до проектування системи. Безліч моделей опису вимог до ІС потім перетворюється в систему моделей, що описують концептуальний проект ІС. Формуються моделі архітектури ІС, вимог до програмного забезпечення (ПЗ) та інформаційного забезпечення (ІЗ). Потім формується архітектура ПЗ та ІЗ, виділяються корпоративні БД та окремі програми, формуються моделі вимог до програм та проводиться їх розробка, тестування та інтеграція.

Проектування

На етапі проектування перш за все формуються моделі даних. Проектувальники в якості вихідної інформації отримують результати аналізу. Побудова логічної та фізичної моделей даних є основною частиною проектування БД. Отримана в процесі аналізу інформаційна модель спочатку перетвориться в логічну, а потім у фізичну модель даних.

Паралельно з проектуванням схеми БД виконується проектування процесів, щоб отримати специфікації (опис) всіх модулей ІС. Обидва ці процесу проектування тісно пов'язані, оскільки частина бізнес-логіки зазвичай реалізується в БД (обмеження, тригери, збережені процедури). Головна мета проектування процесів полягає у відображенні функцій, отриманих на етапі аналізу, в модулі ІС. При проектуванні модулів визначають інтерфейси програм: розмітка меню, вид вікон, гарячі клавіші та пов'язані з ними виклики.

Кінцевими продуктами етапу проектування є:

– схема бази даних (на підставі ER-моделі, розробленої на етапі аналізу);

– набір специфікацій модулів системи (вони будуються на базі моделей функцій).

Крім того, на етапі проектування здійснюється також розробка архітектури ІС, що включає в себе вибір платформи (платформ) та операційної системи (операційних систем). У неоднорідній ІС можуть працювати кілька комп'ютерів на різних апаратних платформах та під управлінням різних операційних систем. Крім вибору платформи, на етапі проектування визначаються наступні характеристики архітектури:

– буде це архітектура "файл-сервер" або "клієнт-сервер";

– буде це трьохрівнева архітектура з наступними шарами: сервер, ПО проміжного шару (сервер додатків), клієнтське ПЗ;

– чи буде БД централізованою або розподіленою. Якщо БД буде розподіленої, то які механізми підтримки узгодженості та актуальності даних

будуть використовуватися;

– чи буде БД однорідної, тобто, чи будуть всі сервери БД продуктами одного й того ж виробника (наприклад, всі сервери тільки Oracle або всі сервери тільки MS SQL). Якщо БД не буде однорідною, то яке ПЗ буде використано для обміну даними між СУБД різних виробників (вже існуюче або розроблене спеціально як частина проекту);

– чи будуть для досягнення належної продуктивності використовуватися паралельні сервери БД (наприклад, Oracle Parallel Server, MS SQL і т.п.).

Етап проектування завершується розробкою технічного проекту ІС.

Реалізація

На етапі реалізації здійснюється створення ПЗ системи, установка технічних засобів, розробка експлуатаційної документації.

Тестування

Етап тестування зазвичай виявляється розподіленим в часі.

Після завершення розробки окремого модуля системи виконують автономний тест, який переслідує дві основні мети:

– виявлення відмов модуля (жорстких збоїв);

– відповідність модуля специфікації (наявність всіх необхідних функцій, відсутність зайвих функцій).

Після того як автономний тест успішно пройдено, модуль включається до складу розробленої частини системи і група згенерованих модулів проходить тести зв'язків, які повинні відстежити їх взаємний вплив.

Далі група модулів тестується на надійність роботи, тобто проходять, по-перше, тести імітації відмов системи, а по-друге, тести напрацювання на відмову. Перша група тестів показує, наскільки добре система відновлюється після збоїв ПЗ, відмов апаратного забезпечення. Друга група тестів визначає ступінь стійкості системи при штатній роботі та дозволяє оцінити час безвідмовної роботи системи. У комплект тестів стійкості повинні входити тести, що імітують пікове навантаження на систему.

Потім весь комплект модулів проходить системний тест – тест внутрішньої приймання товару, що показує рівень його якості. Сюди входять тести функціональності та тести надійності системи.

Останній тест ІС – приймально-здавальні випробування. Такий тест передбачає показ ІС замовнику і повинен містити групу тестів, що моделюють реальні бізнес-процеси, щоб показати відповідність реалізації вимогам замовника.

При використанні різних методологій розробки ПЗ, етапи тестування можуть бути й іншими та сильно залежать від технологій розробки та

тестування у конкретних організаціях.

Необхідність контролювати процес створення ІС, гарантувати досягнення цілей розробки і дотримання різних обмежень (бюджетних, тимчасових і ін.) призвело до широкого використання в цій сфері методів і засобів програмної інженерії: структурного аналізу, об'єктно-орієнтованого моделювання, CASE-систем.

Лекція 3. Стандарти та методології проектування програмного забезпечення

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

Значний внесок у теорію проектування та розробки ІС внесла компанія IBM, запропонувавши 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 ІС, інформаційних об'єктів, документів та БД, запропонований в BSP, використовується сьогодні не тільки в ІТ-проектах, але й в проектах по реінжинірингу бізнес-процесів, зміни організаційної структури. Найважливіші кроки процесу BSP, їх послідовність можна зустріти практично у всіх формальних методиках, а також в проектах, що реалізуються на практиці.

Серед найбільш відомих стандартів та методологій можна виділити наступні:

– ГОСТ 34.601-90 “Информационная технология. Комплекс стандартов на автоматизированные системы. Автоматизированные системы. Стадии создания” – поширюється на автоматизовані системи та встановлює стадії і етапи їх створення. Крім того, в стандарті міститься опис змісту робіт на кожному етапі. Стадії та етапи роботи, закріплені в стандарті, більшою мірою відповідають каскадній моделі ЖЦ. Найближча дата відміни стандарту – 01.01.2022 й в тому числі в Україні (http://csm.kiev.ua/nd/nd.php?b=2&l=1710).

– ГОСТ 34.602-89 “Техническое задание на создание автоматизированной системы”. Найближча дата відміни стандарту – 01.01.2022 й в тому числі в Україні (http://csm.kiev.ua/nd/nd.php?z=34.602-89&st=0&b=2).

– Сімейство стандартів ISO/IEC 12207 – стандарти на процеси та організацію ЖЦ. Поширюється на всі види замовного ПЗ. Найсучасніший з них – ISO/IEC 12207:2008 Systems and software engineering – Software life cycle processes.

– Custom Development Method (CDM, методика Oracle) з розробки прикладних ІС – технологічний матеріал, деталізований до рівня заготівок проектних документів, розрахованих на використання в проектах із застосуванням технологій Oracle. CDM застосовується для класичної моделі ЖЦ (передбачені всі роботи/завдання та етапи), а також для технологій "швидкої розробки" (Fast Track) або "полегшеного підходу", рекомендованих у разі малих проектів.

– 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 мине ті ж фази. Суть роботи в рамках RUP – це створення й супровід моделей на базі UML.

– Microsoft Solution Framework (MSF) подібна до RUP, так само включає чотири фази: аналіз, проектування, розробка, стабілізація, є ітераційної, припускає використання об'єктно-орієнтованого моделювання. MSF в порівнянні з RUP більшою мірою орієнтована на розробку бізнес-додатків.

– Agile software development – гнучка методологія розробки (рис. 3.1), в основі якої лежать так звані 4 основні ідеї та 12 принципів маніфесту, який був прийнятий на початку 2001 року. Основні ідеї наступні.

1. Люди та взаємодія важливіше процесів та інструментів.
2. Працюючий продукт важливіше вичерпної документації.
3. Співпраця з замовником важливіше узгодження умов контракту.
4. Готовність до змін важливіше проходження попереднім планом.

Принципи маніфесту наступні.

1. Задоволення клієнта за рахунок ранньої та безперебійної поставки цінного ПЗ.
2. Вітання змін вимог навіть в кінці розробки (це може підвищити конкурентоспроможність отриманого продукту).
3. Часта поставки робочого ПЗ (кожного місяця або тижня або ще частіше).
4. Тісне, щоденне спілкування замовника з розробниками протягом усього проекту.
5. Проектом займаються мотивовані особистості, які забезпечені потрібними умовами роботи, підтримкою та довірою.
6. Рекомендований метод передачі інформації – особиста розмова (обличчям до обличчя).
7. Працююче ПЗ – кращий вимірювач прогресу.
8. Спонсори, розробники та користувачі повинні мати можливість підтримувати постійний темп на невизначений термін.
9. Постійна увага поліпшенню технічної майстерності та зручному дизайну.
10. Простота – мистецтво не робити зайвої роботи.
11. Кращі технічні вимоги, дизайн та архітектура виходять у самоорганізованої команди.
12. Постійна адаптація до обставин, що змінюються. Команда повинна систематично аналізувати можливі способи поліпшення ефективності і відповідно коригувати стиль своєї роботи.

Сьогодні agile-методи стали фактично узагальненим терміном для цілого

ряду підходів та практик, заснованих на цінностях маніфесту гнучкої розробки.

Найвідомішими гнучкими методологіями є Extreme Programming (XP), scrum, kanban та ін.

Extreme Programming (XP). Екстремальне програмування сформувалося в 1996 році. В основі методології командна робота, ефективна комунікація між замовником і виконавцем протягом усього проекту з розробки ІС, а розробка ведеться з використанням послідовно доопрацьованих прототипів.

Scrum – це “підхід структури”. Над кожним проектом працює універсальна команда фахівців, до якої приєднується ще двоє людей: власник продукту та scrum-майстер. Перший з'єднує команду з замовником та стежить за розвитком проекту; це не формальний керівник команди, а скоріше куратор. Scrum-майстер допомагає першому організувати бізнес-процес: проводить загальні збори, вирішує побутові проблеми, мотивує команду та стежить за дотриманням scrum-підходу.

Scrum-підхід поділяє робочий процес на рівні спринти – зазвичай це періоди від тижня до місяця, в залежності від проекту та команди. Перед спринтом формулюються задачі на даний спринт, в кінці – обговорюються результати, а команда починає новий спринт. Спринти дуже зручно порівнювати між собою, що дозволяє управляти ефективністю роботи.

Kanban – це “підхід балансу”. Його завдання – збалансувати різних фахівців всередині команди та уникнути ситуації, коли дизайнери працюють цілодобово, а розробники скаржаться на відсутність нових завдань.

Вся команда єдина – в kanban немає ролей власника продукту та scrum-майстра. Бізнес-процес поділяється не на універсальні спринти, а на стадії виконання конкретних завдань: «Планується», «Розробляється», «Тестується», «Завершено» та ін.

Головний показник ефективності в kanban – це середній час проходження завдання по дошці. Завдання пройдено швидко – команда працювала продуктивно і злагоджено. Завдання затягнулося – треба думати, на якому етапі і чому виникли затримки та чию роботу треба оптимізувати.

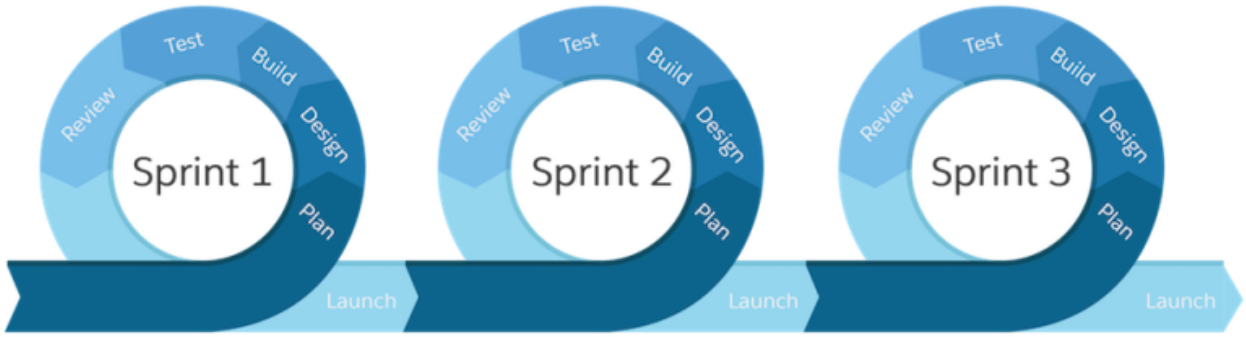


Рис. 3.1. Процес розробки за agile-методологією

Для візуалізації agile-підходів використовують дошки: фізичні та електронні. Вони дозволяють зробити робочий процес відкритим та зрозумілим для всіх фахівців, що важливо, коли у команді немає одного формального керівника.

Відповідно до базового міжнародного стандарту ISO/IEC 12207 всі процеси ЖЦ ПЗ діляться на три групи:

1. Основні процеси:

– придбання;
– поставка;
– розробка;
– експлуатація;
– супровід.

2. Допоміжні процеси:

– документування;
– управління конфігурацією;
– забезпечення якості;
– вирішення проблем;
– аудит;
– атестація;
– спільна оцінка;
– верифікація.

3. Організаційні процеси:

– створення інфраструктури;
– управління;
– навчання;
– удосконалення.

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

Для підтримки практичного застосування стандарту ISO/IEC 12207 розроблений ряд технологічних документів: керівництво для ISO/IEC 12207 (ISO/IEC TR 15271:1998 Information technology – Guide for ISO/IEC 12207) та керівництво по застосуванню ISO/IEC 12207 до управління проектами (ISO/IEC TR 16326:1999 Software engineering – Guide for the application of ISO/IEC 12207 to project management).

У 2002 р опублікований стандарт на процеси ЖЦ 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 та академічних організаціях. Стандарт застосуємо для широкого класу систем, але його основне призначення – підтримка створення комп'ютеризованих систем.

Відповідно до стандарту ISO/IEC серії 15288 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 та цілей.

Більш детально про цикли розробки можна знайти на сторінці:

https://en.wikipedia.org/wiki/Systems_development_life_cycle

Загалом ПЗ, що розробляється, часто розглядається як чорний ящик. Хід процесу проектування і його результати залежать не тільки від складу вимог, а й обраної моделі процесу, досвіду проектувальника.

Модель предметної області накладає обмеження на бізнес-логіку та структури даних.

Залежно від класу створюваного ПЗ, процес проектування може забезпечуватися як «ручним» проектуванням, так і різними засобами його автоматизації. В процесі проектування ПЗ для вираження його характеристик використовуються різні нотації – блок-схеми, ER-діаграми, UML-діаграми, DFD-діаграми, а також макети.

Проектуванню зазвичай підлягають:

– архітектура ПЗ;

– компоненти ПЗ;

– інтерфейси користувача.

SADT (акронім від англ. Structured Analysis and Design Technique) – методологія структурного аналізу та проектування, яка інтегрує процес моделювання, управління конфігурацією проекту, використання додаткових мовних засобів та керівництво проектом зі своєю графічною мовою. Процес моделювання може бути розділений на декілька етапів: опитування експертів, створення діаграм і моделей, поширення документації, оцінка адекватності моделей і прийняття їх для подальшого використання. Цей процес добре налагоджений, тому що при розробці проекту фахівці виконують конкретні обов'язки, а бібліотекар забезпечує своєчасний обмін інформацією.

SADT виникла в кінці 60-х років в ході революції, викликаною структурним програмуванням. Коли більшість фахівців билося над створенням ПЗ, деякі намагалися вирішити більш складну задачу створення великомасштабних систем, що включають як людей, машини, так і програмне забезпечення, аналогічних систем, що застосовуються в сферах телефонного зв'язку, промисловості, управлінні та у військовій справі. У той час фахівці, традиційно займалися створенням великомасштабних систем, стали усвідомлювати необхідність більшої впорядкованості. Таким чином, розробники вирішили формалізувати процес створення системи, розбивши саме на фази аналізу, проектування, реалізації, тестування, встановлення та експлуатації.

IDEF – методології сімейства ICAM (Integrated Computer-Aided Manufacturing) для вирішення завдань моделювання складних систем, дозволяє відображати і аналізувати моделі діяльності широкого спектру складних систем в різних розрізах. При цьому широта та глибина обстеження процесів в системі визначається самим розробником, що дозволяє не перевантажувати створювану модель зайвими даними.

Після опублікування стандарту IDEF він був успішно застосований в

самих різних областях бізнесу, показавши себе ефективним засобом аналізу, конструювання та відображення бізнес-процесів. Більш того, власне з широким застосуванням IDEF (і попередньої методології SADT) і пов'язане виникнення основних ідей популярного нині поняття – BPR (бізнес-процес реінжиніринг).

DFD – загальноприйняте скорочення від англ. Data Flow Diagrams – діаграми потоків даних. Так називається методологія графічного структурного аналізу, що описує зовнішні по відношенню до системи джерела і адресати даних, логічні функції, потоки даних і сховища даних, до яких здійснюється доступ.

Діаграма потоків даних (data flow diagram, DFD) – один з основних інструментів структурного аналізу і проектування інформаційних систем, що існувала до широкого поширення UML.

Незважаючи на те, що в сучасних умовах має місце зміщення акцентів від структурного до об'єктно-орієнтованого підходу аналізу і проектування систем, «старовинні» структурні нотації як і раніше широко і ефективно використовуються як в бізнес-аналізі, так і в аналізі інформаційних систем.

Історично склалося так, що для опису діаграм DFD використовуються дві нотації – Йодана (Yourdon) та Гейне-Сарсона (Gane-Sarson), які відрізняються синтаксисом.

Модель DFD, як і більшість інших структурних моделей є ієрархічною. Кожен процес може бути підданий декомпозиції, тобто розбитий на структурні складові, відносини між якими в тій же нотації можуть бути показані на окремій діаграмі. Коли досягнута необхідна глибина декомпозиції – процес нижнього рівня супроводжується міні-специфікацією (текстовим описом). Нотація DFD підтримує поняття підсистеми – структурної компоненти системи, яка розробляється.

На даний момент велике розповсюдження в процесах проектування інформаційних систем отримала UML (Unified Modeling Language) – уніфікована мова моделювання.

Лекція 4. Введення до UML. Діаграма класів

UML (Unified Modeling Language) – уніфікована мова моделювання – це система позначень, що застосовується для об'єктно-орієнтованого аналізу та проектування, це мова діаграм або позначень для специфікації, візуалізації та документації моделі об'єктно-орієнтованих програмних систем. UML не є методом розробки, тобто він не визначає послідовність дій при розробці ПЗ. Він допомагає описати свою ідею і взаємодіяти з іншими розробниками системи. UML управляється Object Management Group (OMG) і є промисловим стандартом, що описує моделі ПЗ. Повний опис UML є за адресою:

https://www.omg.org/spec/UML/2.5.1/PDF

UML визначає різні види діаграм, наприклад, такі:
1. Діаграма прецедентів.
2. Діаграма класів.
3. Діаграма об'єктів.
4. Діаграма послідовностей.
5. Діаграма взаємодії.
6. Діаграма станів.
7. Діаграма активності.
8. Діаграма розгортання.

Діаграма класів

Це діаграма, на якій показані класи, інтерфейси та відносини між ними. Головний елемент діаграми класів – клас. При проектуванні об'єктно-орієнтованих систем діаграми класів обов'язкові.

Класи використовуються в процесі аналізу предметної області для складання словника предметної області системи, що розробляється. Це можуть бути як абстрактні поняття предметної області, так і класи, на які спирається розробка та які описують програмні або апаратні сутності.

Діаграма класів є набором статичних, декларативних елементів моделі.

Класи зображуються у вигляді прямокутників, зазвичай розділених на дві або три частини (рис. 4.1). У верхній частині знаходиться ім'я класу. Середня частина містить список змінних класу, а нижня частина – методи класу.

Символи, зазначені перед кожною змінною або методом, представляють собою індикатори видимості (visibility indicators). Можливі індикатори видимості та їх значення представлені в таблиці 1. Змінній може присвоюватися відповідне її типу значення за допомогою знака «=» та деякої величини (рис. 4.1).

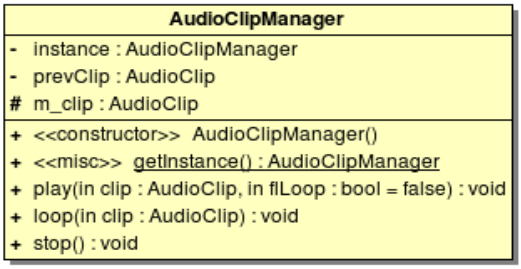


Рис. 4.1. Приклад оформлення класу на діаграмі класів UML

Таблиця 4.1

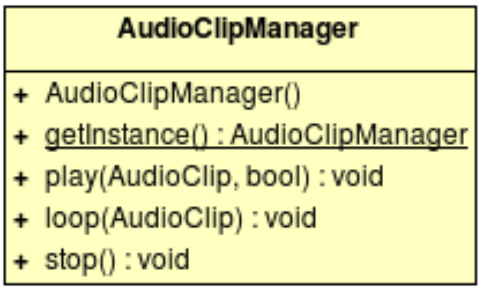
Індикатори видимості

Індикатор	Значення
+	відкритий (public)
#	захищений (protected)
–	закритий (private)
~	для модуля (package)

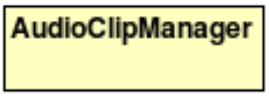
На схемах UML слово, укладене в кутові лапки (дужки), називається *стереотипом*. Стереотип описує те, що за ним випливає. Наприклад, стереотип constructor вказує на те, що наступний за ним метод(и) являє собою конструктор класу. Стереотип misc вказує, що наступний за ним метод(и) – регулярний.

У класі можливий спеціальний елемент – (...) – еліпсис (відображається не у всіх пакетах ПЗ). Він вказує на те, що в класі є додаткові змінні або методи, які не показані на діаграмі.

Як правило, немає необхідності (або просто незручно) показувати клас детально. Тому зображення класу може складатися тільки з двох частин – імені класів та методів або з однієї частини – тільки імені класу (рис. 4.2).



а)



б)

Рис. 4.2. Різновиди зображення класу

Як у методів, так і у змінних можуть бути відсутні індикатори видимості. Однак це не означає, що їх немає.

Інтерфейси зображуються так само, як і класи, тільки у верхній частині імені вказується стереотип interface (рис. 4.3).

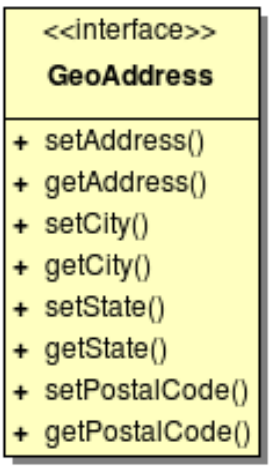


Рис. 4.3. Представлення інтерфейсу

На рис. 4.4, а, б показаний приклад абстрактного класу BaseIO, що є суперкласом для класів BaseFile та TextTerminal. Ім'я абстрактного класу, а також його абстрактні методи записуються курсивом.

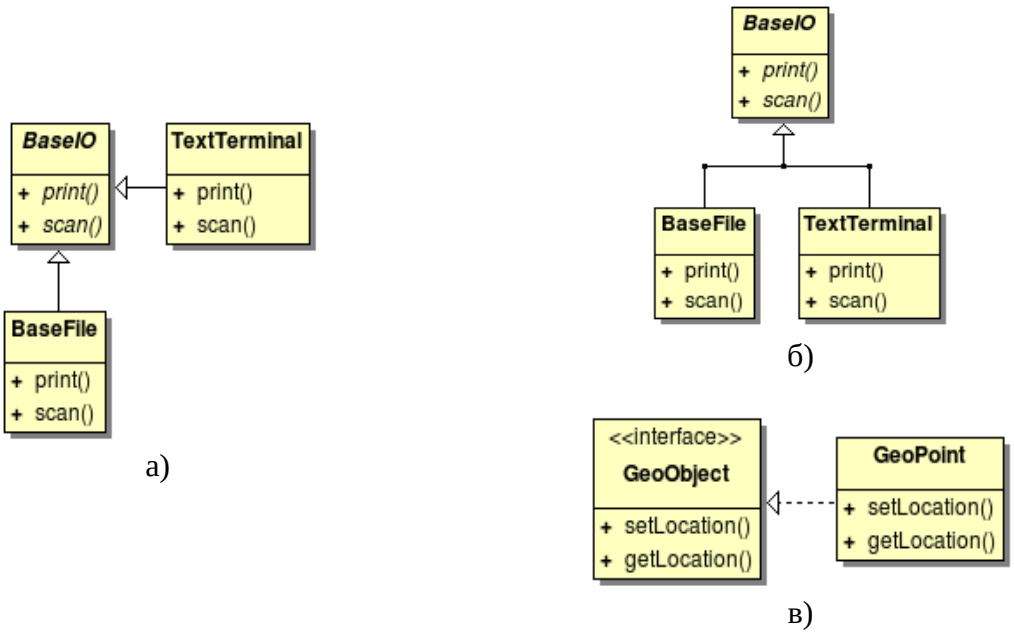


Рис. 4.4. Успадкування підкласів від суперкласу (базового, абстрактного) (а, б) та реалізація класом інтерфейсу (в)

Суцільна лінія зі стрілкою у вигляді замкненого контуру вказує на те, що даний підклас успадковується від суперкласу (рис. 4.4, а, б). Якщо ж клас реалізує інтерфейс, то використовується пунктирна або штрих-пунктирна лінія зі стрілкою у вигляді замкнутого контуру (рис. 4.4, в) – клас GeoPoint реалізує інтерфейс GeoObject.

Якщо між класами використовується звичайна лінія або лінія зі стрілками, то такого типу відносини називаються *асоціаціями*. Разом з асоціаціями можуть зазначатися деякі дані, що представляють інформацію про суть асоціації: ім'я асоціації, стрілки навігації, ім'я ролі, індикатор множинності.

Ім'я асоціації може бути вказано приблизно в середині з'єднання та починатися з великої літери (рис. 4.5). В кінці або на початку імені асоціації може бути зображений трикутник, який вказує напрямок, в якому слід читати асоціацію.

Рис. 4.5. Приклад оформлення асоціації

На рис. 4.5 зображена асоціація Create (Створює) між класами Factory та Product.

Стрілки на кінцях асоціацій називаються *стрілками навігації*. Вони вказують напрямок переміщення по асоціації. Наприклад, стрілка на рис. 4.5 говорить про відповідальність класу Factory за створення екземплярів об'єктів класу Product.

Природа деяких інших асоціацій менш очевидна. Для розуміння суті таких асоціацій може виникнути необхідність в наданні додаткової інформації, що стосується асоціації – задається ім'я ролі, яку кожен клас відіграє в асоціації. Ім'я ролі може бути вказано на будь-якому кінці асоціації. Імена ролей завжди записують малими літерами. Це дозволяє легко відрізнити їх від імен асоціацій.

На рис. 4.6 видно, що клас CreationRequestor бере участь в асоціації в ролі ініціатора запиту, а інтерфейс FactoryIF – в ролі творця.

Індикатор множинності є іншою характеристикою асоціації, яка часто зустрічається та повідомляє про те, скільки примірників кожного класу (об’єктів) беруть участь в асоціації. Такий індикатор може вказуватися на будь-якому кінці асоціації та представляти просто число (наприклад, 0 або 1) або діапазон чисел (наприклад, 0..2). Зірочка використовується замість верхньої межі діапазону та позначає необмежену кількість випадків. Індикатор множинності 1..* вказує принаймні на один екземпляр, 0..* – на будь-яку кількість екземплярів. Просто * еквівалентна 0..*. На рис. 6 все асоціації схеми являють собою співвідношення «один до багатьох».

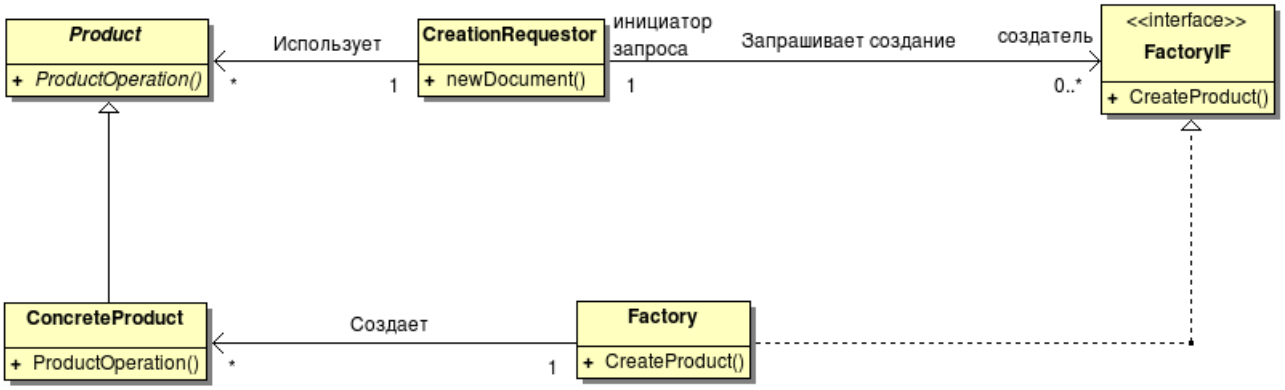


Рис. 4.6. Приклад асоціацій з іменами ролей

Іноді може виникнути необхідність більшої структуризації, ніж в представленій схемі з простим співвідношенням «один до багатьох». Співвідношення «один до багатьох», в якому один об'єкт містить набір інших об'єктів, називається *агрегацією*. На агрегацію вказує контур у вигляді ромба, розташований на тому кінці асоціації, який стикується з класом, що містить екземпляри іншого класу (рис. 4.7).

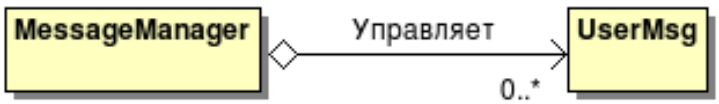


Рис. 4.7. Приклад агрегації

На рис. 4.7 зображено клас MessageManager. З рис. 4.7 випливає, що кожен екземпляр класу MessageManager містить нуль або більше примірників класу UserMsg.

В UML можна використовувати ще одне позначення, яке вказує на більш сильний, ніж агрегація, зв'язок. Цей зв'язок називається *композитною агрегацією*. Щоб агрегація була композитної, повинні виконуватися дві умови:

1. У якийсь момент часу екземпляри, що агрегуються, повинні належати тільки одному складеному (рос.: составному) об'єкту.

2. Деякі операції повинні передаватися у спадок від складеного об'єкта до його екземплярів, що агрегуються. Наприклад, якщо складений об'єкт клонується, то зазвичай клонуються і його агрегуємі екземпляри. Тому новий складений об’єкт володіє власними клонами вихідних агрегованих примірників.

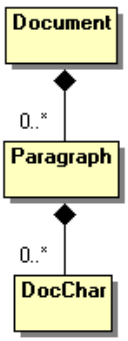


Рис. 4.8. Композитна агрегація

На рис. 4.8 представлена діаграма класів, яка містить композитну агрегацію. Об'єкти класу Document можуть містити об'єкти класу Paragraph, які можуть мати в своєму складі об'єкти класу DocChar. Завдяки композитної агрегації очевидно, що об'єкти класу Paragraph не використовують спільно об'єкти класу DocChar, а об'єкти класу Document не використовують спільно об'єкти класу Paragraph.

Розглянемо більш детально приклади з агрегацією і композицією.

На діаграмі (рис. 4.9) показано, що існує клас University, з яким буде пов'язаний один екземпляр об'єкту та який інкапсулює 20-ть об'єктів типу Department. Сам клас Department такий, що реалізації відповідного класу може бути в межах University 20, однак по відношенню до об'єктів класу Professor їх може бути скільки завгодно. Клас Department агрегує масив з 5-ти покажчиків на об'єкти типу Professor. Кодом на C ++ дана ситуація буде описана наступним чином.

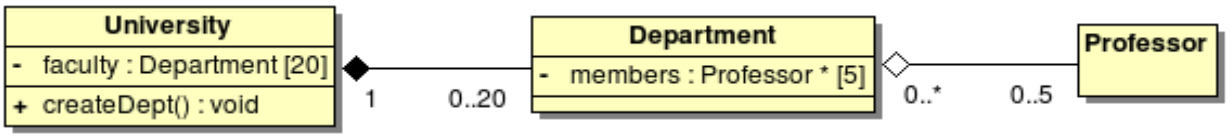


Рис. 4.9. Приклад агрегації та композиції

```
class Professor;
class Department
{
  ...
  private:
       Professor* members[5]; // Звичайна агрегація – у складі кафедр може
                              //  і не бути професорів.
                              //  Об’єкт типу Professor створюється окремо.
  ...
};
class University
{
  ...
  private:
            Department faculty[20];  // Композитна агрегація – університет
                                     //      не може існувати без кафедр.
                                     // Це окрема одиниця університету, яка з ним
                                     // жорстко пов’язана.
```

```
  ...
  public:
  void University( )
  {
   ....
     // Composition
     faculty[0] = createDepartment(....);
     faculty[1] = createDepartment(....);
   ....
  }
};
```

Інший приклад представлений на рис. 4.10. В автомобілі (клас Car) може бути чи ні (якщо він не створений) карбюратор (об'єкт класу Carburetor). У ставку (клас Pond) може бути безліч качок. Авто не може працювати без карбюратору. Ставок може існувати й без качок.

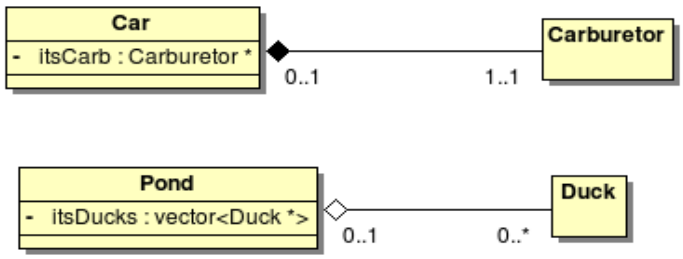


Рис. 4.10. Приклад агрегації та композиції

Приклад коду на C++:

```
// Composition
class Car
{
  private:
    Carburetor itsCarb;
  ...
};
// Aggregation
class Pond
{
  private:
    vector<Duck*> itsDucks;
  ...
};
```

Приклади з життя: комп’ютер може працювати без клавіатури (агрегація), клавіатура не може працювати без клавіш (композитна агрегація).

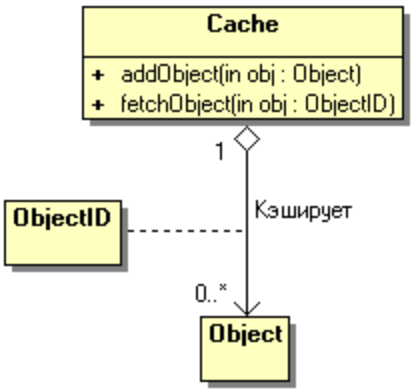


Рис. 4.11. Клас асоціації

Деякі асоціації є непрямими. Замість прямого зв'язування класів один з одним, вони зв'язуються побічно через третій клас. Наприклад, на рис. 4.11 асоціація показує, що екземпляри класу Cache посилаються на екземпляри класу Object через екземпляр класу ObjectID.

Асоціація між класами або інтерфейсами передбачає наявність залежності, яка містить об'єктне посилання, що пов'язує два об'єкти. Мається на увазі, що один клас або інтерфейс містить в собі посилання на інший клас або інтерфейс, і це взаємодія відображається на діаграмі. Можливі також залежності інших видів. Для вказівки залежності більш загального вигляду, використовується пунктирна лінія. Приклад подібної залежності представлений на рис. 4.12.

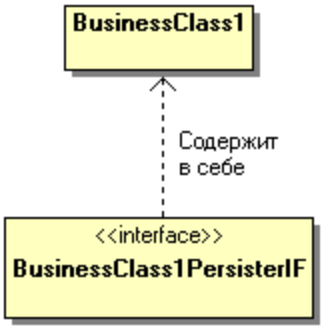


Рис. 4.12. Залежність

Класи на діаграмі можуть бути організовані у пакети. Пакет зображується як великий прямокутник з маленьким прямокутником нагорі, в якому вказується ім'я пакета (рис. 4.13).

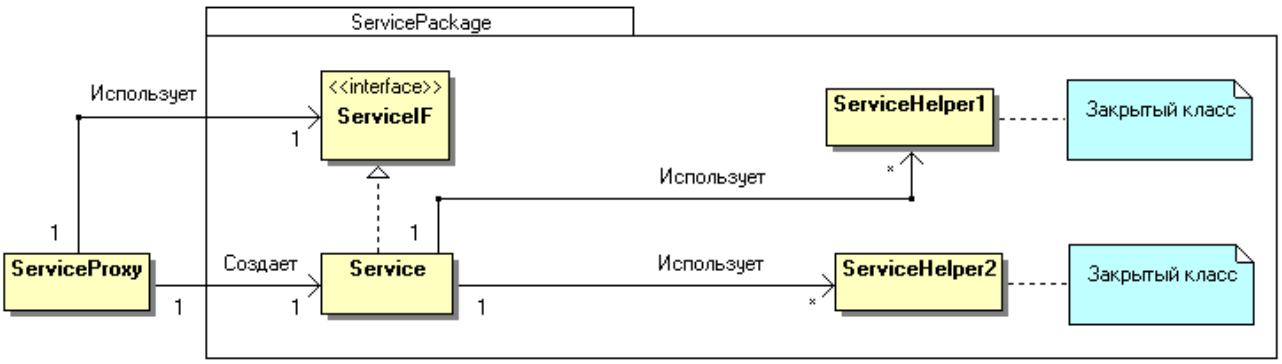


Рис. 4.13. Пакет

На рис. 4.13 показаний пакет з ім'ям ServicePackage. Перед ім'ям класу або інтерфейсу, що знаходиться всередині пакету, може вказуватися індикатор видимості. Відкриті класи доступні для класів за межами пакета, закриті – ні.

Іноді при проектуванні виявляються обставини, незрозумілі без коментаря на діаграмі. В UML коментар зображується у вигляді прямокутника з загнутим правим верхнім кутом і приєднується до того елемента діаграми, до якого вони належать.

Послідовність створення діаграми класів та генерації коду в середовищі BOUML (вер. 4.18.2). Зауважимо, що на момент 2020 року поточною версією є 7.10 (від 18.09.2020).

1. Створити каталог Projects / uml.
2. Запустити Bouml (джерело: https://www.bouml.fr/).
3. Project / New.
4. У каталозі Projects / uml створити проект, наприклад, test. Натиснути OK в діалозі інформації.
5. Вибрати Languages / C ++ ... та Languages / Verbose code generation.
6. У вікні browser в контекстному меню test вибрати New class view.
7. У вікні ввести ім'я виду діаграми класів – ClassView1.
8. У контекстному меню ClassView1 вибрати New class diagram.
9. У вікні імені діаграми класів ввести ClassDiagram1.
10. Двічі клацнути по пункту ClassDiagram1.
11. У вікні діаграми натиснути кнопку Add Class, перемістити курсор в поле малювання діаграми і натиснути ліву кнопку миші. У діалог ввести ім'я класу, наприклад, A.
12. У контекстному меню діаграми класу A вибрати Add operation(s). У вікні ввести ім'я операції (методу класу) – наприклад, print.
13. У діалозі операції вказати в поле value type тип: void. Натиснути на Enter.
14. Перейти в каталог Projects / uml / test і створити каталог code.
15. Виконати Project / Edit / Edit drawing settings.
16. Установити групу опцій в закладці class від show class members full definition до show attribute modifiers в значення yes і натиснути на OK.
17. Виконати Project / Edit / Edit generation settings.
18. Перейти до закладки Directory та навпроти поля C ++ root dir натиснути на кнопку Set it absolute. В кінці згенерованого шляху дописати ім'я каталогу code та натиснути на Enter.
19. Виконати Project / Save.
20. У контекстному меню test вибрати пункт New deployment view та в діалозі ввести ім'я DeploymentView1.
21. У контекстному меню DeploymentView1 вибрати New artifact та в діалозі ввести ім'я артефакту: A.

22. Двічі клацнути по артефакту A та в поле stereotype вибрати source й натиснути OK.

23. Двічі клацнути по класу A та в поле artifact вказати A.

24. Перейти до закладки C++ і, в разі відсутності прикладу згенерованого коду, натиснути на кнопку Default declaration й натиснути на OK.

25. Перейти до властивостей артефакту A та переконатися, що в закладках C++ header і C ++ source є приклад згенерованого коду. Якщо він відсутній, то необхідно скористатися кнопками Default definition в зазначених закладках та натиснути на OK.

26. Виконати Tools / Generate C++.

27. У разі правильності виконаних дій, в каталозі Projects / uml / test / code будуть згенеровані два файли: A.h та A.cpp, що представляють інтерфейсну частину (h-файл) і частина реалізації (cpp-файл) класу A.

Лекція 5. UML. Діаграма прецедентів

Будь-які (в тому числі і програмні) системи проектуються з урахуванням того, що в процесі своєї роботи вони будуть використовуватися людьми та/або взаємодіяти з іншими системами. Сутності, з якими взаємодіє система в процесі своєї роботи, називаються дійовими особами або акторами (actor) причому кожен актор очікує, що система буде вести себе строго певним, передбачуваним чином.

Actor – це безліч логічно пов'язаних ролей, виконуваних при взаємодії з прецедентами або сутностями (система, підсистема або клас). Актором може бути людина або інша система, підсистема або клас, які представляють щось поза сутності.

На рис. 5.1 представлено зображення актора засобами UML.

Рис. 5.1. Приклади зображення актора в UML

Актори не є частиною системи – вони являють собою кого-то або щось, що має взаємодіяти з системою. Актори можуть:

– тільки постачати ІС;

– тільки отримувати інформацію з ІС;

– забезпечувати інформацією та отримувати інформацію з системи.

Зазвичай актори визначаються з опису завдання або шляхом переговорів з замовниками та експертами. Для виявлення акторів може бути використана наступна група питань:

1. Хто зацікавлений в певній системній вимозі?
2. Яку роль система буде виконувати в організації?
3. Хто отримає переваги від використання системи?
4. Хто буде забезпечувати систему інформацією, використовувати інформацію та отримувати інформацію від системи?
5. Хто буде здійснювати підтримку та обслуговування системи?
6. Чи використовує система зовнішні ресурси?
7. Чи виступає будь-який учасник системи в декількох ролях?
8. Чи виступають різні учасники в одній ролі?
9. Чи буде нова система взаємодіяти зі старою?

Прецедент (use-case) – опис окремого аспекту поведінки системи з точки зору користувача (по Бучу).

Прецедент (use case) – опис безлічі послідовних подій (включаючи варіанти), що виконуються системою, які призводять результату, який бачить

актор. Прецедент являє поведінку сутності, описуючи взаємодію між акторами та системою. Прецедент не показує, "як" досягається певний результат, а тільки "що" саме виконується.

Прецеденти зображуються у вигляді еліпса, всередині якого або під ним вказується його назва. Прецеденти та актори з'єднуються за допомогою суцільних ліній. Часто на одному з кінців лінії зображують стрілку, причому спрямована вона до того, у кого запитують сервіс (чиїми послугами користуються). Приклад діаграми прецедентів представлений на рис. 5.2.

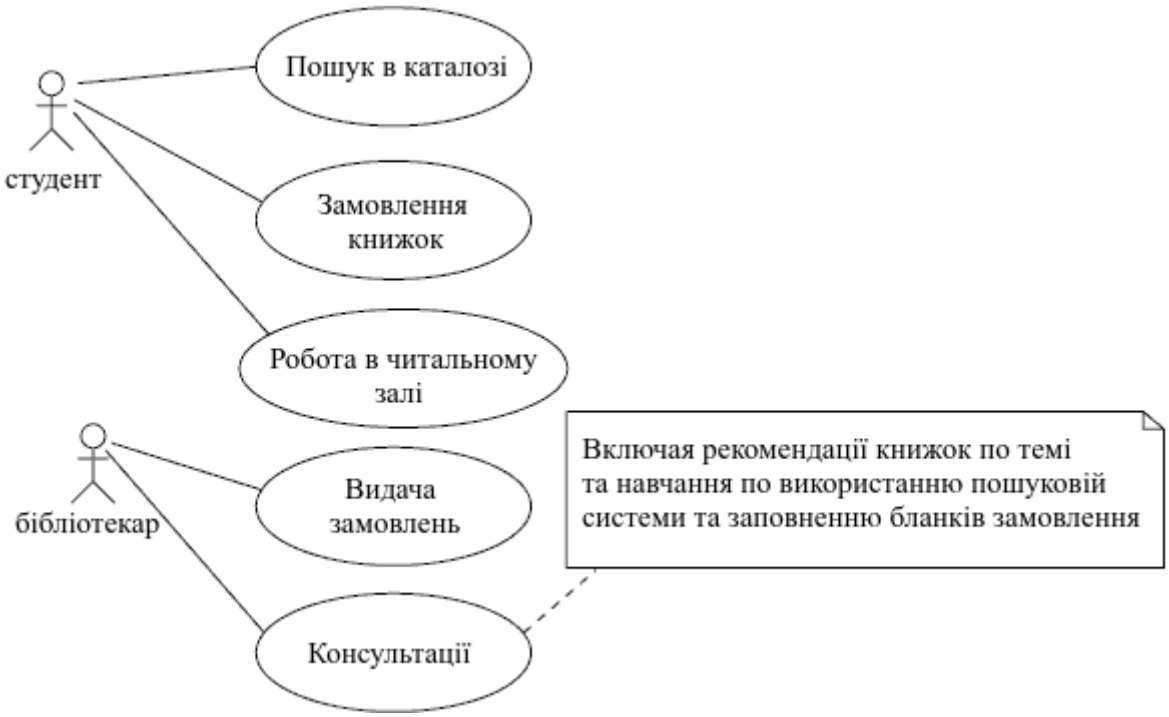


Рис. 5.2. Актори та прецеденти

Діаграма прецедентів (use case diagram) – це графічне представлення всіх або частини акторів, прецедентів та їх взаємодій в системі. У кожній системі зазвичай є головна діаграма прецедентів, яка відображає межі системи (акторів) та основну функціональну поведінку системи (прецедентів). Інші діаграми прецедентів можуть створюватися при необхідності.

Для створення потрібних відносин між прецедентами використовуються стереотипи, наприклад: <<include>>, <<includes>> (включати, включає), <<extend>>, <<extends>> (розширювати, розширює), <<uses>> (використовує), <<communicate>> (повідомляти) та ін.

Ставлення <<include>> зображується як відношення залежності, спрямоване від базового прецеденту до то того, яке використовується. Таке ставлення створюється, коли один з прецедентів використовує інший. Наприклад, кожен прецедент в системі реєстрації починається з аутентифікації користувача. Такі дії можна об'єднати в один прецедент, що застосується

іншими користувачами. Ставлення <<extend>> застосовується для відображення:

– додаткових режимів;

– режимів, які запускаються тільки за певних умов, наприклад, сигнал тривоги;

– альтернативних потоків, які запускаються на вибір актора. Таке ставлення зображується як відношення залежності, спрямоване від додаткового прецеденту до базового. Описані два види відносин зображуються пунктирною лінією зі стрілкою.

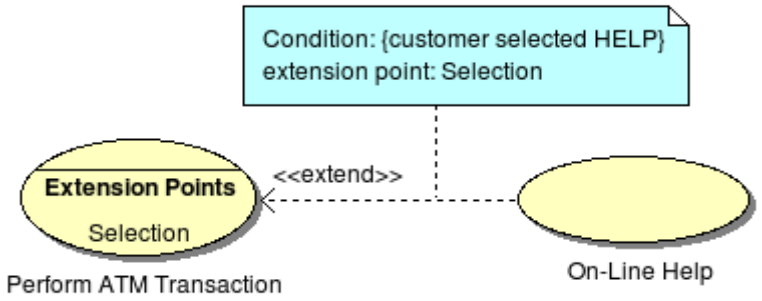
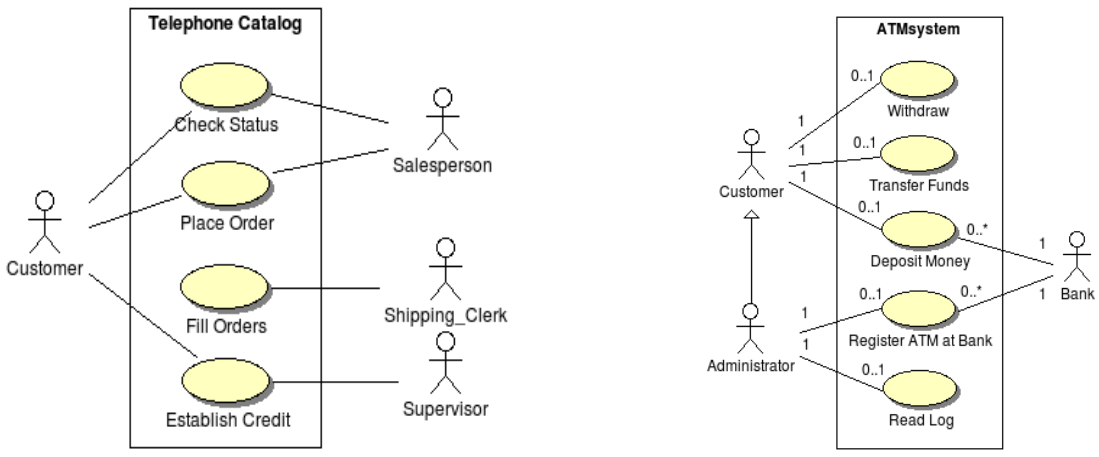


Рис. 5.3. Приклад відношення <<extend>> між прецедентами (розширення з умовою)

На рис. 5.3 прецедент Perform ATM Transaction (виконання транзакції через банкомат) має точку розширення Selection. Цей прецедент пропонує клієнту використовувати опцію банкомату Selection для вибору пункту HELP та використання таким чином прецеденту On-Line Help. Прецедент Perform ATM Transaction визначається незалежно від прецеденту On-Line Help.

Часто на діаграмах прецедентів кордони системи позначають прямокутником, у верхній частині якого може бути зазначена назва системи. На рис. 5.4 та 5.5 представлені приклади діаграм прецедентів.



а) б)

Рис. 5.4. Приклади діаграм прецедентів:
а) система телефонного каталогу; б) система банкоматів

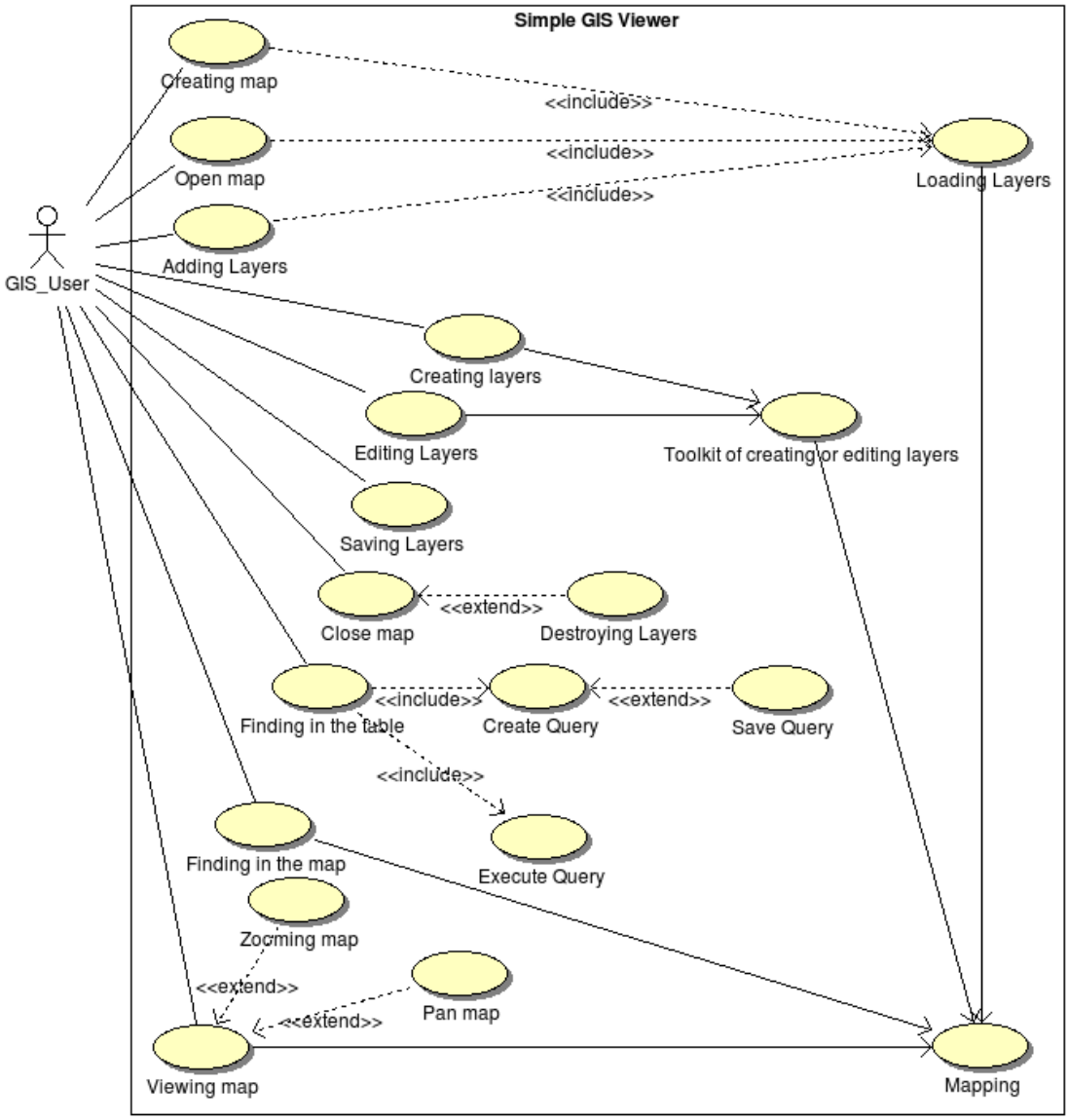


Рис. 5.5. Діаграма прецедентів простішого візуалізатору геопросторових даних

Таким чином, діаграми прецедентів відносяться до тієї групи діаграм, які представляють динамічні або поведінкові аспекти системи та виступають як засіб для досягнення взаєморозуміння між розробниками, експертами та кінцевими користувачами продукту.

Виділяють наступні цілі створення діаграм прецедентів:

1. Визначення кордону та контексту предметної області, що моделюється на ранніх етапах проектування.
2. Формування загальних вимог до поведінки проектованої системи.
3. Розробка концептуальної моделі системи для її подальшої деталізації.
4. Підготовка документації для взаємодії з замовниками та користувачами системи.

Лекція 6. UML. Діаграма компонентів

Діаграма компонентів служить частиною фізичного представлення моделі системи та грає важливу роль в процесі об'єктно-орієнтованого аналізу і проектування, є необхідною для генерації програмного коду. Вона показує різні компоненти системи і залежності між ними.

Компонент являє собою фізичний модуль програмного коду. Компонент часто вважають синонімом пакету, але ці поняття можуть відрізнятися, оскільки компоненти являють собою фізичне об'єднання програмного коду. Хоча окремий клас може бути представлений в цілій сукупності компонентів, цей клас повинен бути визначений тільки в одному певному пакеті. Наприклад, клас String в мові Java є частиною пакета java.lang, але він може бути виявлений в ряді компонентів.

Компонент є фізичною частиною системи, який може бути замінений та має один набор інтерфейсів і забезпечує реалізацію будь-якого іншого компоненту. Приклади зображення компонента представлені на рис. 6.1.

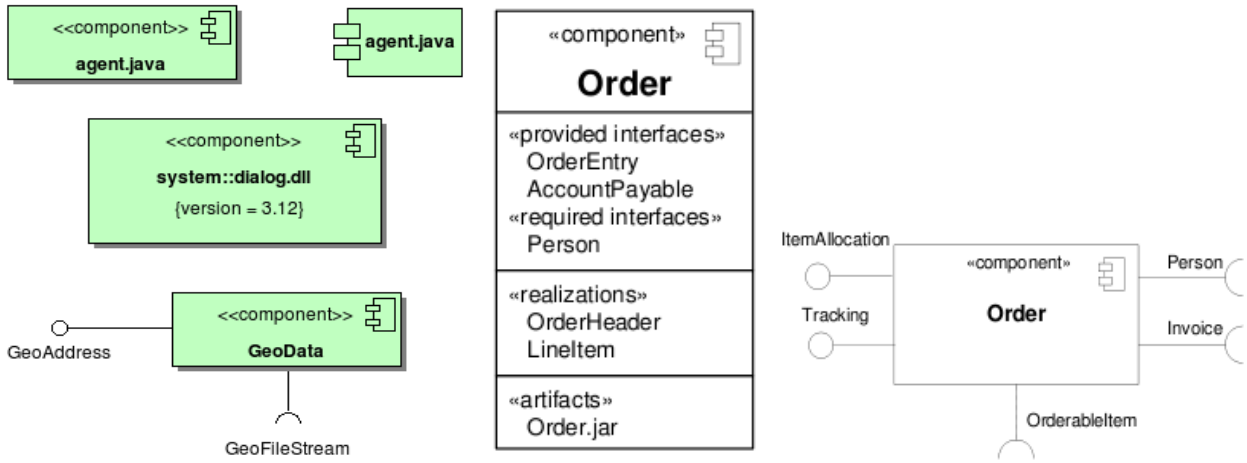


Рис. 6.1. Приклади зображення компонентів

Залежності між компонентами показують, як зміни одного компонента можуть вплинути на зміни інших компонентів. Кількість видів залежностей обмежена.

У CASE-засобах діаграми компонентів, як правило, представлені в Component View.

Часто компоненти містять або використовують інтерфейси. Інтерфейс – це набір операцій, які описують послуги, що надаються класом або компонентом. Істотним є ставлення між компонентом та інтерфейсом. Всі популярні компонентні засоби операційних систем (такі як COM+, CORBA та Enterprise JavaBeans) використовують інтерфейси для "склеювання" різних компонентів.

Відносини між компонентом та його інтерфейсами можна зобразити двома способами. Перший, найбільш поширений, полягає в тому, що інтерфейс малюється в згорнутій (elided) формі. Компонент, який реалізує інтерфейс, приєднується до нього за допомогою відносини згорнутої реалізації. У другому випадку інтерфейс малюється в розгорнутому вигляді, можливо з розкриттям операцій. Компонент, який реалізує його, приєднується за допомогою відносини повної реалізації. В обох випадках компонент, який одержує доступ до послуг інших компонентів через цей інтерфейс, підключається до нього за допомогою відносини залежності.

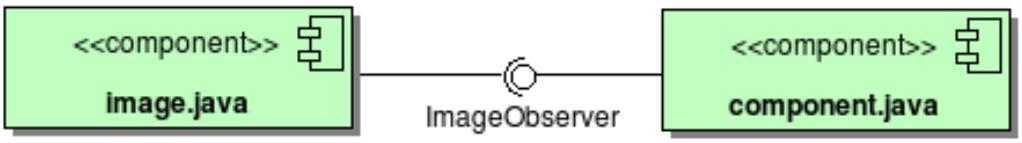


Рис. 6.2. Компоненти та інтерфейси

Інтерфейс, що реалізується компонентом, називається інтерфейсом, що експортується (export interface, component implements interface). Це означає, що компонент через даний інтерфейс надає ряд послуг для інших компонентів. Компонент може експортувати багато інтерфейсів.

Інтерфейс, яким компонент користується, називається інтерфейсом, що імпортується (import interface, component uses interface). Це означає, що компонент сумісний з таким інтерфейсом та залежить від нього при виконанні своїх функцій. Компонент може імпортувати різні інтерфейси, причому йому дозволяється одночасно експортувати та імпортувати різні інтерфейси (рис. 6.1).

На рис. 6.2 компонент, який реалізується в component.java, надає інтерфейс ImageObserver, який використовується компонентом з image.java.

Компоненти можна розділити на три види.

1. Компоненти розгортання (deployment components), які необхідні і достатні для побудови системи, яка виконується. До їх числа відносяться спільні бібліотеки (DLL) та виконувані програми (EXE). Визначення компонентів в UML досить широко, щоб охопити як класичні об'єктні моделі (на зразок COM+, CORBA та Enterprise JavaBeans), так і альтернативні, що можливо містять динамічні Web-сторінки, таблиці БД та модулі для виконання, де використовуються закриті механізми комунікації.

2. Компоненти-робочі продукти (work product components). По суті, це побічний результат процесу розробки. Сюди можна віднести файли з вихідними текстами програм і даними, з яких створюються компоненти розгортання. Такі компоненти не беруть безпосередньої участі в роботі системи, але є робочими продуктами, з яких система створюється.

3. Компоненти виконання (execution components). Вони створюються як наслідок роботи системи. Прикладом може служити об'єкт COM+, екземпляр якого створюється з DLL.

Компоненти, як і класи, можуть бути згруповані в пакети. При організації компонентів між ними можна уточнювати відносини залежності, узагальнення, асоціації (включаючи агрегування) та реалізації.

Стосовно до компонентів, в UML існують такі типи стереотипів:

– executable (для виконання) – визначає компонент, який може виконуватися у вузлі;

– library (бібліотека) – визначає статичну або динамічну бібліотеку;

– table (таблиця) – визначає компонент, який представляє таблицю БД;

– file (файл) – визначає компонент, який представляє документ, що містить вихідний текст або дані;

– document (документ) – визначає компонент, який представляє документ.

З виходом нових версій UML, можливе розширення цього списку.

На рис. 6.3 показаний набір компонентів, що входить до складу інструментальної програми, яка працює на одному ПК: програма та чотири бібліотеки. На діаграмі представлені також залежності між компонентами.

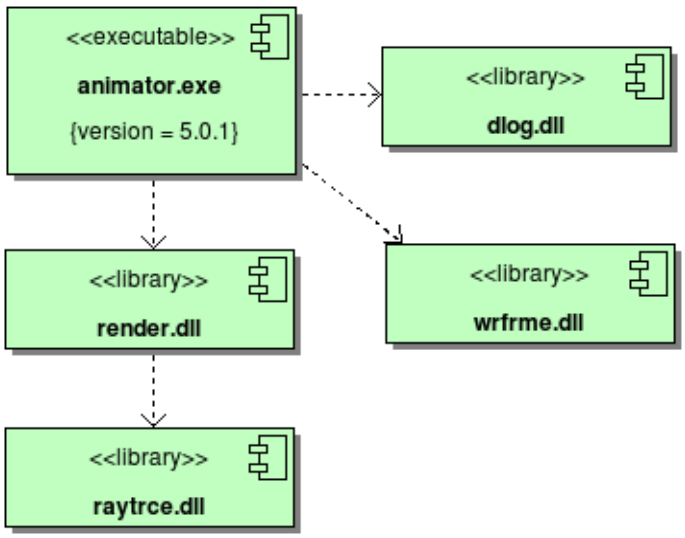


Рис. 6.3. Діаграма компонентів програми з бібліотеками

Безпосередній показ залежності між двома компонентами – це насправді згорнута форма подання реальних відносин між ними. Компонент рідко залежить від іншого компоненту безпосередньо. Частіше він імпортує один або декілька інтерфейсів, що експортуються іншим компонентом. Можна було б, наприклад, змінити представлену на рис. 6.3 діаграму, явно вказавши інтерфейси, які реалізує (експортує) бібліотека render.dll та використовує (імпортує) програма animator.exe. Однак, для простоти ці деталі можна приховати, показавши лише, що існує залежність між компонентами.

Можливий розширений показ компонента з елементами діаграми класів (рис. 6.4).

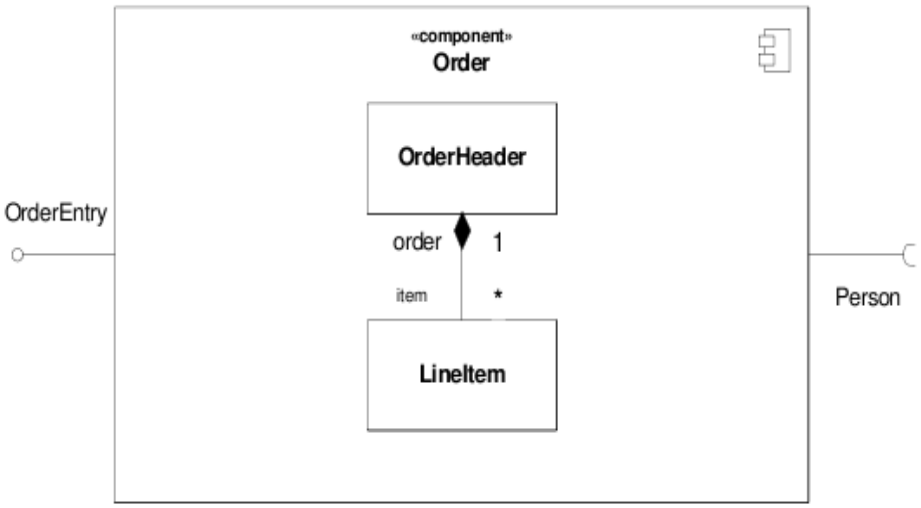


Рис. 6.4. Розгорнуте представлення компонента

При зображенні компонента в UML рекомендується керуватися наступними правилами:

1. Застосовувати згорнуту форму інтерфейсу, якщо тільки не виникає гострої необхідності розкрити операції, які пропоновані цим інтерфейсом.
2. Показувати тільки ті інтерфейси, які необхідні для розуміння призначення компонента в даному контексті.
3. У тих випадках, коли використовуються компоненти для моделювання бібліотек і вихідного коду, вказувати помічені значення, які стосуються контролю версій.

Лекція 7. UML. Діаграма розгортання

Діаграма розгортання показує, яким чином ПЗ розгортається на обчислювальні елементи. Для розробки діаграм розгортання в CASE-засобах зазвичай призначене спеціальне подання Deployment View. На рис. 7.1 представлений приклад діаграми розгортання – зображені два обчислювальних елементи, помічені як Server1 та Server2.

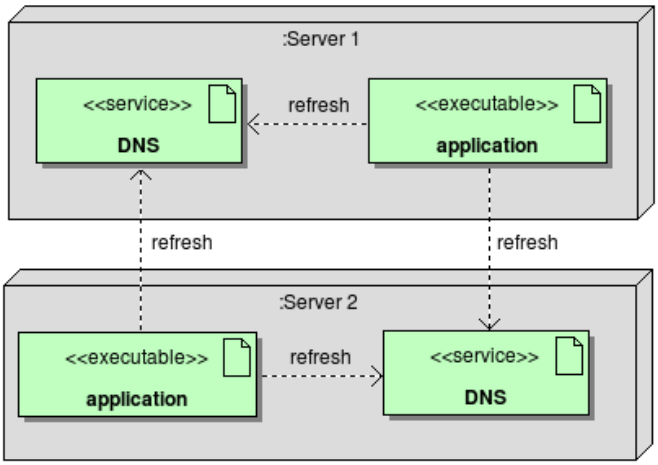


Рис. 7.1. Приклад діаграми розгортання

Відповідно до термінології UML, обчислювальний елемент – це вузол. Прямокутники всередині обчислювальних елементів являють собою компоненти ПЗ. Компонент – це колекція об'єктів, які розгортаються всі разом. На цій діаграмі кожен обчислювальний елемент має два компоненти: компонент DNS та компонент application.

Зв'язки між компонентами позначені пунктирними лініями. Як показано на рис. 7.1, пунктирні лінії, помічені словом refresh, свідчать про те, що компоненти application відправляють повідомлення на розв'язання мережевих імен компонентам DNS.

На рис. 7.2 показані комунікаційні зв'язки між двома типами вузлів з розгорнутими артефактами.

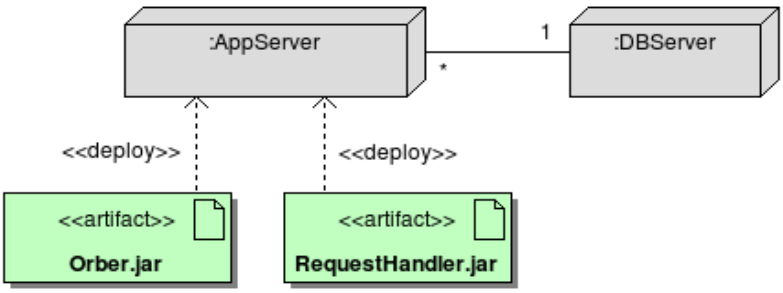


Рис. 7.2. Приклад діаграми розгортання з використанням артифактів

На рис. 7.3 представлені вихідні модулі, які складають графічне ядро ГІС, а на рис. 7.4 – спрощена діаграма розгортання клієнтського місця на базі ArcGIS.

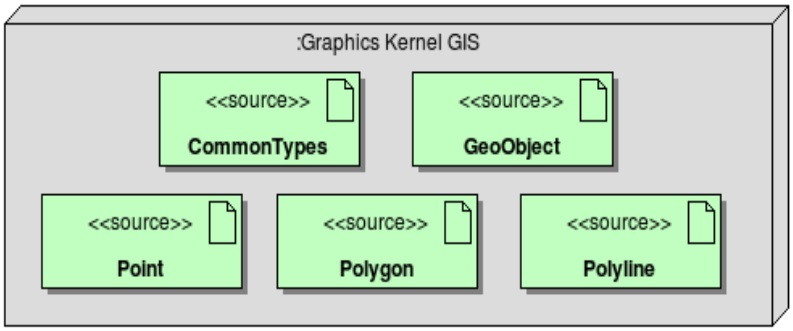


Рис. 7.3. Елементи графічного ядра ГІС

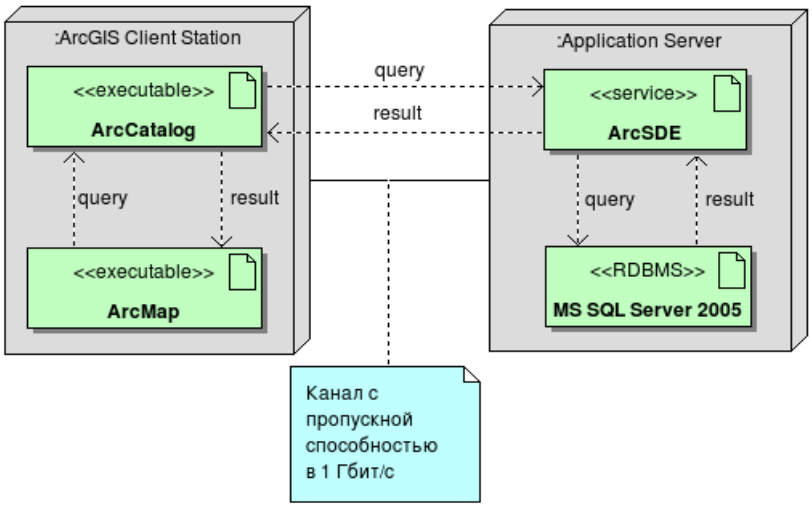


Рис. 7.4. Спрощена діаграма розгортання клієнтського місця ArcGIS

В якості каналу зв'язку (рис. 7.4) між вузлами виступає фізичне з'єднання (наприклад, кручена пара провідників), або може бути також і посилання на сайт в Internet, супутниковий зв'язок та т.ін.

З'єднання показуються між вузлами у вигляді асоціації та зображуються без стрілок. Наявність такої лінії вказує на необхідність організації фізичного каналу для обміну інформацією між відповідними вузлами.

Інший приклад діаграми розгортання мережі міських банкоматів (АТМ), представлений на рис. 7.5.

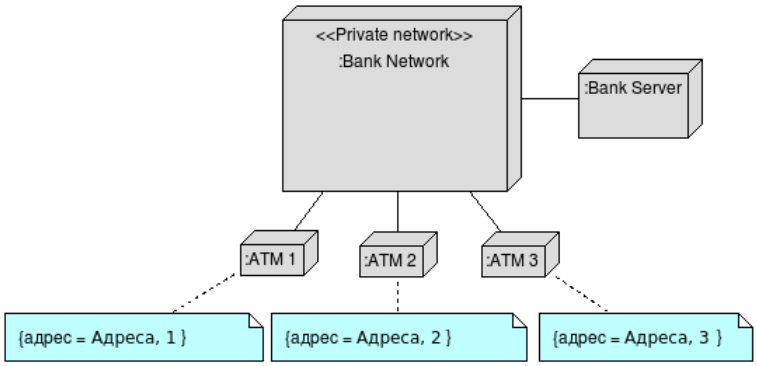


Рис. 7.5. Діаграма розгортання моделі банкоматів

При розробці діаграми розгортання слід дотримуватися наступних правил та рекомендацій.

1. Перед розробкою діаграми необхідно ідентифікувати:

– категорії (типи) користувачів (для кожної категорії повинні бути визначені кількість користувачів та необхідні для роботи компоненти системи);

– апаратні, технічні та інші типи пристроїв, необхідні для виконання системою своїх функцій;

– види і необхідну пропускну здатність каналів зв'язку.

2. Повинні бути розглянуті варіанти прокладки нової або модернізації існуючої корпоративної мережі організації.

3. З метою наочного уявлення розподіленої інформаційної системи на діаграмі рекомендується відображати компоненти, інтерфейси та зв'язку між ними.

Таким чином, діаграма розгортання, як і діаграма компонентів, є складовою частиною фізичного представлення моделі і розробляється, як правило, для територіально розподілених систем. Вона відображає фізичні взаємозв'язку між програмними та апаратними компонентами системи, що розробляється і є хорошим засобом представлення маршрутів переміщення об'єктів і компонентів в розподіленої системі.

Хоча діаграми розгортання та діаграми компонентів можна зображувати окремо, також допускається поміщати діаграму компонентів в діаграму розгортання. Це доцільно робити, щоб показати які компоненти виконуються і на яких вузлах.

Розробники часто зображують на фізичних діаграмах спеціальні символи, які за своїм зовнішнім виглядом нагадують різні елементи. Наприклад, застосовуються спеціальні піктограми для серверів, комп'ютерів та баз даних. В рамках мови UML це цілком допускається: кожну з піктограм можна розглядати як стереотип відповідного елемента діаграми. Зазвичай такі піктограми сприяють кращому розумінню діаграми, хоча і ускладнюють її, якщо одночасно зображуються вузли та компоненти.

Лекція 8. Java. Компоненти бібліотеки Swing. Загальні відомості

Swing являє собою набір класів, що застосовуються для створення графічних інтерфейсів користувача (Graphical User Interface – GUI) сучасних програм, в тому числі Web-програм. Swing представляє собою набір візуальних компонентів, наприклад, кнопок, полів редагування, смуг прокрутки, прапорців опцій та таблиць, розроблених так, щоб їх можна було успішно застосовувати в самих різних додатках.

Першою віконною підсистемою Java була AWT (Abstract Window Toolkit). В AWT було визначено базовий набір керуючих елементів та вікон, що дозволяє створювати графічні інтерфейси з обмеженими можливостями. Одним з обмежень була платформно-орієнтована підтримка візуальних компонентів. В результаті зовнішній вигляд інтерфейсних елементів визначався не засобами Java, а платформою, яка використовується.

Платформно-орієнтована підтримка інтерфейсних елементів стала джерелом низки проблем. Обмеження AWT були занадто серйозні і був потрібний новий підхід. В результаті в 1997 році з'явився набір Swing. Він був включений до складу JFC (Java Foundation Classes).

Незважаючи на те, що Swing усуває ряд обмежень, властивих AWT, він не замінює даний інструмент. Більш того, в основу Swing покладені деякі основні рішення, прийняті в AWT. Зокрема, в Swing використовується такий же механізм обробки подій, як і в AWT.

Компоненти Swing побудовані згідно модифікованої архітектури «модель-уявлення-контролер» (MVC: Model-View-Controller).

Перевага даної архітектури полягає в тому, що кожна її складова в точності відповідає одній з характеристик компонента. Модель задає стан компонента. Уявлення визначає, як компонент буде відображатися на екрані і як буде представлено поточний стан моделі. Контролер забезпечує реакцію компонента на дії користувача.

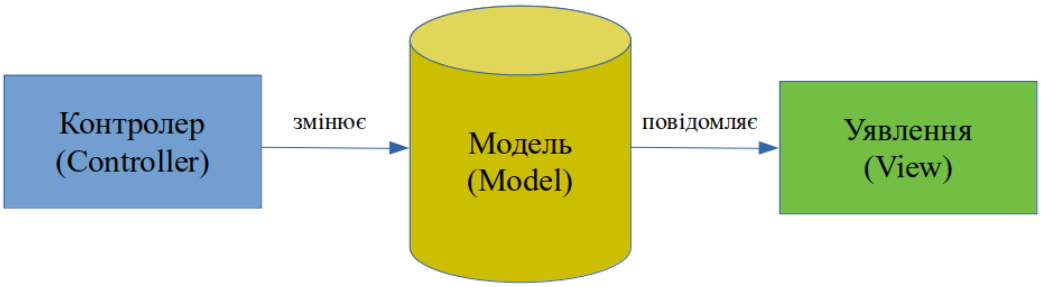


Рис. 8.1. Архітектура MVC

Модифікація MVC в Swing полягає в тому, що уявлення (вид) та

контролер об'єднані в єдиний елемент, що називається представником інтерфейсу користувача (делегатом). Такий підхід відомий, як архітектура «модель-представник» (делегат-модель) або архітектура з виділеної моделлю.

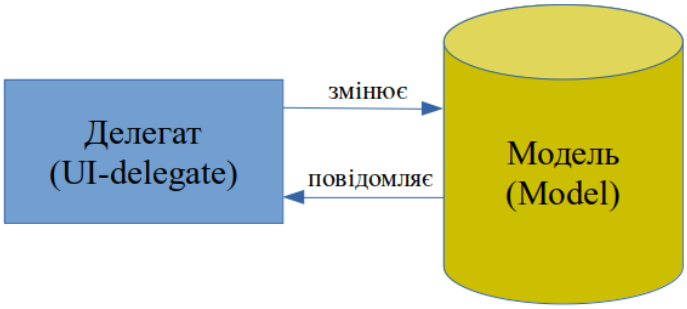


Рис. 8.2. Архітектура делегат-модель в компонентах Swing

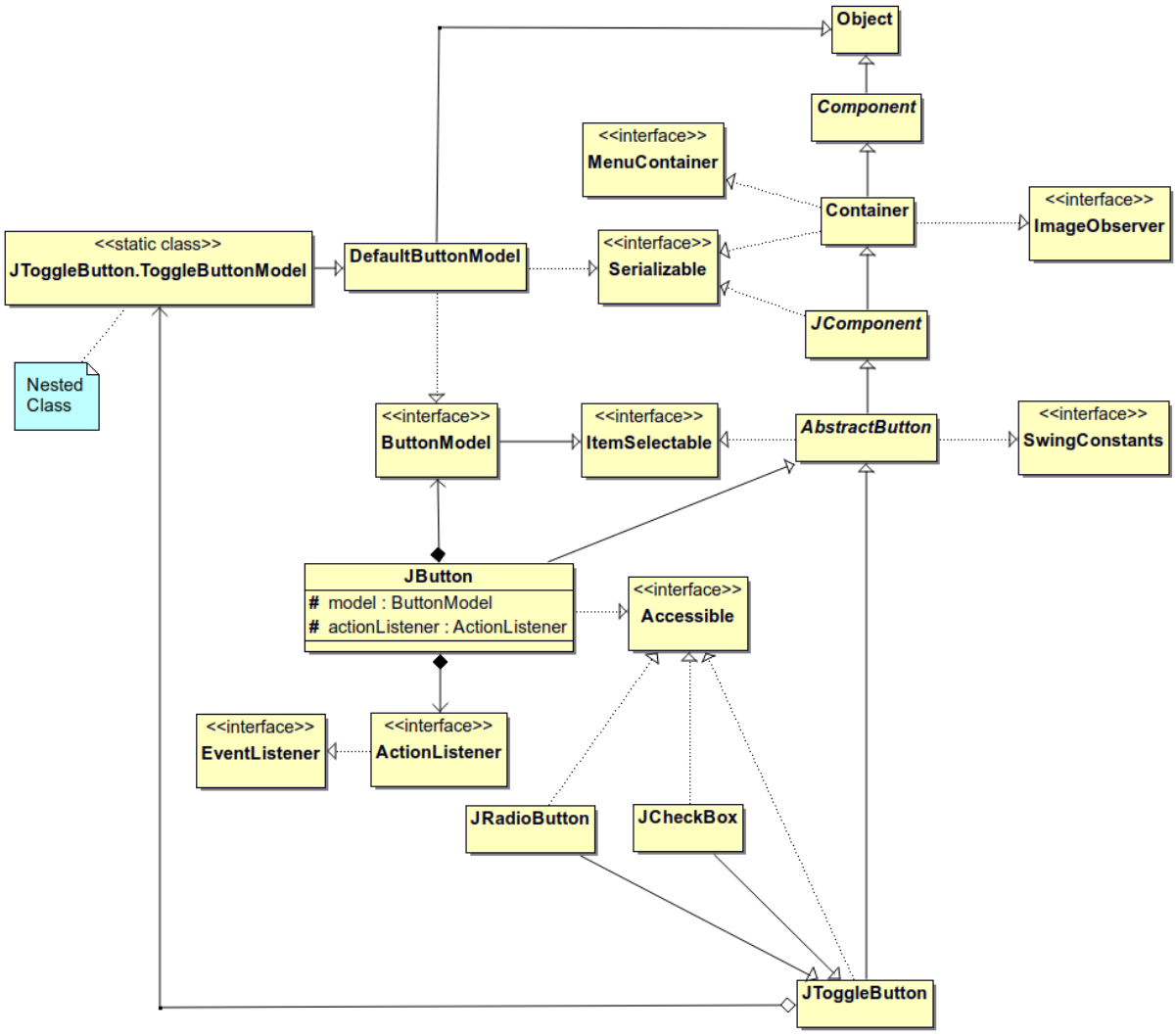


Рис. 8.3. Діаграма класів невеличкої частини бібліотеки Swing

Моделі визначаються за допомогою інтерфейсів. Наприклад, модель для кнопки визначена за допомогою інтерфейсу *ButtonModel* (рис. 8.3). Клас *JButton* є делегатом. Інтерфейс *ButtonModel* містить інформацію про стан, наприклад, чи була кнопка *JButton* натиснута, доступна, а також який список слухачів дій *ActionListener* містить. Об'єкт *JButton* забезпечує графічне представлення (наприклад, прямокутник на екрані з написом і рамкою) та модифікує стан компоненту.

Представниками інтерфейсу користувача є підкласи абстрактного класу *ComponentUI*. Наприклад, в ролі представника для кнопки виступає абстрактний клас *ButtonUI*., у якого є певні нащадки. Наприклад, клас *BasicButtonUI*.

До складу GUI, створеного засобами Swing, входять елементи двох типів: *компоненти та контейнери*. Такий поділ багато в чому умовний, оскільки контейнери є в той же час і компонентами. Різниця між ними в їх призначенні.

Компоненти – це незалежні елементи. Наприклад, до них можна віднести кнопки або лінійні регулятори.

Класи, що представляють всі компоненти Swing, містяться в пакеті *javax.swing*. Переважна більшість компонентів Swing створюється за допомогою класів-нащадків *JComponent*.

Контейнер може містити в собі кілька компонентів і являє собою спеціальний тип компонента. Щоб компонент відобразився на екрані, його треба помістити в контейнер. Таким чином, у складі GUI повинен бути присутнім хоча б один контейнер.

У Swing визначені два типи контейнерів. До першого типу відносяться контейнери верхнього рівня: *JFrame*, *JApplet*, *JWindow* та *JDialog*. Перераховані контейнери є «важкими». Контейнери другого типу – це легкі контейнери, які є нащадками *JComponent*, наприклад, *JPanel* і *JRootPane*. Вони можуть включатися в інші контейнери і часто використовуються для об'єднання групи пов'язаних один з одним компонентів.

Класи компонентів бібліотеки Swing:

JApplet, JColorChooser, JDialog, JFrame, JLayeredPane, JMenuItem,
JButton, JCheckBoxMenuItem, JComboBox, JDesktopPane, JCheckBox
JComponent, JEditorPane, JFileChooser, JInternalFrame, JLabel,
JMenu, JList, JOptionPane, JPanel, JFormattedTextField, JLayer,
JMenuBar, JPasswordField, JPopupMenu, JRootPane, JSlider,
JProgressBar, JScrollBar, JSpinner, JTextArea, JToolBar, JWindow,
JTable, JToggleButton, JViewport, JRadioButtonMenuItem, JSeparator,
JTabbedPane, JTextPane, JTree, JRadioButton, JScrollPane, JSplitPane,
JTextField, JToolTip.

У*правління компонуванням*

Компоненти розміщуються всередині фреймів за допомогою спеціального механізму компонування – всі компоненти контейнера управляються одним з менеджерів компонування (*layout manager*), наприклад, менеджером потокового компонування (*flow layout manager*), що застосовується до компоненту панелі за замовчуванням. Такий менеджер розміщує компоненти горизонтально, поки вистачає місця в рядку, а потім починає новий рядок. Якщо користувач автоматично змінив розміри контейнера, то менеджер компонування автоматично переміщує компоненти, заповнюючи вільний простір (рис. 8.4).

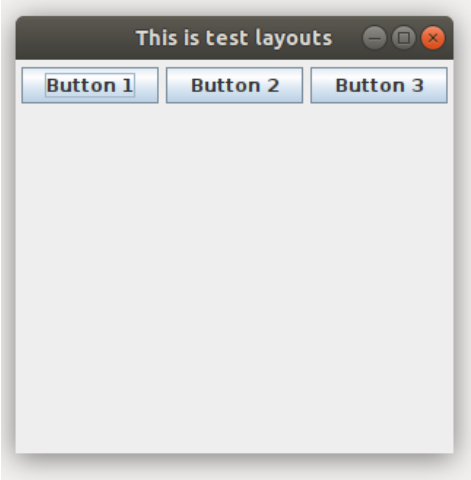


Рис. 8.4. Приклад розташування кнопок
при використанні компонування *flow layout*

Приклад:

```
import java.awt.*;
import javax.swing.*;

public class Main implements Runnable {

    public Main(String[] args)
    {
        panel = new JPanel();
        frm = new JFrame();
    }

    public Rectangle getCenterRect(int width, int height)
    {
        GraphicsEnvironment ge = GraphicsEnvironment.getLocalGraphicsEnvironment();
        Rectangle r = ge.getMaximumWindowBounds();
        return new Rectangle(r.width/2-width/2, r.height/2-height/2, width, height);
    }

    public void run() {
        panel.add(new JButton("Button 1"));
        panel.add(new JButton("Button 2"));
        panel.add(new JButton("Button 3"));

        frm.setTitle("This is test layouts");
        frm.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
        frm.setBounds( getCenterRect() );
```

```
        frm.add( panel );
        frm.setVisible( true );
    }

    public static void main(String[] args) {
        SwingUtilities.invokeLater( new Main(args) );
    }

    private JFrame frm;
    private JPanel panel;
}
```

Для запуску програми з елементами інтерфейсу Swing, основний клас повинен реалізовувати інтерфейс *java.lang.Runnable*. Для цього потрібно реалізувати в класі один метод *run( )*, в якому потрібно прописати всі основні дії по підготовці та відображенню вікна програми. Основний клас програми може бути також нащадком класу *java.lang.Thread*, якій також реалізовую інтерфейс *Runnable*. В цьому випадку потрібно перевантажити метод *run( )*:

```
public class Main extends Thread {
      . . .
      @Override
      public void run( ) {
            . . .
      }
      . . .
}
```

Запуск програм на базі Swing може бути виконаний різними способами. Один із розповсюджених прийомів – передача об’єкта класу з реалізацією інтерфейсу *Runnable* в статичний метод *javax.swing.SwingUtilities.invokeLater()*. Він призводить до асинхронного виконання програми в потоці диспетчеризації подій AWT.

В коді програми створюється об’єкт класу JFrame, в якому розміщується об’єкт панелі – JPanel. На панелі розміщуються три кнопки – об’єкти JButton. В цьому прикладі за кнопками не закріплено ніяких обробників подій. Про це буде йтись мова в наступній лекції. Вікну-фрейму задаються основні властивості. Відображається вікно встановленням властивості *visible* – *setVisible(true)*.

Для завершення роботи по кнопці закриття вікна, програма повинна за замовчанням передати значення *JFrame.EXIT_ON_CLOSE* в метод *JFrame.setDefaultCloseOperation( )*. Якщо у вікна буде свій обробник події закриття, то потрібно передавати значення *JFrame.DO_NOTHING_ON_CLOSE*.

Метод *getCenterRect( )* може бути відсутнім. На його прикладі в програмі показується механізм звернення до об’єкту *GraphicsEnvironment* з метою отримання розміру поточного екрану монітора. За допомогою цієї інформації можна визначити розміри прямокутника вікна програми, яке повинно розташуватися посередині екрану.

Для зміни вирівнювання компонентів необхідно створити об'єкт

FlowLayout із зазначенням необхідного вирівнювання. Наприклад:

```
panel = new JPanel( new FlowLayout(FlowLayout.LEFT) );
```

Менеджер компонування рамок (*border layout manager*) призначається за замовчуванням для кожної панелі вмісту в кожному об'єкті класу *JFrame*. На відміну від менеджера потокового компонування, він дозволяє обрати місце для кожного компоненту – компонент можна розмістити всередині панелі, по центру, нагорі, внизу, зліва та справа:

<table>
<tr><td colspan="3">Верхня частина (BorderLayout.NORTH)</td></tr>
<tr><td>Ліва частина
(BorderLayout.WEST)</td><td>Центральна частина
(BorderLayout.CENTER)</td><td>Права частина
(BorderLayout.EAST)</td></tr>
<tr><td colspan="3">Нижня частина (BorderLayout.SOUTH)</td></tr>
</table>

Приклад:

```
panel = new JPanel( new BorderLayout( ) );
…
panel.add( new JButton("Button 3"), BorderLayout.SOUTH );
```

Менеджер мережевого компонування (*grid layout manager*) розташовує всі компоненти по рядках та стовпцях, при цьому розмір усіх осередків однаковий. Бажана кількість рядків та стовпців вказується в конструкторі об'єкту мережевого компонування:

```
panel = new JPanel( new GridLayout(3, 3, 5, 5) );
…
panel.add(new JButton("Button 1"));
…
panel.add(new JButton("Button 6"));
```

Результат представлений на рис. 8.5. Два останніх параметра конструктора задають відстань між компонентами по горизонталі та вертикалі. На рис. 8.5 показано, як буде виглядати вікно при додаванні меншої та більшої ніж 9-ть (3x3) кількості кнопок. Компоненти додаються порядково.

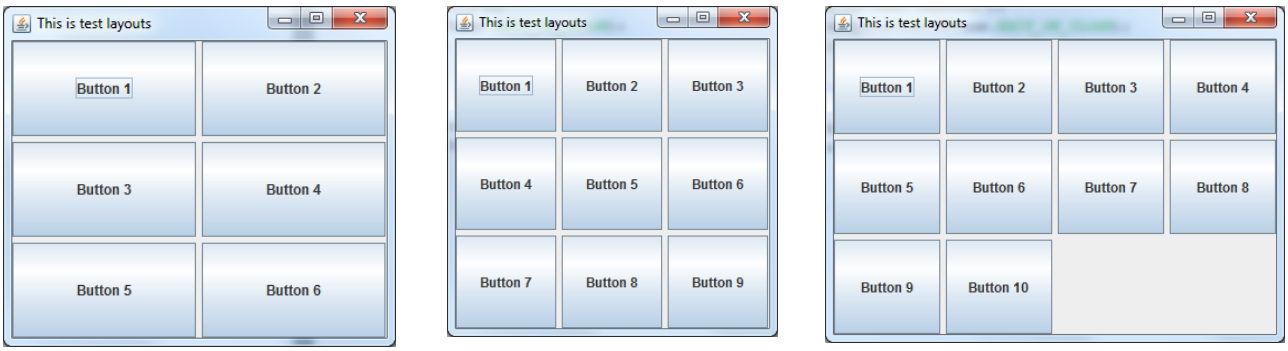


Рис. 8.5. Результат мережевого компонування

Блокове компонування дозволяє створити більш гнучкий рядок, що складається з компонентів. Це досягається за рахунок класу-контейнеру *javax.swing.Box*. Щоб створити новий контейнер, потрібно використовувати один з двох методів:

```
Box hbox = Box.createHorizontalBox( );
Box vbox = Box.createVerticalBox( );
```

Після цього додають компоненти в контейнер. Наприклад: hbox.add(btn1). За замовчуванням компоненти в блоці розташовуються впритул один до одного. Для того щоб розсунути компоненти, додаються невидимі наповнювачі (fillers). Існує три види наповнювачів:

– розпірки (struts);
– фіксовані області (rigid areas);
– склейки (glue).

Розпірка просто додає певний простір між компонентами. Наприклад:

```
hbox.add( Box.createHorizontalStrut( 10 ) );
```

Фіксована область схожа на пару розпірок. Вона також розсовує суміжні компоненти, але, крім того, додає мінімальне значення висоти або ширини в іншому напрямку. Наприклад, наступний оператор додає невидиму фіксовану область з заданими мінімальною та максимальною шириною, рівними 5 пікселям та висотою, що дорівнює 20 пікселям, а також задає вирівнювання по центру:

```
box.add( Box, createRigidArea( new Dimension(5, 20) ) );
```

Додавання склеювань призводить до того, що компоненти розсуваються на максимально можливу відстань. Приклад:

```
hbox.add( btn1);  hbox.add( Box.createGlue( ) );  hbox.add( btn2 );
```

У цьому прикладі, якщо блок не містить інших компонентів, кнопка btn1 весь час буде переміщатися вліво, а btn2 – вправо.

У наступному прикладі показано як буде зроблено розташування компонентів при блоковому компонуванні з рис. 8.6.

Рис. 8.6. Модель блокового компонування

```
. . .
private JFrame frm;
private JPanel panel;
private JTextField textA, textB;
private JButton btn1;
. . .
frm = new JFrame( );
panel = new JPanel( );
textA = new JTextField(10);
textB = new JTextField(10);
btn1 = new JButton( "Get Result A/B" );
. . .
textA.setText( "10" );
textB.setText( "5" );
textA.setToolTipText( "Input value" );
textB.setToolTipText( "Input value" );
. . .
Box hbox = Box.createHorizontalBox( );
hbox.add( new JLabel("A:") );  hbox.add( textA );
hbox.add( Box.createHorizontalStrut(10) );
hbox.add( new JLabel("B:") ); hbox.add( textB );
. . .
Box hbox2 = Box.createHorizontalBox( );
hbox2.add( btn1 );
. . .
Box vbox = Box.createVerticalBox( );
vbox.add( hbox );
vbox.add( Box.createVerticalStrut(10) );
vbox.add( hbox2 );
. . .
panel.add( vbox );
. . .
frm.add( panel );
frm.setResizable( false );
frm.setVisible( true );
. . .
```

Існують й інші менеджери компонування.

Використання менеджеру UI Swing

В Java Swing існує механізм L&F (look and feel), за допомогою якого можна змінювати зовнішній вигляд Swing-компонентів, їх стиль відображення (look) та поведінку (feel).

Найбільш відомими й присутніми в бібліотеці Swing є наступні стилі:

– Metal;
– Nimbus;
– Motif;
– Windows (відображення у стилі MS Windows для ОС Windows);
– GTK (відображення для Unix-сумісних при наявності GTK-підтримки).

Приклади вигляду вікон з різним L&F представлені на рис. 8.6.

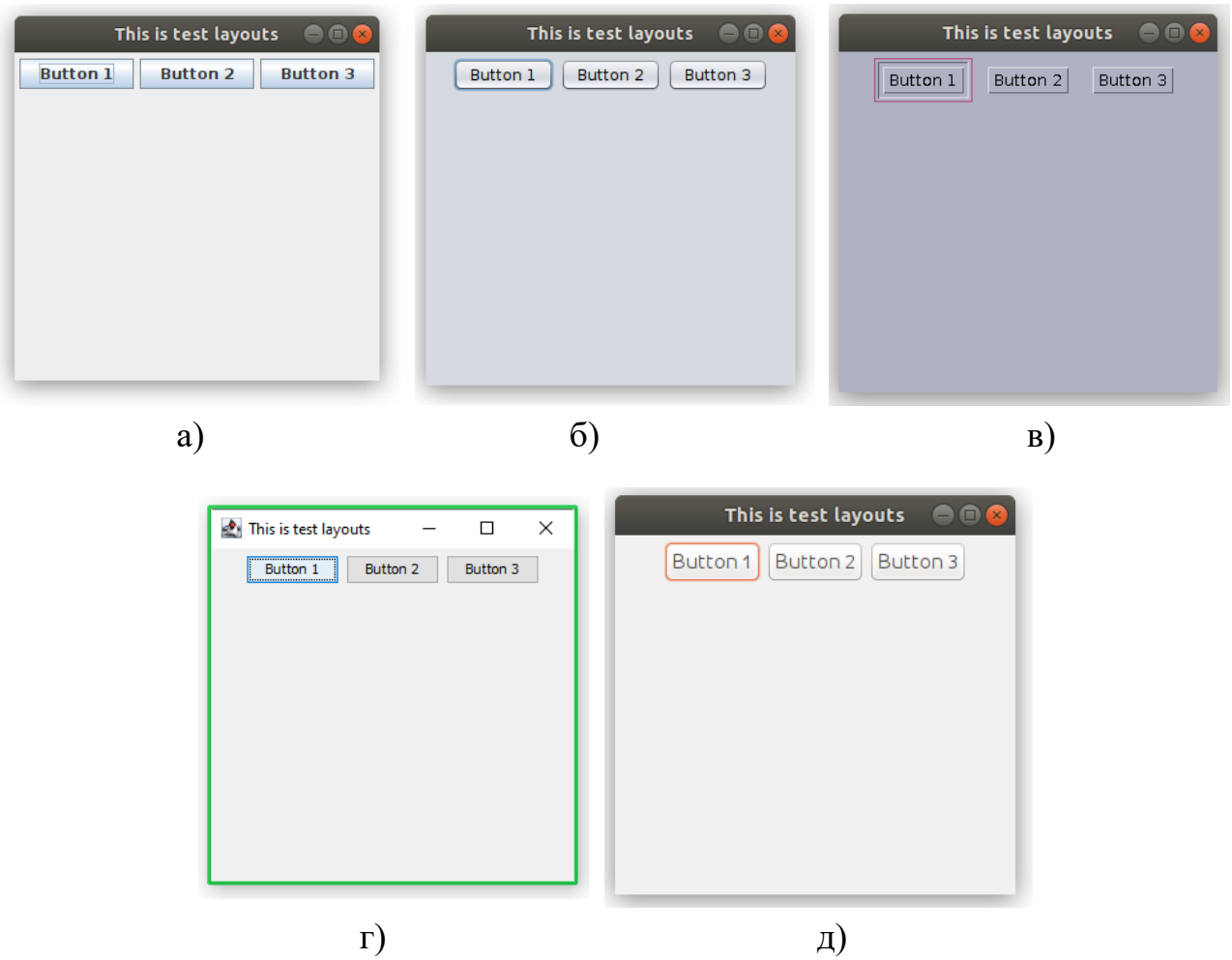


Рис. 8.6. UI Swing LookAndFeel:
а) Metal; б) Nimbus; в) Motif; г) Windows; д) GTK

Встановлення L&F контролюється класом *javax.swing.UIManager*.

Для використання певного стилю для коду вище, потрібно внести зміни в метод *main( )*.

Наприклад:

```
public static void main(String[] args)
throws ClassNotFoundException, InstantiationException,
IllegalAccessException, UnsupportedLookAndFeelException
{
         UIManager.setLookAndFeel("javax.swing.plaf.nimbus.NimbusLookAndFeel");
        SwingUtilities.invokeLater(new Main(args));
}
```

Для перелічених вище стилів потрібно вказати в методі *setLookAndFeel( )* відповідні рядки:

– javax.swing.plaf.metal.MetalLookAndFeel;

– javax.swing.plaf.nimbus.NimbusLookAndFeel;

– com.sun.java.swing.plaf.motif.MotifLookAndFeel;

– com.sun.java.swing.plaf.windows.WindowsLookAndFeel;

– com.sun.java.swing.plaf.gtk.GTKLookAndFeel.

На Linux-сумісних системах з GTK-інтерфейсом робочого оточення рекомендується встановити пакет canberra-gtk або подібний.

Наприклад (команди виконуються від root):

– для Ubuntu: apt install libcanberra-gtk-module

– для Fedora: dnf install libcanberra-gtk3

Лекція 9. Java. Компоненти бібліотеки Swing. Обробка подій

Керуючи елементи Swing реагують на дії користувача, і події (рос.: события), що генеруються користувачами, повинні бути оброблені. Наприклад, події генеруються переміщенням миші, натисканням клавіш на клавіатурі, при виборі елемента списку та т.ін. Існують події, які безпосередньо не пов'язані з діями користувачів. Наприклад, подія генерується після закінчення інтервалу часу, встановленого для таймеру.

У Swing використовується механізм обробки подій, який є і в AWT. Він носить назву *модель делегування подій*.

Джерело (event sources) генерує подію, яка передається одному або декільком обробникам – слухачам подій (event listener, рос.: слушатели событий).

Обробники очікують виникнення події, обробляють подію та повертають управління. Перевага такого підходу в тому, що логіка обробки подій відділена від логіки GUI, що генерує ці події. Елемент інтерфейсу «делегує» обробку події окремому фрагменту коду.

У моделі делегування подій обробник, щоб отримувати сповіщення про події, повинен бути зареєстрований в джерелі.

Джерела подій містять методи, що дозволяють пов'язувати їх зі слухачами. Коли відбувається відповідна подія, джерело посилає повідомлення всім об'єктам слухача, зареєстрованих для цього.

Суперкласом всіх подій є *java.util.EventObject*. Багато подій оголошені в пакеті *java.awt.event*, а деякі містяться в *javax.swing.events*.

Різні джерела можуть породжувати різні види подій. Наприклад, кнопка може посилати об'єкти класу *java.awt.event.ActionEvent*, а вікно – об'єкти класу *java.awt.event.WindowEvent*.

Для реєстрації об'єкта слухача джерелом події застосовується наступний оператор:

об'єктДжерелаПодії.add*Подія*Listener(*об'єктСлухачаПодії*)

Наприклад (з використанням безіменного класу):

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
. . .
JButton btnMsg = new JButton("Message Button");
. . .
btnMsg.addActionListener(
     new ActionListener()
     {
```

```
        public void actionPerformed(ActionEvent event)
        {
            JOptionPane.showMessageDialog( panel, "This is test click button!" );
        }
      }
);
. . .
panel.add(btnMsg);
. . .
```

В цьому коді створюється кнопка (об’єкт *JButton*) з написом “Message Button”. Виклик *btnMsg.AddActionListener( )* реєструє об’єкт-слухач подій, які описані інтерфейсом *java.awt.event.ActionListener*. Власно цей інтерфейс містить тільки один метод – *actionPerformed( )*. Тобто клас, що буде реалізовувати цей інтерфейс, повинен реалізувати цей метод. В наведеному коді створюється об’єкт безіменного класу, який і реалізовує цей метод, що буде обробляти подію, яка генерується при натисканні на кнопку.

В методі *actionPerformed( )* вказано, що при натисканні на кнопку, буде виведено на екран діалогове вікно сповіщення з текстом: “This is test click button!”. Діалогове вікно буде сформовано статичним методом *javax.swing.JOptionPane.showMessageDialog( )*. Першим параметром в цьому методі вказаний об’єкт GUI, який є батьківським для діалогу (це панель верхнього рівня).

Після того, як обробник події описаний, кнопка додається на панель. Зауважимо, що реєстрація об’єкта-слухача події не обов’язково може бути виконана до додавання кнопки в контейнер.

Якщо обробник події буде застосований в декількох місцях коду, тобто буде багато разів для різних об’єктів-джерел подій відбуватися реєстрація об’єктів-слухачів однієї і тієїж події, то має сенс створити окремий клас для опису слухача події. Приклад використання класу-слухача, що реалізує інтерфейс *ActionListener*:

```
// модуль  MyListener.java
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class MyListener implements ActionListener {
    public MyListener(Component parentComponent, String textMessage)
    {
        m_parentComponent = parentComponent;
        m_textMessage = textMessage;
    }
    public void actionPerformed(ActionEvent event)
    {
        JOptionPane.showMessageDialog(m_parentComponent, m_textMessage);
    }
    private Component m_parentComponent = null;
    private String m_textMessage = “”;
}
```

```
// модуль основного коду
. . .
JButton btnMsg1 = new JButton("Message Button 1");
JButton btnMsg2 = new JButton("Message Button 2");
btnMsg1.addActionListener(new MyListener(panel, "Test 1"));
btnMsg2.addActionListener(new MyListener(panel, "Test 2"));
. . .
panel.add(btnMsg1); panel.add(btnMsg2);
. . .
```

Таким чином, в окремому модулі описаний новий клас *MyListener*, який реалізує інтерфейс *ActionListener*. Щоб передати певні особливості в обробник події, використовуються параметри конструктора класу – батьківський об'єкт GUI (*panel*) та об'єкт-рядок, що буде визначати те, що виводиться в діалоговому вікні – "Test 1" або "Test 2".

Зрозуміло, що це приклад і в *actionPerformed( )* повинно бути описаний код, який виконується при натисканні на певні кнопки.

Джерело будь-якої події розпізнається за допомогою методу *java.util.EventObject.getSource( )*. Деякі програмісти для того, щоб розпізнати джерело подій в слухачі, загалом для декількох джерел, користуються методом *java.awt.event.ActionEvent.getActionCommand( )*, який повертає командний рядок, пов'язаний з дією. У кнопки – це текстовий напис.

Приклади обробки подій від декількох кнопок одним слухачем:

```
. . .
public void actionPerformed( ActionEvent event )
{
   String command = event.getActionCommand( );
   JButton btn = (JButton)event.getSource( );

   if( command.equals("Test 1") ) btn.setBackground( Color.red );
   else if( command.equals("Test 2") ) btn.setBackground( Color.blue );
}
. . .
JButton btnMsg1 = new JButton("Test 1");
JButton btnMsg2 = new JButton("Test 2");
. . .
btnMsg1.addActionListener( new MyListener( ) );
btnMsg2.addActionListener( new MyListener( ) );
. . .
panel.add(btnMsg1);
panel.add(btnMsg2);
. . .
```

Тобто для двох кнопок використовується єдиний об'єкт-слухач, в якому за допомогою перевірки напису на кнопці перевіряється що це за кнопка і відповідно від цього вже приймається рішення, в який колір змінювати фон кнопки. Причому для доступу до поточного об'єкту *JButton*, який викликав подію, використовується метод *getSource( )* з приведенням *Object* (що повертається методом) до того типу, з яким ми працюємо (тобто в даному випадку до *JButton*).

На рис. 9.1 представлений фрагмент діаграми класів бібліотеки Java для обробників подій, який показує розташування у класах вказаних методів.

Якщо користувач намагається закрити фрейм вікна, то об'єкт *JFrame* стає джерелом події *WindowEvent*. Слухач вікна повинен бути об'єктом класу, який реалізовує інтерфейс *java.awt.WindowListener*, в який входять сім методів:

```
public interface WindowListener extends EventListener
{
     void windowActivated(WindowEvent e);
     void windowClosed(WindowEvent e);
     void windowClosing(WindowEvent e);
     void windowDeactivated(WindowEvent e);
     void windowDeiconified(WindowEvent e);
     void windowIconified(WindowEvent e);
     void windowOpened(WindowEvent e);
}
```

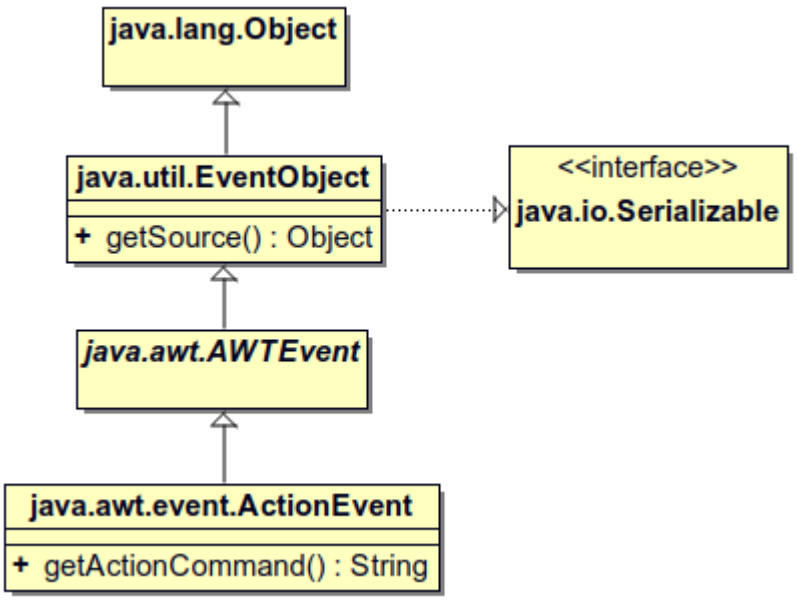


Рис. 9.1. Фрагмент діаграми класів бібліотеки Java для обробників подій

У разі, якщо не всі методи *WindowListener* повинні бути реалізовані, використовують клас-адаптер *java.awt.event.WindowAdapter*, який реалізує всі ці методи, але вони є пусті. Наприклад, нехай необхідно перед закриттям програми видавати користувачу запит:

```
. . .
frm.setDefaultCloseOperation(JFrame.DO_NOTHING_ON_CLOSE);
. . .
frm.addWindowListener( new WindowAdapter() {
```

```
            @Override
            public void windowClosing(WindowEvent e)
            {
               if(JOptionPane.showConfirmDialog(panel, "Close program?", "My Program",
                  JOptionPane.OK_CANCEL_OPTION)==JOptionPane.OK_OPTION)

               System.exit(0);
            }
         }
);
. . .
```

Тобто у цьому випадку об’єкт-слухач події – це об’єкт безіменного класу, який є нащадком *WindowAdapter*, в якому перезавантажується метод *windowClosing( )*.

Найбільш відомі на практиці типи подій з пакету AWT, що передаються слухачам:

– ActionEvent (клацання на кнопці, вибір пункту меню, вибір пункту в списку, натиснення клавіші <ENTER> в поле введення тексту);

– AdjustmentEvent (переміщення бігунка на панелі прокрутки);

– ComponentEvent (зміна розміру компонента, його переміщення, показ або приховування);

– ContainerEvent (додавання та видалення компоненту);

– FocusEvent (переміщення фокусу уваги на компонент і зняття фокуса з компонента);

– WindowEvent (активування, деактивування, згортання, розгортання та закриття вікна);

– ItemEvent (установка прапорця або вибір пункту зі списку);

– KeyEvent (натискання або відпускання клавіші);

– MouseEvent (натискання, відпускання клавіші миші, а також переміщення курсора та буксирування об'єкту);

– MouseWhellEvent (обертання коліщатка миші);

– TextEvent (зміна змісту текстового поля).

Пакет javax.swing.event містить додаткові події, характерні для компонентів пакету swing.

Сім інтерфейсів слухача з пакету AWT, які мають кілька методів, супроводжуються класами-адаптерами, що реалізують всі "порожні" методи інтерфейсу:

– ComponentAdapter;

– ContainerAdapter;

– FocusAdapter;

– KeyAdapter;

– MouseAdapter;

– MouseMotionAdapter;

– WindowAdapter.

Деякі методи інтерфейсів слухачів семантичних подій:
– ActionListener: void actionPerformed(ActionEvent e);
– AdjustmentListener: void adjustmentValueChanged(AdjustmentEvent e);
– ItemListener: void itemStateChanged(ItemEvent e);
– TextListener: void textValueChanged(TextEvent e).

Деякі методи інтерфейсів слухачів деяких низькорівневих подій:
– ComponentListener: void componentResized(ComponentEvent e);
– KeyListener: void keyPressed(KeyEvent e);
void keyReleased(KeyEvent e);
– MouseListener: void mouseClicked(MouseEvent e);
void mouseEntered(MouseEvent e);
void mouseExited(MouseEvent e);
void mousePressed(MouseEvent e);
void mouseReleased(MouseEvent e);
– MouseMotionListener: void mouseMoved(MouseEvent e);
void mouseDragged(MouseEvent e);
– MouseWheelListener: void mouseWheelMoved(MouseWheelEvent e);
– WindowStateListener: void windowStateChanged(WindowEvent e).

Описи деяких методів класу *MouseEvent*:

int getButton( ) – повертає одну з констант NOBUTTON, BUTTON1, BUTTON2 or BUTTON3 в разі якщо була натиснута клавіша миші;

Point getPoint( ), int getX( ), int getY( ) – повертає x, y координати позиції покажчика на поточному компоненті;

Point getLocationOnScreen( ), int getXOnScreen( ), int getXOnScreen( ) – повертає абсолютні координати x, y покажчика миші на екрані;

static String getMouseModifiersText(int modifiers) – повертає клавішу, натиснуту спільно з кнопкою миші, використовується спільно з InputEvent.

Описи деяких методів класу *KeyEvent*:

char getKeyChar( ) – повертає символ, що асоціюється з клавішею;

int getKeyCode( ) – повертає код клавіші;

static String getKeyText(int keyCode) – повертає назву клавіші по її коду, наприклад, "HOME", "F1", "A" і т.д.

Метод класу *ComponentEvent*: *Component getComponent( )* – повертає об'єкт, що послав повідомлення. Приклад:

```
. . .
frm.addComponentListener( new ComponentAdapter() {

     public void Msg(String message)
     {
         JOptionPane.showMessageDialog(panel, message);
```

```
      }

      @Override
      public void componentResized(ComponentEvent e)
      {
          Component c = e.getComponent();
          Msg("Size:" + c.getWidth() + "x" + c.getHeight());
      }
} );
. . .
```

Приклад зміни кольору кнопки при переміщенні на неї покажчика миші:

```
. . .
btnMsg1.addMouseListener( new MouseAdapter() {

      @Override
      public void mouseEntered(MouseEvent e) {
          m_color = ((JButton)(e.getSource())).getBackground();
          ((JButton)(e.getSource())).setBackground(Color.GREEN);
      }

     @Override
      public void mouseExited(MouseEvent e) {
          ((JButton)(e.getSource())).setBackground(m_color);
      }
});
. . .
private Color m_color;
```

Приклад обробки події таймера (кнопка, що моргає):

```
. . .
btnMsg1 = new JButton("Test 1");
. . .
int delay = 1000; //milliseconds
ActionListener taskPerformer = new ActionListener()
{
      public void actionPerformed(ActionEvent evt)
      {
          if( btnMsg1.getBackground() == Color.RED)
              btnMsg1.setBackground(Color.GREEN);
          else
              btnMsg1.setBackground(Color.RED);
      }
};
new Timer(delay, taskPerformer).start();
. . .
private JButton btnMsg1;
. . .
```

В останньому прикладі створюється безіменний об’єкт типу *javax.swing.Timer*, через конструктор якого передається значення періоду активування таймера і для цього створений об’єкт-слухач події таймера, який повинен реалізовувати інтерфейс *ActionListener*. У прикладі по виклику таймера кнопка *btnMsg1* змінює свій фон з червоного кольору на зелений та навпаки через кожну секунду (1000 мілісекунд).

Лекція 10. Java. Компоненти бібліотеки Swing. Діалогові вікна

Клас *JOptionPane* підтримує чотири основні типи діалогових вікон:

1. Вікна для виведення повідомлень.
2. Вікна для отримання підтвердження від користувача.
3. Вікна для введення даних.
4. Вікна для установки опцій.

Всі діалогові вікна, що створюються за допомогою *JOptionPane*, є модальними. Якщо необхідно створити немодальне діалогове вікно, то використовують клас *JDialog*.

Для створення діалогових вікон на основі *JOptionPane* використовують один з фабричних методів *show...( )*:

showMessageDialog( )
showConfirmDialog( )
showInputDialog( )
showOptionDialog( )

static void showMessageDialog(Component parent, Object msg, String title, int msgT)

Параметр *msgT* задає тип повідомлення. Його значення (константи *JOptionPane*):

ERROR_MESSAGE – відображає повідомлення про помилку.
INFORMATION_MESSAGE – інформаційне повідомлення.
PLAIN_MESSAGE – повідомлення без піктограми.
QUESTION_MESSAGE – повідомлення-питання.
WARNING_MESSAGE – попередження.

static int showConfirmDialog(Component parent, Object msg, String title, int optT, int msgT)

За замовчуванням формується вікно типу QUESTION_MESSAGE, в якому відображаються три кнопки: Yes, No та Cancel. Метод повертає значення цілого типу, що дорівнює одній з наступних констант JOptionPane:

CANCEL_OPTION, CLOSED_OPTION, YES_OPTION, NO_OPTION

У більшості програм значення CLOSED_OPTION обробляється так само, як і NO_OPTION.

Параметр optT повинен бути однією з наступних констант JOptionPane:

YES_NO_OPTION, YES_NO_CANCEL_OPTION

Існує кілька варіантів методу *showInputDialog( )*. Найпростіший з них створює вікно з полем редагування, в якому користувач може ввести текстовий

рядок:

static String showInputDialog(Object msg)

За замовчуванням буде виведено діалогове вікно з піктограмою знаку оклику та кнопками OK і CANCEL. При виборі кнопки CANCEL метод повертає *null*. Порожній чи ні рядок, який повертається після натискання на кнопку OK, можна перевірити за допомогою методу *isEmpty( )*:

```
String str = JOptionPane.showInputDialog("Input value:");
if(str!=null)
   if(!str.isEmpty())
   {
     . . .
   }
```

Для ініціалізації поля введення необхідним значенням використовується метод:

static String showInputDialog(Component parent, Object msg, Object initVal)

Існує варіант методу *showInputDialog( )*, що дозволяє задати список, з якого користувач може вибрати потрібний пункт, а також вказати піктограму для відображення у вікні:

static Object showInputDialog(Component parent, Object msg,
String title, int msgT, Icon icon, Object[] vals, Object initVal)

Таке вікно особливо корисно, коли необхідно обмежити вибір користувача набором визначених відповідей. Наприклад:

```
String[] names = {"Dnepr", "Orel", "Samara"};
String river = (String)JOptionPane.showInputDialog(m_parent, "Select river:",
               "Select", JOptionPane.QUESTION_MESSAGE, null, names, names[0]);
```

Результат:

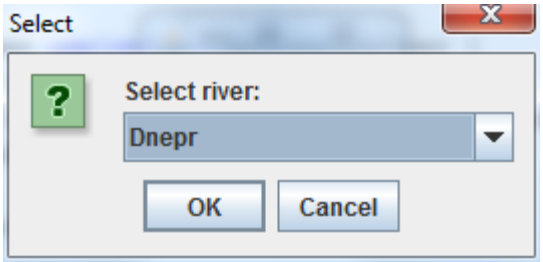


Рис. 10.1. Приклад `JOptionPane.showInputDialog`

Метод створення діалогу для установки опцій один:

```
static int showOptionDialog(Component parent, Object msg, String title, int optT, int msgT,
                            Icon icon, Object[] options, Object initVal)
```

Зазвичай в якості параметра *options* передається масив рядків. Кожен рядок інтерпретується як напис на кнопці. Ви можете надсилати й масив піктограм, які будуть відображатися на кнопках. Приклад діалогового вікна, що дозволяє користувачеві вибирати спосіб з'єднання з мережею:

```
String[] connectOptions = {"Modem", "Wireless", "Satellite", "Cable"};
int res = JOptionPane.showOptionDialog(m_parent, "Choose One:",
                "Connection Type",
                JOptionPane.DEFAULT_OPTION,
                JOptionPane.QUESTION_MESSAGE, null,
                connectOptions, connectOptions[0]);
switch(res)
{
   case 0: . . . break;
   case 1: . . . break;
   case 2: . . . break;
   case 3: . . . break;
   case JOptionPane.CLOSED_OPTION: . . .
}
```

Результат:

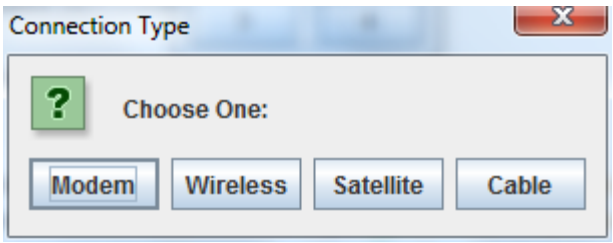


Рис. 10.2. Приклад `JOptionPane.showOptionDialog`

Метод повертає індекс цілого типу обраної опції, зіставленої з об'єктом в масиві.

Приклад реалізації нестандартного діалогу:

```
Object[] opt = {  new JLabel("Name"),
                  new JTextField(10),
                  new JLabel("Phone Number"),
                  new JTextField(10),
                  "OK", "Cancel" };
int res = JOptionPane.showOptionDialog(m_parent, "Enter Info:",
                "Get Name and Telephone",
```

```
                    JOptionPane.OK_CANCEL_OPTION,
                    JOptionPane.QUESTION_MESSAGE, null, opt, "Cancel");
    if(res==4) { // OK
          String txtName = ((JTextField)(opt[1])).getText();
          String txtNum  = ((JTextField)(opt[3])).getText();
          JOptionPane.showMessageDialog(m_parent, txtName+" "+txtNum);
    }
```

Результат:

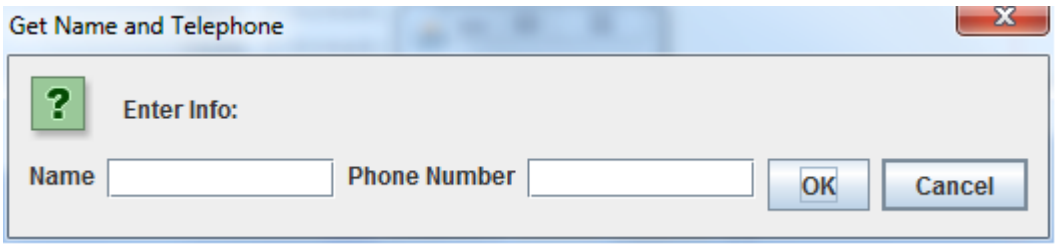


Рис. 10.3. Приклад нестандартного `JOptionPane.showOptionDialog`

Для створення діалогового вікна на базі *JDialog* необхідно виконати наступні дії:

1. Створити об'єкт *JDialog*.
2. Визначити диспетчер компонування, розмір і, можливо, політику закриття діалогового вікна.
3. Включити необхідні компоненти в панель вмісту.
4. Показати вікно, викликавши метод *setVisible(true)*.

Для видалення діалогового вікна з екрану викликають метод *setVisible(false)*. Якщо діалог в подальшому не використовується, то його закривають методом *dispose( )*.

Основні конструктори класу *JDialog*:

JDialog(Frame owner) // створює немодальне діалогове вікно
JDialog(Frame owner, boolean model)
JDialog(Frame owner, String title)
JDialog(Frame owner, String title, boolean model)

Приклад:

```
. . .
private JDialog dlg;
. . .
dlg = new JDialog( mainFrm, “TestDlg”, true );
dlg.setBounds( MainRun.getCenterRect(300, 300) );
dlg.setLayout( new FlowLayout() );
dlg.add( new JButton(“My Button”) );
. . .
dlg.setVisible( true );
```

Лекція 11. Шаблони проектування

Шаблони проектування (зразки, патерни, patterns) – один з найважливіших складових частин об'єктно-орієнтованої технології розробки ПЗ. Вони широко застосовуються в інструментах аналізу. Шаблон можна визначити як спільне рішення деякої проблемної ситуації в певному контексті. Шаблон складається з чотирьох основних елементів.

1. Ім'я.
2. Проблема.
3. Рішення.
4. Наслідки.

Пославшись на ім'я зразка, можна відразу описати проблему проектування, її рішення та їх наслідки. Присвоєння шаблонам імен, дозволяє проектувати на більш високому рівні абстракції.

Проблема – це опис задачі, яку розв’язують. Необхідно сформулювати завдання та її контекст. Може описуватися конкретна проблема проектування, може включатися перелік умов, при виконанні яких має сенс застосовувати даний шаблон.

Рішення – це опис елементів проектного рішення, зв'язків між ними та функцій кожного елементу. Зазвичай дається абстрактний опис завдання проектування і того, як воно може бути вирішене за допомогою якогось вельми узагальненого поєднання елементів (класів та об'єктів).

Наслідки – це опис області застосування, переваг і недоліків зразка. Хоча при описі проектних рішень про наслідки часто не згадують, знати про них необхідно, щоб можна було вибрати між різними варіантами і оцінити переваги та недоліки застосування даного шаблону.

Шаблони можуть розглядатися на різних рівнях абстракції та в різних предметних областях. Найбільш загальними категоріями шаблонів ПЗ є:

– шаблони бізнес-моделювання;
– шаблони аналізу;
– шаблони поведінки;
– шаблони проектування;
– архітектурні шаблони;
– шаблони програмування.

Під шаблоном проектування розуміється опис взаємодії об'єктів та класів, адаптованих для вирішення загальної задачі проектування в конкретному контексті.

Найбільш поширені 23 патерни проектування.

1. *Abstract Factory* (абстрактна фабрика) – надає інтерфейс для створення сімейств, пов'язаних між собою, або незалежних об'єктів, конкретні класи яких невідомі.

2. *Adapter* (адаптер) – перетворює інтерфейс класу в деякий інший інтерфейс, очікуваний клієнтами. Забезпечує спільну роботу класів, яка була б неможлива без даного патерну через несумісність інтерфейсів.

3. *Bridge* (міст) – відокремлює абстракцію від реалізації, завдяки чому з'являється можливість незалежно змінювати те й інше.

4. *Builder* (будівельник) – відокремлює конструювання складного об'єкту від його уявлення, дозволяючи використовувати один і той же процес конструювання для створення різних уявлень.

5. *Chain of Responsibility* (ланцюжок обов'язків) – можна уникнути жорсткої залежності відправника запиту від його одержувача, при цьому запитом починає оброблятися один з декількох об'єктів. Об'єкти-одержувачі зв'язуються в ланцюжок, і запит передається по ланцюжку, поки якийсь об'єкт його не обробить.

6. *Command* (команда) – інкапсулює запит у вигляді об'єкта, дозволяючи тим самим параметризувати клієнтів типом запиту, встановлювати черговість запитів, протоколювати їх і підтримувати скасування виконання операцій.

7. *Composite* (компонувач) – групує об'єкти в деревовидні структури для представлення ієрархій типу «частина-ціле». Дозволяє клієнтам працювати з одиничними об'єктами так само, як з групами об'єктів.

8. *Decorator* (декоратор) – динамічно покладає на об'єкт нові функції. Декоратори застосовуються для розширення наявної функціональності і є гнучкою альтернативою породження підкласів.

9. *Facade* (фасад) – надає уніфікований інтерфейс до безлічі інтерфейсів в деякій підсистемі. Визначає інтерфейс більш високого рівня, що полегшує роботу з підсистемою.

10. *Factory Method* (фабричний метод) – визначає інтерфейс для створення об'єктів, при цьому обраний клас інстанцирується підкласами.

11. *Flyweight* (пристосуванець) – використовує поділ для ефективної підтримки великого числа дрібних об'єктів.

12. *Interpreter* (інтерпретатор) – для заданої мови визначає уявлення його граматики, а також інтерпретатор пропозицій мови, що використовує це уявлення.

13. *Iterator* (ітератор) – дає можливість послідовно обійти всі елементи складеного об'єкту, не розкриваючи його внутрішнє представлення.

14. *Mediator* (посередник) – визначає об'єкт, в якому інкапсульоване знання про те, як взаємодіють об'єкти з деякої безлічі. Сприяє зменшенню числа зв'язків між об'єктами, дозволяючи їм працювати без явних посилань один на одний. Це, в свою чергу, дає можливість незалежно змінювати схему взаємодії.

15. *Memento* (зберігач) – дозволяє, не порушуючи інкапсуляції, отримати і зберегти у зовнішній пам'яті внутрішній стан об'єкту, щоб пізніше об'єкт можна було відновити точно в такому ж стані.

16. *Observer* (спостерігач) – визначає між об'єктами залежність типу один-до-багатьох, так що при зміні стану одного об'єкту всі залежні від нього отримують повідомлення та автоматично оновлюються.

17. *Prototype* (прототип) – описує види створюваних об'єктів за допомогою прототипу і створює нові об'єкти шляхом його копіювання.

18. *Proxy* (заступник) – підміняє інший об'єкт для контролю доступу до нього.

19. *Singleton* (одинак) – гарантує, що певний клас може мати тільки один екземпляр, і надає глобальну точку доступу до нього.

20. *State* (стан) – дозволяє об'єкту варіювати свою поведінку при зміні внутрішнього стану. При цьому створюється враження, що змінився клас об'єкту.

21. *Strategy* (стратегія) – визначає сімейство алгоритмів, інкапсулюючи їх всі та дозволяючи підставляти один замість іншого. Можна міняти алгоритм незалежно від клієнта, який ним користується.

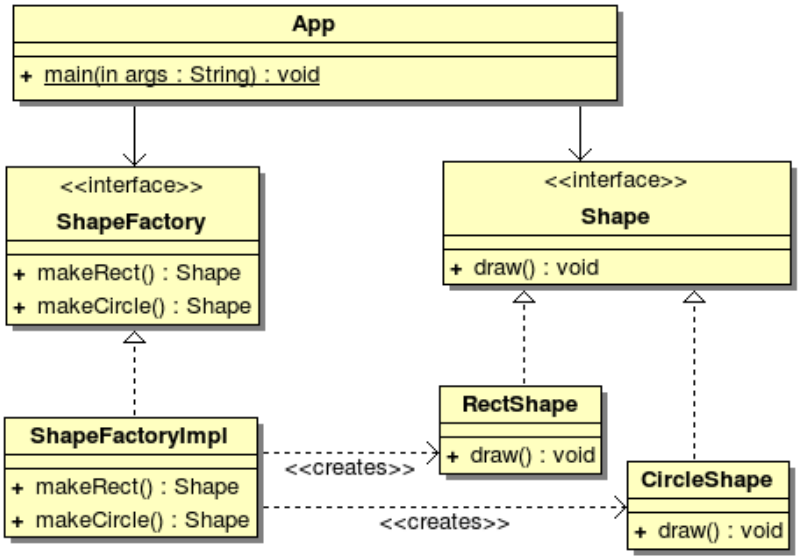
22. *Template Method* (шаблонний метод) – визначає скелет алгоритму, перекладаючи відповідальність за деякі його кроки на підкласи. Дозволяє підкласам перевизначити кроки алгоритму, не змінюючи його загальної структури.

23. *Visitor* (відвідувач) – являє операцію, яку треба виконати над елементами об'єкта. Дозволяє визначити нову операцію, не змінюючи класи елементів, до яких він застосовується.

Простір шаблонів проектування (шаблони, що розглянуті на прикладах нижче, підкреслені):

Рівень	*Ціль*		
	Шаблони, що породжують	*Структурні шаблоні*	*Шаблони поведінки*
Рівень класу	Фабричний метод	Адаптер (класу)	Інтерпретатор Шаблонний метод
Рівень об'єкту	Абстрактна фабрика Одинак Прототип Будівельник	Адаптер (об'єкту) Декоратор Заступник Компонувач Міст Пристосуванець Фасад	Ітератор Команда Спостерігач Відвідувач Посередник Стан Стратегія Зберігач Ланцюжок обов'язків

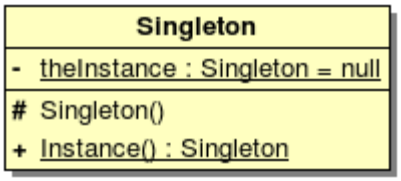
Шаблон Abstract Factory (на прикладі)



Приклад реалізації *Abstract Factory*:

```
public interface ShapeFactory {
    public Shape makeRect( );
    public Shape makeCircle( );
}

public class ShapeFactoryImpl implements ShapeFactory {
    @Override public Shape makeRect() { return new RectShape( ); }
    @Override public Shape makeCircle() { return new CircleShape( ); }
}

public interface Shape {
    public void draw( );
}

public class RectShape implements Shape{
    @Override public void draw( ) { System.out.println("Draw Rectangle"); }
}

public class CircleShape implements Shape {
    @Override public void draw( ) { System.out.println("Draw Circle"); }
}
. . .
ShapeFactory shapes = new ShapeFactoryImpl( );
Shape circle = shapes.makeCircle( );
Shape rectangle = shapes.makeRect( );
circle.draw( );
rectangle.draw( );
```

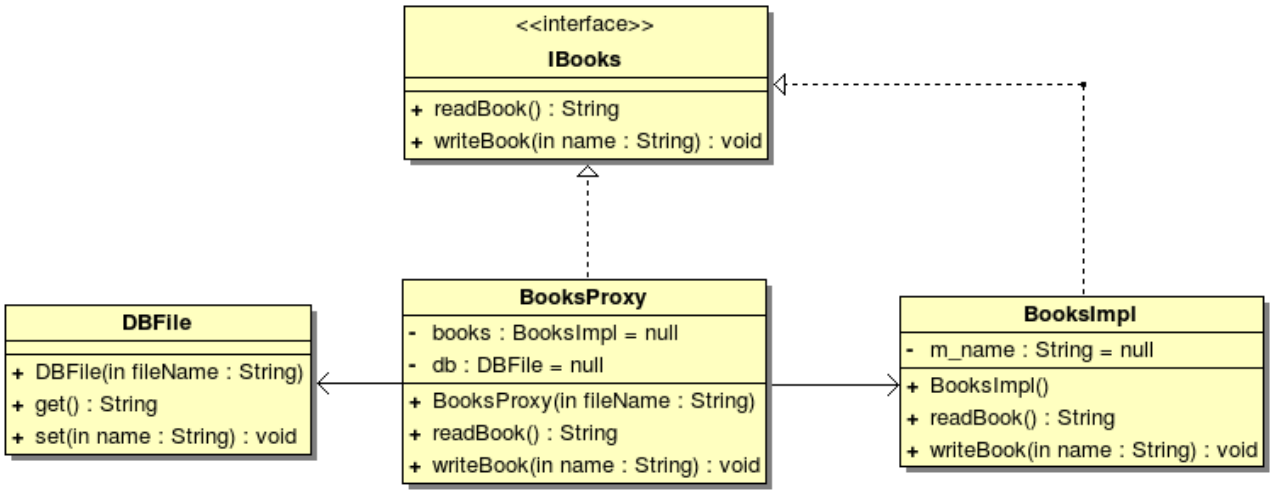
Шаблон Singleton



Приклад реалізації *Singleton*:

```
public class Singleton {
    private static Singleton theInstance = null;
    // або: private Singleton( ){ }
    protected Singleton( ){ }
    public static Singleton Instance( )
    {
     if(theInstance == null) { theInstance = new Singleton( ); }
     return theInstance;
    }
}
. . .
Singleton s = Singleton.Instance( );
```

Шаблон Proxy (на прикладі)

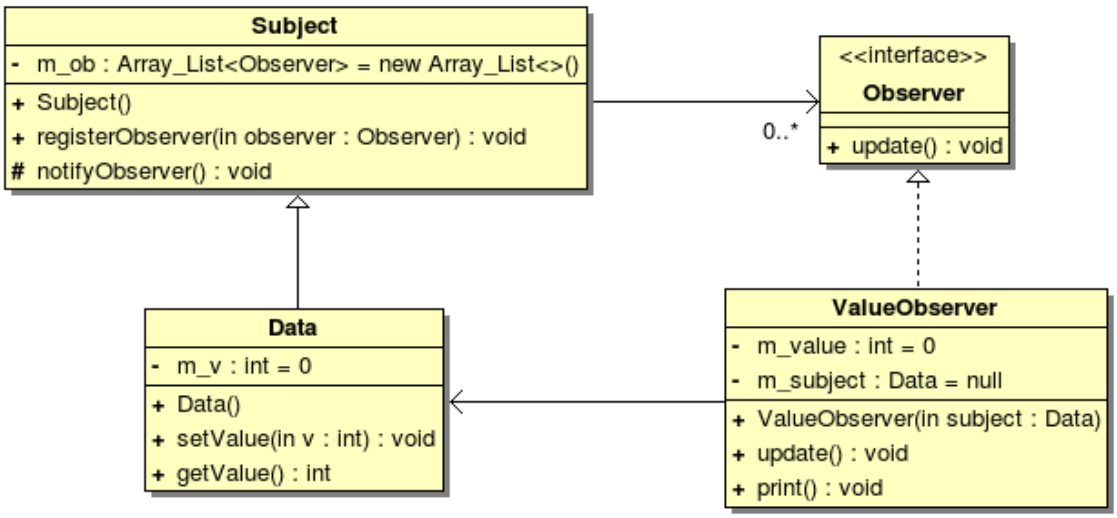


Приклад реалізації *Proxy*:

```
public class DBFile {
    public DBFile(String fileName) { }
    public String get( ) { return ""; }
    public void set(String name) { }
}
public interface IBooks {
```

```
    public String readBook( );
    public void writeBook(String name);
}
public class BooksImpl implements IBooks
{
    private String m_name = null;
    public BooksImpl( ) { }
    @Override public String readBook( ) {  return m_name; }
    @Override public void writeBook(String name) {  m_name = name; }
}

public class BooksProxy implements IBooks
{
    private BooksImpl books = null;
    private DBFile db = null;

    public BooksProxy(String fileName)
    { books = new BooksImpl( ); db = new DBFile(fileName); }

    @Override public String readBook( ) {
        books.writeBook( db.get( ) );
        return books.readBook( );
    }

    @Override public void writeBook(String name) {
        books.writeBook(name);
        db.set(name);
    }
}
. . .
IBooks books = new BooksProxy("datafile.dat");
books.writeBook("my book");
```

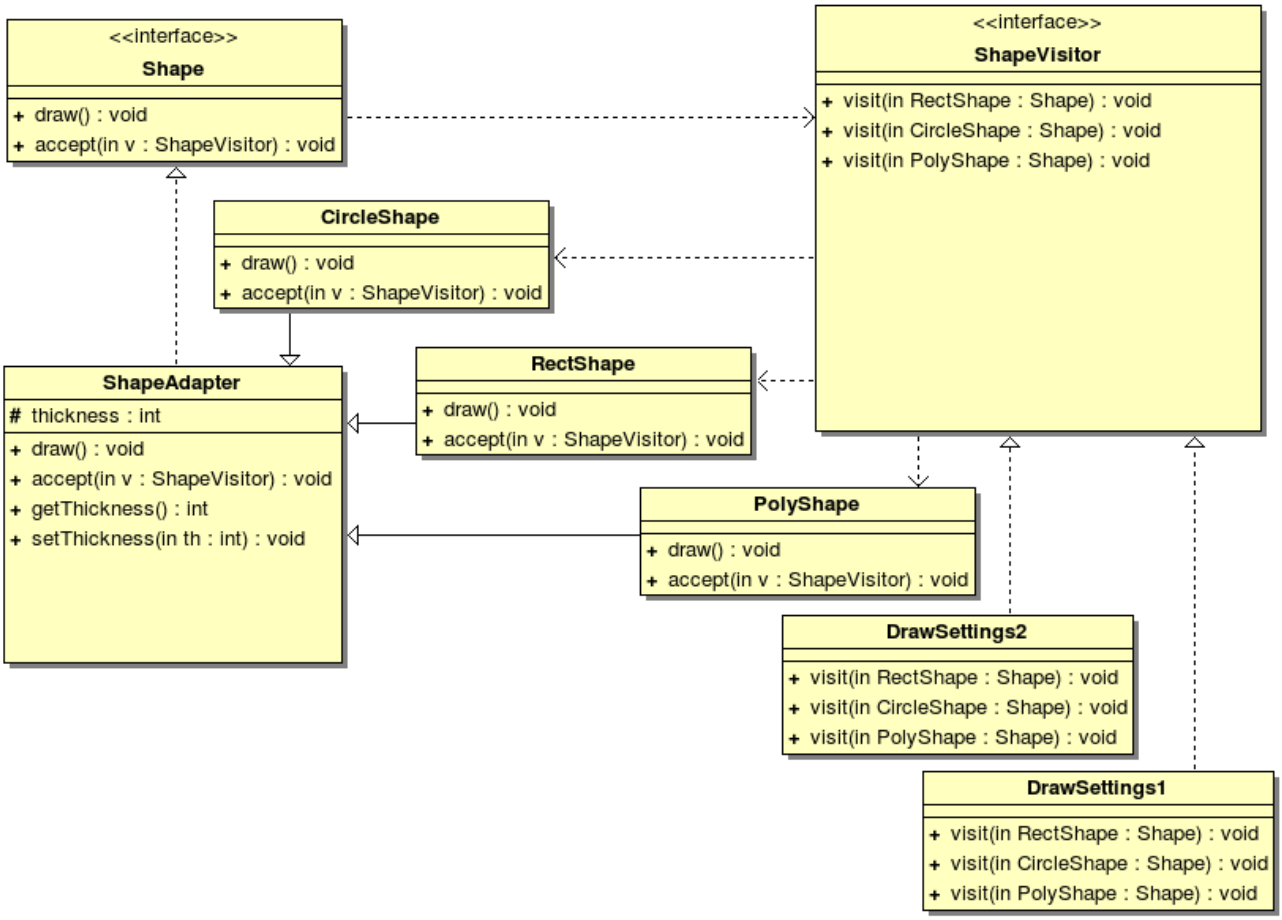
Шаблон Observer (на прикладі)



Приклад реалізації *Observer*:

```
public interface Observer
{
    public void update( );
}

// ==========================
import java.util.ArrayList;
public class Subject
{
  private ArrayList<Observer> m_ob = new ArrayList<>( );
  public Subject( ) { }
  public void registerObserver(Observer observer) { m_ob.add(observer); }
  protected void notifyObserver( )
  {
        for(Observer observer : m_ob) observer.update( );
  }
}

// ==========================
public class Data extends Subject
{
  private int m_v = 0;
  public Data( ) { }
  public void setValue(int v) { m_v = v; this.notifyObserver( ); }
  public int getValue( ) { return m_v; }
}

// ==========================
public class ValueObserver implements Observer
{
    private int m_value = 0;
    private Data m_subject = null;
    public ValueObserver(Data subject) {  m_subject = subject; }
    @Override public void update( ) {  m_value = m_subject.getValue( ); }
    public void print( ) { System.out.println(m_value); }
}

// ==========================
. . .
Data data = new Data( );
ValueObserver observer1 = new ValueObserver(data);
ValueObserver observer2 = new ValueObserver(data);
data.registerObserver(observer1);
data.registerObserver(observer2);
for(int i=1; i<=10; i++)
  { data.setValue(i); observer1.print( ); observer2.print( ); }
```

Шаблон Visitor (на прикладі)



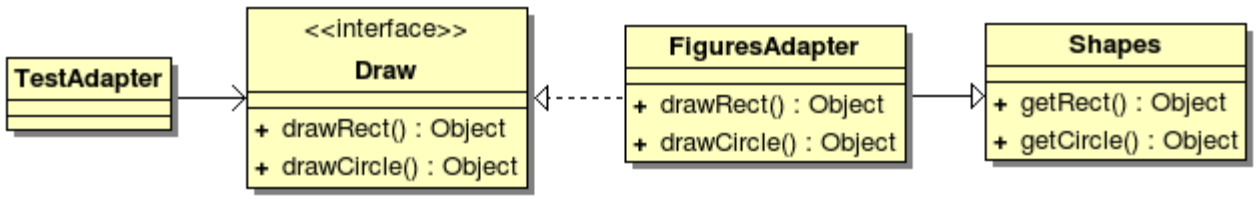
Приклад реалізації *Visitor*:

```
public interface Shape {
    public void draw( );
    public void accept(ShapeVisitor v);
}

public class ShapeAdapter implements Shape
{
    protected int thickness;
    @Override public void draw( ) { }
    @Override public void accept(ShapeVisitor v) {  }
    public int getThickness( ) { return thickness; }
    public void setThickness(int th) { thickness = th; }
}

public class RectShape extends ShapeAdapter
{
  @Override public void draw( )
  { System.out.println("Draw Rect with thickness " + getThickness( ) ); }

  @Override public void accept(ShapeVisitor v) { v.visit(this); }
}
```

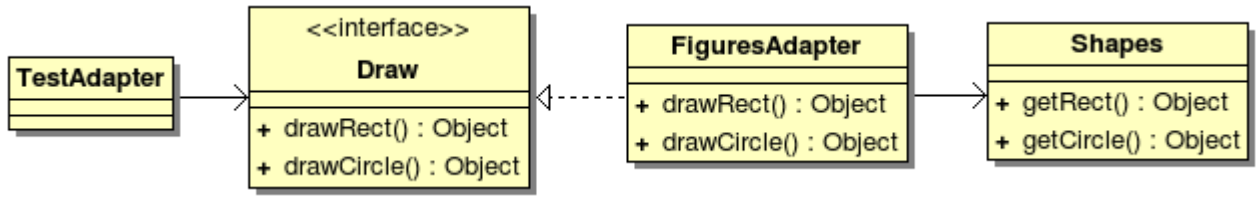
```
public class CircleShape extends ShapeAdapter
{
@Override public void draw( )
{ System.out.println("Draw Circle with thickness " + getThickness( ) ); }

@Override public void accept(ShapeVisitor v) { v.visit(this); }
}

public class PolyShape extends ShapeAdapter
{
@Override public void draw( )
{ System.out.println("Draw Poly with thickness " + getThickness( ) ); }

@Override public void accept(ShapeVisitor v) { v.visit(this); }
}

public interface ShapeVisitor {
    public void visit(RectShape shape);
    public void visit(CircleShape shape);
    public void visit(PolyShape shape);
}

public class DrawSettings1 implements ShapeVisitor
{
 @Override public void visit(RectShape shape) { shape.setThickness(3);}
 @Override public void visit(CircleShape shape){ shape.setThickness(2);}
 @Override public void visit(PolyShape shape)  { shape.setThickness(1);}
}

public class DrawSettings2 implements ShapeVisitor
{
 @Override public void visit(RectShape shape){ shape.setThickness(1); }
 @Override public void visit(CircleShape shape) { shape.setThickness(2);}
 @Override public void visit(PolyShape shape) { shape.setThickness(1); }
}
. . .
RectShape rectangle = new RectShape( );
CircleShape circle = new CircleShape( );
PolyShape poly = new PolyShape( );
DrawSettings1 settings1 = new DrawSettings1( );
DrawSettings2 settings2 = new DrawSettings2( );
rectangle.accept(settings1);
circle.accept(settings1);
poly.accept(settings2);
rectangle.draw( );
circle.draw( );
poly.draw( );
```

Шаблон Adapter (на прикладі)

Class Adapter:



Object Adapter:



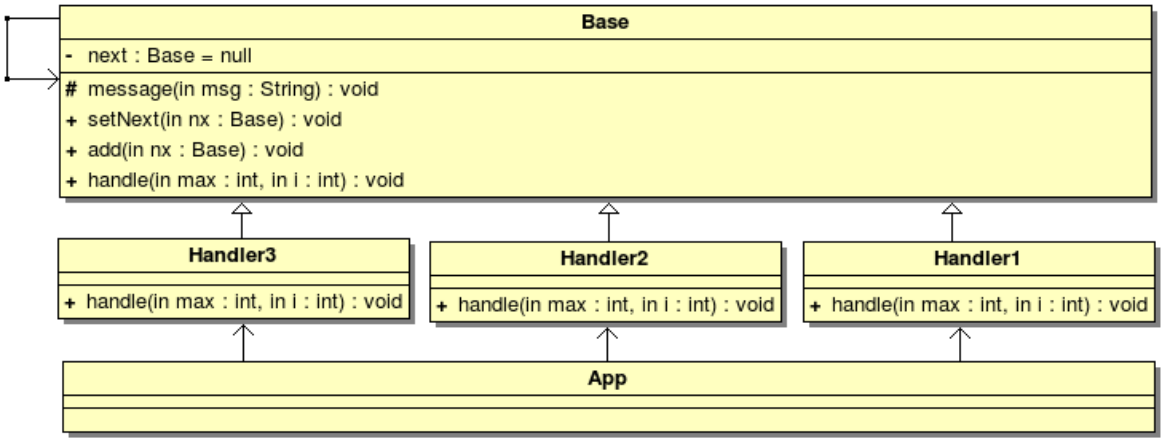
Приклад реалізації *Class Adapter*:

```
public interface Draw {
    public Object drawRect( );
    public Object drawCircle( );
}

public class Shapes {
    public Object getRect( ) { return new Object( ); }
    public Object getCircle( ) { return new Object( ); }
}

public class FiguresAdapter extends Shapes implements Draw
{
    @Override public Object drawRect( ) { return getRect( ); }
    @Override public Object drawCircle( ) { return getCircle( ); }
}
. . .
Draw obj = new FiguresAdapter( );
Object rectangle = obj.drawRect( );
Object circle = obj.drawCircle( );
```

Приклад реалізації *Object Adapter*:

```
public class FiguresAdapter implements Draw {
    private Shapes shapes = new Shapes( );
    @Override public Object drawRect( ) { return shapes.getRect( ); }
    @Override public Object drawCircle( ) { return shapes.getCircle( ); }
}
```

Шаблон Chain of Responsibility (на прикладі)



Приклад реалізації *Chain of Responsibility*:

```
public class Base {
    private Base next = null;
    protected void message(String msg) { System.out.print(msg+" "); }
    public void setNext(Base nx) { next = nx; }
    public void add(Base nx)
    { if(next!=null) next.add(nx); else next = nx; }
    public void handle(int max, int i) { next.handle(max, i); }
}
public class Handler1 extends Base {
    @Override public void handle(int max, int i)
    { message("Handler1 "+i); if(max>0) {max--; super.handle(max, i); } }
}
public class Handler2 extends Base {
   @Override public void handle(int max, int i)
   { message("Handler2 "+i); if(max>0) {max--;  super.handle(max, i); } }
}
public class Handler3 extends Base {
   @Override public void handle(int max, int i)
   { message("Handler3 "+i); if(max>0) {max--;  super.handle(max, i); } }
}
. . .
Handler1 root = new Handler1( );
Handler2 two = new Handler2( );
Handler3 thr = new Handler3( );
root.add(two);
two.add(thr);
thr.setNext(root);
int max = 10;
for(int i=1; i<=max; i++) { root.handle(2, i); System.out.println( ); }
```

РЕКОМЕНДОВАНА ЛІТЕРАТУРА

1. Bhuvan Unhelkar. Software Engineering with UML. – Auerbach Publications, CRC PRESS, 2018. – 427 p. ISBN-10: 1138297437, ISBN-13: 978-1-138-29743-2.

2. Bernhard Rumpe. Modeling with UML: Language, Concepts, Methods. – Springer International Publishing, 2016. – 288 p. ISBN-13: 978-3-319-33933-7.

3. Bernhard Rumpe. Agile Modeling with UML: Code Generation, Testing, Refactoring. – Springer, 2017. – 394 p. ISBN-10: 3319588613, ISBN-13: 978-3319588612.

4. Hassan Gomaa. Software Modeling and Design: UML, Use Cases, Patterns, and Software Architectures. – Cambridge University Press, 2011. – 578 p. ISBN-10: 0521764149, ISBN-13: 9780521764148.

5. Martin Fowler. Patterns of enterprise application architecture. – Addison-Wesley, 2015. – 558 p. ISBN-10: 0321127420, ISBN-13: 9780321127426.

6. Martin Fowler. UML Distilled: A Brief Guide to the Standard Object Modeling Language (3rd Edition). – Addison-Wesley Professional, 2003. – 208 p. ISBN-10: 9780321193681, ISBN-13: 978-0321193681.

7. Herbert Schildt. Java: The Complete Reference, Eleventh Edition. – McGraw-Hill Education, 2019. – 1882 p. ISBN-13: 978-1260440232.

8. Cay S. Horstmann. Core Java Volume I. Fundamentals. – Pearson, 2018. – 928 p. ISBN-10: 0135166306, ISBN-13: 978-0135166307.

9. Cay S. Horstmann. Core Java Volume II. Advanced Features. – Pearson, 2019. – 960 p. ISBN-10: 0135166314, ISBN-13: 978-0135166314.

10. Програма та методичні рекомендації щодо виконання проектно-технологічної практики для студентів третього курсу навчання спеціальності 126 “Інформаційні системи та технології” / Гаркуша І.М. – Д.: НТУ «ДП», 2020. – 18 с. URL: https://it.nmu.org.ua/ua/scientific_method_materials/books/ICT_Програма_проектно-технологічної_практики_2020.pdf (дата звернення: 04.05.2020).

Навчальне видання

Гаркуша Ігор Миколайович

Конспект лекцій
з дисципліни
“Проектування інформаційних систем”
для студентів галузі знань 12 “Інформаційні технології”
спеціальності 126 “Інформаційні системи та технології”

Електронний ресурс

Видано
у Національному технічному університеті
«Дніпровська політехніка».
Свідоцтво про внесення до Державного реєстру ДК №1842 від 11.06.2004.
49005, м. Дніпро, просп. Дмитра Яворницького, 19.