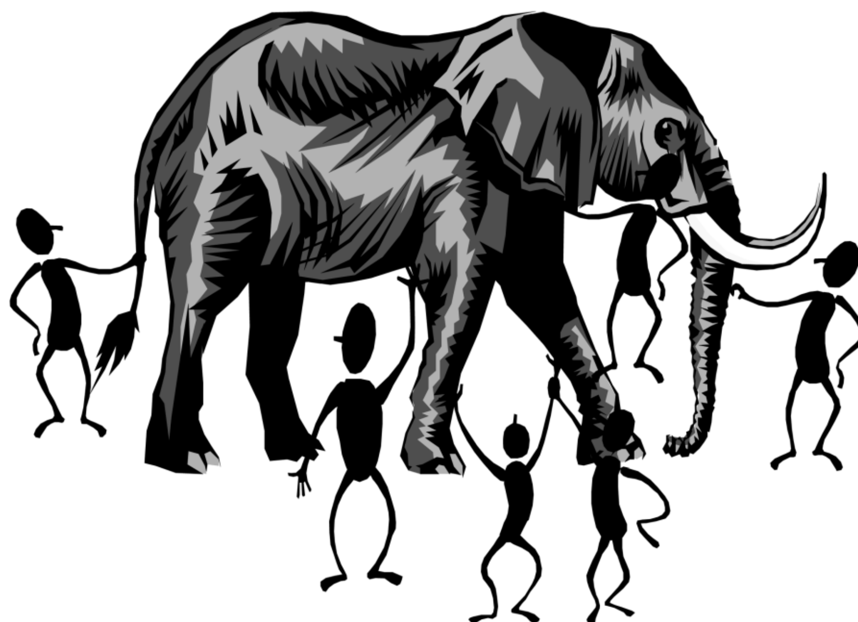
М.Петрик, О.Петрик

Моделювання програмного забезпечення

Науково-методичний посібник

напрям підготовки 6.050103 «Програмна інженерія»,

напрям підготовки 7.05010302, 805010302 «Іінженерія програмного забезпечення»,



Тернопіль
2015

УДК 519.4+681.3
ББК 22.18
П30

Рекомендовано до друку вченою радою Тернопілдьського національного технічного університету імені Івана Пулюя протокол № 1 від 10 лютого 2015 р.

Рецензенти:
С.А. Лупенко, докт. техн. наук, проф.,
О.А. Пастух, докт. техн. наук, проф.

П30 Петрик М.Р. Моделювання програмного забезпечення : науково-методичний посібник / М.Р. Петрик, О.Ю. Петрик – Тернопіль : Вид-во ТНТУ імені Івана Пулюя, 2015. – 200 с.

Навчальний матеріал посібнику викладено у вигляді професійного практикуму з дисципліни професійно-орієнтованого циклу «Моделювання та аналіз програмного забезпечення» для студентів напряму підготовки 6.050103 «Програмна інженерія».

Метою посібника є навчання студентів основним принципам і методам проектування програмного забепечення (ПЗ), їх практичного застосування з використанням сучасних інструментальних засобів побудови моделей, що використовуються на різних етапах життєвого циклу ПЗ та характеризують різні властивості ПЗ.

Посібник доцільно також використовувати і для інших суміжних професійно-орієнтованих дисциплін, зокрема, таких, як «Аналіз вимог», «Об’єктно-орієнтоване програмування», «Архітектура та проектування програмного забезпечення», а також для курсового і дипломного проектування та виконання кваліфікаційних робіт магістра зі спеціальності 8.05010302 «Інженерія програмного забезпечення».

УДК 519.4+681.3
ББК 22.18

© М.Р.Петрик, О.Ю.Петрик 2015
© Тернопільський національний технічний університет імені Івана Пулюя, 2015

Зміст

Переднє слово…………………………………………………………………. 4
1. Загальні підходи до моделювання ПЗ………………………………. 5
1.1. Основні підходи та концепції моделювання…………………………….. **5**
1.2. Принципи моделювання……………………………………………………... **8**
1.3. Об’єктне моделювання………………………………………………………. **10**
1.4. Введення в UML……………………………………………………………… **10**
1.5. Концептуальна модель UML………………………………………………… **12**
1.6. Діаграми UML………………………………………………………………... **17**
1.7. Правила UML…………………………………………………………………. **18**
2. Основи структурного моделювання ПЗ……………………………. 20
2.1. Моделювання класів, об’єктів та екземплярів…………………………... **20**
2.1.1. Класи………………………………………………………………….. **20**
2.1.2. Основні механізми доповнення і розширення будівельних блоків UML…... **31**
2.1.3. Розширені класи. Класифікатори ……………………......……… **36**
2.1.4. Екземпляри ………………………………………………………... **44**
2.2. Моделювання зв’язків та відношень……………………………………... **48**
2.2.1. Типові зв’язки………………………………………………………………. **49**
2.2.2. Розширені зв’язки………………………………………………………….. **56**
2.2.3. Інтерфейси, типи, ролі……………………………………………………... **66**
2.2. Діаграми для моделювання статичних характеристик системи……………... **71**
2.2.1. Основні моделі статиччних представлень системи……………………... **71**
2.2.2. Діаграми класів…………………………………………………………… **76**
2.2.3. Діаграми об’єктів………………………………………………………….. **83**
2.2.4. Компоненти та інтерфейси. Діаграми компонентів……………………….. **86**
3. Основи моделювання поведінки ПЗ……………………………………… 95
3.1. Моделювання варіантів використання………………………………………... **95**
3.1.1. Стратегія розробки варіантів використання як кооперації класів. Поділ, реалізація і специфікація поведінки……………………………………………….. **95**
3.1.2. Діаграми варіантів використання……………………………………………. **102**
3.2. Моделювання взаємодії об’єктів……………………………………………. **107**
3.2.1. Стратегія моделювання взаємодії як обміну поведінками між об’єктами…. **107**
3.2.2. Діаграми взаємодії………………………………………………………… **115**
3.2.3. Діаграми діяльності………………………………………………………... **126**
4. Основи моделювання подій……………………………………………….. 137
4.1. Моделювання подій сигналів………………………………………………... **137**
4.2. Кінцеві автомати. Моделювання поведінки спільно об’єктів……………….. **143**
4.3. Процеси і потоки керування………………………………………………… **156**
4.4. Моделювання систем реального часу………………………………………. **162**
4.5. Діаграми станів……………………………………………………………… **167**
5. Основи моделювання архітектури ПЗ……………………………………. 172
5.1. Поняття про архітектуру та раціональний уніфікований процес……………. **172**
5.2. Моделювання архітектурних зразків……………………………………….. **176**
5.3. Моделювання кооперації……………………………………………………. **184**
5.4. Моделювання пакетів як спосіб організації елементів моделі……………… **192**
6. Перелік використаних джерел……………………………………………. 199

Переднє слово

Метою даного посібника є навчання студентів напряму підготовки 6.050103 – «Програмна інженерія» основним принципам і методам проектування програмного забезпечення (ПЗ) із використанням сучасних інструментальних засобів побудови моделей, що використовуються на різних етапах життєвого циклу ПЗ та характеризують різні властивості та сторони використання для різних учасників процесу створення ПЗ. У результаті засвоєння теоретичного матеріалу, виконання лабораторних робіт, що передбачають створення окремих елементів проектів ПЗ, студенти набувають практичні навички побудови різних моделей ПЗ, основу яких складають UML-діаграми як засоби візуалізації, специфікування, конструювання і документування з допомогою інструментальних засобів проектування ПЗ компанії IBM Rational IBM Software Architect. Одним з найпродуктивніших напрямків у проектуванні ПЗ стало використання візуальних схем, змальованих на папері й екрані комп'ютера, що відображають ті чи інші вигляди ПЗ.

Візуальний підхід до проектування з використанням раціонального уніфікованого підходу та уніфікованої мови моделювання (Unified Modeling Language, UML) дозволяє ефективно вести боротьбу із постійно зростаючою складністю ПЗ, здійснювати їх аналіз, будувати стабільну архітектуру складних програмних систем різного призначення. Система є складною, якщо розробники для складання деякого цілісного уявлення про неї розглядають її не з однією, а з багатьох різних точок зору: з позиції об'єктів і відносин між ними, бізнес- та інших процесів. Крім того, програмне забезпечення розглядається програмними інженерами з позицій глобальних і локальних змінних, однозначно ідентифікованих імен змінних, інкапсуляції частин програмних кодів і багатьох інших точок зору.

Поряд з викладенням теоретичних основ моделювання програмного забезпечення у посібнику розглянуто багато прикладів прикладного застосування на базі UML-діаграм. На цій основі також побудований цикл лабораторних робіт, що проводяться у спеціалізованому комп'ютерному класі кафедри програмної інженерії з встановленим ліцензійним ПЗ програмним забезпеченням Rational IBM Software Architect. Даний інструментарій та низка навчально-методичних матеріалів Лабораторії IBM дозволяє студентам поряд із успішним виконанням лабораторних робіт проходити відповідне тестування на знання та використання окремих продуктів IBM у сертифікаційному центрі та отримувати відповідні сертифікати компанії. Обсяг і зміст лабораторних занять охоплює усі основні теми навчальної програми дисципліни професійно-орієнтованого циклу «Моделювання та аналіз програмного забезпечення» для напряму підготовки 6.0501.03–«Програмна інженерія», визначеному галузевим стандартом та міжнародним куррікулом «Software Curriculum 2004».

Реалізований у посібнику міждисциплінарний підхід робить доцільність його використання і при вивченні інших суміжних професійно-орієнтованих дисциплін, таких, як «Аналіз вимог», «Обєктно-орієнтоване програмування», «Архітектура та проектування програмного забезпечення» та ін., а також при курсовому і дипломному проектуванні та виконанні кваліфікаційних робіт магістра зі спеціальності 8.05010302 «Інженерія програмного забезпечення».

1. Загальні підходи до моделювання програмного забезпечення

Основні питання:

- *Значення моделювання, основні підходи і принципи*
- *Найважливіші моделі програмних систем*
- *Об'єктно-орієнтоване моделювання*
- *Огляд UML, три кроки до розуміння UML*
- *Архітектура програмного забезпечення*
- *Процес розроблення ПЗ*

Софтверна компанія є успішною на ринку, якщо програмне забезпечення (ПЗ) є високоякісним, вчасно розробленим і відповідає вимогам користувачів. Для цього необхідно постійно контактувати з користувачем, з'ясовуючи реальні вимоги до створюваної системи. Якісне ПЗ повинно ґрунтуватися на міцній архітектурі, стійкій до можливих змін і удосконалень.

Для швидкого й ефективного розроблення ПЗ потрібно залучати високопрофесійну команду розробників, вибрати правильні методи та інструменти роботи. Необхідно, щоб процес розроблення був ретельно продуманий і адаптований до мінливих потреб користувачів, технології розроблення та життєвого циклу (ЖЦ) системи.

Основою створення якісного ПЗ є **моделювання,** що дозволяє:

- наочно продемонструвати бажану структуру і поведінку системи;
- здійснювати візуалізації та керування її архітектурою;
- забезпечити краще розуміння створюваної системи, що найчастіше призводить до її спрощення й забезпечує можливість повторного використання;
- мінімізувати ризики.

1.1. Основні підходи та концепції моделювання

Три різні точки зору до розроблення ПЗ. Різні колективи розробників ПЗ, виходячи із своїх можливостей, досвіду роботи, кваліфікації персоналу, фінансової спроможності по-різному підходять до процесу розроблення ПЗ.

Підхід 1. **Принцип собачої будки**. *Процес створення простого за змістом ПЗ для наочності можна продемонструвати на прикладі будівництва собачої будки. До спорудження такого об'єкта можна приступити відразу, маючи в наявності лише купу дощок, жменю цвяхів і молоток. Кілька годин роботи після невеликого попереднього планування – і будка готова, без сторонньої допомоги. Якщо будка вийде досить велика і не буде сильно протікати, собака залишиться задоволеним. У крайньому випадку, ніколи не пізно почати все спочатку, або придбати менш примхливого пса. [2]*

Підхід 2. **Принцип будинку**. *Якщо потрібно побудувати котедж для сім'ї, то, звичайно, можна скористатися тим же набором інструментів, але часу на це піде значно більше і майбутні мешканці виявляться вимогливішими, ніж собака. Перш ніж забити перший цвях, тут слід ретельно все продумати. Необхідно зробити різні креслення майбутньої будівлі. Житло має бути якісним і задовольняти вимоги усіх членів сім'ї, не порушуючи будівельних норм і правил. Потрібно врахувати призначення кожної кімнат,а також такі деталі, як освітлення, опалення і водопровід. На підставі цих планів можна правильно розрахувати необхідний для роботи час і вибрати необхідні будматеріали. Можна побудувати будинок і самому, але вигіднішим є найняти фахівців з відповідним досвідом для виконання ключових робіт. При дотриманні такого плану і кошторису вся сім'я буде задоволена. Якщо ж щось не*

зладиться, то навряд чи варто міняти сім'ю. Краще завчасно врахувати її побажання. [2]

Підхід 3. **Принцип хмарочоса**. *Задавшись метою побудувати хмарочос для офісу, було б зовсім нерозумно братися до роботи, маючи в розпорядженні лише купу дощок, цвяхів і молоток. Оскільки в цьому випадку будуть залучатися значні капіталовкладення, то інвестори зажадають, щоб були враховані усі їх побажання щодо розміру, стилю і форми будівлі. Їх рішення може змінюватись і після початку будівництва. Враховуючи великий ризик прорахунків, задачі планування та вибору колективу проектувальників і будівельників є першочерговими. Для успішної взаємодії та координації робіт потрібно розробити велику кількість креслень і моделей будинку.*

Підібравши потрібні команду та інструментарій і приступивши до реалізації наміченого плану, можна з упевненістю стверджувати, що об'єкт буде побудований з урахуванням усіх вимог майбутніх мешканців. При потребі надалі зводити подібні споруди, потрібно шукати компроміс між побажаннями замовників, можливостями сучасної інженерії та технології будівництва, з розумінням ставитися до команди, не ставити нереальних завдань, здійснювати ефективний менеджмент персоналу тощо. [2]

Такі, на перший погляд, тривіальні ситуації мають місце при розробленні ПЗ. Багато команд, що розробляють ПЗ, прагнуть створити програмний «хмарочос», коли у той же час їхній підхід до справи дуже нагадує будівництво собачої будки. Зіштовхуючись зі зростаючими вимогами до швидкості розроблення ПЗ, команди розробників часто обмежуються тим, що вони по-справжньому вміють робити – *написанням нових рядків програмного коду*, що витрачаються безпосередньо на програмування. Побутує навіть думка, що єдиною відповіддю на труднощі в розробленні ПЗ є інтенсивніша робота з програмування і написання програмного коду, на що й спрямовуються «героїчні» зусилля багатьох команд.

Написаний програмний код не завжди допоможе розв'язати проблему, а проект може бути настільки грандіозним, що навіть збільшення робочого дня на кілька годин виявиться недостатнім для його успішного завершення.

Написання правильного коду мінімального обсягу. Якщо команда дійсно прагне створити ПЗ, що за масштабністю задуму достойне житлового будинку або хмарочосу, то її завдання не повинно зводитися до написання обширного коду. Насправді проблема полягає в тому, щоб написати *правильний* код *мінімального* обсягу. При такому підході розроблення якісного ПЗ зводиться до питань вибору необхідної архітектури, інструментарію і засобів керування процесом. Багато проектів, задуманих за принципом «будки», швидко виростають до розмірів хмарочоса, стаючи жертвою власного успіху. Якщо такий ріст не враховано в архітектурі розроблюваного ПЗ, технологічному процесі або при виборі інструментарію, то неминуче настає час, коли «будка», яка виросла до розмірів величезного «хмарочоса», обвалиться від власної ваги. Але якщо зруйнується «будка», те це розлютить тільки «собаку», а якщо завалиться «хмарочос», це зачепить матеріальні інтереси значно серйозніших клієнтів – усіх його інвесторів і орендарів.

Невдалі проекти терплять крах із найрізноманітніших причин. І навпаки, успішні проекти, як правило, мають багато спільного. Хоча успіх програмного проекту забезпечується множиною різних складових, одним із головних чинників є застосування моделювання.

Моделювання програмного забезпечення є усталеною і загальноприйнятою інженерною методикою. Розробляють архітектурні моделі будинків, аби допомогти їхнім майбутнім мешканцям у всіх деталях уявити готовий об'єкт. Також застосовують і

математичне моделювання об'єктів, що полягає в урахуванні впливу різних фізичних навантажень (*сильного вітру, вібрацій автостради, землетрусу тощо*).

Моделювання застосовується не тільки в будівництві. Неможливо налагодити випуск нових літаків чи автомобілів, не випробувавши свій проект на моделях: від комп'ютерних моделей і креслень до фізичних моделей в аеродинамічній трубі та повномасштабних прототипів. Електричні прилади від мікропроцесорів до телефонних комутаторів також вимагають моделювання для кращого розуміння системи й організації спілкування розробників. Навіть у кінематографії успіх фільму неможливий без попередньо написаного сценарію (*своєрідна форма моделі*).

Що таке модель системи? Модель є спрощений вигляд реальності, креслення системи, куди може входити як детальний план, так і більш абстрактний вигляд, так би мовити, «з висоти пташиного польоту». Якісна модель завжди включає елементи, які суттєво впливають на результат і не включає ті, які малозначні на даному рівні абстракції. [4]

Кожна система може бути описана з різних точок зору, для чого використовуються різні моделі, кожна з яких є семантично замкненою абстракцією системи. Модель може бути структурною, що підкреслює організацію системи, або моделлю поведінки, що відображає її динаміку. [2,3,5]

Мета моделювання ПЗ. Модель ПЗ будується для того, щоб краще розуміти розроблювану систему. Моделювання дозволяє вирішити чотири різні завдання:

1. Візуалізувати систему в її поточному або бажаному для замовника стані.
2. Описати структуру або поведінку системи.
3. Отримати шаблон, що дозволяє сконструювати систему.
4. Документувати прийняті рішення, використовуючи отримані моделі.

Чим більша і складніша система, тим більшого значення набуває моделювання при її розробленні. Без моделі складну систему неможливо сприйняти як одне ціле (*навіть програмний еквівалент собачої будки суттєво виграє від застосування моделювання*). Людське сприйняття складних сутностей є обмеженим. Моделювання системи чи об'єкта дозволяє звузити проблему, зосередивши увагу в кожен момент тільки на певних її аспектах (принципу «*розділяй і володарюй»*). Складне завдання завжди легше вирішити, якщо розділити його на менші. Моделювання підсилює можливості людського інтелекту. Правильно обрана модель дозволяє створювати проекти на вищих рівнях абстракції.

Неформальне і формальне моделювання. Навіть для найпростішого проекту застосовується моделювання хоча б неформально. Для візуалізації частини системи її проектувальник може намалювати щось на дошці чи на клаптику паперу (*моделювання на серветці, обговорення за чашкою кави*). Команда може застосовувати і інші подібні неформальні підходи, використання яких є виправданим при певних умовах. Однак такі неформальні моделі часто створюються для одноразового застосування і не забезпечують загальної мови, яка б була зрозумілою іншим учасникам проекту. В будівельній інженерії існує загальна і зрозуміла для всіх формалізована мова креслень. Є загальноприйняті мови в електротехніці, математичному моделюванні. Команда розробників ПЗ також може мати суттєву користь при використанні певної формалізованої загальної мови моделювання ПЗ.

Від моделювання виграє будь-який проект. Моделювання допоможе команді краще уявити план системи, а, отже, виконати проект швидше й створити саме те, що передбачалося задумом. Чим складніший проект, тим більша ймовірність, що через

відсутність моделювання він потерпить невдачу завчасно, або ж буде створено не те, що потрібно. Усі системи з часом ускладнюються. Нехтуючи моделюванням на самому початку створення системи, розробник ризикує наразитися на низку небезпек, пов'язаних з природним процесом ускладнення системи тощо. [1,2,10]

1.2. Принципи моделювання

Моделювання має багату історію застосування в усіх галузях інженерії та наукових дослідженнях. Тривалий досвід його використання дозволив сформулювати такі основні принципи моделювання.

Принцип 1. **Вибір моделі впливає на підхід до розв'язання проблеми і на те, як буде виглядати це рішення.** Правильно вибрана модель висвітить найпідступніші проблеми розроблення й дозволить проникнути в саму суть завдання, що при іншому підході було б просто неможливо. Неправильна модель може завести розробника у безвихідь, оскільки основна увага буде приділена несуттєвим питанням. (*Правильна постановка – це дві третіх її розв'язання /Коші/*).

Для прикладу, потрібно розв'язати задачу щодо кінетики процесів молекулярного транспорту в нанопористих середовищах. Такі проблеми, як поглинання чи адсорбція мікропорами окремих газів потребують проведення великої кількості складних і дорогих нанофізичних експериментів. Але варто лише використати модель, побудовану на рівняннях переносу Ленгмюра-Хіншелвуда, реалізовану у вигляді відповідного моделюючого програмного комплексу, то можна отримати просторово-розподілену картину градієнтних полів у макро- і мікропорах середовища тощо. Аналогічно, при проектуванні будинку чи телетрансляційної вежі потрібно знати поведінку об'єкта при сильному вітрі. Побудувавши макети таких об'єктів і випробувавши їх в аеродинамічній трубі, можна довідатися багато цікавого з цього приводу, хоча дослідження були проведені на їх зменшених копіях, а не на самих об'єктах. Імітаційне моделювання з використанням математичної моделі дає додаткову інформацію і, можливо, дозволить пробробити більше різних сценаріїв, ніж це вдалося б при роботі з фізичним макетом. Ретельно вивчивши поведінку об'єкта на подібних моделях, можна отримати детальне бачення поведінки розроблюваної системи в реальних умовах, навіть у тих, які неможливо або важко відобразити в реальності (різні екстремальні умови та ситуації тощо).[2]

Різні точки зору призводять до створення різних систем. Світогляд розробника ПЗ залежить від обраної моделі. Розробники баз даних (БД) основну увагу приділять UML-моделям «*сутність – зв'язок*», де поведінка інкапсульована в процедурах збереження та сховищах. Аналітик структурного підходу створив би модель, у центрі якої є алгоритми і передача даних від одного процесу до іншого тощо. Результатом роботи розробника об'єктно-орієнтованої парадигми проектування буде система, архітектура якої заснована на *множині класів і зразках взаємодії*, що визначають, кооперації цих класів у реалізації певних сценаріїв. Кожен із цих варіантів може виявитися придатним для конкретного застосування і методики розроблення, хоча досвід підказує, що об'єктно-орієнтована точка зору є найефективнішою при створенні гнучких архітектур, навіть для великих БД чи складних математичних розрахунків. Різні точки зору призводять до створення різних систем зі своїми перевагами й недоліками.

Принцип 2. **Кожна модель може бути представлена з різним ступенем точності.** При будівництві хмарочоса може виникнути необхідність показати його з висоти пташиного польоту, наприклад, щоб із проектом могли ознайомитися інвестори. В інших випадках, навпаки, потрібен детальний опис, наприклад, щоб показати яку-

небудь складну конфігурацію каналу чи незвичайний елемент конструкції сонячної батареї.

Кращою моделлю є та, що дозволяє вибрати необхідний рівень деталізації ПЗ. Іноді проста й нашвидкуруч створена модель програмного інтерфейсу є найприйнятнішим варіантом. В інших випадках доводиться працювати на рівні бітів (*специфікація міжсистемних інтерфейсів тощо*). У кожному випадку кращою моделлю буде та, яка дозволяє вибрати рівень деталізації залежно від того, хто і з якою метою на неї дивиться. Для аналітика або користувача найбільший інтерес представляє питання «*що робити*», а для розробника – «*як зробити*». В обох випадках необхідна можливість розглядати систему на різних рівнях деталізації в різний час.[2,7]

Принцип 3. **Кращі моделі ті, які ближче до реальності**. Фізична модель будинку, що поводиться не так, як виготовлена з реальних матеріалів, має лише обмежену цінність. Математична модель літака, для якої передбачаються ідеальні умови роботи і бездоганне складання, може й не мати деяких характеристик, властивих справжньому виробу, що в ряді випадків призводить до фатальних наслідків. Розроблювані моделі мають певним чином співвідноситися з реальністю. Оскільки модель завжди спрощує реальність, завдання в тому, щоб це спрощення не спричинило якихось істотних втрат.

При об'єктно-орієнтованому підході створення ПЗ можна об'єднати всі майже незалежні вигляди системи в єдине семантичне ціле. «Ахіллесова п'ята» структурного аналізу – невідповідність прийнятої в ньому моделі й моделі системного проекту. Якщо цей розрив не буде усунутий, то поведінка створеної системи з часом буде все більше й більше відрізнятися від задуманого. [1,2,3]

Принцип 4. **Не можна обмежуватися створенням тільки однієї моделі**. Найкращий підхід при розробленні будь-якої нетривіальної системи – використовувати сукупність кількох моделей, майже незалежних одна від одної. При конструюванні будинку ніякий окремий комплект креслень не допоможе прояснити всі деталі до кінця. Знадобляться як мінімум плани кожного поверху, вигляди у розрізі, схеми електропроводів, центрального опалення, водопроводу тощо. [1,2]

Моделі створюються й досліджуються окремо, хоч є взаємозалежними. Тут ключовим моментом є визначення «майже незалежні». Можна вивчати тільки схеми електропроводів проектованого будинку, але при цьому вони мають бути накладені на плани поверхів чи навіть розглянуті разом із прокладкою труб на схемі водопостачання.

Вигляди архітектурного розуміння ПЗ. Такий підхід справедливий і в об'єктно-орієнтованих програмних системах. Для розуміння архітектури подібної системи потрібно кілька взаємодоповнюючих виглядів: з погляду варіантів використання (*щоб виявити вимоги до системи*), з погляду проектування (*щоб побудувати словник предметної області й області рішень*), з погляду взаємодій (*щоб змоделювати взаємодії між частинами системи, системою в цілому і середовищем її функціонування*), з погляду реалізації (*що дозволяє розглянути фізичну реалізацію системи*) і з погляду розміщення (*що допомагає зосередитися на питаннях системного проектування*). Кожен із перерахованих виглядів має цілу множину структурних аспектів поведінки, які у своїй сукупності становлять детальне креслення програмної системи.[11]

Залежно від природи системи, деякі моделі можуть бути важливіші за інші. При створенні систем для опрацювання великих обсягів даних важливішими є статичні моделі. У застосуваннях, орієнтованих на інтерактивну роботу користувача, на перший план виходять статичні й динамічні вигляди варіантів використання. У системах реального часу істотнішими будуть вигляди з позиції динамічних процесів. Нарешті, у

розподілених системах, таких, як WEB-застосування, основну увагу потрібно приділяти моделям реалізації й розміщення.

1.3. Об'єктне моделювання

Інженери-будівельники створюють велику кількість моделей різних виглядів (структурні моделі - для візуалізації і специфікування частини системи, їх співвідношення між собою; динамічні моделі для вивчення поведінки конструкції при землетрусі). Ці два типи моделей відрізняються організацією і тим, на що, в першу чергу, звертається увага при проектуванні. При розробленні ПЗ також існує кілька підходів до моделювання. Найважливіші з них – алгоритмічний і об'єктно-орієнтований.

Алгоритмічний підхід [9] підхід до створення ПЗ. Основним будівельним блоком є *процедура* і *функція*, а увага приділяється насамперед питанням передавання керування й декомпозиції більших алгоритмів на менші. Складні системи *не зовсім легко адаптуються*. При зміні вимог або збільшенні розміру додатка супроводжувати їх стає складніше. [14]

Об'єктно-орієнтований підхід в якості основного будівельного блоку має об'єкт або клас. В найзагальнішому сенсі **об'єкт** – це сутність, що отримується зі **словника предметної області** чи області рішень, а **клас** – опис множини однотипних об'єктів. Кожен об'єкт відрізняється своєю **ідентифікацією** (*від інших об'єктів*), **станом** (*з об'єктом пов'язані деякі дані*) і **поведінкою** (*з ним можна щось робити або він сам може щось робити з іншими об'єктами*). [1,2,3,11]

ООП до розроблення ПЗ найбільш широко використовуються, що демонструє свою придатність при побудові систем довільного розміру і складності для різних галузей. Сучасні мови програмування, інструментальні засоби й операційні системи є тією чи іншою мірою об'єктно-орієнтованими, що дозволяють дивитись на світ у термінах об'єктів («*Щоб побачити життя в рожевому слід подивитись на світ через бокал де Шамбертен» /Атос, граф де Ля Фер/*). Об'єктно-орієнтовані методи розроблення покладено в основу ідеології складання систем з окремих компонентів (технології J2EE, .NET). [1,2]

Вибравши об'єктно-орієнтовану точку зору розроблення ПЗ, необхідно водночас відповісти на запитання «Яка повинна структура бути якісної об'єктно-орієнтованої архітектури?» «Які артефакти повинні бути створені в процесі роботи над проектом?» «Хто повинен створювати їх?» «Як оцінити результат?»

Полегшить відповіді на ці запитання застосування **уніфіковананої мови моделювання** UML, основним призначенням якої є візуалізація, специфікація, конструювання й документування (ВСКД) об'єктно-орієнтованих систем.

1.4. Введення в UML

Отже, уніфікована мова моделювання (Unified Modeling Language - UML) – це стандартний інструмент для розроблення «креслень» ПЗ, що використовується для візуалізації, специфікації, конструювання й документування артефактів ПЗ. UML підходить для моделювання будь-яких систем – від складних інформаційних систем до розподілених Web-додатків і вбудованих систем реального часу. Це дуже виразна мова, що дозволяє розглянути систему з усіх точок зору, що має відношення до її розроблення і сприяє наступному її розгортанню. Незважаючи на багатство засобів, UML є простою для розуміння й застосування. Вивчення ефективного використання UML починається з формування концептуальної моделі мови, основу якої складають три основні елементи:

базові будівельні блоки UML, правила, що визначають, як ці блоки можуть сполучатися між собою, та деякі загальні механізми мови.

UML є однією зі складових процесу розроблення ПЗ. Хоча UML не залежить від процесів, що моделюються, найкраще її застосовувати у випадках, коли процес моделювання орієнтований на застосування варіантів використання (ВВ), сконцентрований на архітектурі системи, є ітеративним і покроковим. [1,3,6]

UML є мовою для ВСКД артефактів програмних систем. [1,11,12]UML як мова моделювання включає словник і комунікативні правила комбінування слів, які до нього входять, орієнтована на концептуальне і фізичне зображення системи. UML – стандартний засіб створення «креслень» ПЗ. Жодна модель системи не є абсолютно достатньою. Щоб зрозуміти більшість систем, потрібна ціла множина взаємозалежних моделей. Відносно ПЗ мова моделювання повинна володіти засобами, з допомогою яких можна описати архітектуру системи з різних точок зору впродовж усього ЖЦ її розроблення. Словник і правила UML пояснюють, як створювати і читати добре визначені моделі, але нічого не кажуть у яких випадках які саме моделі потрібно створювати. Це завдання всього процесу розроблення ПЗ. Добре організований процес повинен сам підказати, які будуть потрібні робочі продукти, які ресурси знадобляться для їхнього створення й керування ними, як їх використовувати для оцінювання виконаної роботи й керування проектом у цілому.

UML – мова візуалізації.[1,2] Деякі речі найкраще виражаються безпосередньо в коді мовою програмування: текст програм – найпряміший і найкоротший спосіб написання виразів і алгоритмів. *Думати – значить, кодувати*. Реалізації проекту майже еквівалентні написанню коду (*думки окремих програмістів*). Але й при цьому програміст насправді займається моделюванням, хоча й робить це неформально (*намітки ідей на серветці тощо*). Однак при цьому виникають деякі **проблеми розроблення.** *По-перше*, обговорення таких концептуальних моделей з іншими учасниками розроблення може супроводжуватися помилками й непорозуміннями, якщо тільки всі учасники дискусії не розмовляють однією мовою. *По-друге*, ряд речей розроблення ПЗ важко моделювати лише засобами текстових мов програмування (*Ієрархію класів можна зрозуміти, вивчивши код кожного із класів в ієрархії. Однак вивчення коду системи не дозволяє скласти цілісне уявлення про фізичний розподіл і можливі міграції об'єктів тощо*). *По-третє*, якщо розробник, який писав цей код, ніколи не втілював у ньому моделі, що існували в його голові, то інформація про них може бути загублена назавжди.

UML як явна модель полегшує спілкування. Опис моделей мовою UML дозволяє розв'язати третю проблему: явна модель полегшує спілкування. Деякі речі краще моделювати в тексті, інші – графічно. Насправді у більшості систем існують структури, які неможливо виразити мовою програмування. UML – графічна мова, що дозволяє розв'язати другу з описаних вище проблем. UML – щось більше, ніж просто набір графічних символів, кожен з яких має чітку певну семантику. І це означає, що один розробник може описати модель мовою UML, а інший може її однозначно інтерпретувати її, що вирішує першу зі згаданих проблем (комунікації). [1,2,7,11]

UML – мова специфікації. У даному контексті специфікація – це побудова точних, недвозначних і повних моделей. Зокрема, UML дозволяє специфікувати усі важливі розв'язки, що стосуються аналізу, дизайну й реалізації, прийнятих у процесі розроблення і упровадження програмних систем.

UML – мова конструювання. [2,12] UML не є візуальною мовою програмування, але її моделі можуть бути безпосередньо асоційовані з різними мовами програмування. А це означає, що існує можливість відобразити UML-модель на таку мову, як C++ , Java, C#, а за необхідності навіть на таблиці реляційної бази даних або об'єкти, що зберігаються в об'єктно-орієнтованій базі даних. Ті речі, які простіше виразити графічно, виражаються мовою UML, а ті, що легше виразити у вигляді тексту, – мовою програмування.

Відображення моделі мовою програмування дозволяє здійснити **пряме проектування** (forward engineering) – ґенерацію коду мовою програмування з моделі UML. Зворотне також можливо: відновити модель UML на основі існуючої програмної реалізації. У **зворотному проектуванні** (reverse engineering) інформація в реалізації може губитися при переході від моделі до коду. Тому зворотне проектування, що виконується інструментальними засобами, все ж вимагає певного втручання людини. Комбінація цих двох засобів – прямого і зворотного проектування – забезпечує можливість роботи як із графічним, так і з текстовим виглядом моделі; при цьому забезпечується погодженість між ними.

Наостанок, до прямого відображення в мовах програмування, UML, завдяки своїй виразності й однозначності, дозволяє безпосередньо виконувати моделі, імітувати поведінку проектованих систем, а також управляти діючими системами.

UML – мова документування. [2,11,12] Успішні компанії, що спеціалізуються на програмному забезпеченні, окрім програмного коду, що виконується, створюють також інші артефакти, включаючи такі (але не обмежуючись ними):

- вимоги;
- архітектуру;
- проектні розв'язки (дизайн);
- вихідний код;
- проектні плани;
- тести;
- прототипи;
- релізи (версії).

Залежно від рівня культури розроблення, прийнятого в компанії, деякі із цих продуктів виражаються формальніше, ніж інші. Перераховані продукти не тільки доставляються як складові проектів; вони необхідні для керування, оцінювання результатів і взаємодії в процесі розробленні системи й після її впровадження.

UML призначена для документування архітектури системи й усіх її деталей. Крім того, *це мова для вираження вимог* до системи й опису тестів. Вона підходить для моделювання робіт на етапі проектування й керування версіями.

Галузі використання UML як засобу моделювання й розроблення програмних систем у різних галузях (корпоративні інформаційні системи, телекомунікації, транспорт, авіація і космонавтика, наука, освіта, розподілені Web-сервіси тощо). Застосування UML не обмежене моделюванням ПЗ, а може застосовуватись і до непрограмних систем.

1.5. Концептуальна модель UML

Концептуальна модель мови включає три основних елементи: будівельні блоки мови, правила, що визначають їх комбінації й загальні для всієї мови механізмів. У міру

набуття досвіду у використанні UML можна будувати моделі, використовуючи розширені засоби.

Будівельні блоки UML. Словник UML включає три види будівельних блоків:

1. Сутності.
2. Зв'язки.
3. Діаграми.

Сутності (things) – це абстракції, які є основними елементами моделі, **зв'язки** (relationships) з'єднують їх між собою, а **діаграми** (diagrams) ґрупують набори сутностей, що являють спільний інтерес.

Є чотири види сутностей UML:

1. Структурні.
2. Сутності поведінки.
3. Сутності ґрупування.
4. Анотаційні сутності.

Усі вони є базовими об'єктно-орієнтованими будівельними блоками моделей UML, що використовуються для описування коректних моделей.

Структурні сутності – «іменники» у моделях UML. В основному, це статичні частини моделі, що представляють концептуальні або фізичні елементи системи.

Клас (class) – це опис набору об'єктів з однаковими атрибутами, операціями, зв'язками й семантикою. Клас реалізує один або кілька інтерфейсів. Графічно клас зображується у вигляді прямокутника, що, як правило, включає ім'я, атрибути й операції (рис. 1.5.1). [1,2,3,11,12]

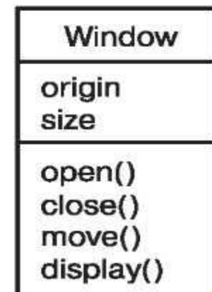


Рис. 1.5.1. Класи

Інтерфейс (interface) – набір операцій, що специфікує сервіс (набір послуг) класу або компоненти. Інтерфейс описує видиму ззовні поведінку елемента. Він може представляти повну поведінку класу або компоненти, або тільки частину такої поведінки. Інтерфейс визначає набір специфікацій операцій (тобто їхню *сигнатуру*), але ніколи не визначає деталі їх реалізації. Оголошення інтерфейсу зображується як клас із ключовим словом (стереотипом) «interface» над його іменем; атрибути несуттєві, за винятком іноді показуваних констант. Інтерфейс, однак, не часто існує сам по собі.

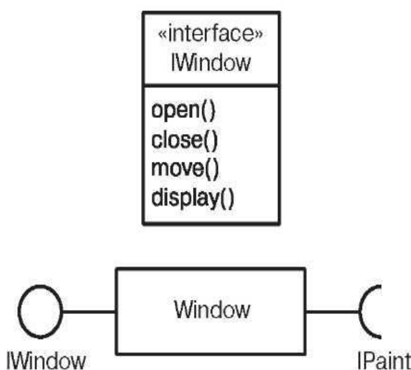


Рис. 1.5.2. Інтерфейси

Інтерфейс, що надається класом для зовнішнього світу, зображується у вигляді маленького *кола*, з'єднаного лінією з рамкою класу.

Інтерфейс, що вимагається класом від деякого іншого класу, представлений маленьким *півколом*, з'єднаним з рамкою класу лінією (рис. 1.5.2).

Кооперація (collaboration) – визначає взаємодію між сутностями (класами та об'єктами) і являє собою сукупність ролей та інших елементів, які функціонують разом, забезпечуючи деяку спільну поведінку, що є дещо більшим, ніж просто сума поведінок окремих елементів. [1,2,13] Кооперації

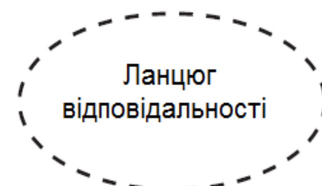


Рис. 1.5.3. Кооперації

мають як структурний, так і поведінковий аспект. Конкретний клас або об'єкт може брати участь у кількох коопераціях. Останні, у такий спосіб, є реалізацією **зразків** (patterns), складових системи. Кооперація зображується у вигляді еліпса, нарисованого пунктирною лінією, що іноді включає в себе лише її ім'я (рис. 1.5.3).

Варіант використання (use case) – опис послідовності дій, які виконує система, що приносять значний результат для конкретного діючого суб'єкта або актанта структурування поведінкових сутностей моделі й реалізуються за допомогою кооперацій. Графічно варіант використання (ВВ) представлений еліпсом, намальованим суцільний лінією, що містить у собі зазвичай тільки ім'я (рис. 1.5.4). [1,2,4,12]

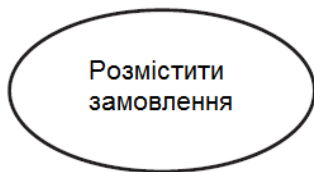


Рис. 1.5.4. Варіант використання

Три сутності, що залишилися – активні класи, компоненти й вузли (nodes) – є подібними до класів; це свідчить про те, що вони також описують сукупності об'єктів, які мають ті ж самі атрибути, операції, зв'язки й семантику. Однак ці три поняття достатьою мірою відрізняються один від одного та від класів, і є необхідними для моделювання певних аспектів об'єктно-орієнтованих систем, тому потребують спеціального розгляду.

Активний клас (active class) – клас, об'єкти якого задіяні в один або кілька процесів або ниток (threads) і, таким чином, можуть ініціювати керуючі впливи. [2,12] Активний клас у всьому подібний до простого класу, за винятком того, що його об'єкти є елементами, поведінка яких здійснюється паралельно з поведінкою інших.

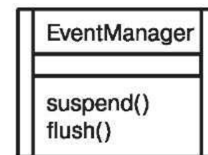


Рис. 1.5.5. Активні класи

Активний клас зображується як клас із *подвійними бічними лініями*; містить у собі ім'я, атрибути й операції (рис. 1.5.5).

Компонента (component) – модульна частина системи, що приховує свою реалізацію за набором зовнішніх інтерфейсів. [2] Компоненти системи, що розділяють загальні інтерфейси, можуть заміщати один одного, зберігаючи при цьому однакову логічну поведінку. Реалізація компоненти може бути виражена об'єднанням **частин** і **конекторів**. [1,3,4,11,12] При цьому частини можуть містити в собі дрібніші компоненти. Графічно компонента представлена як клас зі спеціальною піктограмою в правому верхньому куті (рис. 1.5.6).

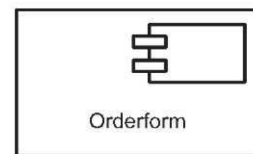


Рис. 1.5.6. Компоненти

Артефакт (artifact) – це фізична частина, що заміщається, система, що несе фізичну інформацію («біти»). [2,3,10] У системі ви можете зустріти різні види артефактів, таких, як файли вихідного коду, що виконують програми й скрипти. Зазвичай артефакт є фізичним пакетом із вихідним або виконавчим кодом. Зображується як прямокутник, постачений ключовим словом «artifact», розташованим над його іменем (рис. 1.5.7).

Рис 1.5.7. Артефакти

Вузол (node) – це фізичний елемент, який існує під час виконання і представляє обчислювальний ресурс, що зазвичай має деяку пам'ять і обчислювальні можливості. [1,2,6] Набір

Рис.1.5.8. Вузли

компонентів може перебувати на вузлі, а також мігрувати з одного вузла на інший. Вузол зображується у вигляді куба, який, як правило, містить лише його ім'я (рис. 1.5.8).

Артефакти і вузли є фізичними сутностями на відміну від попередніх п'яти, що відносяться до логічних чи концептуальних сутностей.

Усі ці елементи – класи, інтерфейси, кооперації, варіанти використання (ВВ), активні класи, компоненти, артефакти й вузли є базовими структурними сутностями, які можуть бути включені в UML-модель. Існують також різні варіації: дійові особи (actors), **сигнали** й **утиліти** (різновид класів), **процеси** й **потоки** (*різновиди активних класів*), додатки, документи, файли, бібліотеки, сторінки й таблиці (різновиди артефактів).

Поведінкові сутності (Behavioral things) – динамічні частини UML-моделей. Це – «дієслова» моделей, що представляють поведінку в часі й просторі. [2] Усього існує три основні види поведінкових сутностей.

Взаємодія (Interaction) – це поведінка, яка полягає в обміні повідомленнями (Messages) між наборами об'єктів або ролей у рамках певного контексту для досягнення деякої мети. Поведінка сукупності об'єктів або індивідуальна операція можуть бути виражені взаємодією. [1,2,6] Взаємодія включає множину інших елементів, таких, як повідомлення, дії (actions) і конектори (з'єднання між об'єктами). Повідомлення зображується у вигляді *лінії зі стрілкою*, майже завжди супроводжуваною *іменем операції* (рис. 1.5.9).

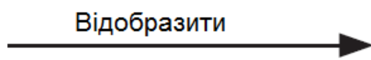


Рис.1.5.9. Повідомлення

Автомат (state machine) – це *алгоритм поведінки*, що, визначає послідовність станів, через які об'єкт проходить протягом життєвого циклу у відповідь на події разом з його реакцією на ці події. [1,2] Поведінка індивідуального класу або кооперації класів може бути описана в термінах автомата. Автомат містить у собі множину інших елементів: стани, переходи (*з одного стану в інший*), **події** (*сутності, які ініціюють переходи*), а також дії (реакції на переходи). Графічно стан представлений прямокутником із закругленими кутами, зазвичай із вказівкою імені й підстанів, якщо такі є (рис. 1.5.10).

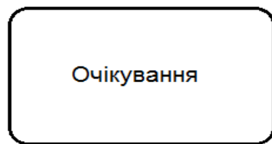


Рис. 1.5.10 Стан

Діяльність (activity) – поведінкова сутність, що специфікує послідовність кроків процесу обчислень. У взаємодії увага зосереджена на наборі взаємодіючих об'єктів, в автоматі – на ЖЦ одного об'єкта; для діяльності ж у центрі уваги – послідовність кроків безвідносно до об'єктів, що виконують кожен крок. Окремий *крок діяльності* називають **дією** (action). Зображується вона у вигляді прямокутника із заокругленими кутами, що включає ім'я, яке відображає його призначення (рис. 1.5.11). Стани і дії можна відрізнити за контекстами. [2]

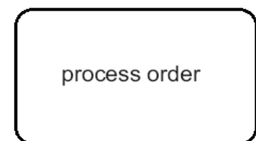


Рис. 1.5.11. Дії

Ці три елементи – взаємодії, автомати й діяльності; є базовими сутностями поведінки, які можна включити в UML-модель. Семантично ці елементи звичайно пов'язані з різними структурними елементами, у першу чергу, класами, коопераціями й об'єктами.

Сутності, які ґрупують, – організаційна частина моделей UML. Це – «ящики», по яких можна розкласти модель. Головна із сутностей, які ґрупують, – пакет. [1,2]

Пакет (package) – механізм загального призначення для організації проектних розв'язків, який упорядковує конструкції реалізації. [1,3] Структурні сутності, поведінкові сутності, сутності, що групують, та ін. можуть бути вміщені в пакет. На відміну від компонентів (*існуючих тільки під час виконання*), пакети повністю концептуальні, тобто існують лише на етапі розроблення. Пакет зображується у вигляді папки із закладкою, зазвичай тільки із вказівкою імені, але іноді й у місту (рис. 1.5.12).

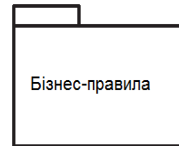


Рис. 1.5.12 Пакети

Пакети – основна сутність організації UML-модель. Існують і такі варіації, як **каркаси** (frameworks), моделі та підсистеми (різновид пакетів).

Сутності анотування – складові, що пояснюють UML-моделі, або коментарі, які можна застосувати для описування, виділення й пояснення будь-якого елемента моделі. Основна із сутностей анотування – **примітка** (note). [2,4] Це – простий символ, що служить для описування обмежень і коментарів, які ставляться до елемента або набору елементів. Графічно представлений прямокутником із загнутим кутом, всередині міститься текстовий або графічний коментар (рис. 1.5.13).

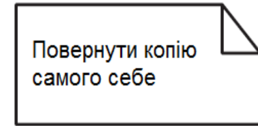


Рис.1.5.13.Примітки

Цей елемент є базовою сутністю анотування UML-моделі, що використовується для доповнення діаграм обмеженнями або коментарями, які найкраще виражаються у вигляді формального або неформального тексту. Існують також різні варіації цього елемента, такі, як вимоги, що специфікують деяку бажану поведінку з погляду ззовні стосовно моделі.

Зв'язки в UML. Існує чотири типи зв'язків у UML:

1. Залежність.
2. Асоціація.
3. Узагальнення.
4. Реалізація.

Ці зв'язки є базовими будівельними блоками для описування відношень у UML, що використовуються для розроблення добре узгоджених моделей.

Залежність (dependence) – семантично є зв'язок між двома елементами моделі, у якому зміна одного елемента (незалежного) може призвести до зміни семантики іншого елемента (залежного). Графічно представлена пунктирною лінією зі стрілкою (як правило); може бути постачена міткою (рис. 1.5.14). [1,2,3]

Рис. 1.5.14. Залежності

Асоціація (association) – структурний зв'язок між класами, який описує набір зв'язків, що існують між об'єктами, – екземплярами класів. **Агреґація** (aggregation) – особливий різновид асоціації, що представляє структурний зв'язок цілого з його частинами. Зображується суцільною лінією, іноді зі стрілкою, що визначає навігацію; іноді позначена міткою, часто містить інші позначки, такі, як потужність і кінцеві імена (рис. 1.5.15). [2,4]

Рис. 1.5.15.Асоціації

Узагальнення (generalization) – виражає спеціалізацію або узагальнення, у якому спеціалізований елемент (нащадок) будується за специфікаціями узагальненого елемента (батька). Нащадок розділяє структуру й поведінку батька. Графічно узагальнення представлене у вигляді

Рис. 1.5.16. Узагальнення

суцільної лінії з порожньою стрілкою, що вказує на батька (рис. 1.5.16). [2,4,7]

Реалізація (realization) – семантичний зв’язок між класифікаторами, коли один із них специфікує угоду, яку інший зобов’язаний дотримувати. [2,5] Зв’язки реалізації зустрічаються у двох випадках: між інтерфейсами і класами або компонентами, які реалізують ці інтерфейси, а також між варіантами використання, що й реалізуються їхніми коопераціями. Зв’язок реалізації в графічному виконанні – гібрид зв’язків узагальнення й залежності (рис. 1.5.17).

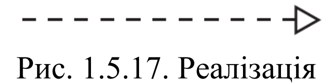


Рис. 1.5.17. Реалізація

Ці чотири елементи представляють основні сутності відносин, які можуть бути включені в UML-моделі. Є також різні їхні варіації: уточнення (refinements), слід (trace), включення (include) й розширення (extend).

1.6. Діаграми UML

Діаграма – графічне уявлення набору елементів, найчастіше зображеного у вигляді зв’язного графа вершин (сутностей) і шляхів (відношень/зв’язків). Діаграми використовуються для візуалізації системи з різних точок зору, тому окрема діаграма – це проекція системи. Діаграма дає певний згорнутий погляд про елементи системи. Один і той же елемент (сутність/відношення) може з’являтися або в усіх діаграмах, або в деяких. Теоретично діаграма може містити в собі будь-яку комбінацію сутностей і відношень/зв’язків. На практиці використовується лише невелике число загальних комбінацій. Перші п’ять із цих діаграм найчастіше застосовуються для побудови **архітектурних виглядів** програмних систем. Виходячи з цього, подаємо короткий опис 13 видів найчастіше використовуваних для розроблення моделей та архітектурних виглядів ПЗ UML -діаграм:

1. Діаграма класів.
2. Діаграма об’єктів.
3. Діаграма компонентів.
4. Діаграма складової структури.
5. Діаграма варіантів використання.
6. Діаграма послідовності.
7. Діаграма комунікації.
8. Діаграма станів.
9. Діаграма діяльності.
10. Діаграма розміщення.
11. Діаграма пакетів.
12. Тимчасова діаграма.
13. Діаграма огляду взаємодій.

Діаграма класів (Class diagram) показує набір класів, інтерфейсів і кооперацій, а також їх зв’язки. Діаграми цього виду найчастіше використовуються для моделювання об’єктно-орієнтованих систем і призначені для статичного зображення системи. Діаграми класів, що включають активні класи, описують статичний вигляд процесів системи. **Діаграми компонентів** є різновидом діаграм класів.

Діаграма об’єктів (Object diagram) показує набір об’єктів і їх зв’язки. Діаграми об’єктів представляють статичні копії станів екземплярів сутностей, описаних у діаграмі класів. Також представляють статичний вигляд проектування або процесів системи (як і діаграми класів, але з погляду реальних або прототипних ситуацій).

Діаграма компонентів (Component diagram) демонструє інкапсульовані класи та їх інтерфейси, порти і внутрішні структури, що складаються із вкладених компонентів і коннекторів. Діаграми компонентів описують статичний вигляд з точки зору реалізації системи. Вони важливі при побудові складних систем. Поряд з діаграмою компонетів UML дозволяє створювати **діаграму складеної структури** (composite structure diagram), що застосовуються до будь-якого класу.

Діаграма варіантів використання (Use case diagram) демонструє набір ВВ і діючих осіб-акторів (актантів) (які є спеціальним видом класів), а також їх зв'язки. Вони описують статичний вигляд ВВ системи і важливі для організації та моделювання поведінки системи.

Діаграма взаємодії (Interaction diagram) показує взаємодію, що складається з набору об'єктів і виконуваних ними **ролей**, включаючи повідомлення, які можуть передаватися між ними. Вони призначені опису динамічного стану системи. Діаграми взаємодії поділяються на діаграми послідовностей і діаграми комунікацій.

Діаграма послідовності (sequence diagram) – різновид діаграми взаємодії, що показує тимчасову послідовність повідомлень.

Діаграма комунікацій (communication diagram) – різновид діаграми взаємодії, що показує структурну організацію об'єктів або ролей, що відправляють і приймають повідомлення. Діаграми послідовності й діаграми комунікації (діаграми кооперації) представляють схожі базові концепції, але з різних точок зору. Діаграми послідовності описують тимчасову послідовність, а комунікаційні діаграми – структури даних, через які проходить потік повідомлень.

Діаграма станів (State diagram) – **автомат** (state machine), що включає в себе стани, переходи, події і діяльності. Діаграми станів описують динамічний вигляд об'єкта і є важливі для моделювання поведінки інтерфейсів, класів або кооперацій та підкреслюють подвійно-залежну поведінку об'єкта, що особливо зручно для моделювання реактивних систем.

Діаграма діяльності (Activity diagram) показує структуру процесу або інших обчислень як покроковий потік керування і даних. Діаграми діяльності описують динамічний вигляд системи, є важливі при моделюванні функцій системи, виділяючи потік керування між об'єктами.

Діаграма розміщення (Deployment diagram) показує конфігурацію вузлів-процесорів, а також розташовані на них компоненти. Діаграми розміщення дають статичний вигляд розміщення архітектури. Вузли містять один або кілька артефактів.

Діаграма артефактів (Artifact diagram) показує фізичний склад комп'ютерної системи. Артефакти представляють файли, бази даних і подібні до них фізичні набори бітів. Діаграми даного типу часто застосовуються в комбінації з діаграмами розміщення. Також показують класи і компоненти, реалізовані ними, трактує діаграми артефактів, є різновидом UML-діаграм розміщення.

Діаграма пакетів (Package diagram) показує декомпозицію моделі на організаційні одиниці та їх залежності.

Тимчасова діаграма (Timing diagram) – діаграма взаємодій, що показує реальний час життя різних об'єктів або ролей, на противагу простої послідовності повідомлень.

Діаграма огляду взаємодій (Interaction overview diagram) – гібрид діаграми діяльності й діаграми послідовності. Ці два типи мають спеціалізоване застосування.

Ц є неповним переліком діаграм. Інструментальні засоби можуть використовувати UML для зображення інших типів діаграм.

1.7. Правила UML

UML включає набір правил, що вказують, як повинна виглядати добре розроблена модель – модель, яка семантично самопогоджена й перебуває в гармонії з усіма іншими моделями, пов'язаними з нею. UML включає синтаксичні й семантичні правила для:

- імен (*сутностей, відношень/зв'язків і діаграм*);
- областей дії (*контексти, що надають іменам специфічні значення*);

- видимості (*як імена можуть бути видимі й використані іншими*);
- цілісності (*як сутності мають правильно і узгоджено співвідноситись між собою*);
- виконання (*що означає запуск або імітування деякої динамічної моделі*).

Моделі, побудовані в процесі розроблення ПЗ, еволюють у часі і можуть використовуватися багатьма зацікавленими учасниками проекту для різних цілей і в різний час. Із цієї причини команди розробників створюють не тільки добре розроблені моделі, але й моделі, які:

- містять приховані елементи (*для спрощення сприйняття*);
- неповні (*деякі елементи можуть бути пропущені*);
- непогоджені (*цілісність моделі не гарантована*).

Поява таких не дуже добре розроблених моделей є неминучою в процесі розроблення, поки всі деталі системи не прояснять повною мірою. Правила UML змушують прояснювати найважливіші питання аналізу, дизайну й реалізації, що дозволяють отримувати розроблені моделі високої якості.

Загальні механізми UML. Усяке будівництво спрощується і здійснюється ефективніше, якщо дотримуватися низки узгоджень (*будинок може бути вибудуваний у вікторіанському чи французькому стилі, якщо виконувати його за певним архітектурним зразкам*). Процес розроблення ПЗ суттєво спрощує послідовне застосування таких загальних UM–механізмів, як:

- специфікації;
- доповнення (Adornments);
- прийняті поділи (Common divisions);
- механізми розширення (Extensibility mechanisms).

Специфікації (Specifications)**. Візуалізація системи та опис деталей**. За кожною частиною графічної UML -нотації стоїть специфікація, що містить текстове зображення синтаксису і семантики певного будівельного блоку. Піктограмі класу відповідає специфікація, що повністю описує набір її атрибутів, операцій (*включаючи їх повні сигнатури*) і поведінку. Візуально піктограма може відображати лише малу частину цієї специфікації. Може існувати інший вигляд цього класу, що відображає зовсім інші його аспекти, відповідні тій же специфікації.

Графічна UML-натація дозволяє візуалізувати систему, а UML-*специфікації* - описувати її деталі. Побудова моделі може здійснюватися послідовно – крок за кроком, починаючи зі створення діаграми, а потім додається семантика до специфікацій моделі, або навпаки, коли, в першу чергу, створюються специфікації (*зворотнє проектування*), а потім будуються діаграми, що визначають їхні проекції. Специфікації UML створюють певний семантичний задній план, що містить складові всіх моделей системи, погоджені між собою. Діаграми UML – це візуальні проекції на цей задній план, кожна з яких розкриває деякий істотний аспект системи.

Доповнення (Adornments). Більшість елементів UML мають унікальну й просту графічну нотацію, яка дає візуальне уявлення найважливіших аспектів елемента. Позначення класу легко зображувати, оскільки клас є часто вживаний елемент при моделюванні об'єктно-орієнтованих систем. Нотація класу показує його найважливіші аспекти: ім'я, атрибути й операції.

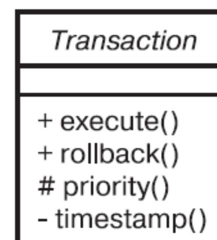


Рис. 1.7.1. Доповнення

Специфікація класу може містити інші деталі, такі, як видимість атрибутів і операцій, що можуть зображатися у вигляді графічних або текстових *доповнень* до базового прямокутника, що описує клас. Рис. 1.7.1 демонструє клас, у позначення якого включені доповнення, які вказують, що цей клас абстрактний, має дві відкриті, одну захищену і одну закриту операції.

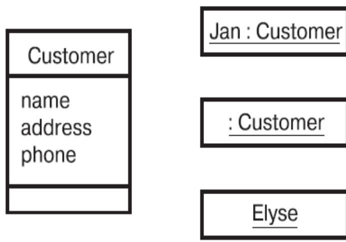


Рис.1.7.2. Класи й об'єкти

Загальноприйняті поділи (Common divisions). UML дозволяє моделювати як класи, так і об'єкти (рис. 1.7.2). Клас є абстракцією, а об'єкт – її конкретною матеріалізацєю. У графічному вигляді об'єкта UML використовує той же символ, що і для його класу, але з підкресленням імені об'єкта. При моделюванні об'єктно-орієнтованих систем реальності можуть бути розділені кількома способами. На рис. 1.7.2 показано один клас Customer (Покупець) разом із трьома об'єктами: Jan, явно вказаний як об'єкт класу Customer, анонімний об'єкт класу Customer) і Elyse, специфікація якого відносить його до класу Customer, хоча це не відображено явно.

Майже кожен будівельний блок UML характеризується подібною дихотомією «клас/об'єкт». Наприклад, існують варіанти використання й екземпляри ВВ, компоненти й екземпляри компонентів, вузли й екземпляри вузлів тощо.

Рис.1.7.3. Інтерфейси та реалізація

По-друге, існує поділ інтерфейсу і реалізації. Інтерфейс визначає угоду, а реалізація – його конкретне втілення й зобов'язується точно дотримуватися повної семантики інтерфейсу. В UML можна моделювати як інтерфейси, так і їх реалізації (рис. 1.7.3). Тут присутня одна компонента з іменем spellingwizard.dll (МайстерПеревіркиОрфографії.dll), що реалізує два інтерфейси – IUnknown і ISpellling. Вона також запитує інтерфейс IDictionary (Словник), який повинен бути наданий іншою компонентою. Отже, для досягнення мобільності коду використовуємо таку ж дихотомію- «*інтерфейс/реалізація*». Зокрема, ВВ реалізуються *коопераціями*, а операції – *методами*.

По-третє, існує поділ на тип і роль. **Тип** декларує клас сутності: об'єкт, атрибут або параметр. **Роль** описує значення сутності всередині контексту (класі, компоненті чи кооперації). Будь-яка сутність, що формує частину структури іншої сутності (атрибут), володіє обома характеристиками. Вона успадковує деякий зміст від типу (класу), якому належить, а інший зміст – від його ролі в даному контексті (рис. 1.7.4).

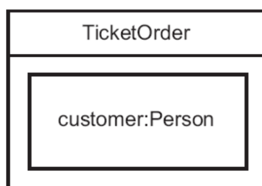


Рис.1.7.4. Частина з роллю і типом

Механізми розширення (Extensibility mechanisms). UML як стандартна мова розроблення «креслень» ПЗ не може охопити всі нюанси всіх можливих моделей у всіх областях застосування в будь-який час. Тому UML є відкритою мовою і допускає контрольовані механізми розширення, що включають:

- стереотипи;
- присвоєні значення;
- обмеження.

Стереотип (Stereotype) розширює словник UML, дозволяючи створювати нові види будівельних блоків, які успадковуються від існуючих і є специфічними для розв'язання конкретної проблеми. Наприклад, працюючи з мовами C++ чи Java, доводиться моделювати **виключення** (exceptions), що можуть трактуватися як звичайні класи. [2,4,7] Потрібно, щоб виключення можна було збуджувати і перехоплювати. Виключення моделюються, позначивши їх відповідним стереотипом і зробивши їх подібними до базових будівельних блоків (клас Overflow) (рис. 1.7.5).

Присвоєне значення (Tagged value) розширює властивості стереотипу UML, дозволяючи включати нову інформацію в специфікацію стереотипу. Можна відслідковувати певні параметри певних важливих абстракцій римітивами). [4,5] Такі параметри, що не єпримітивами UML, можуть бути додані до будь-якого будівельного блока (класу) за рахунок введення в нього нових присвоєних значень. Клас EventQueue розширюється явною позначкою його версії й автора (рис. 1.7.5).

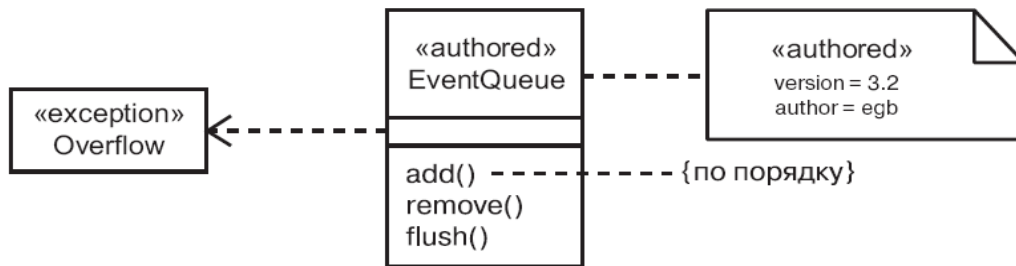


Рис.1.7.5. Механізми розширення

Обмеження (Constraints) розширюють семантику будівельного блока UML, дозволяючи додавати нові правила чи модифікувати існуючі. [4,5,6] Клас EventQueue обмежується, щоб усі події додавалися в чергу одна за одною шляхом додавання обмеження, що явно задає таке правило для операції add (рис. 1.7.5).

Усі ці три механізми розширення мови спільно дозволяють модифікувати UML відповідно до вимог проекту та адаптувати UML до нових програмних технологій (*мов розподіленого програмування тощо*). Можна додавати нові будівельні блоки, модифікувати специфікацію існуючих і змінювати їхню семантику в нормальних межах.

Початки роботи. «*Єдиний спосіб вивчити нову мову програмування – це писати на ній програми*» /Брайан Керніган (Brian Kernighan) і Деніс Річі (Dennis Ritchie), авт. C/. Єдиний спосіб вивчити UML – писати на ній моделі. Перша програма, яку багато розробників пишуть, приступаючи до засвоєння нової мови програмування, – найпростіша. Вона робить дещо більше, ніж просто надрукувати текстовий рядок, типу «Salut, UML!». Цей тривіальний додаток наочно ілюструє шлях руху вперед і дозволяє розкрити всю необхідну інфраструктуру для її складання й запуску. В простоті цього створюваного додатка лежить ряд цікавих механізмів, що забезпечують його роботу.

Ключові абстракції. Аплет мовою Java, що виводить рядок «Salut, UML!». У Web-браузері, досить простий:

```
import java.awt.Graphics;
class SalutUML extends Java.applet.Applet
 {
  public void paint (Graphics g)
    {g.drawstring("Salut, UML!", 10, 10);}
```

```
}
```

Перший рядок коду забезпечує доступ до класу Graphics (Графіка) для наступного коду. Префікс java.awt. специфікує пакет Java, у якому знаходиться клас Graphics. Другий рядок коду представляє новий клас по імені SalutUML і вказує, що він є нащадком класу Applet, що перебуває в пакеті java.applet. Наступні рядки коду повідомляють операцію по імені paint, реалізація якої викликає іншу операцію-drawstring (від рядка), відповідальну за виведення рядка «Salut, UML!» по заданих координатах. drawstring є операцією об'єкта g класу Graphics.

Моделювання цього додатка на UML здійснюється діаграмою класів (рис. 1.7.6). Клас SalutUML графічно зображений у вигляді прямокутної піктограми. Його операція paint (малювання) показана тут же, причому її формальні параметри упущені, а реалізація зазначена в приєднаній примітці. Ця діаграма класів передає основну суть додатка «Salut, UML!», але водночас залишає без уваги багато речей. Як бачимо із коду, у додатку беруть участь два інших класи – Applet (Аплет) і Graphics (Графіка). Причому, вони використовуються різними способами. Клас Applet використовується в якості батьківського для SalutUML, а клас Graphics використовується в сигнатурі й реалізації його операції paint.

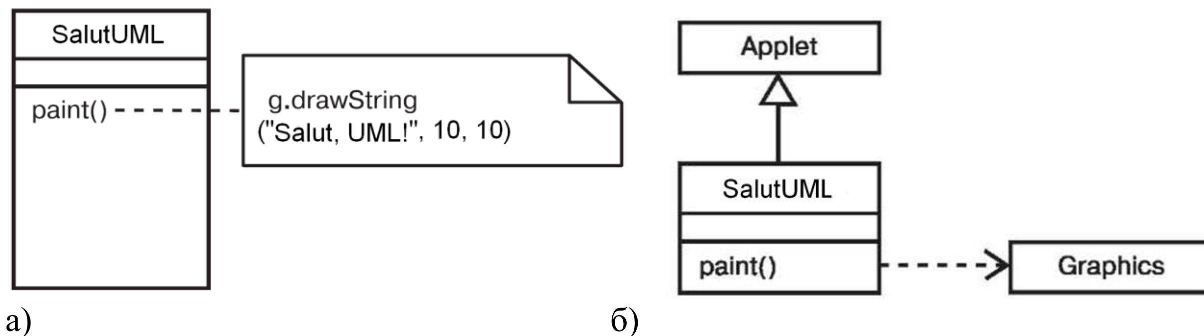


Рис. 1.7.6. Ключові абстракції класу SalutUML (а) та діаграма класів (б)

Класи Applet і Graphics зображені у вигляді прямокутних піктограм. Ніякі операції для них не показані. Лінія з порожньою стрілкою від SalutUML до Applet є узагальненням, яке вказує, що SalutUML є дочірнім стосовно Applet. Пунктирна лінія, направлена від SalutUML до Graphics, є відношенням залежності, яка означає, що SalutUML використовує Graphics. І це ще не вся структура, що лежить в основі SalutUML. Як випливає з бібліотеки Java, що містить Applet і Graphics, ці обидва класи є частинами ще більшої ієрархії. Переглядаючи тільки ті класи, які розширює й реалізує Applet, можна розробити іншу діаграму класів (рис. 1.7.7).

UML не є візуальною мовою програмування, але вона дозволяє забезпечити тісний зв'язок з різними мовами програмування, зокрема Java (рис. 1.7.6). UML дозволяє трансформувати моделі в код і назад – з коду в модель. Деякі деталі (математичні вирази) краще виразити текстовою мовою програмування, а інші (ієрархія класів) – візуалізувати графічно на UML.

Зв'язок між Imageobserver (Оглядач Зображень) і Component трохи відрізняється від попередніх зв'язків узагальнення, і діаграма класів відображає цю відмінність (рис. 1.7.7). У бібліотеці Java ImageObserver – це інтерфейс. Звідси випливає, що він сам по собі не має реалізації, а замість цього вимагає, щоб інші класи реалізовували його.

Можна показати, що клас Component реалізує інтерфейс Imageobserver за допомогою суцільної лінії, проведеної від прямокутника (Component) до кружка, який символізує інтерфейс (Imageobserver). Клас SalutUML безпосередньо взаємодіє тільки із двома класами (Applet і Graphics), які є лише малою частиною великої бібліотеки визначених класів Java (рис. 1.7.7).

Пакетування. Для управління цією великою колекцією, Java організовує свої інтерфейси і класи в цілу множину пакетів. Кореневий пакет середовища Java має ім'я java, у якому вкладено кілька пакетів, що містять, у свою чергу, інші пакети, інтерфейси й класи. Клас Object включений у пакет lang, тому повний шлях до нього має такий вигляд: java.lang.Object.

Аналогічним чином Panel, Container і Component перебувають у пакеті awt, клас Applet – у пакеті applet. Інтерфейс ImageObserver перебуває в пакеті image, який, у свою чергу, розміщений в awt, тому повний шлях до нього – java.awt.image.Imageobserver.

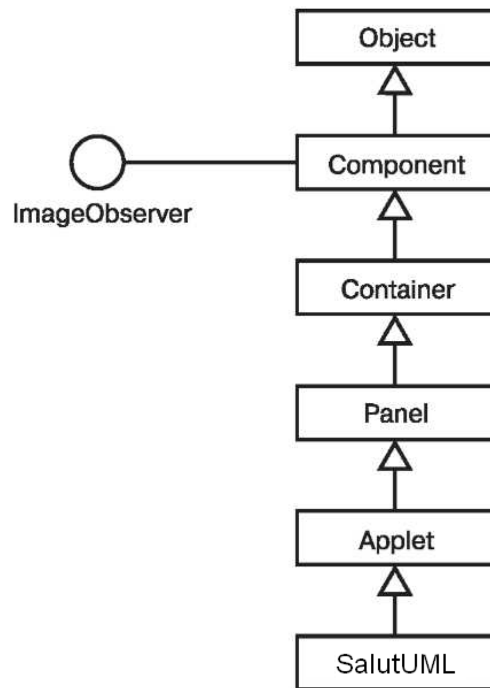


Рис. 1.7.7. Ієрархія успадкування

Можна зобразити таке пакетування на діаграмі класів (рис. 1.7.8). Пакети представлені в UML у вигляді папок із закладками. Пакети можуть бути вкладеними, і пунктирні лінії зі стрілками виражають залежності між ними. Наприклад, клас SalutUML залежить від пакета Java.applet, а Java.applet – від Java.awt.

Механізми. Найважче завдання, що виникає при освоєнні такої багатої бібліотеки, як бібліотека Java, – зрозуміти, як її частини працюють разом. Наприклад, як викликається операція paint класу SalutUML. Які операції слід використовувати, щоб змінити поведінку цього аплета, наприклад, надрукувати рядок іншого кольору? Для відповіді на подібні запитання потрібно мати концептуальну модель, що показує, як класи працюють спільно в динаміку. Як випливає із бібліотеки Java, операція paint класу SalutUML успадковується від Component. Але залишається відкритим питання, як вона викликається. Відповідь у тому, що paint викликається як частина потоку, у якому виконується аплет (рис. 1.7.9).

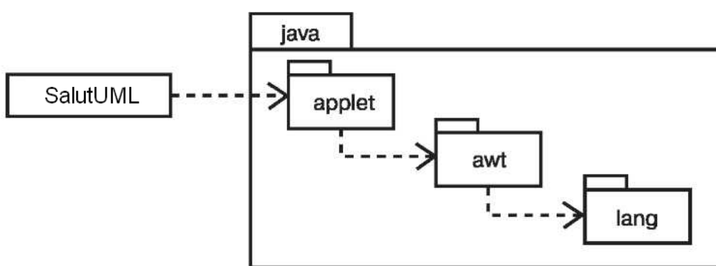


Рис. 1.7.8. Пакетування SalutUML

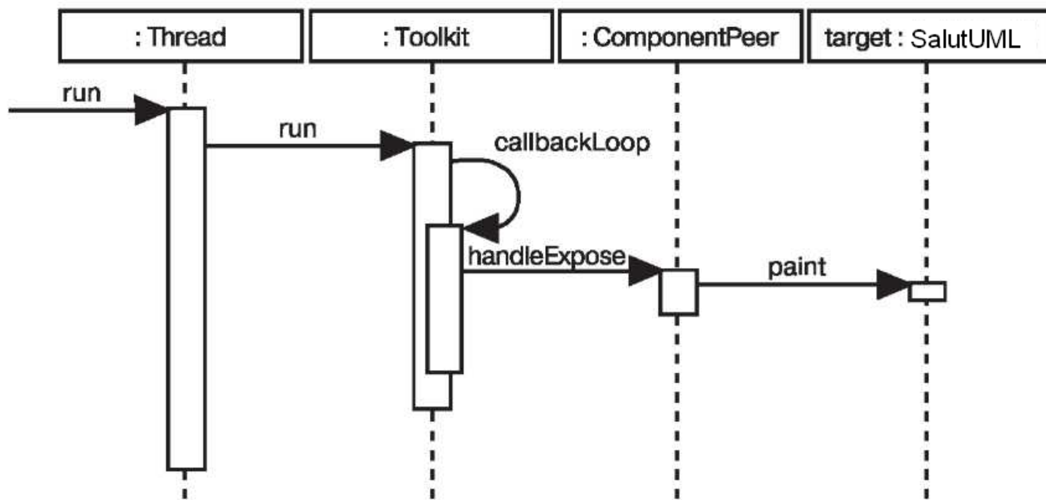


Рис. 1.7.9. Зображення механізму

Порядок подій тут моделюється при допомозі діаграми послідовності (рис. 1.7.9). Тут послідовність починається із запуску об'єкта Thread, який, у свою чергу, викликає операцію run об'єкта Toolkit. Об'єкт Toolkit потім викликає одну зі своїх власних операцій (callbackloop), яка потім викликає операцію handleexpose об'єкта Componentpeer. Об'єкт Componentpeer припускає, що його ціль є екземпляром Component, але в цьому випадку – це об'єкт дочірнього класу Component, а саме SalutUML, тому поліморфно викликається операція paint класу SalutUML.

Артефакти. Програма «Salut, UML!» реалізована у вигляді аплета, тому вона ніколи не запускається самостійно, а тільки у складі деякої Web-сторінки. Аплет стартує, коли сторінка, що містить його, відкривається механізмом браузера, котрий запускає об'єкт Thread цього аплета. Однак не існує класу SalutUML, який би був безпосередньою частиною Web-сторінки. Є лише бінарна форма цього класу, створена компілятором Java, який трансформував вихідний код, що представляє цей клас, у виконуваний артефакт. Таким чином, формується зовсім інший погляд на систему. У той час, як усі попередні діаграми пропонували логічний вигляд аплета, те, що ми бачимо тепер, – вигляд його фізичних артефактів.

Можна змоделювати цей фізичний вигляд за допомогою діаграми артефактів (рис. 1.7.10). Подана діаграма взаємодії демонструє кооперацію кількох об'єктів, включаючи один екземпляр класу SalutUML. Інші зображені тут об'єкти є частиною середовища Java, тому в основному перебувають на задньому плані створюваних аплетів. Це є кооперацією між екземплярами класів, яка може застосовуватися багаторазово. Кожен стовпець показує роль у кооперації, тобто частина, яка може бути виконана різними об'єктами при кожному запуску. В UML ролі зображуються так само, як класи, з тією відмінністю, що для них вказуються тип і ім'я ролі (назва об'єкта). Середні дві ролі на даній діаграмі є анонімними, тому що їх типу досить для того, аби ідентифікувати їх у межах кооперації (але двокрапка і підкреслення вказують на те, що це ролі). Перший об'єкт Thread називається root, а роль SalutUML іменована target (як назва об'єкта) і відома об'єкту Componentpeer.

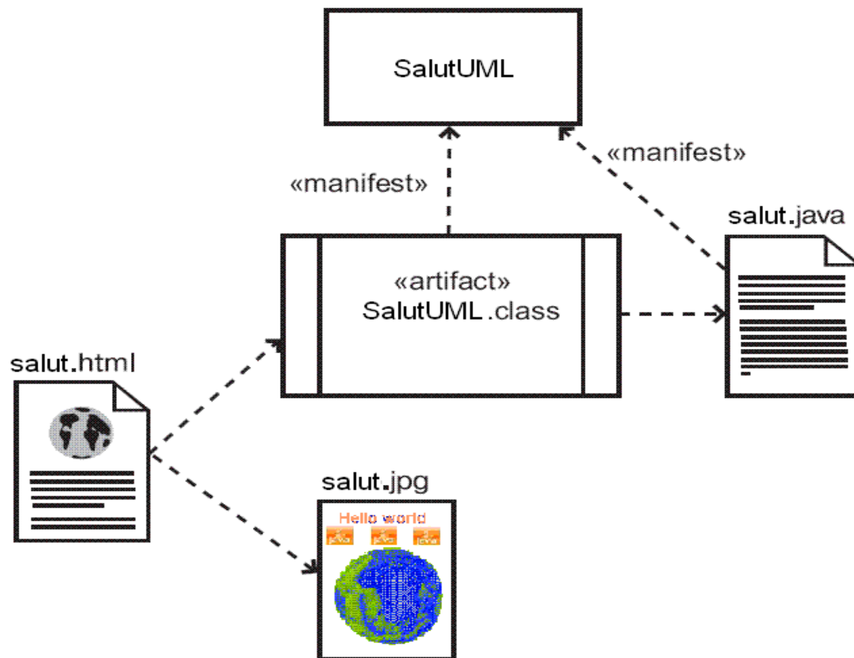


Рис. 1.7.10. Артефакти

Логічний клас SalutUML показаний у верхньому прямокутнику. Усі інші піктограми на рисунку символізують артефакти UML у вигляді реалізації системи. **Артефакт** – це фізична сутність, така, як файл. Артефакт по імені salut.java представляє вихідний код логічного класу SalutUML; цим файлом може маніпулювати середовище розроблення, а також інструменти керування конфігурацією. Вихідний код може бути трансформований у бінарний аплет salut.class за допомогою компілятора Java, що робить його придатним для запуску в середовищі віртуальної машини Java. Як вихідний текст, так і бінарний аплет фізично реалізують логічний клас. Це показано пунктирними стрілками, позначеними ключовим словом «manifest».

Піктограмою артефакту є прямокутник із ключовим словом «artifact», розташованим над іменем. Бінарний аплет SalutUML.class – варіація цього базового символу, але зі стовщеними лініями, що вказують на те, що це артефакт, що виконується (подібно активному класу). Піктограма артефакту salut.java замінена користувацькою іконкою, що представляє текстовий файл. Піктограма Web-сторінки salut.html оформлена аналогічним чином, з розширенням нотації UML. Як випливає із рисунка, ця Web-сторінка містить інший артефакт, salut.jpg, який теж представлений користувацькою іконкою – тепер уже графічним файлом, бо останні три артефакти представлені обумовленими користувачем графічними символами, їх імена вміщені поза піктограмами. Залежності між артефактами зображені пунктирними стрілками.

2. Основи структурного моделювання

2.1. Моделювання класів, об’єктів та екземплярів

2.1.1. Класи

Основні питання:

- *Класи, атрибути, операції, обов’язки*
- *Моделювання словника системи*
- *Моделювання розподілу обов’язків у системі*
- *Моделювання непрограмних сутностей*
- *Створення якісних абстракцій*

Класи (Class) є найважливішими абстракціями і будівельними блоками створюваної об’єктно-орієнтованої системи і описують множини об’єктів з однаковими властивостями і поведінкою. [2] Класи характеризуються спільними атрибутами, операціями, зв’язками і семантикою. Клас є **концептуальним виглядом** цілої множини об’єктів. [1,4,5]

Вимоги до проектування класів. Класи використовуються для побудови словника системи і включають абстракції, що є частиною проблемної області та інші абстракції, на яких ґрунтується реалізація ПЗ. Класи можна застосовувати для описування концептуальних, програмних та апаратних сутностей. Добре структуровані класи мають чітко окреслені границі і формують збалансований розподіл обов’язків у системі. Моделювання системи включає ідентифікацію сутностей, важливих для конкретного вигляду її реалізації, що формують **словник системи**. *При побудові будинку важливо знати, якими будуть стіни, двері, вікна, освітлення тощо. Кожна із цих сутностей відрізняється від іншої і має свої властивості. Стіни мають певну висоту, ширину, є суцільними. Двері характеризуються тими ж ознаками, але відрізняються специфічною поведінкою – відчиняються в один бік. Вікна частково схожі з дверими – утворюють проріз у стіні, але їх функціональність інша – пропускати сонячне світло… Окремі стіни, двері, вікна не існують самі по собі – це конкретні екземпляри цих сутностей, з’єднаних між собою. Ці сутності і зв’язки залежать від використання кімнат, дизайнерського вирішення тощо. Для будівельників важливим є розташування водостоків, вентиляції, електричних кабелів тощо. Домовласник не обов’язково зверне увагу на такі речі (за винятком видимих порушень: ванну вмонтовано у вітальні, водостік – у робочому кабінеті).*

В UML сутності моделюються класами, що є їх абстракціями і частиною словника системи. Клас концептуально описує цілу множину об’єктів (а не індивідуальний об’єкт: *«стіна» є клас об’єктів зі спільними властивостями і поведінкою: висота, довжина, товщина; чи є стіна опорною тощо. Конкретна стіна у вітальні є реальним об’єктом).* Створювані UML-абстракції безпосередньо можна виражати мовою ООП.

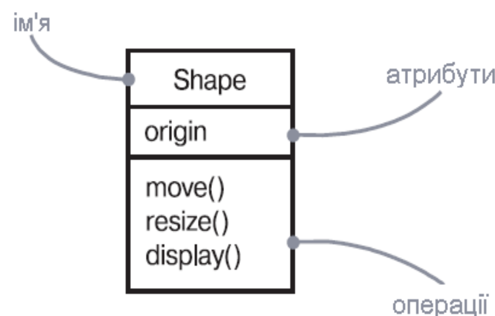


Рис. 2.1.1. Загальна нотація класу

Графічна UML-нотація класу (рис. 2.1.1) дозволяє візуалізувати абстракцію незалежно від мови ООП, підкресливши її найважливіші частини: ім’я, атрибути і операції класу (*опису множини об’єктів з однаковими атрибутами, операціями, зв’язками і семантикою*). **Ім’я** (name) є унікальною назвою кожного класу у вигляді текстового рядка, що відрізняє його від інших і

відображає його сутність. Окреме ім'я класу є простим, а *ім'я класу з префіксом* – з іменем пакета, в який включено клас, є **кваліфікованим**. Клас на найзагальнішому рівні абстрагування зображується прямокутником із вказанням його імені (рис. 2.1.2).

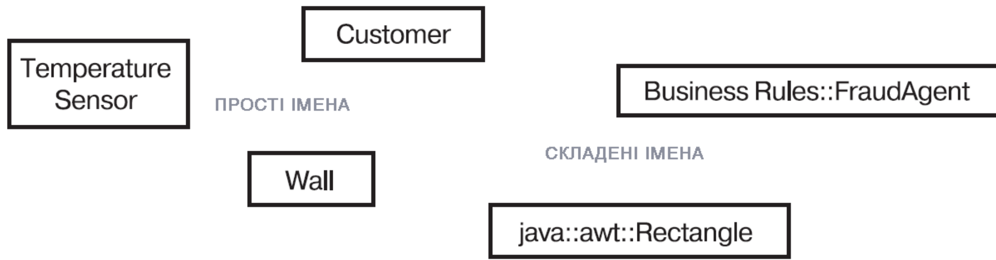


Рис. 2.1.2. Прості й кваліфіковані імена

Атрибут (attribute) – іменована **властивість** класу, що описує діапазон значень, які може набувати **екземпляр** атрибута. [2] Клас може мати будь-яке число атрибутів або не мати жодного. Атрибут є певною властивістю модельованої сутності, спільної для усіх об'єктів класу (*у кожного співробітника є ім'я, адреса, номер телефону, дата народження*). Атрибут є абстракцією **виду даних** або **стану**, яким може бути наділений об'єкт класу. У кожен певний момент об'єкт класу характеризується конкретними значеннями (екземплярами) кожного з атрибутів. Атрибути перераховані в розділі нотації безпосередньо під іменем класу (рис. 2.1.3).

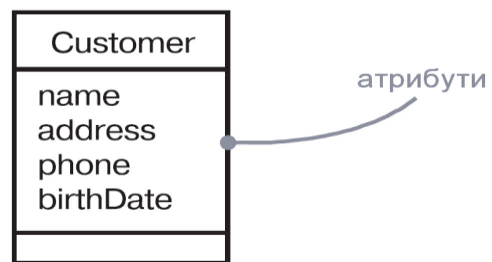


Рис. 2.1.3. Атрибути

Специфікацію атрибута можна уточнити, вказавши його тип і початкове значення (рис. 2.1.4).

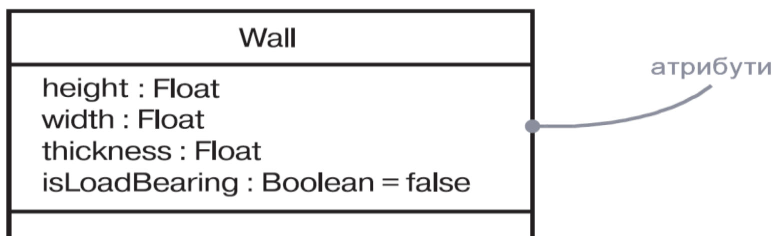


Рис. 2.1.4. Атрибути з їхніми типами

Операція (operation) – це **реалізація послуги** об'єкта класу, на яку може бути запит з боку іншого об'єкта чи актора (екземпляра актанта), щоб викликати певну **поведінку об'єкта** класу. [1,2,13] Операція є абстракцією того, що можна зробити з конкретним об'єктом або з усіма об'єктами класу. Клас може мати будь-яке число операцій або не мати жодної. Виклик операції об'єкта може змінювати його дані (стан). Список операції класу вказуються в нотації безпосередньо під списком атрибутів (рис. 2.1.5). Для імені операції використовується **коротке дієслово** або **дієслівний зворот**, що відповідає певній *поведінці* класу. [13] Кожне слово в операції слід писати з *великої букви*. Операцію можна специфікувати повною сиґнатурою, що включає ім'я, тип, значення за замовчуванням

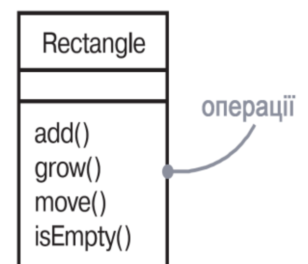


Рис. 2.1.5. Операції

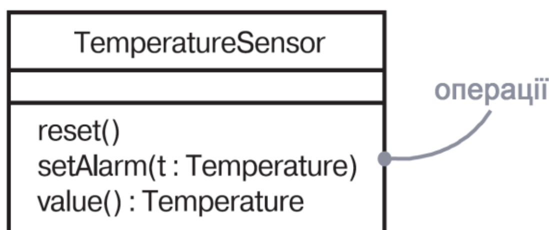


Рис. 2.1.6. Операції, їх сигнатури

усіх параметрів, а також тип (клас) значення, що повертається операцією (рис. 2.1.6).

Організація атрибутів і операцій. При моделюванні класу немає сенсу зразу вказувати всі його атрибути й операції, що не завжди можливо вмістити на одній діаграмі. Для конкретного вигляду класу істотною є тільки частина атрибутів і операцій, для чого використовується спрощене його зображення, що включає окремі його атрибути й операції чи не включає взагалі. [2] Якщо окрім вказаних є інші атрибути й операції, то це можна показати, закінчуючи список *трикрапками*. Можна зовсім не позначати відповідний розділ, не вказуючи, чи має клас атрибути або операції. Для кращої організації довгих списків атрибутів і операцій, кожна категорія з них обладнується префіксом (*іменем стереотипу*) (*детальніше у п. 2.3*).

Відповідальність (responsibility) – це угода або зобов'язання класу щодо виконання ним своїх обов'язків із забезпечення певних функцій (ВВ) системи. [1]

Атрибути й операції як засоби виконання обов'язків. На більш абстрактному рівні атрибути й операції є засобами виконання класом своїх обов'язків. Клас Wall (стіна) відповідає за інформацію про висоту, ширину, товщину; клас Fraudagent (АгентЗапобіганняКраж), що опрацьовує кредитні карти за опрацювання запитів і визначення їх законності; клас TemperatureSensor (СенсорТемператури) – за вимір температури і подачу сигналу небезпеки при перевищенні допустимої межі. [2]

Формулювання обов'язків класу. Для моделювання класів потрібно добре сформулювати обов'язки сутностей зі словника предметної області (*на основі аналізу ВВ, CRC-карт*). [2] Добре структурований клас має чітко сформульований не надто обтяжливий набір обов'язків. У міру деталізації моделей потрібно **відображати обов'язки на множину атрибутів і операцій**, що найбільше їм відповідають. Обов'язки класу подаються у нижній частині піктограми класу (рис. 2.1.7).

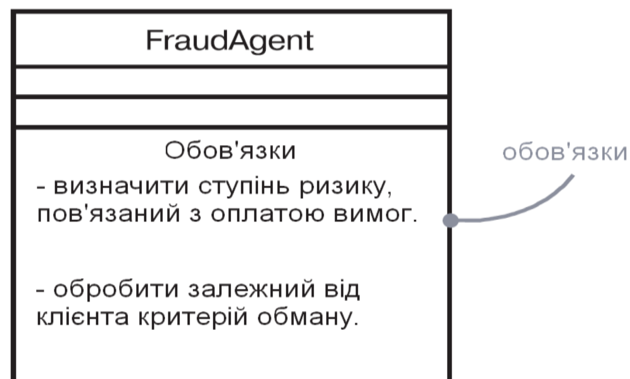


Рис. 2.1.7. Обов'язки

Інші характеристики. При моделюванні ПЗ виникають потреби у візуалізації інших характеристик: видимість окремих атрибутів операцій; властивостей операцій, залежних від мови ООП (*поліморфізм, константність*); виключень, які об'єкт класу може ґенерувати або опрацьовувати. Все це можна описати додатковими засобами UML.

Специфікація та реалізація класу. При побудові UML-моделей ПЗ *реалізацію класу* необхідно *відокремлювати від його специфікації* за допомогою **інтерфейсів**. Клас може реалізувати один або кілька інтерфейсів. При проектуванні реалізації класу потрібно моделювати її внутрішню структуру як набір взаємозалежних частин. [2, 13]

Кооперації класів. При побудові моделей акцентується увага на ґрупах класів, що взаємодіють між собою (*а не існують самі по собі*). В UML такі співтовариства класів формують *кооперації*, що візуалізуються в діаграмах класів та кооперації. [2,6]

Активні класи, класифікатори, вузли. При проектуванні складніших моделей використовують ряд інших сутностей: *активні класи*, що описують паралельні процеси і потоки, *класифікатори*, артефакти, що описують фізичні сутності (*аплети, компоненти ПЗ, файли, Web-сторінки*), вузли (*описують апаратне забезпечення*), що є важливими архітектурними абстракціями. [1,2,11]

Моделювання словника системи. Для користувачів ідентифікувати більшість абстракцій не так складно, оскільки останні походять від понять, що описують систему. Для розробників подібні абстракції є **технологічними сутностями реалізації проекту**. Щоб змоделювати словник системи, необхідно ідентифікувати сутності, якими оперують користувачі і розробники для описування проблеми та її розв'язання. Для кожної визначеної абстракції ідентифікується набір обов'язків, збалансовано розподілені між ними (класами). Необхідно визначити атрибути й операції, необхідні для виконання обов'язків кожного класу.

На рис. 2.1.8 наведено набір множини (основних абстракцій), призначений для реалізації системи роздрібної торгівлі, що містить класи Customer (Покупець), Order (Замовлення) і Product (Продукт) та інші класи зі словника проблемної області, пов'язані з ними: Shipment (Поставка) – для відстеження замовлень; Invoice (Рахунок-фактура) – для оплати замовлень; Warehouse (Склад) – місце зберігання товарів перед відправкою. Клас Transaction (Транзакція) застосовується до замовлень і поставок.

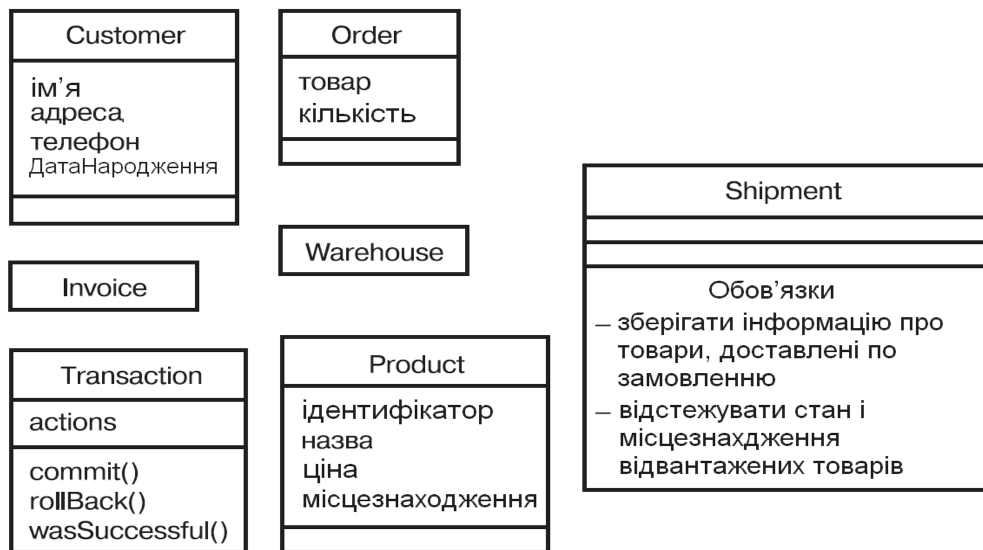


Рис. 2.1.8. Моделювання словника предметної області системи

В міру розширення моделей виявляється, що багато класів мають тенденцію збиратися в концептуально і семантично зв'язані ґрупи (кластери). В UML для моделювання таких кластерів використовуються **пакети**.

Моделювання розподілу обов'язків у системі. При моделюванні великої множини класів необхідно переконатись у забезпеченні балансу обов'язків між розроблюваними абстракціями. Кожен із них не повинен бути надто великим чи надто малим і добре виконувати одне завдання. Для дуже великих класів модель ПЗ буде важко піддаватися змінам, що ускладнить її повторне використання. Малі класи визначають надто багато абстракцій, важких для сприйняття і керування.

Щоб змоделювати розподіл обов'язків у системі, необхідно ідентифікувати множину класів, що корпоративно працюють для досягнення деякої поведінки. Потрібно розглянути способи взаємодії цих класів, ідентифікувавши набір обов'язків кожного з них. Класи, на які припадає занадто велика частка обов'язків, потрібно розчленувати на менші абстракції. Дрібні класи з тривіальними обов'язками необхідно об'єднати в більші, рівномірно перерозподіливши обов'язки, щоб для кожної абстракції був відведений оптимальний їх набір.

На рис. 2.1.9 поданий набір класів, що працюють у кооперації з рівноміно розподіленими обов'язками: Model (Модель), View (Вигляд) і Controller (Контролер).

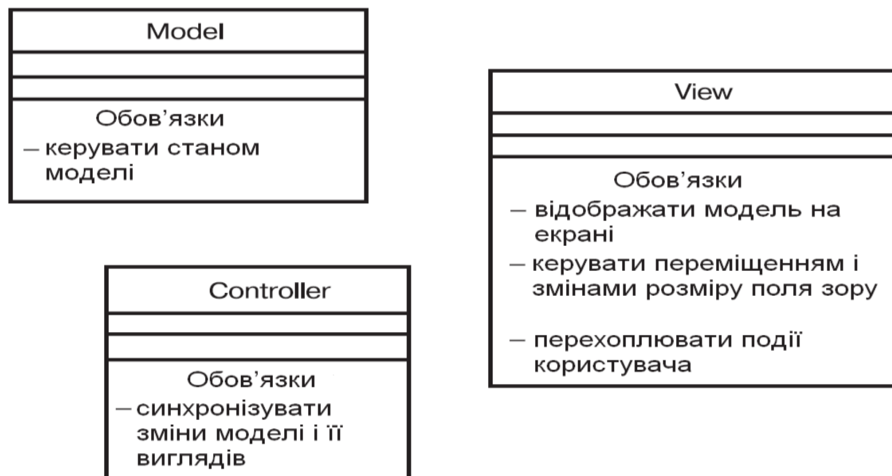


Рис. 2.1.9. Моделювання розподілу обов'язків у системі

Моделювання непрограмних сутностей, які не мають аналогів у ПЗ (*актанти, роботи,…*), але є частиною **потоку робіт** (workflow) для системи здійснюється **класами зі стереотипом** для їх специфікації. Моделювання апаратного забезпечення здійснюється через вузли.

Рис. 2.1.10. Класи непрограмних сутностей

Дійові особи (Accountsreceivableagent (АгентПоДебеторськійЗаборгова-ності) й апаратне забезпечення (Robot) абстрагуються класами об'єктів спільної структури і поведінки (рис. 2.1.10). *Поряд з UML може використовуватися спеціальна текстова мова VHD, розширення Sysml консорціуму OMG. [15]*

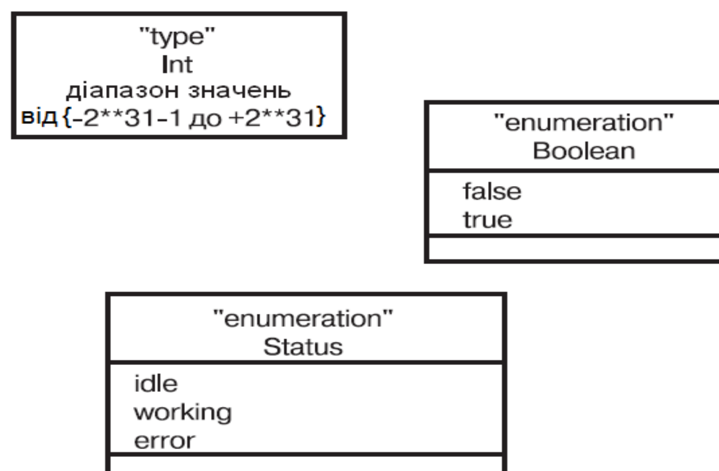


Рис. 2.1.11. Моделювання примітивних типів як класів зі стереотипами

Моделювання примітивних типів – сутностей мови ООП (*цілі числа, рядки, перераховані типи, типи користувача*) можна здійснювати у вигляді класів або перерахувань (enumeration) з відповідним стереотипом (рис. 2.1.11). Для специфікації діапазону їх значень використовуються обмеження (constraints). [1,2,10]

Добре структурований клас є чіткою концептуальною абстракцією поняття зі словника проблемної області для користувача чи розробника; включає короткий, чітко виражений набір обов'язків та їх забезпечення; має чіткий поділ специфікації й реалізації цієї абстракції; є зрозумілим і простим, розширюваним і добре адаптованим. [11]При моделюванні класу в UML слід показувати найважливіші для розуміння абстракції властивості в даному контексті; організувати списки атрибутів і операцій, ґрупуючи їх за категоріями; представити взаємозалежні класи на одній діаґрамі.

2.1.2. Основні механізми доповнення й розширення будівельних блоків UML

Основні питання :

- *Примітки*
- *Стереотипи, присвоєні значення й обмеження*
- *Моделювання коментарів*
- *Моделювання нових будівельних блоків*
- *Моделювання нових властивостей, семантики і розширення UML*

Іноді в модель потрібно включити інформацію, що виходить за рамки звичайних формальностей. *Архітектор робить примітки на комплекті креслень будинку, щоб звернути увагу будівельників на окремі тонкощі. Композитор застосовує спеціальну музичну нотацію для вираження особливих ефектів для виконавця...* При моделюванні ПЗ та ВСКД різних артефактів і обміні інформацією виникає необхідність у модифікації й розширенні UML. Це забезпечується завдяки наступним загальним механізмам: специфікаціям, доповненням, загальним засобам поділу і засобам розширення.

Примітка (note). UML передбачає механізм включення в модель різних коментаріїв і обмежень, що пояснюють її суть. [1,2] Ці артефакти відіграють важливу роль у ЖЦ розроблення ПЗ, особливо для розроблення вимог, оглядів чи пояснень, виражених у вільній формі, тощо. Примітка є графічною UML-нотацією у вигляді прямокутника з одним загнутим кутом, що включає текстовий або графічний коментар, приєднаний до елемента чи набору елементів і розміщуваних безпосередньо на діаграмі (рис. 2.1.12). У комбінації й іншими інструментами примітки дають можливість вказати обмеження, посилання або безпосередньо вмонтувати в модель інші документи.

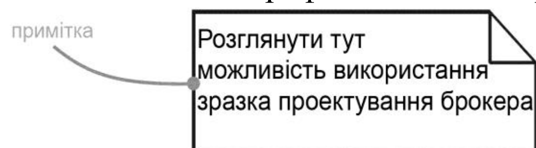


Рис. 2.1.12. Примітки

Механізми розширень UML (*стереотипи, присвоєні значення, обмеження*) забезпечують можливість її розширення строго контрольованим чином. Стереотипи розширюють словник UML і є механізмами створення нових будівельних блоків і властивостей, специфікації нової семантики і добре «*підігнані*» під проблемну область. Присвоєні значення розширюють властивості стереотипів UML, дозволяючи описувати нову інформацію в специфікації елемента. Обмеження розширюють семантику будівельних блоків UML, додаючи нові правила або модифікувати існуючі, адаптувати UML для конкретної області й методики роботи. При моделюванні мережі потрібні нові символи для позначення маршрутизаторів, використовуючи вузли зі стереотипами. При тестуванні й поставці версій ПЗ потрібно

відслідковувати номери версій і результати тестів для кожної з підсистем, для чого використовуються присвоєні значення. При моделюванні систем реального часу (СРЧ) є необхідність доповнення моделей інформацією про граничні терміни тощо.

Стереотип (stereotype) – це розширення словника UML, що дозволяє створювати нові види будівельних блоків, подібних до існуючих, але специфічних для проблемної області. UML передбачає текстове зображення стереотипів, присвоєних значень і обмежень (рис. 2.1.13). Стереотипи дозволяють вводити нові графічні символи, візуально підкреслюючи той факт, що створювані моделі «говорять» мовою специфічної предметної області та сформованої культури розроблення. [2,10]

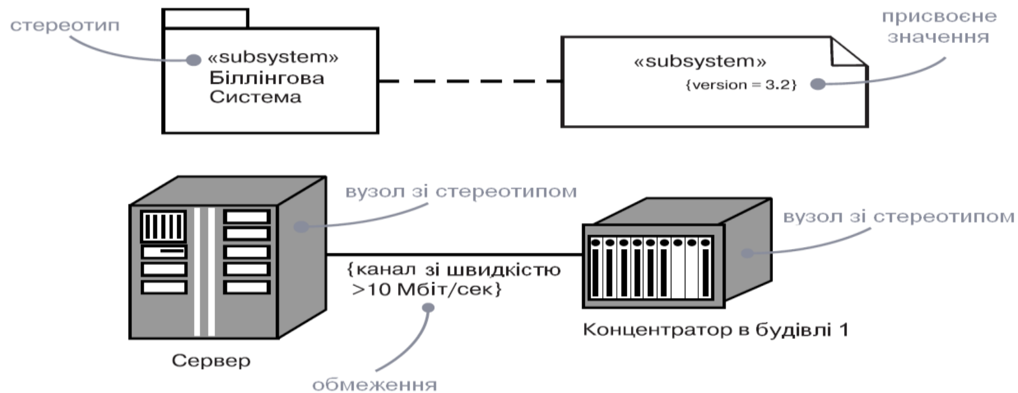


Рис. 2.1.13. Стереотипи, присвоєні значення й обмеження

Стереотип зображується іменем у кутових лапках і розміщується над іменем елемента. Елементи зі стереотипом можуть зображатися новою піктограмою, асоційованою з цим стереотипом. Стереотип є **метатип** (*визначає інші типи*) і створює еквівалент нового класу в UML-метамоделі. При використанні процесу розроблення Rational Unified Process потрібно моделювати граничні (boundary) класи, класи управління (control) і класи-сутності (entities), виражені *стереотипами*. [2,6] Присвоюючи стереотип класу чи вузлу, здійснюється створення нових будівельних UML-блоків з особливими властивостями (*свій набір присвоєних значень*), семантикою (*власні обмеження*) і нотацією (*окремою піктограмою*).

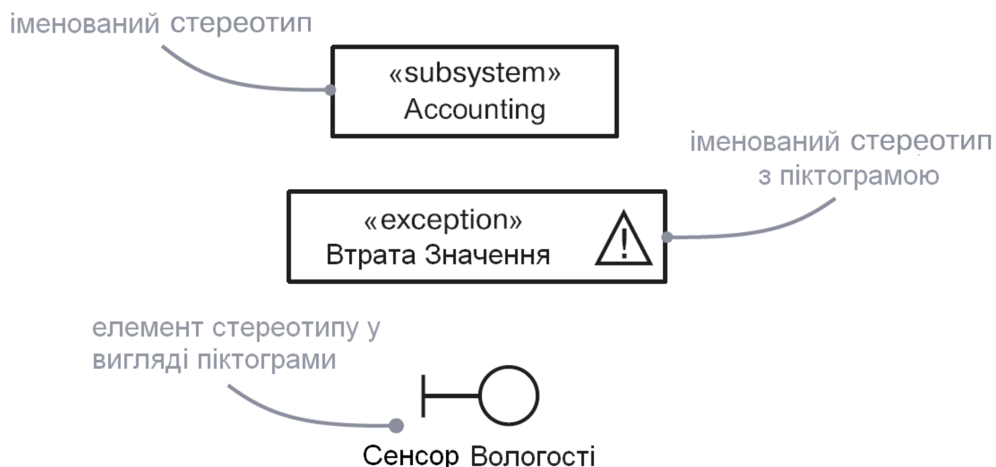


Рис. 2.1.14. Види стереотипів

У найпростішій формі стереотип зображений як ім’я в кутових лапках («stereotypname») і вміщене над іменем елемента. Для візуалізації стереотипу можна вибрати будь-яку піктограму і розмістити її справа від імені (*при використанні базової нотації*) чи застосувати її як базовий символ для відображення елементів стереотипної сутності (рис. 2.1.14).

Присвоєне значення (tagged value) – це властивість стереотипу, що дозволяє вводити нову інформацію для елемента. [1,6] Присвоєні значення входять у примітки, приєднані до елемента (рис. 2.1.15). Кожне присвоєне значення містить у собі рядок, що складається з імені (**тега**), роздільника (:) і значення. Присвоєні значення можуть вводитись всередині індивідуальних стереотипів. Присвоєні значення є **метадані**, що стосуються специфікації елементів UML, а не їхніх екземплярів. Можна вказати необхідну продуктивність класу Сервер, вимагати певного типу сервера (рис. 2.1.15).

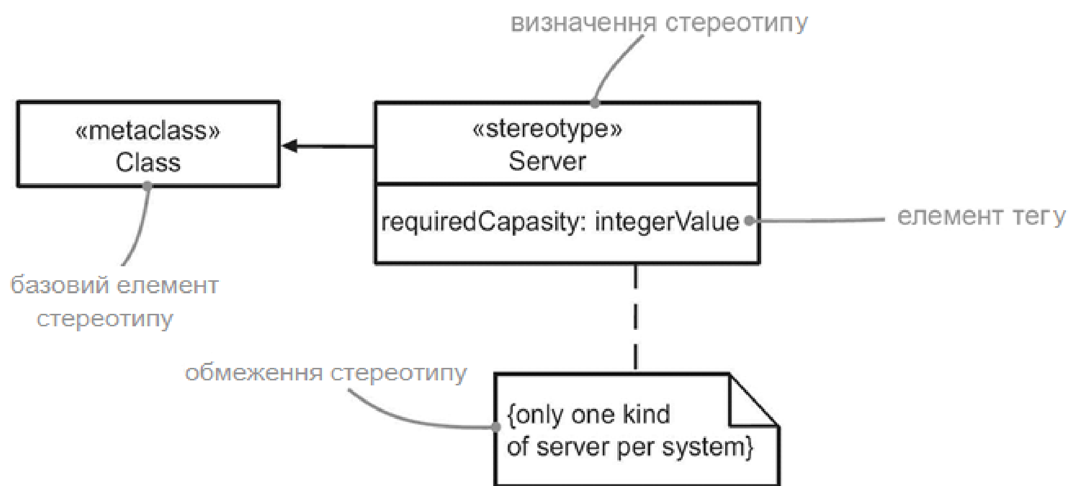


Рис. 2.1.15. Стереотипи і визначення тегів

Обмеження (constraint) – це текстова специфікація семантики елемента UML, що забезпечує додавання нових правил чи модифікацію (нову семантику) до існуючих, описана рядком у фігурних дужках і розміщується поруч з елементом або пов’язаною з ним залежністю. [2,13] В якості альтернативи можна зображувати обмеження, як примітку з текстом і графікою. Обмеження визначають **умови**, яким повинна задовольняти система. Можна розмістити всередині коментаря URL-адресу або посилання на інший документ чи вмонтувати сам документ (рис. 2.1.16).

Рис. 2.1.16. Примітки й обмеження

Рис. 2.1.17 ілюструє приклад використання обмеження для описування взаємодії за окремою асоціацією, яка повинна бути безпечною, і конфігурації, що порушують це обмеження, не відповідають моделі, а певний екземпляр може мати зв'язок тільки з однією асоціацією з множини асоціацій, з'єднаних з деяким класом (Банківський рахунок).

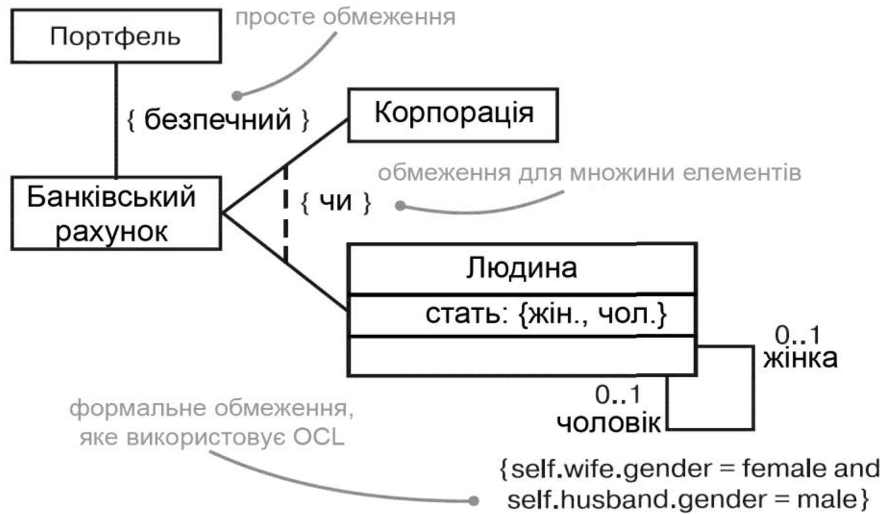


Рис. 2.1.17. Обмеження

Доповнення (adornments) – це текстові або графічні об'єкти, що додаються до базової нотації і використовуються для візуалізації деталей специфікації. *Базова нотація для асоціації – лінія може бути доповнена описом ролі й множинності кожного кінця. [1]*

Додаткові розділи. Більшість доповнень описується текстом, розміщеним біля елемента чи графічним символом, доданим до базової нотації. Для детальнішого описування класів, компонентів і вузлів використовуються додаткові розділи (*іменовані й анонімні*), що розміщаються безпосередньо під звичайними (рис. 2.1.18).

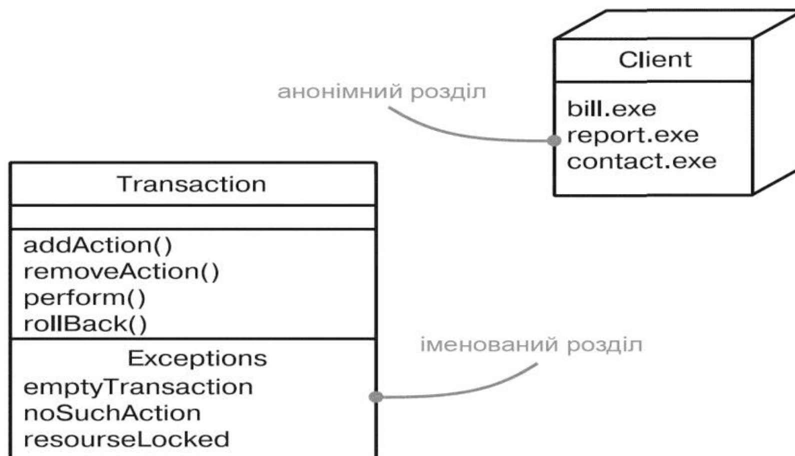


Рис. 2.1.18. Додаткові розділи

UML визначає ряд стандартних стереотипів (stereotype)для моделювання інших стереотипів (розроблення інструментальних засобів). Він визначає, що даний класифікатор є стереотипом і може застосовуватися до інших елементів. UML можна адаптовувати під конкретні цілі або предметну область. Зокрема, при використовуванні UML-моделі для ґенерації коду, доцільним є визначення стереотипів для підказки генератору коду (*Java,* C++).

Профіль (profile) – це UML-модель із набором готових стереотипів, присвоєних значень, обмежень і базових класів. [1] Профілі визначають спеціалізовані версії UML для моделювання певної предметної області. Профілі створюються розробниками інструментальних засобів, каркасів для різних моделей (*підтримки мов програмування БД, різних платформ, інструментів моделювання, бізнес-додатків*).

Моделювання коментарів. Використання приміток у моделі перетворює її в «склад» корисних артефактів розроблення, в тому числі для візуалізації вимог і їх зв’язку з частинами моделі. Довгі коментарі розміщаються у зовнішньому документі й зв’язуються приміткою, приєднаною до моделі. У міру розвитку моделі слід зберігати коментарі, що відображають тільки істотні рішення, які неможливо витягти із самої моделі. Рис. 2.1.19 демонструє модель, яка перебуває в процесі розроблення ієрархії класів, показуючи деякі вимоги, що формують модель, та деякі примітки з оцінювання дизайну. Один із коментарів (*примітка внизу*) є гіперпосилання на інший документ.

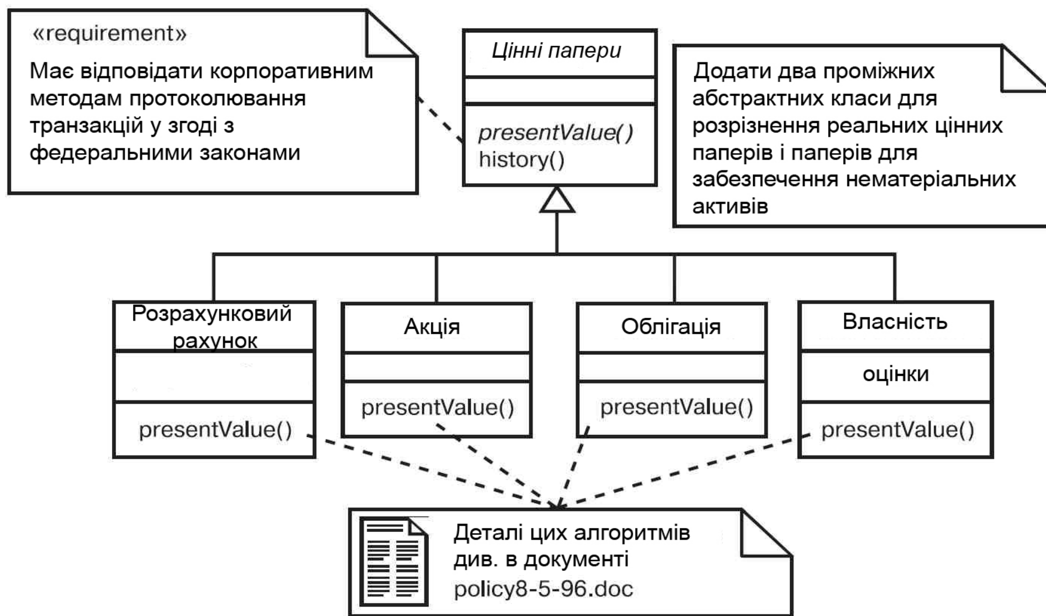


Рис. 2.1.19. Моделювання коментарів

Із використанням обмежень до елементів та залежностей між ними можна отримати нову семантику UML. У системі управління персоналом (рис. 2.1.20) кожна людина (Person) може бути членом однієї чи кількох команд (Team) або не входити в жодну; у кожній команді повинен бути як мінімум один керівник; кожна людина може бути керівником однієї або кількох команд чи не керувати ні однією. Засобами стандартного UML не можна виразити те, що керівник повинен

Рис. 2.1.20. Моделювання нової семантики

бути членом команди, яку він очолює, це стосується кількох асоціацій. Потрібно описати обмеження, яке показує, що керівник належить до членів команди, і зв'язати обидві асоціації обмеженням. В обмеженні додатково вказується, що керівник повинен бути членом команди не менше року.

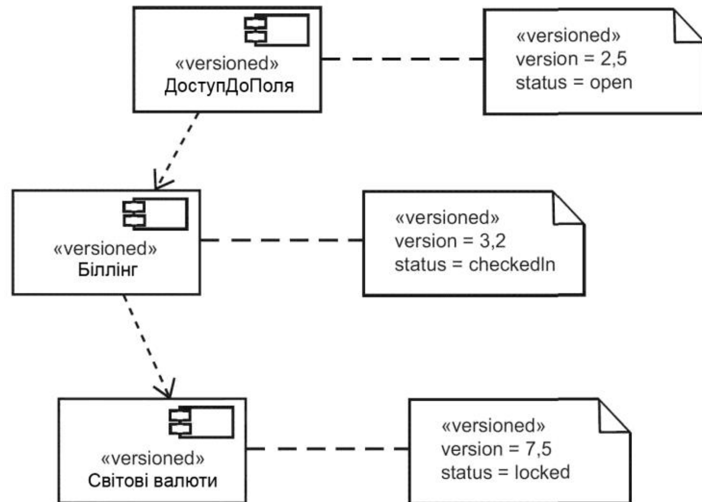


Рис. 2.1.21. Моделювання нових властивостей

Практичне використання розширень. Прив'язка створеної моделі до системи керування конфігурацією проекту вимагає відслідковувати номери версій, поточний статус «check in/check out», дату і час створення/модифікації кожної з підсистем. Ця інформація і не є базовою частиною UML і не виражається атрибутами класів. Номер версії – частина метаданих (не моделі). Рис. 2.1.21 демонструє три підсистеми, кожна з яких розширена стереотипом «versioned» для вказівки номера версії і статусу.

2.1.3. Розширені класи. Класифікатори

Основні питання:

- *Класифікатори, особливості атрибутів і операцій, різні види класів*
- *Моделювання семантики класу*
- *Вибір правильного типу класифікатора*

Класи є різновидом більш загальних будівельних блоків UML - класифікаторів.

Класифікатор (classifier) – механізм описування структурних і поведінкових властивостей елемента системи, які, крім стандартних властивостей (*атрибутів і операцій*), мають багато розширених, що дозволяють моделювати **множинність, видимість, сигнатури, поліморфізм** та інші характеристики. UML дозволяє на будь-якому рівні формалізації моделювати семантику абстракцій опису сутностей реального світу та складових проектного рішення. [5,7] *На початку проектування будинку ухвалюється рішення щодо будматеріалів (дерево, бетон, арматура). Вибір матеріалу диктується вимогами проекту (арматура і бетон підходять для будівництва в сейсмічних районах). Вибраний на початковому етапі матеріал диктує проектні рішення (вибір між деревом і металом вплине на масу будови…). Продовжуючи роботу над проектом, уточнюються базові проектні рішення і* додаються деталі, істотні для інженера-проектувальника *(перевірка безпеки конструкцій) і будівельника (виконання будівельних робіт).*

Розширені властивості класів. На ранній стадії програмного проекту достатньо включити в систему один або кілька класів, що будуть виконувати певні обов'язки. В міру уточнення архітектури і переходу до конструювання приймаються рішення щодо структури і поведінки (*атрибутів* і *операцій*) цих класів, необхідних для виконання їх зобов'язань.

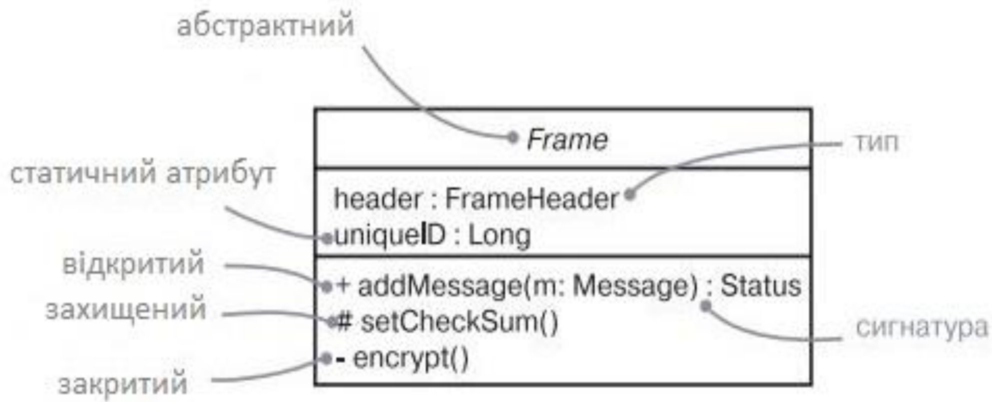


Рис. 2.1.22. Розширені класи

На етапі реалізації потрібно моделювати такі деталі, як видимість окремих атрибутів і операцій, семантика паралелізму класу в цілому й окремих операцій та інтерфейси, які він реалізує. Розширені властивості UML (рис. 2.1.22) дозволяють ВСКД класи на будь-якому рівні деталізації.

Види класифікаторів. Основний вид класифікаторів для ООМ є клас (*як опис набору об'єктів з однаковими атрибутами, операціями, зв'язками і семантикою*). Іншими важливими для ООМ класифікаторами UML є:

інтерфейс – набір операцій, що використовуються для специфікації сервісу класу або компоненти; [2]

тип даних – тип, значення якого незмінні: примітивні вбудовані типи (числа, рядки), типи перерахувань (Boolean та ін.); [2]

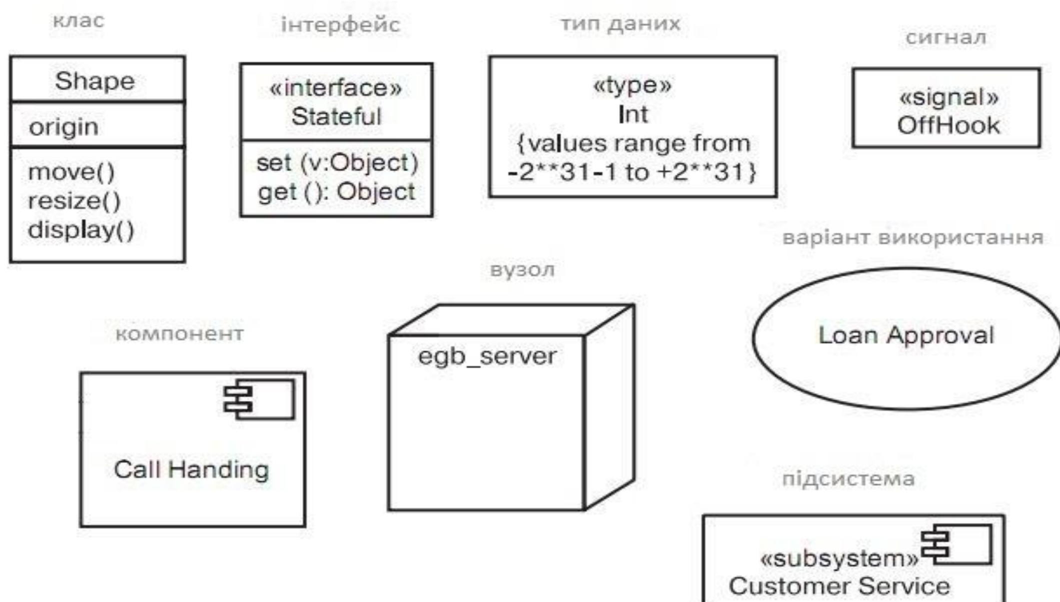


Рис. 2.1.23. Графічні UML-нотації різних видів класифікаторів

асоціація – опис набору посилань, кожне з яких з'єднує ряд об'єктів; [13]

сиґнал – специфікація асинхронного повідомлення, переданого між екземплярами; [2]

компонента – модульна частина системи, що приховує свою реалізацію за набором зовнішніх інтерфейсів; [1,2]

вузол – фізичний елемент, що існує під час виконання, і представляє обчислювальний ресурс, наділений пам'яттю й обчислювальними можливостями; [1,13]

варіант використання – опис послідовності дій (*включаючи їх різновиди*), які здійснює система, що породжують значущий результат для певної діючої сутності; [5]

підсистема компонентів, що представляє головну частину системи. [1]

При моделюванні систем за допомогою кожного з цих класифікаторів можна використовувати всі розширені властивості для забезпечення достатнього рівня деталізації й передавання змісту абстракції (рис. 2.1.23).

Видимість (visibility) – деталь проектування атрибута чи операції, що вказує на можливість використання даного атрибута чи операції іншими класифікаторами. [6] В UML можна специфікувати чотири рівні видимості:

public (*відкритий*) – позначається символом «+» перед іменем атрибута або операції. Будь-який зовнішній класифікатор може використовувати цю властивість;

protected (*захищений*) – позначається символом «#» (*дієз*) перед іменем атрибута або операції. Будь-який спадкоємець класифікатора може використовувати дану властивість;

private (*закритий*) – позначається символом «-» (*дефіс*) перед іменем атрибута або операції. Цю властивість може використовувати тільки сам класифікатор;

package (*пакетний*) – позначається символом «~» (*тильда*) перед іменем атрибута або операції. Тільки класифікатори, *оголошені в тому ж пакеті*, можуть використовувати дану властивість. На рис. 2.1.24 показана сукупність відкритих, захищених і закритих атрибутів і операцій класу Toolbar.

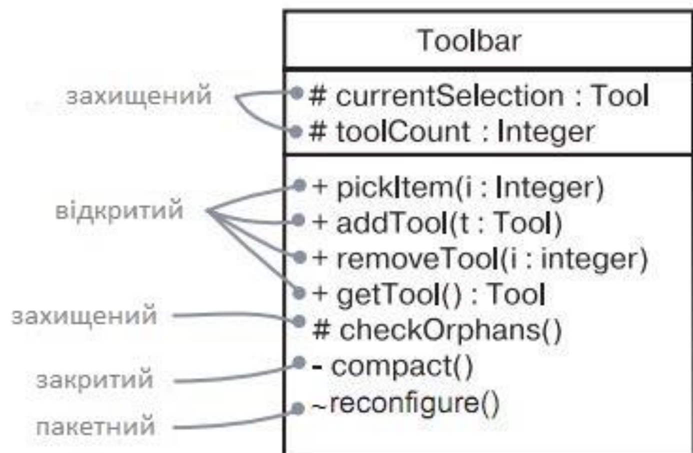


Рис. 2.1.24. Видимість атрибутів та операцій

Керування реалізацією класифікатора здійснюється через використання засобів видимості класифікатора, приховуючи певні деталі реалізації і роблячи видимими тільки ті властивості, які необхідні йому для виконання своїх обов'язків. Це основа інкапсуляції (*приховання інформації*), що забезпечує побудову стійкої системи. Якщо властивість класифікатора не визначена явно символом видимості, то за замовчуванням береться public.

Область дії екземпляра й статична область дії. Область дії (scope) властивості (*атрибута, операції*) класифікатора вказує на те, чи кожен екземпляр класифікатора має власне значення цієї властивості, чи повинно бути тільки одне її значення для всіх екземплярів класифікатора (*статичні члени класу в* C++). В UML є два типи області дії:

instance – **область дії екземпляра** (exemplar scope): кожен екземпляр класифікатора має власне значення цієї властивості. Це є варіант за замовчуванням (*не вимагає додаткової нотації*);

static – **статична область дії** – область дії класу (class scope): є тільки *одне значення властивості для всіх екземплярів* класифікатора.

Атрибут зі статичною областю дії (class scope) позначається **підкресленням** (рис. 2.1.25). Для відображення області дії екземпляра *не використовуються ніякі доповнення*. Більшість властивостей (*атрибутів і операцій*) класифікаторів, що моделюються, мають область дії екземпляра. У статичній області дії перебувають *закриті атрибути*, однакові для всіх екземплярів класу.

Статичні операції. Операція екземпляра має неявний параметр, який вказує на об'єкт, що її викликає. *Статичні операції прив'язані до класу*, а не до екземплярів і використовуються для роботи зі статичними атрибутами.

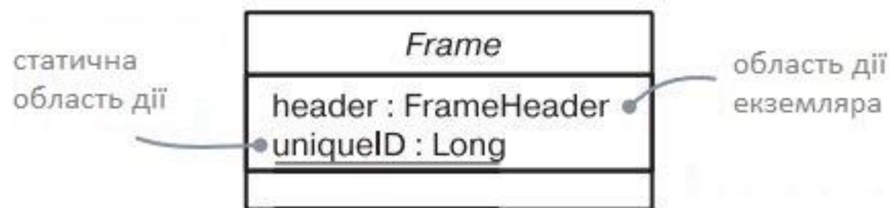


Рис. 2.1.25 Область дії атрибута, операції

Абстрактні, листові й поліморфні елементи (*класи*). Зв'язки узагальнення використовуються для моделювання структури класів із загальнішими абстракціями, розташованими на вершині ієрархії і детальнішими – внизу. У межах такої ієрархії деякі класи часто визначаються як **абстрактні** – вони не можуть мати **прямих (***безпосередніх***) екземплярів (***хоча екземпляри його нащадків можуть існувати в будь-якій кількості*). Класи Icon (Піктограма), RectangularIcon (ПрямокутнаПіктограма) і ArbitraryIcon (ПіктограмаДовільноїФорми) (рис. 2.1.26) – абстрактні класи. На відміну від них, конкретні класи Button (Кнопка) і Okbutton (КнопкаОК) можуть супроводжуватися екземплярами.

Новий клас створюється з метою успадкування його властивостей від інших, більш загальних класів, а також щоб він мав нащадків – більш спеціалізовані класи, що успадковують його властивості (*це нормальна семантика класів у UML*). Однак можуть бути випадки, що клас не повинен мати нащадків. Такий клас називається **листовим** (*якого заборонено успадковувати*) і позначається в UML властивістю leaf, записаним під іменем класу. Клас Okbutton є листовим класом (рис. 2.1.26).

Абстрактні, листові й поліморфні операції. Операції мають аналогічні властивості. Як правило, операція є **поліморфною**, тобто її можна специфікувати операцію *з тією же сигнатурою* в різних місцях ієрархії класів. **Операція в дочірньому класі** *скасовує поведінку* такої ж операції батьківського класу (*пріоритетна перед нею*). Коли повідомлення посилається під час виконання, конкретна операція, що викликається, вибирається з ієрархії **поліморфно**: *визначається під час виконання відповідно до класу об'єкта*. Операції display (відобразити) і IsInside (всередині) поліморфні (рис. 2.1.26). Операція *Icon::display()* є **абстрактна** операція, що не реалізована в класі *Icon* і «*вимагає*» від нащадків надання *її* **власних реалізацій**.

Імена абстрактних операцій в UML за аналогією з абстрактними класами виділяються *курсивом*. Операція *Icon*::getid() є **листовою** операцією (*забороненою для успадкування*), на що вказує ключове слово leaf. Вона **не є поліморфною** і не може бути перевизначена в класах-нащадках. В Java такі операції називаються final (*кінцевими*).

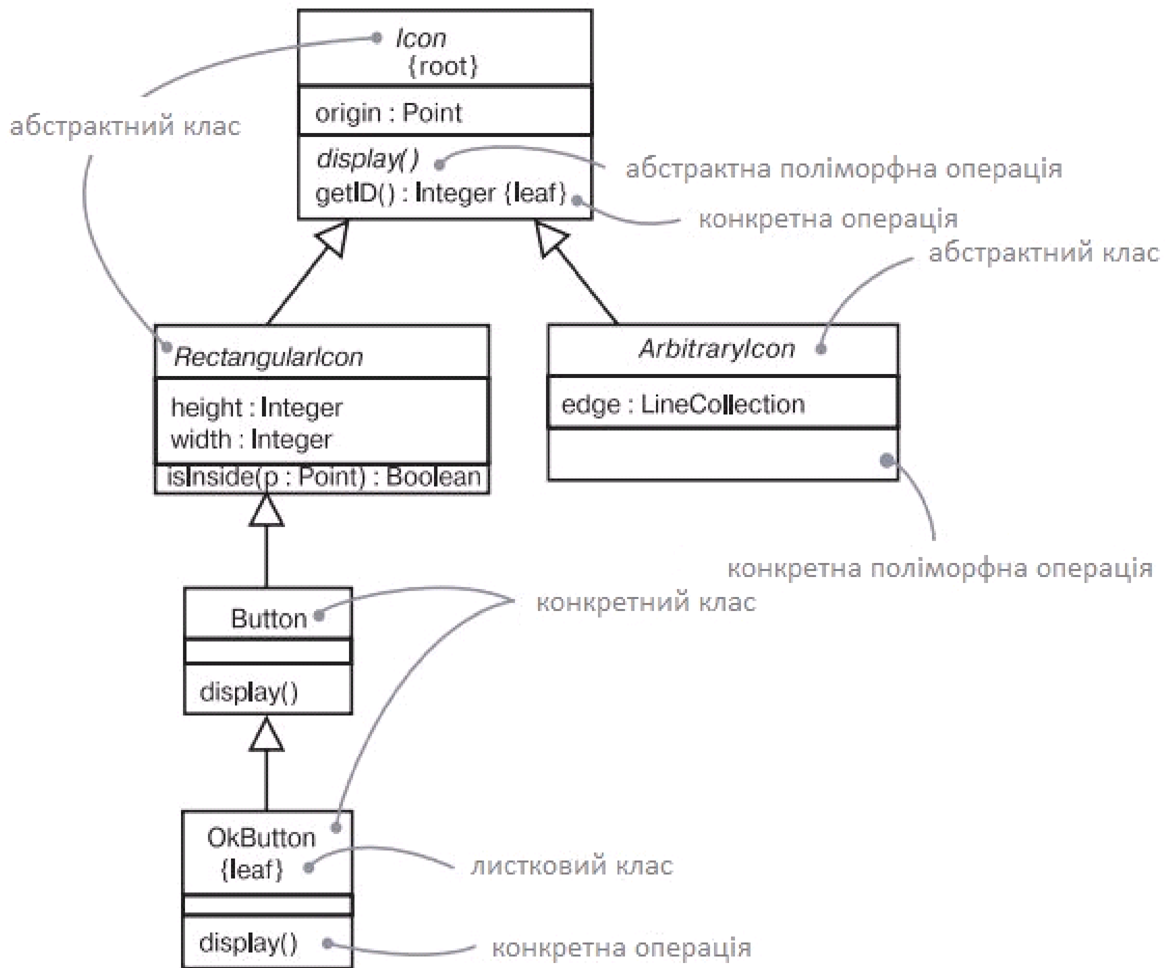


Рис. 2.1.26. Абстрактні й конкретні класи та операції

Множинність (multiplicity) – це діапазон допустимих значень кількості сутностей (*екземплярів та атрибутів*). Число екземплярів класу може бути довільне (*необмеженим згори, якщо він не є абстрактним*). [1,2] Можна обмежити кількість екземплярів класу або вказати, що воно дорівнює нулю (при використанні **класу-утиліти**, що представляє тільки статичні операції і атрибути), одиниці (**клас-одинак**), певному числу або невизначеній множині (*за замовчуванням*).

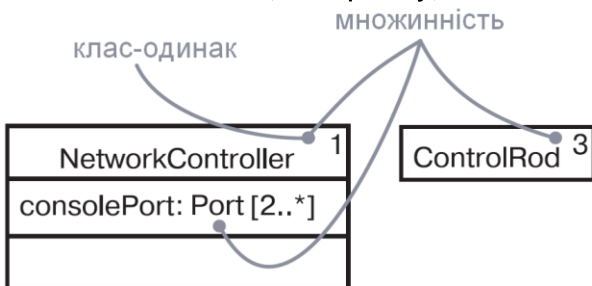


Рис. 2.1.27. Множинність

Множинність вказується в правому верхньому куті піктограми класу (рис. 2.1.27). Клас NetworkController (КонтролерМережі) є клас-одинак (singleton), для класу ControlRod (РегулюючийСтержень) повинно бути рівно три екземпляри.

Для атрибутів множинність встановлюється відповідним виразом у квадратних дужках відразу після імені й типу атрибута. Всередині екземпляра consolePort може бути кілька екземплярів (рис. 2.1.27).

Розширення для атрибутів. На найабстрактнішому рівні при моделюванні атрибутів вказуються тільки їх імена, що є достатнім для розуміння моделі пересічним користувачем. Однак можна специфікувати видимість, область дії, множинність, а також тип, початкове значення й мінливість кожного атрибута, використовуючи повний синтаксис атрибута в UML:

[видимість] ім'я [':' тип] ['['множинність']']
['='початкове значення] [рядок властивостей {, рядок властивостей}]

Приклади коректних оголошень атрибута origin:

- origin тільки ім'я
- + origin видимість та ім'я
- origin : Point ім'я і тип
- name : tnng[0..10] ім'я, тип і множинність
- origin : Point = (0,0) ім'я, тип і початкове значення
- id : Integer readonly ім'я і властивість (*незмінності*)

Константні атрибути (readonly). За замовчуванням атрибути вважаються завжди змінювані, якщо нічого явно не зазначено. Щоб вказати, що значення атрибута не може бути змінене після ініціалізації об'єкта, використовується властивість readonly.

Операції. На найвищому рівні абстракції при моделюванні поведінкових властивостей класу (*операцій і сигналів*) просто вказується ім'я кожної операції, що є достатнім для загального розуміння моделі. *Сиґнатура операції* включає ім'я, разом з її параметрами, тип значення, що повертається (якщо воно є). Можна специфікувати видимість і область дії кожної операції, її параметри, тип значення, що повертається, семантику паралелізму та інші властивості. Повний синтаксис операції в UML:

[видимість] ім'я ['(' список параметрів ')'] [':' тип значення, що повертається,]
[рядок властивостей {, рядок властивостей}]

Приклади коректних оголошень операції:

- display тільки ім'я
- + display видимість та ім'я
- set(n : Name, s : String) ім'я і параметри
- getID : Integer ім'я і тип (клас) повернення
- restart () {guarded} ім'я і властивість

Синтаксис параметрів. У сиґнатурі операції можна вказати нуль, один або декілька параметрів, кожний з яких виражається наступним синтаксисом:

[direction] name : type [=default-value]

direction (*напрямок*) може набувати одне із значень:

- in - вхідний параметр, не може бути модифікований

- out – вихідний параметр, може бути модифікований для передавання інформації коду, що його викликав;
- inout – вхідний параметр, може бути модифікований для передавання інформації коду, що його викликав.

Розширені властивості операцій (*в доповнення до* leaf і abstract):

guery (*запит*): виконання операції залишає стан системи без змін: операція є простою операцією, що не має побічних ефектів;

sequential (*послідовна*): код, що її викликав, повинен бути скоординований поза об'єктом, щоб існував тільки один потік в об'єкті в одиницю часу. При наявності множини потоків керування семантика і цілісність об'єкта не ґарантуються;

guarded (*захищена*): в результаті тільки одна операція об'єкта може бути викликана в одиницю часу, що зумовлює послідовну семантику. Семантика і цілісність об'єкта ґарантуються за наявності множини потоків керування за рахунок вибудовування послідовності всіх викликів захищених операцій об'єкта;

concurrent (*паралельна*): при паралельній операції для одного об'єкта можуть здійснюватися одночасно множинні виклики з паралельних потоків керування, які повинні оброблятися паралельно, з коректною семантикою;

static (*статична*): операція не має цільового об'єкта, що її викликає (*є глобальною для всіх екземплярів класу: викликається шляхом вказання імені класу /* Icon::getid() /).

Властивості паралелізму (sequential, guarded і concurrent) призначені для підтримки семантики паралелізму операцій. Ці властивості є суттєвими тільки в присутності активних об'єктів, процесів або потоків.

Шаблонні класи. Шаблон – це параметризований елемент. C++, Java дозволяють створювати шаблонні класи і шаблонні методи, кожен з яких визначає множину класів та методів. Шаблони можуть включати **слоти** (параметри шаблонів) для класів, об'єктів і значень. Шаблон сам по собі не використовується: спочатку слід створити **екземпляр шаблона.** [3] Цей процес припускає *зв'язування формальних параметрів шаблона з реальними*. В результаті для шаблонного класу створюється конкретний клас (*реалізація шаблона*), який можна використовувати як будь-який інший реальний клас. Шаблонні класи застосовуються для опису контейнерів, екземпляри яких використовуються при створені елементів інформаційних систем, забезпечуючи високий рівень безпеки даних. Поданий нижче фрагмент коду на C++ оголошує про створення параметризованого класу Map:

```
template <class Item, class Vtype, int Buckets>
class Map
 {
   public:
     virtual map(const Item&, const &Vtype); // віртуальний конструктор
     virtual Boolean isMappen(const Item&) const;
     …
  };
```

На підставі цього можна створити *екземпляр цього шаблона* для створення контейнера типу **Map** (відображення), що визначає відношення відображення об'єктів класу Customer (Покупець) на об'єкти класу Order (Замовлення):

m : **Map** <Customer, Order, 3>;

UML дозволяє ефективно моделювати шаблонні класи (рис. 2.1.28), що зображуються як звичайні, але з додатковою *пунктирною рамочкою* у верхньому правому куті піктограми класу, де перераховуються **слоти** - *параметри шаблона*.

Створення екземпляра шаблонного класу. Явне і неявне зв'язування. При моделюванні створення екземпляра шаблонного класу можна двома способами: явно і неявно. У першому випадку вказується ім'я що описує зв'язування; у другому – застосовується *стереотип залежності* (bind), який вказує, що джерело створює екземпляр *цільового шаблона*, використовуючи реальні параметри (рис. 2.1.28). Для обох варіантів зв'язування на діаграмі для створюваного контейнера слід у кутових дужках ***стрілочками*** вказувати зв'язки відповідності між шаблонними параметрами та реальними параметрами (*класами об'єктів, якими наповнюватиметься контейнер*) (рис. 2.1.28).

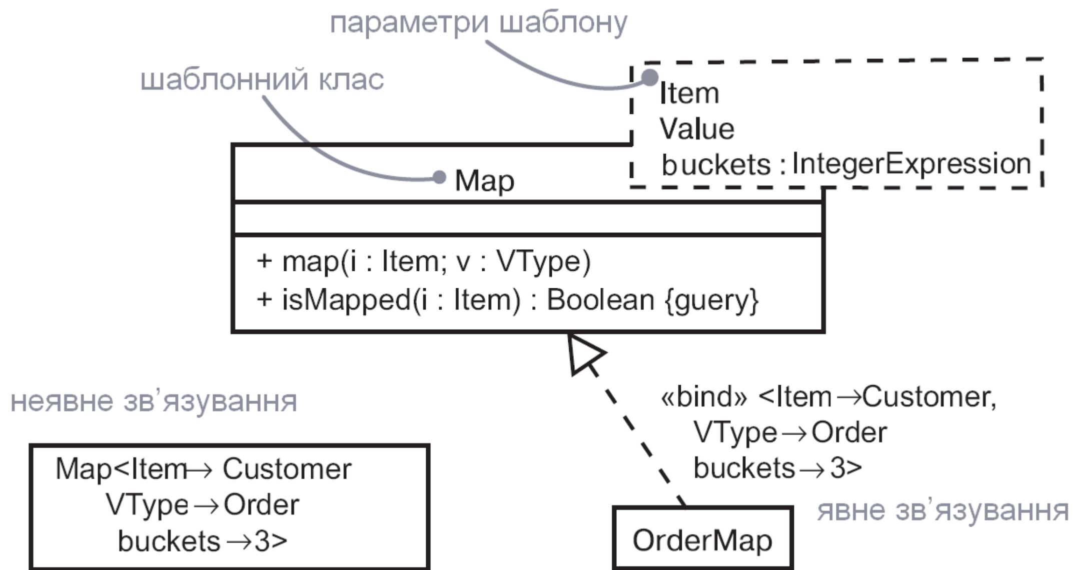


Рис. 2.1.28. Шаблонні класи

Стандартні стереотипи розширення класифікаторів. UML визначає чотири стандартні стереотипи розширення властивостей класів:

metaclass – описує класифікатор, об'єктами якого є всі класи;

powertype – описує класифікатор, об'єктами якого є дочірні класи заданого батьківського класу;

stereotype – вказує на те, що класифікатор є стереотипом, який може бути застосований до інших елементів;

utility – описує клас, атрибути й операції якого мають статичну область дії.

Для розширення властивостей класу можуть використовуватися присвоєні значення (*специфікація версії класу*) і стереотипи для специфікації нових видів компонентів (*специфічних для моделі*).

Моделювання семантики класу. Головна мета використання класів – моделювання абстракцій прикладної проблеми й технології її розв'язання. Наступним кроком після ідентифікації абстракцій є **специфікація їх семантики,** основними стратегіями формалізування якої є: визначити обов'язки класу (*контракт або зобов'язання класу);* специфікувати семантику класу як єдиного цілого, використовуючи структурований текст, поданий у вигляді примітки зі стереотипом semantics,

приєднаним до класу; *описати тіло кожного методу*, використовуючи структурований текст або мову програмування у формі примітки, з'єднаної з операцією зв'язком залежності; задати перед- і постумови кожної операції, використовуючи структурований текст (*у примітках відповідно до стереотипів* precondition, postcondition і invariant, *з'єднаних з операціями або класами залежностями*); специфікувати для класу поведінку через автомат (state machine), що описує послідовність станів об'єкта за час свого ЖЦ у відповідь на події, разом з його реакцією на ці події; проробити внутрішню структуру класу та специфікувати кооперацію, що представляє клас. *Кооперацією є співтовариство ролей і інших елементів, які працюють спільно для організації певної загальної поведінки, більшої від поведінки простої суми цих елементів.*

Кооперація має як структурну, так і динамічну частини, тому її можна використовувати для описування всіх сторін семантики класу.

Основні характеристики добре структурованого класифікатора: визначені структурні й поведінкові аспекти; висока погодженість і слабка зв'язність; розкриває тільки ті властивості, які необхідні клієнтам, щоб його використовувати, і приховує всі інші; є недвозначний щодо призначення і семантики; має достатній для розуміння рівень деталізації.

2.1.4. Екземпляри

Основні питання:

- *Екземпляри й об'єкти*
- *Моделювання конкретних екземплярів*
- *Моделювання прообразу екземплярів*
- *Реальний і абстрактний світ екземплярів*

Терміни «екземпляр» і «об'єкт» у більшості випадків є синонімами і часто бувають взаємозамінними.

Екземпляр (instance) – конкретне втілення абстракції, до якого можуть бути застосовані певний набір операцій, який має стан, що володіє можливістю збереження результатів цих операцій. Екземпляри використовують для моделювання конкретних сутностей реального світу. [2] Майже всі будівельні блоки UML виражаються через дихотомію «*клас/об'єкт*»: ВВ, вузли, асоціації, їх екземпляри тощо. *Ще одна аналогія з будівництва будинку. Використовуючи слово «будинок», а не «автомобіль» чи інше поняття, заздалегідь звужується словник предметної області. Будинок є абстракцією «постійного або тимчасового житла, призначення якого – надати притулок». У ході роботи, що полягає в з'єднанні різних, найчастіше суперечливих вимог, модель будинку поступово конкретизується: можна вибрати будинок із трьома спальнями і гараж …*

Отримавши ключі від готового будинку і увійшовши усередину, замовник поринає в реальну обстановку – оцінює будинок не просто як будинок із трьома спальнями, а як «власний будинок із трьома спальнями, розташований за такою адресою». Між «будинком із трьома спальнями» і «власним будинком із трьома спальнями» існує фундаментальна відмінність: перший – абстракція, що описує певний тип будинку, другий є конкретним

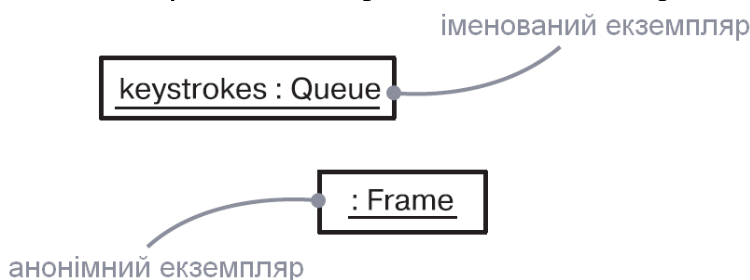


Рис. 2.1.29. Екземпляри

екземпляром цієї абстракції, втілений у реальній формі й кожна його властивість має реальне значення.

Абстракція та **екземпляр**. Абстракція описує *ідеальну суть* предмета, екземпляр – *конкретне втілення*. В однієї абстракції може бути скільки завгодно екземплярів. Для конкретного екземпляра завжди існує абстракція, що визначає характеристики усіх подібних екземплярів. UML дозволяє представляти і здійснювати поділ на абстракції та екземпляри для всіх сутностей моделювання. Практично всі UML – будівельні блоки (*класи /крім абстрактних/, компоненти, вузли, ВВ і асоціації*) можуть бути промодельовані в *термінах своєї сутності* або *своїх екземплярів*. Здебільшого працюють з ними як з абстракціями, але при моделюванні конкретних втілень мають справу з екземплярами. Графічна нотація екземплярів дозволяє візуалізувати як *іменовані*, так і *анонімні* екземпляри (рис. 2.1.29). Екземпляри всіх абстракцій зображують з *підкресленими іменами*.

Екземпляри не існують самі по собі, вони *завжди пов'язані з абстракцією*. Об'єкт займає деяке місце в реальному чи концептуальному світі і над ним можна виконувати певні операції. *Екземпляром вузла є комп'ютер, фізично розташований у якомусь приміщенні; екземпляром компоненти – файл, розміщений у певному каталозі; екземпляр запису про клієнта займає якийсь обсяг оперативної пам'яті комп'ютера. Екземпляр траєкторії польоту літака є конкретний слід, що піддається математичному опису*.

Прямі й непрямі екземпляри. За допомогою UML можливо моделювати не тільки безпосередні фізичні екземпляри, а й менш конкретні сутності. Абстрактний клас (*за означенням*) не може мати *безпосередніх екземплярів*, однак дозволяється моделювати *непрямі екземпляри* абстрактних класів, показуючи як даний абстрактний клас можна використовувати в якості прототипу і як його абстрактні операції поліморфно перетворюються в екземпляри операцій тих же непрямих екземплярів абстрактних класів. Хоча об'єкта абстрактного класу не існує, *з практичної точки зору він дозволяє надати ім'я будь-якому потенційному екземпляру конкретного нащадка цього абстрактного класу*.

Аналогія абстрактних класів та інтерфейсів. Аналогічно до абстрактних класів, **інтерфейси** (*за означенням*) також не мають безпосередніх екземплярів. Можна змоделювати **екземпляр-прототип** інтерфейсу, що представлятиме один з *потенційних екземплярів конкретних класів, які реалізують даний інтерфейс*. Модельовані екземпляри вміщаються у **діаграми об'єктів** (*для показу їх структурних деталей*), або в **діаграми взаємодії та діяльності** (*для візуалізації поведінки*). Їх можна включати у діаграми класів, якщо треба показати зв'язок об'єкта і його абстракції.

У кожного екземпляра є свій **тип**. Тип екземпляра повинен бути його **конкретним класифікатором**, але специфікація екземпляра (*що не є конкретним екземпляром*) може мати абстрактний тип. Класифікатор екземпляра є **статичним**. [2] Після створення екземпляра класу останній не зміниться протягом усього часу існування об'єкта. Однак у деяких ситуаціях моделювання і мовах програмування можлива зміна абстракції екземпляра (*об'єкт* Caterpillar (Гусениця) *може стати об'єктом іншої абстракції* Butterfly (Метелик)). Це той же об'єкт, але вже належний до іншої абстракції.

Ім'я екземпляра. Екземпляр (об'єкт) існує в контексті операції, компонента або вузла. [2] У текстовому рядку спочатку вказується його ім'я (*що відрізняє його в даному контексті від інших екземплярів*) і через двокрапку – тип абстракції (objT : Transaction). Ім'я екземпляра може включати тільки ім'я екземпляра без вказання типу абстракції (objT, *myCustomer* (мійПокупець), а також назву типу абстракції, якому

передує двокрапка без назви імені екземпляра (:Transaction - *безіменний екземпляр*), що є простими іменами (рис. 2.1.30). Абстракція екземпляра може мати кваліфіковане ім'я (Multimedia::AudioStream), що утворюється шляхом додавання перед іменем абстракції *імені пакета*, що її включає.

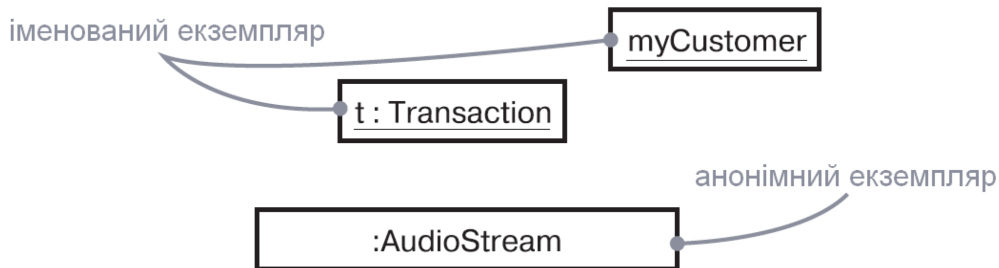


Рис. 2.1.30. Іменовані та неіменовані екземпляри

Об'єкт можна іменувати явно, роблячи його ім'я осмисленим для користувача. Можна дати йому просте ім'я і сховати його абстракцію, якщо вона очевидна з контексту. Однак найчастіше справжнє ім'я об'єкта відомо тільки комп'ютеру, який з ним працює. У таких випадках з'являється **анонімний об'єкт** (:Audiostream). Кожна поява анонімного об'єкта є відмінною від усіх інших його появ. Якщо навіть невідома абстракція об'єкта, то ім'я потрібно дати йому явне. Ім'я і тип об'єкта записуються в один рядок, що увесь є *підкреслений* (t:Transaction), на відміну від *ролі* (t:Transaction).

Операції. Об'єкт не просто займає певне місце в реальному світі – він піддається певним маніпуляціям. Операції, виконувані об'єктом, оголошуються в його абстракції. Наприклад, якщо клас Transaction (Трансакція) містить операцію commit (зробити) і в нього є екземпляр t:Transaction, то можна написати вираз t.commit(). Його виконання означає, що над об'єктом t здійснюється операція commit. Залежно від пов'язаної із цим класом ієрархії успадкування, дана операція може бути викликана поліморфно.

Стан об'єкта. Кожен об'єкт ще має стан, що визначається сукупністю усіх його властивостей і їх поточних значень (*включаючи також посилання й зв'язані об'єкти, залежно від точки зору*). До числа властивостей входять атрибути й асоціації об'єкта, а також усі його аґреговані частини. Отже, стан об'єкта динамічний і при його візуалізації фактично описується *значення його стану в цей момент часу і в даній точці простору*.

Зміна стану об'єкта. Процес зміни стану об'єкта можна відобразити графічно, якщо на одній і тій же взаємодії зобразити його кілька разів, причому кожне зображення буде відображати новий стан. Виконуючи над об'єктом операцію, змінюється його стан, однак **при запиті об'єкта його стан не змінюється**. При бронюванні квитка на літак (об'єкт r:Замовлення), визначається значення одного з атрибутів (наприклад, ціна квитка = 395.75). Згодом, змінивши умови замовлення, скажімо, додавши до маршруту ще одну пересадку, тим самим змінюється його стан: наприклад, ціна квитка дорівнює 1024.86.

Рис. 2.1.31 демонструє, як зображувати значення атрибутів об'єкта засобами UML. Значення атрибута id об'єкта myCustormer дорівнює "432-89-1783". У цьому

випадку тип ідентифікатора (SSN – номер соціального страхування) показаний явно, хоча його можна й вилучити (як це зроблено для атрибута active = True), оскільки тип утримується в оголошенні id в асоціййованому класі об'єкта myCustomer.

Автомат. Із класом можна асоціювати також автомат, що необхідно при моделюванні систем, керованих подіями або ЖЦ класу. Для таких випадків можна показати стан автомата для даного об'єкта в цей момент часу. Стан зображується у квадратних дужках після типу. [2] Об'єкт c – екземпляр класу Phone (Телефон) перебуває в стані Waitingforanswer (Чекає відповіді), певному в автоматі для класу Phone (рис. 2.1.31).

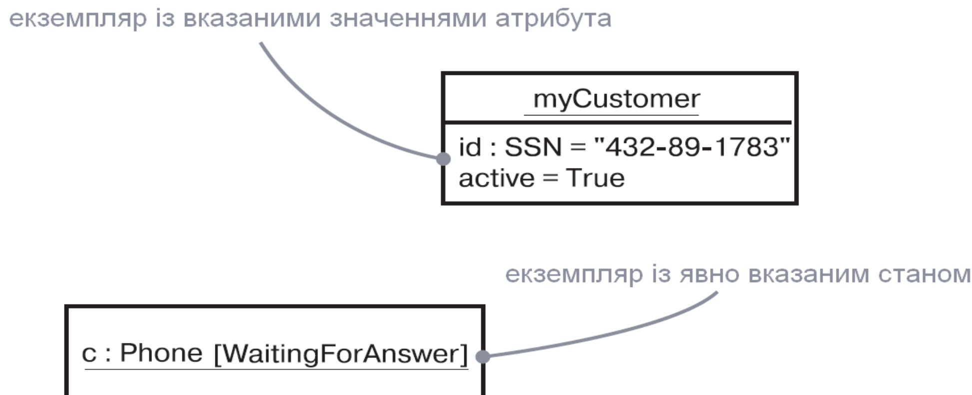


Рис. 2.1.31. Стани об'єкта

Активні й пасивні елементи. В UML є графічна нотація для розрізнення активних і пасивних елементів, що визначають важливі компоненти керування системи – процеси і потоки. Елемент вважається **активним**, якщо він є частиною процесу або потоку і являє собою вихідну точку потоку керування. [2] Можна оголосити **активні класи**, що матеріалізують процес або потік і, відповідно, виділити **екземпляр активного класу** (рис. 2.1.32).

Посилання й асоціації. В UML існують ще два елементи, у яких можуть бути екземпляри. Перший – це асоціація. **Екземпляром асоціації** є **посилання** (link), що є семантичним з'єднання між об'єктами. [2] Як і асоціація, вона зображується у вигляді лінії; від асоціації її можна відрізнити за тим, що **посилання з'єднує тільки об'єкти**.

Рис. 2.1.33. Активні об'єкти

Статичний атрибут – атрибут, область дії якого обмежена класом. Статичний атрибут – це, по суті, об'єкт у цьому класі, спільно використовуваний усіма екземплярами класу. [2] В оголошенні класу *статичний атрибут* **підкреслюється**.

Рис. 2.1.34. Об’єкти зі стереотипами

Об’єкти зі стереотипами. До екземплярів не приписують стереотип безпосередньо і не зв’язують з ними позначених значень. Замість цього стереотипи й позначені значення об’єкта виводяться з асоційованих з ним абстракцій. Можна явно приписати стереотип самому об’єкту або його абстракції (рис. 2.1.34).

Зв’язки залежності між об’єктами і класами. В UML визначено два стандартні стереотипи, що застосовуються до зв’язків залежності між об’єктами і класами: instanceof – об’єкт-клієнт є екземпляром класифікатора-сервера, instantiate – клас-клієнт, створює екземпляри класифікатора-сервера.

Моделювання конкретних екземплярів. При моделюванні конкретних екземплярів візуалізуються сутності реального світу. Неможливо побачити екземпляр класу Client, якщо даний клієнт не є знаходиться в полі зору, однак можна побачити у відладчику вигляд цього об’єкта. Для моделювання топології комп’ютерної мережі, використовують діаграмами розміщення, що містять екземпляри вузлів, для моделювання компонентів, розташованих у фізичних вузлах – діаграмами компонентів, що містять їх екземпляри. Якщо до системи підключений відладчик, то можна представити структурні зв'язки між екземплярами за допомогою діаграм об’єктів.

Рекомендації при моделюванні екземплярів: ідентифікувати необхідні екземпляри, дати кожному власне ім’я (*якщо можна, або зобразити його як анонімний об’єкт*); виявити для кожного з них необхідні стереотипи, позначені значення й атрибути (разом з їхніми значеннями); зобразити ці екземпляри і зв’язки між ними на діаграмі об’єктів. На рис. 2.1.35 показана діаграма об’єктів, взята із системи перевірки роботоздатності кредитних карток (*вона є такою, як у відладчику тестування додатків*).

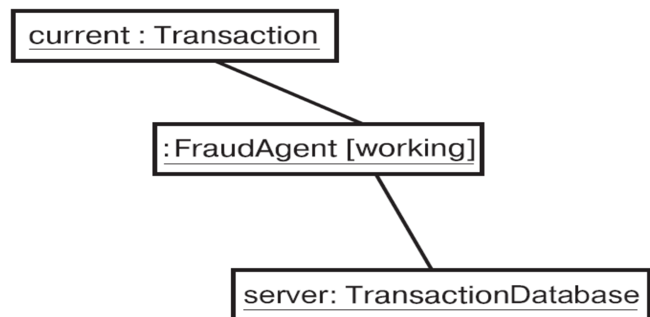


Рис. 2.1.35. Моделювання екземплярів

Моделювання екземплярів на UML означає моделювання конкретного втілення деякої абстракції (*класу, компоненти, вузла, ВВ або асоціації*). Добре структурований екземпляр є явно асоційований з конкретною абстракцією; має унікальне ім’я, взяте зі словника предметної області. При зображенні екземплярів в UML слід показувати ім’я абстракції, до якої належить екземпляр, стереотип, роль або стан екземпляра, якщо це необхідно для розуміння об’єкта в даному контексті; організовувати списки атрибутів екземпляра, ґрупуючи їх разом з їхніми значеннями відповідно до категорій.

2.2. Моделювання зв'язків і відношень

Основні питання:

- *Зв’язки: залежності, узагальнення, асоціації*
- *Моделювання простих залежностей*
- *Моделювання одиничного спадкування*
- *Моделювання структурних зв’язків*
- *Створення мереж зв’язків*

2.2.1. Типові зв'язки

При моделюванні системи потрібно не тільки ідентифікувати сутності, що формують її словник, але й моделювати їх відношення. Класи кооперуються з іншими класами найрізноманітнішими способами. Побудова мереж зв'язків є подібною до створення збалансованого розподілу обов'язків між класами: їх надмірне ускладнення робить модель незрозумілою; при надмірному спрощенні можна втратити все те багатство можливостей, що забезпечує систему кооперації. *Стіни, двері, вікна, освітлення формують частину словника системи. Однак жодна з цих речей не існує окремо: стіна з'єднується з іншими, двері й вікна вмонтовані в стіни, системи освітлення кріпляться до стін і стелі. Всі ці об'єкти в сукупності формують більш високорівневі сутності (кімнати), між якими є не тільки структурні, а й інші зв'язки: будинок має вікна, які є неоднакові (велике вікно у вітальні, маленьке кухонне…). Незалежно від цих відмінностей, у кожному вікні присутня певна ознака «віконності»: усі вони відкривають проріз у стіні й призначені для пропускання світла, повітря, іноді й мешканців.*

Детальніше про відношення і зв'язки. В об'єктно-орієнтованому моделюванні (ООМ) UML способи з'єднання сутностей між собою (*логічних і фізичних*) моделюються зв'язками, що описують найпоширеніші способи взаємодії і комбінування абстракцій:

залежності – зв'язком використання між класами, включаючи *уточнення, трасування і зв'язування* (*труби залежать від нагрівача води, що по них передається*); [2]

асоціації – структурним зв'язком між екземплярами (*кімнати складаються зі стін та інших об'єктів; у стіни вмонтовані двері, вікна*); [2]

узагальнення – зв'язком між узагальненими класи з їх спеціалізаціями («*клас-підклас*» або «*батько-нащадок*») (*вітраж – вікно з дуже великими, жорстко фіксованими панелями; патіо – різновид вікна, що відчиняється вбік*). [2]

Графічна UML-нотація дозволяє візуалізувати ці види зв'язків (рис. 2.2.1) і описувати найважливіші їх параметри (*ім'я сутності й властивості*) незалежно від конкретної мови ООП.

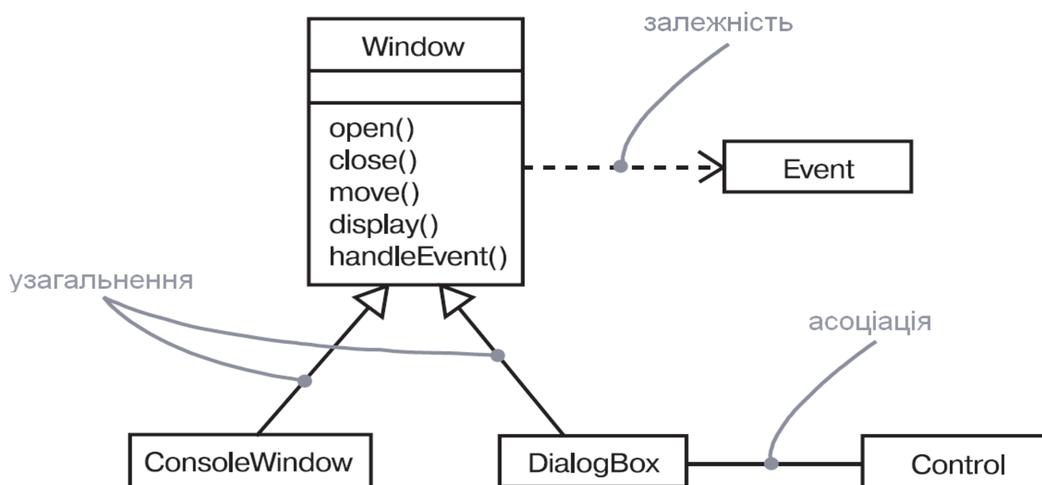


Рис. 2.2.1. Типи зв'язків

Зв'язок (relationship) – з'єднання сутностей, що зображується у вигляді шляху з використанням різноманітних типів ліній, що відповідають певним видам зв'язку. [1,2]

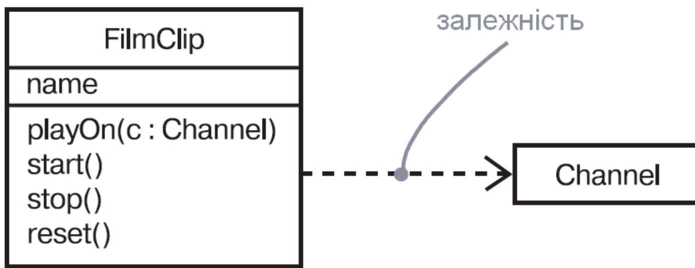


Рис. 2.2.2. Залежності

Залежність (dependence) – зв'язок, що встановлюється як одна сутність (*клас* Window /Вікно/), використовує інформацію і сервіс (*операцію* або *послугу*), надану іншою (*класом* Event /Подія/). Навпаки – не обов'язково. Залежність зображується у вигляді пунктирної лінії *зі стрілкою, спрямованої на залежну сутність*. [2] Залежність найчастіше застосовується при моделюванні, щоб показати як один клас використовує *операції чи атрибути іншого* класу (рис. 2.2.2). Це є **зв'язок використання**. Якщо використовуваний клас змінюється, то це стосуватиметься і операцій інших класів, що його використовують. UML допускає створення залежностей і між іншими елементами, зокрема між примітками і пакетами.

Узагальнення (generalization) – це зв'язок між сутністю загального характеру (суперкласом, батьківським класом) і більш специфічною сутністю (підкласом, дочірнім класом або нащадком). [2] Узагальнення ще називають зв'язком типу «є»: (*клас* Baywindow (Вікно-«ліхтар») *є різновидом більш загальної сутності – класу* Window). Об'єкти дочірнього класу можуть бути використані всюди, де зустрічаються об'єкти батьківського класу, а не навпаки. **Дочірня сутність** може бути підставлена там, де оголошена батьківська. Вона успадковує властивості батька: його атрибути й операції. Часто, хоча не завжди, нащадок має додаткові атрибути й операції, крім батьківських. [1,13] Реалізація операції в дочірньому класі заміщає реалізацію тієї ж операції батька (**поліморфізм**). Однакові операції повинні мати однакову **сигнатуру** (ім'я і параметри).

Узагальнення використовується для моделювання зв'язку «батько-нащадок» і графічно представлене суцільною лінією зі стрілкою у формі порожнього трикутника, *що вказує на батька* (рис. 2.2.3). Клас може мати одного або кількох батьків або не мати зовсім. Клас, що не має батьків, але має одного або кількох нащадків, називається **кореневим** (root) або **базовим**. Клас, що не має нащадків, називається **листовим** (leaf). Про клас, у якого є тільки один батько, кажуть, що він використовує **одиничне успадкування**, на відміну від класу, у якого більше, ніж один батько (**множинне успадкування**).

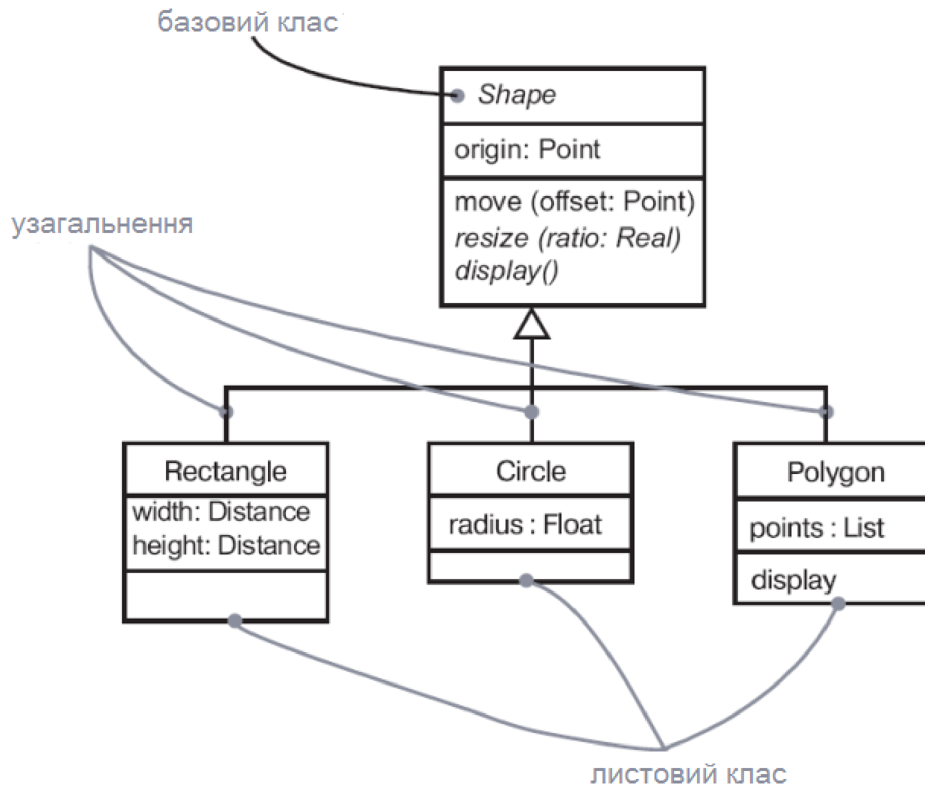


Рис. 2.2.3. Узагальнення

Найчастіше узагальнення використовується між класами та інтерфейсами (*відношення успадкування*). В UML можна описати узагальнення між іншими елементами (актантами, ВВ, вузлами).

Асоціація – є структурний зв’язок, який вказує, що об’єкти однієї сутності (класу) з’єднуються з об’єктами іншої. Допускається, щоб обидва кінці асоціації з’єднували **один і той же клас**: *один об’єкт класу може зв’язуватися з іншим об’єктом того ж класу*. Асоціація, що зв’язує два класи, називається **бінарною**. Можливі також **парні асоціації**, які з’єднують більше двох класів. UML-нотацією асоціації є суцільна лінія, два кінці якої з’єднують *один або різні* класи.

Розширення асоціації. Є п’ять доповнень, що застосовуються до асоціацій. Асоціація може мати **ім’я,** що використовується для описування природи **зв’язку** і яке **не** повинно бути двозначним. Використовуючи стрілочку трикутника, можна вказати напрямок (**навігацію**), у якому слід читати це ім’я (рис. 2.2.4).

Рис. 2.2.4. Імена асоціації

Коли клас бере участь в асоціації, він виконує в цьому зв’язку конкретну роль. **Роль** – це «обличчя» класу, яким клас, що знаходиться на одній стороні асоціації (*ближній кінець*), повернутий до класу, що знаходиться на іншій стороні (*далекий кінець*) асоціації. Можна явно іменувати роль, яку виконує клас в асоціації.

Роль, яку відіграє клас, що перебуває на кінці асоціації, називається **кінцевим іменем** (*у першій версії* UML – **іменем ролі**). На рис. 2.2.5 клас Persone (Особа), що відіграє роль employee (працівник), асоційований з класом Company (Компанія), що відіграє роль employer (роботодавець).

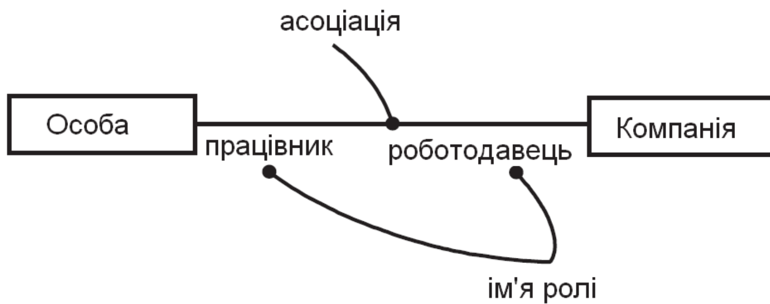


Рис. 2.2.5. Кінцеві імена асоціації (імена ролей)

Множинність ролі (role multiplicity) **асоціації.** Асоціація визначає структурний зв'язок між об'єктами і тому важливо знати, скільки об'єктів може бути з'єднано одним екземпляром асоціації. Цей параметр називається *множинністю ролі асоціації,* яка представляє діапазон цілих чисел, що вказує можливу кількість зв'язаних об'єктів і записується у вигляді виразу з мінімальним і максимальним значеннями, розділених двокрапкою. Встановлюючи множинність далекого кінця асоціації, вказується, скільки об'єктів може існувати на далекому кінці асоціації для кожного об'єкта класу на її близькому кінці. Кількість об'єктів повинна перебувати в межах заданого діапазону. Множинність може бути визначена як *одиниця* (1), *нуль або один* (0..1), *багато* (0..*), *один або багато* (1..*). Можна задавати діапазон цілих значень (2..5) або встановлювати точне число, наприклад 5 (еквівалент запису 5..5).

На рис. 2.2.6 кожна компанія може наймати одного або кількох людей (множинність 1..*); кожній людині зіставлено 0 або більше компаній-роботодавців (множинність * – еквівалент запису 0..*).

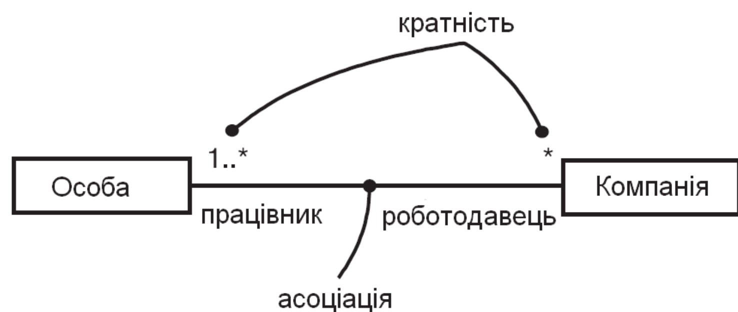


Рис. 2.2.6. Множинність

Проста асоціація між класами описує структурни зв'язок між рівноправними елементами: обидва класи концептуально перебувають на одному рівні (*жоден з них не вважається важливішим за інший*). В реальних системах є багато зв'язків типу «ціле-частина», де один клас описує велику сутність (ціле), що містить у собі дрібніші (частини). Цей тип зв'язків, що ґрунтується на відношеннях володіння, називається **аґреґацією** і вказує, що *об'єкт-ціле* має *об'єкти-частини*. Аґреґація є особливий вид асоціації, що зображується лінією простої асоціації, до якої доданий порожній ромбик з боку об'єкта-цілого.

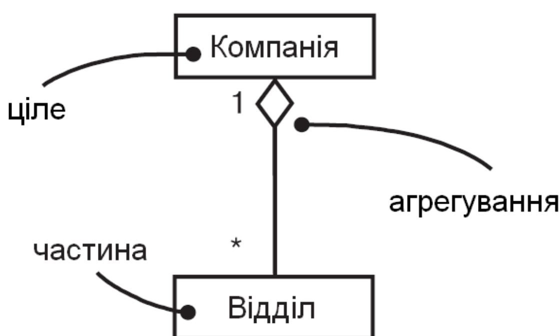


Рис. 2.2.7. Аґреґація

Прості залежності, узагальнення й асоціації з іменами, показниками множинності й ролями є засобами моделювання найзагальнішої семантики зв'язків. Для реальних систем потрібно також ВСКД інших типів зв'язків: *композитна аґреґація, навігація, дискримінанти, асоціації-*

класи, деякі різновиди залежностей і узагальнень.

Посилання (з'єднання для відправлення повідомлень). Залежності, узагальнення й асоціації є *статичними* сутностями, що візуалізуються в UML- діаграмах. При моделюванні об'єктів (*в динамічних коопераціях*) використовуються *посилання* – **екземпляри асоціацій**, що є з'єднаннями між об'єктами, по яких відправляються повідомлення. Зв'язки зображуються лініями, проведеними паралельно до країв сторінки або похилими під будь-яким кутом. З метою уникнення двозначної інтерпретації зв'язків, можливі місця перетину їх ліній позначаються маленьким півколом (рис. 2.2.8).

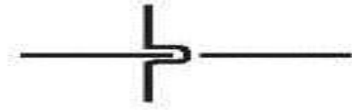


Рис. 2.2.8. Символ перетину ліній

Моделювання зв'язків залежності. Залежність можна показати як зв'язок класу, що використовує інший клас як параметр для своєї операції. Щоб його змоделювати, необхідно створити залежність, спрямовану від *класу з операцією* до класу, що використовується як *параметр операції*. Набір класів системи управління призначення студентів і викладачів на навчальні дисципліни (курси) в університеті (рис. 2.2.9) включає залежність від класу CourseSchedule (РозкладКурсів) до Course (Курс), оскільки Course використовується в операціях add (додати) і remove (вилучити) класу CourseSchedule.

Якщо показана повна сиґнатура операції (рис. 2.2.9), то залежність показувати необов'язково, оскільки використання класу вже присутнє у сиґнатурі. Однак іноді показати таку залежність необхідно, особливо якщо не приводиться сиґнатура операцій або модель показує інші зв'язки з використовуваним класом. Рис. 2.2.9 демонструє також залежність, що не пов'язана з класами в операціях, а моделює одну загальну ідіому C++. Залежність від класу Iterator (Ітератор) вказує те, що клас Iterator використовує клас CourseSchedule. Клас CourseSchedule при цьому нічого не знає про клас Iterator. Залежність позначена стереотипом «permit» («дозволити») подібно пропозиції friend (друг) у C++.

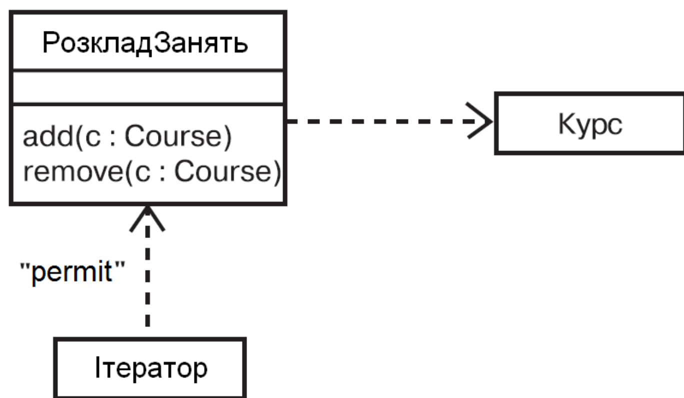


Рис. 2.2.9. Зв'язки залежності

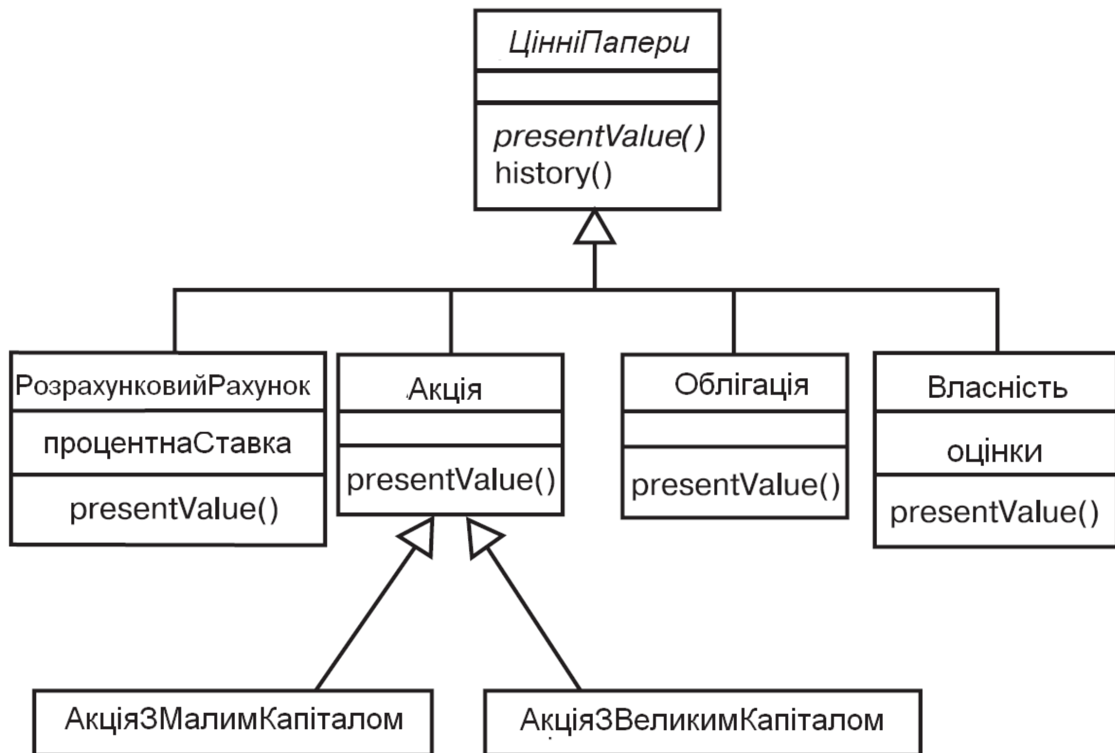


Рис. 2.2.10. Зв'язки успадкування

Моделювання одиничного успадкування при створенні словника системи полягає в знаходженні в множині класів обов'язків, атрибутів і операцій спільних для кількох класів, перенісши їх у загальніший клас. Більш спеціалізовані класи призначаються нащадками загальних класів, провівши зв'язок узагальнення від кожного спеціалізованого класу до його батька. Створювану ієрархію класів не слід перевантажувати багаторівневістю зв'язків успадкування. Рис. 2.2.10 демонструє ієрархію класів, що формують додаток керування торгівлею. Зв'язок узагальнення, проведений тут від чотирьох класів – Cashaccount (Розрахунковий Рахунок), Stock (Акція), Bond (Облігація) і Property (Власність) – до загальнішого класу Securities (Цінні Папери). Клас Securities є батьком, а Cashaccount, Stock, Bond і Property – нащадками (дочірніми класами). Кожен із цих чотирьох спеціалізованих класів є різновидом класу Securities. Останній включає дві операції: presentvalue (Наявна Ціна) і history (Історія). Оскільки Securities є батьком Cashaccount, Stock, Bond і Property, усі вони успадковують ці дві операції, так само як і будь-які інші атрибути й операції Securities, що можуть бути тут не зазначені.

Абстрактні класи. Імена класу Securities й операції presentvalue написані курсивом. Створювані ієрархії (рис. 2.2.10) часто включають **нелистові** класи, які є неповними або для яких не може існувати об'єктів. Це є абстрактні класи (*позначені курсивом*). Усі чотири **безпосередні** нащадки Securities є конкретними (не абстрактними) і кожен з них зобов'язаний представити конкретну реалізацію операції presentvalue. [2]

Множинне успадкування. Ієрархії узагальнення/спеціалізації не обов'язково обмежуються двома рівнями. Їх може бути значно більше (рис. 2.2.10). Можна створювати класи з кількома батьками. [2] Це є множинне спадкування й означає, що класу-нащадкові передаються всі атрибути, операції й асоціації його батьків. У системі успадкування не може бути циклів: *жоден клас не може бути своїм власним предком*.

Асиметричність зв'язків залежності і узагальнення. При використанні зв'язків залежності або узагальнення можна моделювати класи, які перебувають на різних рівнях важливості чи абстракції. [1] У випадку залежності між двома класами один клас залежить від іншого, але останньому нічого не відомо про перший. У випадку узагальнення між класами нащадок успадковує властивості батька, але батько не містить ніяких конкретних відомостей про своїх нащадків. Отже, зв'язки залежності й узагальнення є *асиметричними.* [1,13]

Асоціація як структурний шлях взаємодії об'єктів двох класів. У моделюванні асоціацій беруть участь класи, що перебувають на одному рівні. Два класи, залучені в асоціацію, певним чином зв'язані один з одним. Часто можна здійснювати навігацію по цьому зв'язкові в будь-якому напрямку. [13] У той час, як залежність – **зв'язок використання**, а узагальнення – **зв'язок успадкування**, асоціація визначає *структурний шлях взаємодії об'єктів двох класів*. [2]

Моделювання структурних зв'язків: ІС університету. Щоб змоделювати структурні зв'язки, потрібно встановити асоціацію між кожною парою класів, вказавши навігацію між об'єктами цих класів і проаналізувати як об'єкти одного з них, що потребують взаємодії з об'єктами іншого (*але не у вигляді параметрів операції*). Для кожної із встановлених асоціацій специфікувати множинність та імена ролей. Якщо клас на одному кінці асоціації структурно або організаційно є ціле, а клас на іншому кінці є його частиною, позначити такий зв'язок як включення або композиція.

На основі аналізу ВВ системи можна встановити усі структурні й поведінкові сценарії. Для будь-яких класів, що взаємодіють між собою, встановлюються зв'язки асоціації.

На рис. 2.2.11 показано набір класів інформаційної системи (ІС) навчального закладу. Це класи Student (Студент), Course (Курс або Дисципліна) і Professor (Викладач). Існує асоціація між Student і Course, що свідчить про те, що студенти відвідують курси. Кожен студент може відвідувати багато дисциплін і кожна дисципліна може бути прочитана для багатьох студентів. Показана асоціація між Course і Professor, яка визначає, що викладачі читають дисципліни. Для кожної дисципліни призначений як мінімум один викладач, причому кожен може читати одну або кілька дисциплін або не читати жодної. Кожна дисципліна закріплена за певним факультетом (кафедрою).

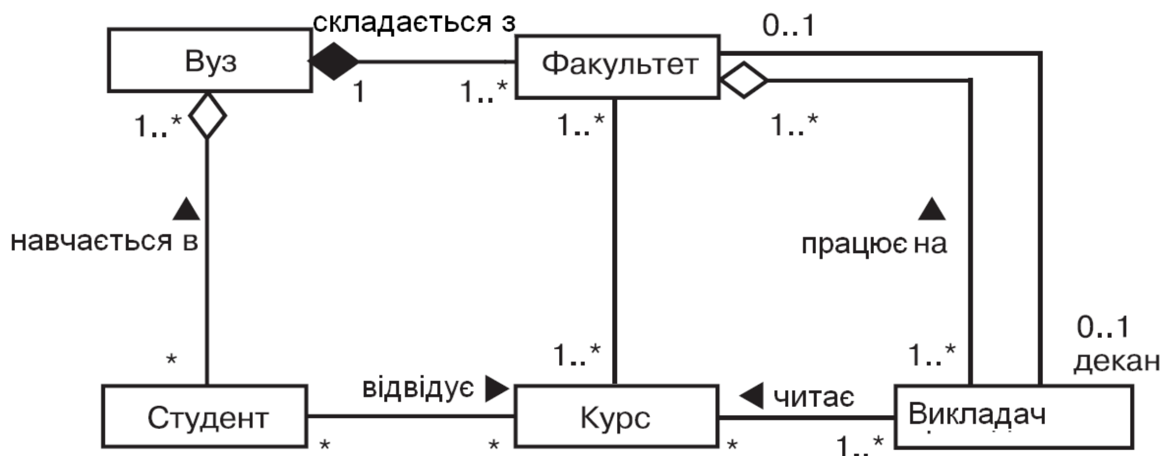


Рис.2.2.11. Діаграма класів, що відображає структурні зв'язки

Зв'язки між класом University (Навчальний заклад) і класами Student і Department (Факультет) набагато інші. Тут є приклади агрегацій. ВНЗ може набрати або не набрати

студентів; кожен студент може бути зареєстрований в одному або кількох вузах; у вузі є один або кілька факультетів, причому кожен ставиться винятково до одного навчального закладу. Можна вилучити додаткові символи аґреґації й застосовувати прості асоціації, але, специфікуючи клас University як ціле, а Student і Department – як його частини, вияснивши, що є головним. З діаграми бачимо, що навчальні заклади в деякому плані визначаються його студентами і факультетами. Студенти і факультети теж не існують самі по собі, поза університетом, до якого вони прикріплені. Вони отримують свій статус тільки у зв'язку з навчальним закладом.

Є також дві асоціації між Department і Professor. Одна з них показує, що кожен викладач працює на одному або кількох факультетах і на кожному факультеті є викладачі (*хоча б один*). Цей зв'язок моделюється як аґреґація, оскільки організаційно факультети в структурі навчального закладу перебувають вище, аніж викладачі. Друга асоціація, встановлена для кожного факультету, визначає, що на факультеті є один начальник – викладач, що займає посаду декана. З даної моделі випливає, що викладач може бути деканом не більше ніж на одному факультеті, а деякі викладачі не є деканами жодного факультету.

При моделюванні зв'язків слід використовувати залежності для **зв'язків використання**, а не структурних; узагальнення – для зв'язків типу «*є*», «*відноситься до*». Множинне успадкування в міру можливості слід замінити аґреґацією, уникати циклічних зв'язків узагальнення; забезпечувати збалансованість зв'язків узагальнення. Ланцюги успадкування не повинні бути ні занадто довгими (*не більше чотирьох рівнів*), ні занадто широкими (*знаходити можливість створення проміжних абстрактних класів*). Для структурних зв'язків між об'єктами використовувати асоціації (*крім тимчасових, зокрема у параметрах або локальних змінних*). Для їх графічного представлення використовувати стилі прямокутних і похилих ліній (*прямокутні виражають з'єднання сутностей з одним загальним батьком, похилі – економлять простір діаграм*). Уникати зайвих зв'язків (*надлишкввих асоціацій*): показувати тільки ті зв'язки, які потрібні для розуміння моделі.

2.2.2. Розширені зв'язки

Основні питання:

- *Розширені зв'язки: залежності, узагальнення, асоціації, реалізації й уточнення.*
- *Моделювання систем зв'язків.*
- *Створення систем зв'язків.*

При моделюванні сутностей та формуванні словника системи необхідно визначити усі зв'язки між ними, які можуть бути досить складними. Це вимагає низки розширених властивостей.

Моделювання семантики зв'язків різних рівнів. Найважливішими UML – блоками зв'язків є залежності, узагальнення і асоціації, що мають множину додаткових властивостей. [2] Це дає можливість моделювати *множинне успадкування*, *навігацію*, *композицію*, *уточнення* та ін. Використовуючи *реалізацію*, можна моделювати *з'єднання між інтерфейсом і класом* або *компонентою* або *між ВВ і кооперацією*. UML допускає моделювання семантики зв'язків на будь-якому рівні формальності. *При проектуванні будинку важливо визначити розташовування кімнат (ванну розмістити на першому поверсі, передбачити можливість внесення продуктів з гаража на кухню, однак не через ванну). Слід скласти детальний план будинку, враховуючи план розташування кімнат*

та інші важливі зв'язки. Однак якщо не враховувати в проекті складніших зв'язків, можуть виникнути серйозні труднощі. Розташування кімнат на кожному поверсі цілком влаштовує замовника, але кімнати на різних поверхах сусідять невдало (кімнату дочки, що вчиться грати на фортепіано, розміщено над галасливою кухнею). План доведеться переглянути. Потрібно ще враховувати комунікації. Вартість конструкції значно зросте, якщо суміжні кімнати не мають спільних стін із прокладеними трубами і водостоками.

Керування складними системами зв'язків при створенні ПЗ аналогічно вимагає використання правильних зв'язків на різних рівнях деталізації з метою недопущення надмірного спрощення чи ускладнення системи. UML включає множину розширених інструментальних засобів для опису таких зв'язків, що дозволяє їх ВСКД на будь-якому рівні деталізації для прямого і зворотного проектування (рис. 2.2.12).

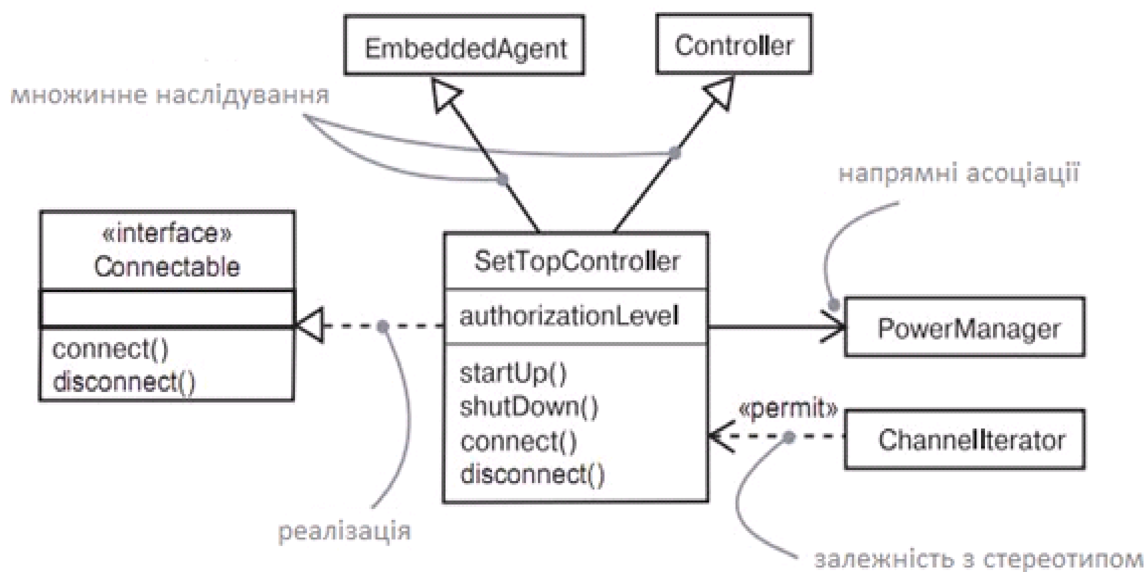


Рис. 2.2.12. UML нотації розширених зв'язків.

Зв'язок є з'єднанням сутностей, що зображуються лініями різної графічної нотації. В ООМ є чотири найважливіші типи зв'язків: залежності, узагальнення, асоціації й реалізації.

Залежність – *зв'язок використання*, який вказує, що зміна специфікацій однієї сутності (*клас* SetTopController) може вплинути на інші сутності, які використовують її (*клас* ChannelIterator) (рис. 2.2.12). Залежність зображується у вигляді пунктирної лінії зі стрілкою, спрямованої до тієї сутності, від якої вона залежить. Залежності потрібно застосовувати, коли необхідно показати, що якась сутність використовує іншу. UML забезпечує набір ґруп стереотипів залежності, щоб підкреслити суттєві особливості цього виду зв'язку між проектованими абстракціями.

Стереотипи зв'язків залежності між класами й об'єктами (на діаграмі класів):

bind – показує, що вихідний об'єкт створює екземпляр цільового, використовуючи задані реальні параметри. Використовується для уточнення подробиць шаблонних класів. Зв'язок між шаблонним контейнерним класом і

його екземпляром моделюється залежністю bind. Вона включає список реальних арґументів, що відповідають формальним аргументам шаблона;

derive – показує, що джерело може бути обчислене по меті. Стереотип derive потрібно використовувати при моделюванні зв'язку між двома атрибутами або двома асоціаціями, одна з яких конкретна, а друга – концептуальна. Клас Person (Особа) може мати конкретний атрибут Birthdate (Дата Народження), а також атрибут Age (Вік), похідний від Birthdate, тому не оголошений окремо в класі. Зв'язок між Birthdate і Age виражається залежністю derive, тобто Age визначається з Birthdate;

permit – показує, що джерело має задану видимість для мети. Така залежність може *використовуватись для забезпечення доступу одного класу до закритих властивостей іншого класу*. В C++ для цього передбачені дружні класи (friends);

instanceof – показує, що вихідний об'єкт є екземпляром цільового. Це відображається текстовою нотацією вигляду Вихідний об'єкт: Цільовий об'єкт;

instantiate – показує, що джерело створює екземпляри мети. Ці два стереотипи дозволяють явно моделювати зв'язки «клас/об'єкт». Можна використовувати instanceof при моделюванні зв'язку між класом і об'єктом на одній і тій же діаграмі або зв'язку між класом і його мета класом. Однак остання відображається в текстовому синтаксисі. Стереотип instantiate вказує, що один клас створює об'єкти іншого;

powertype – повідомляє, що цільовий об'єкт є *супертипом* вихідного. Супертип – це класифікатор, об'єкти якого є дочірніми стосовно заданого батьківського. Слід використовувати powertype, коли потрібно моделювати ті класи, які стосовно інших є класифікаторами (*при моделюванні БД*);

refine – показує, що джерело перебуває на глибшому рівні абстракції, ніж ціль. Використовується при моделюванні класів, що представляють те саме поняття на різних рівнях абстракції. У процесі аналізу можна визначити клас Customer (Покупець), який у процесі проектування буде уточнений до більш деталізованого класу Customer і завершений реалізацією;

use – специфікує, що семантика вихідного елемента залежить від семантики відкритої частини цільового. Слід застосовувати use, коли необхідно явно позначити залежність як *зв'язок використання*, на противагу решті значень залежностей, представлених у стереотипах;

import – показує, що відкритий вміст цільового пакета входить у відкритий простір імен вихідного пакета, немов вони були декларовані у вихідному;

access – показує, що відкритий вміст цільового пакета входить у закритий простір імен вихідного. Некваліфіковані імена можуть застосовуватися всередині вихідного, але не можуть реекспортуватися. Слід використовувати import і access, коли необхідно задіяти елементи, оголошені в інших пакетах. Імпорт елементів дозволяє відмовитися від застосування повних кваліфікованих імен для посилань на елементи інших пакетів у текстових виразах.

extend – показує, що цільовий ВВ розширює поведінку вихідного;

include – показує, що вихідний ВВ явно містить у собі поведінку іншого ВВ у місці, зазначеному у вихідному. Стереотипи зв'язків між ВВ extend і include, а також просте узагальнення слід використовувати, коли потрібно виконати декомпозицію ВВ на повторно використовувані частини;

send – показує, що вихідний клас (об'єкт) посилає повідомлення цільовому. Send слід використовувати, коли потрібно змоделювати операцію (*дію, пов'язану з переходом між станами*), яку передає дана подія цільовому об'єкту, який, у свою чергу, може мати

асоційований кінцевий автомат. Стереотип залежності взаємодії об'єктів send дозволяє ефективно зв'язати між собою два незалежні автомати;

trace – стереотип організації елементів системи в підсистеми і моделі показує, що цільовий елемент є історичним попередником вихідного (*тим же елементом, але на ранній стадії розроблення*). Trace використовується при моделюванні зв'язку між елементами різних моделей. У контексті архітектури системи ВВ у моделі ВВ, що представляє функціональну вимогу, може трасуватися в пакет відповідної моделі проектування з артефактами, які реалізують даний ВВ.

Узагальнення – є зв'язок успадкування загального класифікатора (суперкласу, батька) більш спеціалізованим (підкласом, дочірним класом). MultipanelWindow(Багатопанельне Вікно) (дочірний клас) успадкує всю структуру і поведінку Window (батьківський). Дочірній клас може, на додачу до цього, представити нову структуру і поведінку, або навіть перевизначити поведінку батька. У зв'язку узагальнення екземпляри нащадка можуть бути використані скрізь, де застосовні екземпляри батька – це означає, що екземпляр нащадка може бути підставлений замість екземпляра батька.

Стереотипи зв'язків узагальнення між класами. Одиничне успадкування є достатнім для багатьох систем. Для випадків, коли один клас містить у собі аспекти множини інших, такі зв'язки краще моделювати **множинним успадкуванням**. На рис. 3.2.2 набір класів складають додаток деяких фінансових послуг. Клас Asset (Активи) має три нащадки: Bankaccount (Банківський Рахунок), Realestate (Нерухомість) і Security (Цінні Папери). Два з них – Bankaccount і Security – мають своїх власних нащадків. Stock (Акція) і Bond (Облігація) – дочірні класи Security. Дочірні класи Bankaccount (Банківський Рахунок) і Realestate (Нерухомість) успадковують від множини батьків. Клас Realestate є різновид як класу Asset (Активи), так і класу InsurableItem (Об'єкт Страхування), а клас BankAccount – різновид класів Asset (Активи), InterestBearingItem (Об'єкт Нарахування Відсотків), InsurableItem (Об'єкт Страхування).

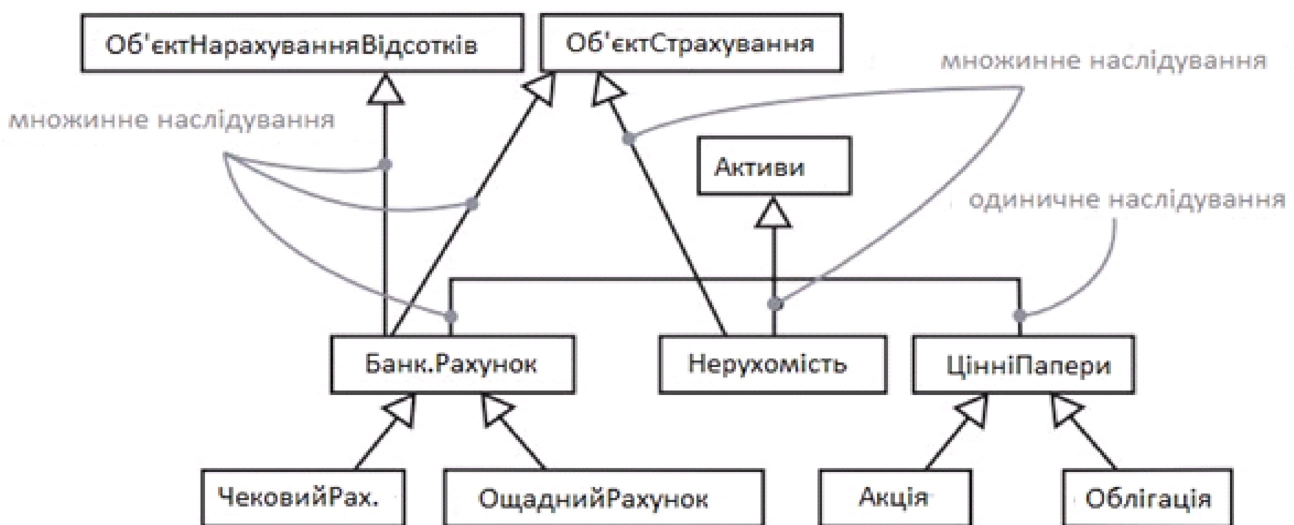


Рис. 2.2.13. Множинне успадкування

Деякі суперкласи можуть використовуватися тільки для того, щоб додати деяку поведінку чи структуру до класів, які успадковують свою основну структуру від звичайних суперкласів. Ці додаткові класи називаються **класами домішків** (mixins). Вони не можуть застосовуватися автономно, а завжди виступають в якості додаткових суперкласів у зв'язку множинним успадкуванням.

Для необхідності підкреслити тонкі суттєві особливості узагальнення слід використовувати існуючі в UML стереотипи обмеження для зв'язків узагальнення:

complete – показує, що всі дочірні класи в узагальненні специфіковані в моделі (*хоча деякі можуть бути не зазначені на діаграмі*) і ніякі додаткові дочірні класи не допускаються;

incomplete – показує, що не всі дочірні класи в узагальненні специфіковані в моделі (*навіть якщо деякі й пропущені*) і допускається створення додаткових дочірніх класів. Якщо тільки не зазначене інше, то можна припускати, що будь-яка діаграма показує тільки частковий вигляд структури успадкування, інша частина пропущена. Обмеження complete слід використовувати, коли необхідно явно показати, що ієрархія успадкування в моделі показана повністю. Обмеження incomplete дозволяє явно вказати, що не встановлюється повна ієрархія в моделі;

disjoint – означає, що об'єкт батьківського класу може належати тільки одному типу класів-нащадків. Клас Person (Особа) може бути спеціалізований шляхом створення disjoint-класів Man (Чоловік) і Woman (Жінка);

overlapping – означає, що об'єкт батьківського класу може належати кільком типам класів-нащадків і саме підкласам, що перекриваються. Клас Vehicle (Транспорт) може бути спеціалізований шляхом створення підкласів Landvehicle (Наземний Транспорт) і Watervehicle (Водний Транспорт); автомобіль-амфібія відноситься до обох.

Останні два обмеження відносяться до множинного успадкування. disjoint слід використовувати, щоб показати, що класи в наборі є *взаємно несумісними* і підклас не може успадковуватись від кількох батьків. Overlapping застосовується для моделювання множинного успадкування класу від більш ніж одного класу.

Асоціації. Асоціація є структурним зв'язком, що показує як об'єкти однієї сутності з'єднані з об'єктами іншої. Клас Library (Бібліотека) може мати асоціацію «один – до багатьох» із класом Book (Книга), повідомляючи таким чином, що кожен екземпляр Book належить одному екземпляру Library. Більше того, можна знайти бібліотеку, де втримується конкретна книга, а в рамках однієї бібліотеки можна здійснити навігацію по всіх її книгах. Асоціація зображується суцільною лінією, що з'єднує різні класи або один сам з собою. Використовується для моделювання структурних зв'язків.

Є чотири базові доповнення до асоціацій: *ім'я*, *роль на кожному кінці асоціації*, *множинність кожного кінця асоціації* й *аґрегація*. Для поглибленого моделювання властивостей асоціації передбачені розширені механізми: *навігація*, *кваліфікація* та різні типи аґрегацій.

Навігація (navigation). Для простої (*без доповнень*) асоціації між двома класами (*класи* Book і Library) можна здійснювати навігацію від об'єктів одного виду до об'єктів іншого, явно задавши напрямом з допомогою стрілки, що вказує в потрібному напрямі. Якщо не зазначено, то навігація по асоціації є двонаправленою. При моделюванні служб операційної системи (рис. 2.2.14) необхідно описати асоціацію між об'єктами User (Користувач) і Password (Пароль). Кожному об'єкту класу User

відповідає конкретний об'єкт класу Password, але за об'єктом Password немає сенсу знаходити відповідний об'єкт User.

Видимість класів і об'єктів (visibility of class end objects). За наявності асоціації між двома класами об'єкти одного класу можуть «бачити» об'єкти іншого і допускати навігацію до них, якщо вона не обмежена явно. За необхідності можна обмежити видимість одних об'єктів стосовно інших у межах асоціації. На рис. 2.2.15 асоціації встановлені між класами UserGroup (ҐрупаКористувачів) і User (Користувач), а також між User і Password (Пароль). Маючи об'єкт класу User, можна ідентифікувати відповідні йому об'єкти класу Password. Однак клас Password є *закритим стосовно класу* User і не може бути доступний ззовні (*якщо тільки користувач не відкриває доступ до пароля, можливо, за допомогою якоїсь відкритої операції*). Маючи об'єкт класу UserGroup, можна перейти до відповідних йому об'єктів класу User і навпаки, але не можна побачити об'єкти класу Password, що належать класу User, тому що вони закриті стосовно класу User.

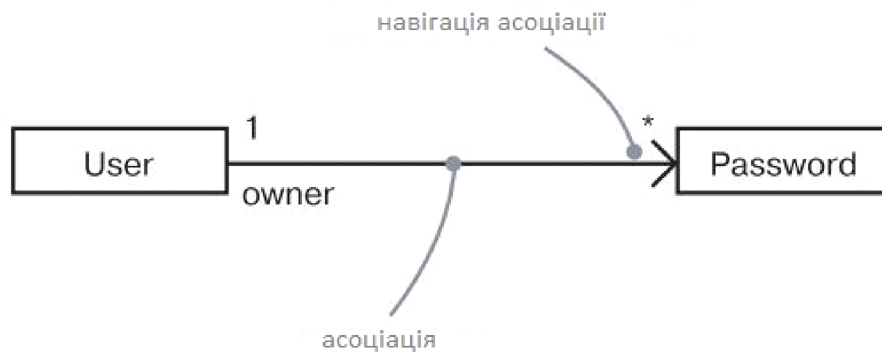


Рис. 2.2.14. Навігація

Рівні видимості кінців асоціації (ролі**).** UML передбачає три рівні видимості для кінця асоціації подібно тому, як це реалізовано для властивостей класу, додаючи символ видимості до імені ролі. Якщо не вказаний символ видимості ролі, то вона є відкритою. Закрита видимість ролі означає, що об'єкти на даному кінці асоціації не доступні ніяким об'єктам поза нею. Захищена видимість означає, що об'єкти на даному кінці асоціації не доступні ніяким об'єктам поза нею, за винятком дочірніх стосовно того, який перебуває на іншому кінці. [2] Пакетна видимість означає, що класи, оголошені в тому ж пакеті, можуть бачити даний елемент. Це не стосується кінців асоціації.

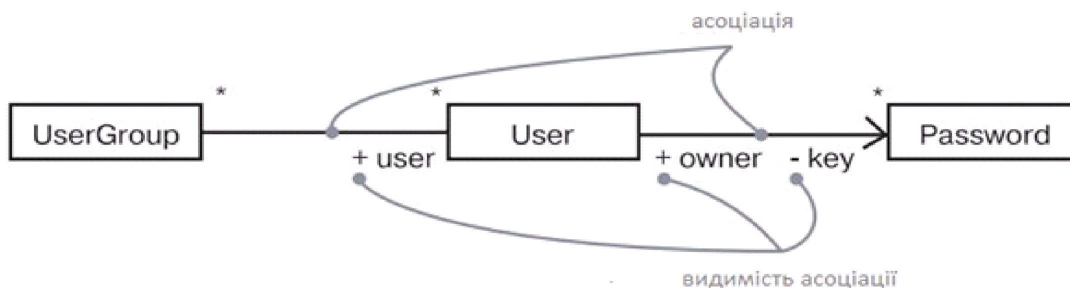


Рис. 2.2.16. Видимість класів і об'єктів

Кваліфікація (qualification). Однією з ідіом, що найчастіше використовуються моделювання є проблема пошуку (lookup) стосовно асоціацій. [2] Виникає питання, як, маючи об'єкт на одному кінці асоціації, ідентифікувати один або множину об'єктів на іншому її кінці? Для прикладу, розглянемо моделювання робочого стола в цеху, на якому сортуються браковані вироби, повернені для ремонту (рис. 2.2.17). Потрібно змоделювати асоціацію між двома класами: Workdesk (Робочий Стіл) і ReturnedItem (Повернений Виріб). Для класу Workdesk буде потрібен jobId – **ідентифікатор** певного об'єкта класу ReturnedItem (Повернений Виріб). Тут jobId виступатиме як **атрибут асоціації**, що не є властивістю класу ReturnedItem (*повернуті вироби не несуть у собі ніякої інформації про ремонт*). Отже, маючи в наявності об'єкт класу Workdesk і певне значення атрибуту асоціації jobId, можна здійснити навігацію до нуля або одного об'єкта класу ReturnedItem. В UML дану ідіому слід моделювати за допомогою **кваліфікаторів** (qualifiers), що є атрибутами асоціацій, значення яких ідентифікує підмножину об'єктів (*частіше – один об'єкт*), пов'язаних з іншим об'єктом по асоціації. Кваліфікатор зображується у вигляді маленького прямокутника, приєднаного до кінця асоціації; у цей прямокутник вміщені *атрибути кваліфікатора* (рис. 2.2.17). *Вихідний об'єкт* (клас Workdesk) разом зі значеннями атрибутів кваліфікатора породжує *цільовий об'єкт* (клас ReturnedItem), якщо множинність цільового об'єкта дорівнює одиниці, або набір цільових об'єктів, якщо множинність більша за одиницю (*множинність вказується в правому верхньому куті загальної нотації класу*).

Композиція (composition). Проста аґрегація концептуально виражає відмінності цілого від частини. Вона не змінює змісту навігації по асоціації між цілим і його частинами і нічого не каж про ЖЦ зв'язку цілого з його частинами. Композиція є особливим різновидом простої агрегації (*яка додає їй важливу семантику*): це форма аґрегації з чітко вираженими відношеннями володіння та збігом ЖЦ частин і цілого. [2,5] Частини з нефіксованою множинністю можуть бути створені після самого композиту, але їх ЖЦ завершується разом цілим. Крім того, такі частини можуть бути явно вилучені перед завершенням ЖЦ композиту.

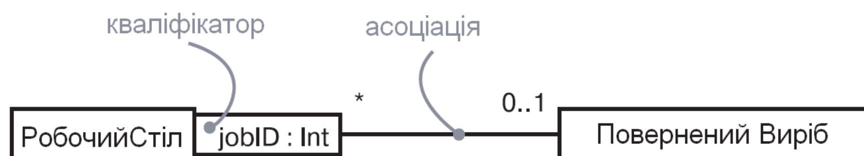


Рис. 2.2.17. Кваліфікація

В композитній аґрегації об'єкт може бути одночасно частиною тільки одного композиту. При моделюванні віконної системи об'єкт Frame (Рама) належить тільки одному об'єкту – Window (Вікно). Цей варіант відрізняється від простої аґрегації, що допускає належність частини кільком цілим. У моделі будинку Wall (Стіна) може бути частиною одного або кількох об'єктів Room (Кімната).

У композитній аґрегації **ціле розпоряджається своїми частинами:** композит повинен управляти створенням і видаленням своїх частин. При створенні об'єкта класу Frame у віконній системі його потрібно приєднати до об'єкта класу Window, що його включає. Аналогічно, при ліквідації об'єкта Window ліквідовується і його частина – об'єкт Frame.

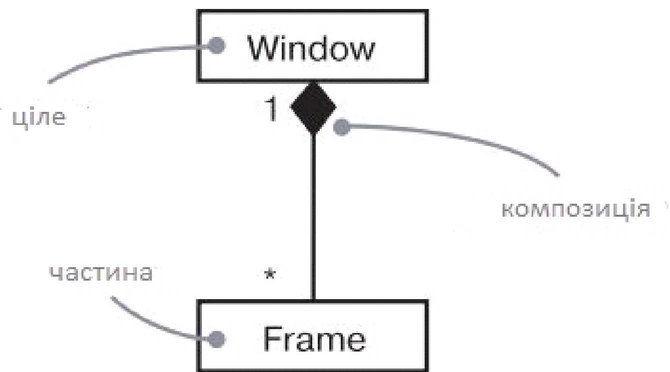


Рис. 2.2.18. Графічна нотація композиції

Композиція є особливим видом асоціації, графічна нотація якої має вигляд простої асоціації, доповненої зафарбованим («*чорним*») ромбом (рис. 2.2.18).

Асоціації-класи (association class). Асоціація, що зв'язує два класи, сама по собі може мати деякі властивості. [2,6] На рис. 2.2.18 у зв'язку «роботодавець/працівник» між класами Company (Компанія) і Person (Особа) є присутнім елемент Job (Посада) – властивість асоціації, яку описує посада працівника в компанії і з'єднує кожну пару об'єктів Company і Person. Неправильним є моделювати цю ситуацію асоціаціями Company з Job і Job з Person, оскільки при цьому не визначається конкретний екземпляр Job, що зв'язує один з одним класи Company і Person. Це можна добре описати в UML асоціацією-класом – елементом, який одночасно має властивості класу й асоціації. Його можна розглядати подвійно: як *асоціацію, яка несе в собі властивості класу*, або ж як *клас, наділений властивостями асоціації*. Зображується він символом класу, з'єднаним пунктирною лінією з лінією асоціації (рис. 2.2.19).

Обмеження асоціації. Для моделювання окремих особливостей зв'язків асоціації UML визначає низку додаткових обмежень:

1) показати, що об'єкти на одному кінці асоціації (*з множинністю більше одиниці*) впорядковані або не впорядковані:

ordered – вказує, що набір об'єктів на кінці асоціації повинен бути впорядкований. В асоціації User/Password об'єкти Password, асоційовані з User, можуть зберігатися у зворотному порядку (тобто першим іде останній використаний пароль) і, відповідно, асоціація може бути позначена як ordered. Якщо це ключове слово відсутнє, то об'єкти не впорядковані;

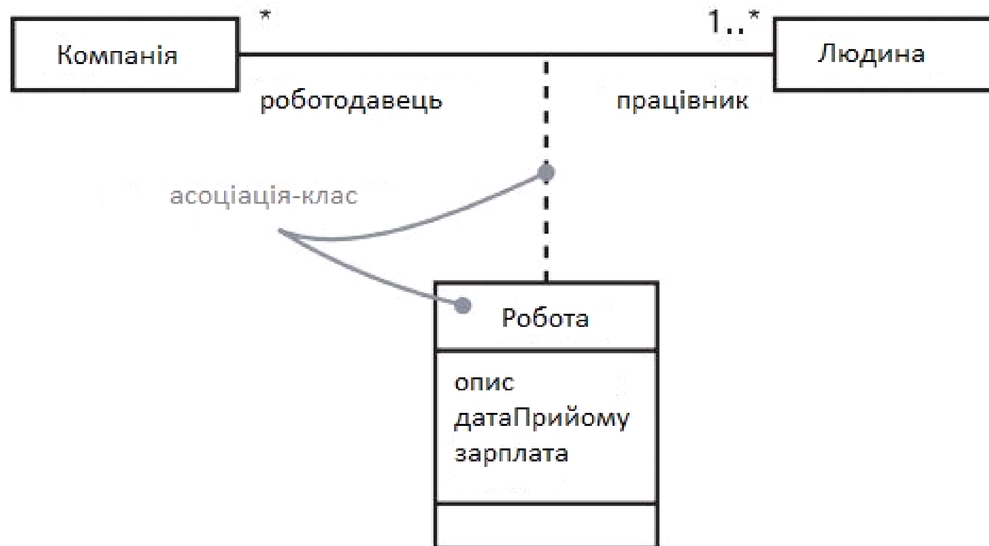


Рис. 2.2.19. Асоціації-класи

2) підкреслити, що об'єкти на одному кінці асоціації повинні бути унікальними (*впорядкованими, не містити дублікатів*) або подані як множини із повторюваними елементами:

set – об'єкти унікальні, без дублікатів;

bag – об'єкти не унікальні (*повторювані*), допускаються дублікати;

ordered set – об'єкти унікальні й упорядковані;

list або sequence – об'єкти впорядковані, можуть бути дублікати;

3) вказати обмеження змінюваності екземплярів асоціації:

readonly – посилання, одного разу встановлене від об'єкта на протилежному кінці асоціації, не може бути модифіковане або вилучене. За замовчуванням при відсутності цього обмеження допускається змінюваність асоціації.

Реалізація – семантичний зв'язок між класифікаторами, один з яких специфікує якийсь контракт, з одного боку, а іншого зобов'язується його виконувати. Зображується *пунктирною лінією з незафарбованою стрілкою-трикутником*, що вказує на класифікатор, який специфікує контракт. Реалізація суттєво відрізняється від зв'язків залежності, узагальнення й асоціації, а тому трактується як окремий тип зв'язку. [1,2] Семантично це є щось на зразок об'єднання між залежністю й узагальненням, а її нотація є комбінацією нотацій залежності й узагальнення. Реалізація використовується у двох контекстах – *інтерфейсів* та *кооперацій*.

В основному реалізація застосовується для описування зв'язку між *інтерфейсом* і *класом* (*або компонентом, який надає операцію чи сервіс для нього*).

Інтерфейс – набір операцій, що використовуються для специфікації сервісу класу або компоненти. [1] Інтерфейс описує *контракт*, який клас або компонента зобов'язані виконувати. Інтерфейс може бути реалізовано багатьма класами або компонентами, і навпаки, клас або компонента можуть реалізовувати багато інтерфейсів. Перевагою інтерфейсів є те, що вони дозволяють відокремлювати **специфікацію контракту** (*сам інтерфейс*) від його **реалізації** (класом чи компонентою).

Інтерфейси зв'язують **логічну** і **фізичну частини архітектури** системи. Клас AccountBusinessRules у системі введення замовлень у вигляді проектування системи може реалізовувати інтерфейс IRuleAgent. Той же самий інтерфейс може бути реалізований компонентою (acctrule.dll) у вигляді реалізації системи (рис. 2.2.20).

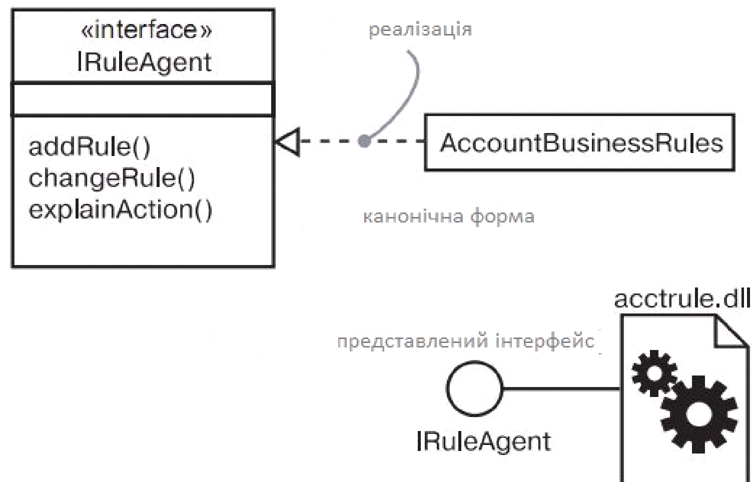


Рис. 2.2.20. Реалізація інтерфейсу (канонічна і скорочена форми)

Реалізацію можна подати двома способами: у **канонічній формі** (*застосовуючи стереотип* interface *і нотацією у вигляді пунктирної лінії з незафарбованою трикутною стрілкою*) або в **скороченій формі** (lollipop notation *для наданого інтерфейсу*). Реалізацію застосовують також для зображення **зв'язку між ВВ** і **кооперацією**, яка його реалізує, графічною нотацією – пунктирною лінією зі стрілкою з незафарбованим трикутником (рис. 2.2.21).

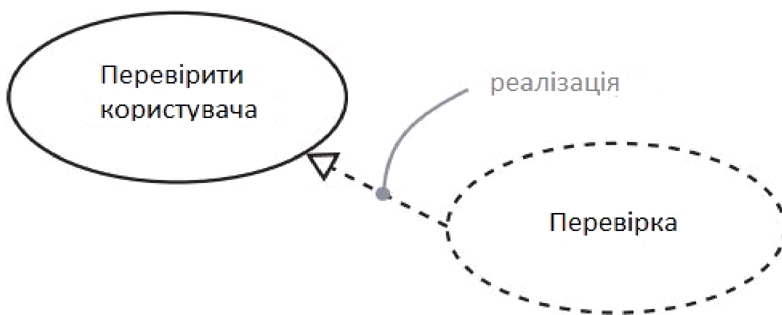


Рис. 2.2.21. Реалізація варіанта використання

Практичні рекомендації для моделювання систем зв'язків. Моделювання словника складної системи включає сотні й тисячі класів, інтерфейсів, компонентів, вузлів і ВВ. Важко встановити чіткі границі між усіма цими абстракціями, але ще складнішим є описати павутину зв'язків між ними. Це вимагає збалансованого розподілу обов'язків у системі в цілому. Для моделювання таких систем зв'язків, слід користуватися такими рекомендаціями:

– здійснювати моделювання окремих зв'язків системно, з урахуванням впливу інших. Застосовувати ВВ і сценарії для дослідження зв'язків серед великої множини абстракцій;

– починати моделювання з реально існуючих структурних зв'язків (*це дає прояснений статичний вигляд системи)*;

–ідентифікувати можливості зв'язків узагальнення/спеціалізації. Обережно застосовувати множинне успадкування;

–після завершення цих дій ідентифікувати залежності (як *більш тонку форму семантичних зв'язків*);

–описуючи кожен вид зв'язків, починати з базової форми, додаючи в міру необхідності розширені засоби;

–не потрібно демонструвати всі існуючі зв'язки в наборі абстракцій на одній діаграмі або на одному вигляді. Набори зв'язків, що викликають особливий інтерес, слід виносити в окремі діаграми, показуючи їх у різних виглядах.

Моделювання складних систем зв'язків здійснюється покроково, в міру уточнення системної архітектури. При зображенні зв'язку в UML слід показувати тільки ті властивості зв'язку, які важливі для розуміння абстракції в її контексті; вибирати версію зі стереотипом, що щонайкраще виражає призначення зв'язку візуально.

2.2.3. Інтерфейси, типи та ролі

Основні питання:

- *Інтерфейси, типи, ролі й реалізація*
- *Моделювання з'єднань у системі*
- *Моделювання статичних і динамічних типів*
- *Забезпечення зрозумілості й доступності інтерфейсів*

Інтерфейси окреслюють границю між *тим, що робить* абстракція, і реалізацією *того, як вона це робить*.

Інтерфейс (interface) – це набір операцій, що використовується для описування сервісу класу або компоненти. [1] Інтерфейси використовуються для ВСКД з'єднань всередині системи. Добре структурований інтерфейс визначає чіткий поділ зовнішнього і внутрішнього виглядів абстракції, забезпечуючи зрозумілість і доступність без необхідності вникати в деталі його реалізації. *Безглуздо проектувати будинок так, щоб при потребі перефарбувати стіни довелось би руйнувати фундамент, чи при заміні світильника - міняти всю електропроводку. Сторіччя будівельного досвіду дозволили виробити множину прагматичних конструктивних прийомів, що дозволяють уникнути подібних проблем реконструкції.*

У термінах ПЗ проектування має супроводжуватися чітким поділом складових компонентів. *У добре структурованому будинку зовнішній вигляд фасаду можна змінити, не переймаючись іншою частиною, зміна меблювання не повинна глобально впливати на інтер'єр… Цьому всьому сприяє множина стандартних інтерфейсів будівництва, що дозволяють використовувати* загальні готові компоненти, *застосування яких суттєво знижує витрати на будівництво й експлуатацію. Розроблено стандарти дверей, електричних розеток тощо.*

З'єднання в системі. У ПЗ важливо будувати системи з чітким поділом аспектів, щоб у міру їх еволюції зміни в одній частині системи не торкалися і не ушкоджували інші її частини. Один із головних способів досягнення цієї мети – визначення чітких з'єднань (*швів*) між частинами, які можуть змінюватися незалежно одна від одної.

В UML інтерфейси використовуються для моделювання з'єднань у системі. Оголошуючи інтерфейс, можна визначити необхідну поведінку абстракції незалежно від її реалізації. Застосовуючи правильні інтерфейси, клієнти можуть будувати навколо них все, що їм потрібно, як і розробники мають право створювати або купувати будь-які реалізації даного інтерфейсу, поки вони задовольнятимуть *обов'язки* і *контракт*, визначений інтерфейсом. *Замість того, щоб заново будувати, доцільніше вибирати* для їхньої реалізації стандартні компоненти, бібліотеки і каркаси. Якщо знайдуться вдаліші

реалізації, то ними завжди можна буде замінити старі, запобігаючи непорозумінь із користувачами.

ООП-мови програмування (C++, Java, CORBA IDE) підтримують концепцію інтерфейсу. Інтерфейси – важливий засіб не тільки для поділу специфікації й реалізації класу або компоненти, але й для визначення зовнішнього вигляду пакета чи підсистеми.

UML нотація інтерфейсу дозволяє роздільно візуалізувати опис *абстракції* та її *реалізацію* (рис. 2.2.22). Механізмом моделювання статичного й динамічного узгодження абстракції з інтерфейсом у певному контексті є типи і ролі.

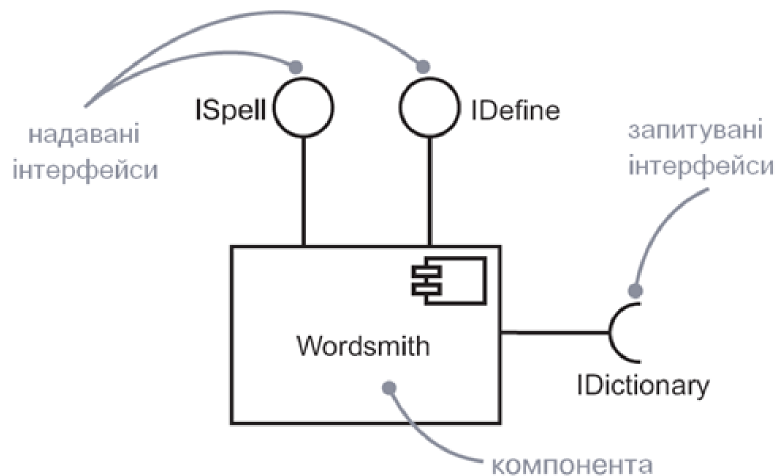


Рис. 2.2.22. Нотації наданих та запитуваних інтерфейсів

Тип (type) – *стереотип класу*, що використовується для специфікації множини об'єктів разом із дозволеними для них операціями (*але не методами*). [1]

Роль (role) – *поведінка сутності* в певному контексті. [1,2]

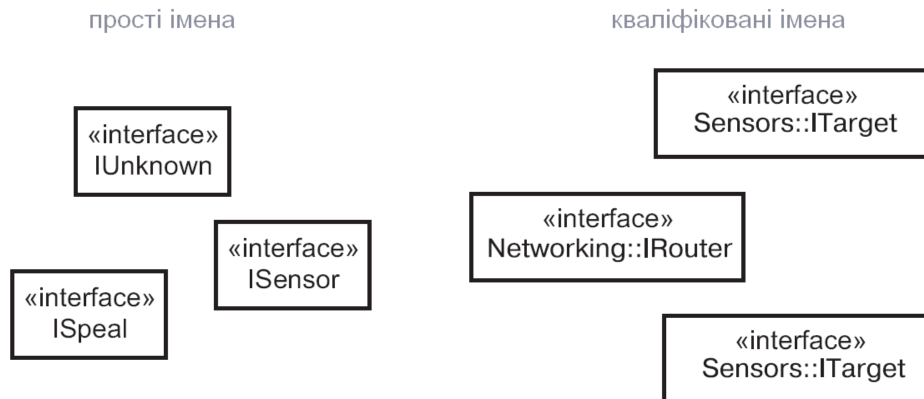


Рис. 2.2.23. Прості й кваліфіковані імена

Інтерфейс як клас зі стереотипом. Графічно інтерфейс може бути представлений як *клас зі стереотипом*, що дозволяє розкрити його операції та інші властивості. Для відображення зв'язку між класом та інтерфейсом використовується спеціальна нотація, розрізняючи **надаваний інтерфейс** (*той, що описує сервіси, які забезпечуютьс класом*) – у вигляді *маленького кола*, приєднаного до прямокутника (класу) та **необхідний інтерфейс** (*той, який один клас вимагає від іншого*) – у вигляді *маленького півкола*, з'єднаного з прямокутником. Інтерфейс має ім'я, відмінне від імен інших інтерфейсів: *просте* ім'я (*текстовий рядок*) або *кваліфіковане* (*з префіксом імені пакета, у якому він знаходиться*) (рис. 2.2.23).

Операції. Інтерфейс – це іменований набір операцій, що застосовується для описування сервісу класу або компонента. На відміну від класів або типів, *інтерфейси не містять описів реалізації операцій*. Інтерфейс може містити будь-яку кількість

операцій, доповнених властивостями видимості, паралельності, стереотипами, присвоєними значеннями й обмеженнями.

Оголошення операцій інтерфейсу. Інтерфейс зображується *як клас зі стереотипом*, перераховуючи операції у відповідному розділі. Операції можуть оголошуватись тільки іменем або повною сиґнатурою й іншими властивостями (рис. 2.2.24).

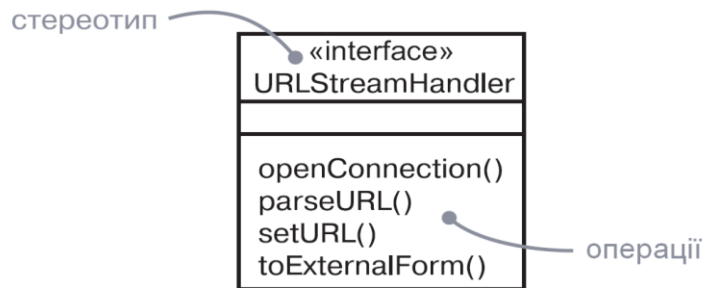


Рис. 2.2.24. Оголошення операцій інтерфейсу

Подібно до класу, інтерфейс може брати участь у зв'язках узагальнення, асоціації й залежності та зв'язку реалізації (*між двома класифікаторами, один з яких описує контракт, а інший зобов'язується його виконувати*).

Інтерфейс специфікує контракт для класу або компоненти, не нав'язуючи його реалізації. Клас або компонента може реалізувати багато інтерфейсів, *зобов'язуючись виконувати всі контракти* точно і в повній мірі – надавати набір методів, які правильно реалізують операції, визначені в усіх цих інтерфейсах.

Надаваний інтерфейс – набір пропонованих сервісів класу. В такий спосіб клас або компонента може залежати від кількох інтерфейсів. При цьому він очікує, що контракти будуть надані деяким набором компонентів, що реалізують інтерфейси.

Необхідний інтерфейс – набір сервісів, які даний клас вимагає від інших класів. Тому інтерфейс є *сполучним елементом*, що зв'язує компоненти системи. Інтерфейс специфікує контракт, дві сторони якого – *клієнт* і *постачальник* – можуть змінюватися незалежно доти, поки кожен з них дотримується своїх договірних обов'язків.

Можна двома способами показати, що елемент реалізує інтерфейс (рис. 2.2.25). По-перше, існує проста форма: інтерфейс і його зв'язок реалізації зображуються у вигляді лінії, що з'єднує прямокутник (*клас*) і маленьке **коло** (*для використання надаваного інтерфейсу*) або маленьке **півколо** (*для необхідного інтерфейсу*). Ця форма зручна і краща, коли необхідно показати з'єднання в системі. Обмеження даної нотації унеможливлює візуалізувати операції чи сигнали, надані інтерфейсом. Можна використовувати розширену форму інтерфейсу – у вигляді *класу зі стереотипом*, що дозволяє візуалізувати операції та інші властивості й зобразити зв'язок реалізації (*для надаваного інтерфейсу*) або залежності (*для необхідного інтерфейсу*) від класифікатора або компоненти до інтерфейсу.

Розуміння інтерфейсів. Перше, що видно при роботі з інтерфейсом, – це набір операцій, які специфікують сервіс класу або компоненти. Можна виявити повні сигнатури цих операцій поряд з їхніми особливими властивостями, такими, як видимість, контекст і семантика паралелізму. Ці властивості важливі, але для складних інтерфейсів їх недостатньо, щоб прояснити семантику надаваного ними

сервісу і дати зрозуміти, як правильно використовувати операції. За відсутності будь-яких інших відомостей потрібно вникнути в деяку абстракцію, яка реалізує інтерфейс, щоб зрозуміти, що саме робить кожна операція і як усі вони повинні працювати разом.

В UML моделі можна вказати значно більше інформації, щоб зробити інтерфейс доступним для розуміння. По-перше, можна супроводити кожну операцію перед- і постумовами, а клас або компоненту в цілому – інваріантами. В результаті клієнт, що бажає використовувати інтерфейс, зможе зрозуміти, що він робить і як його застосовувати, не вникаючи в деталі реалізації. *По-друге*, можна супроводити інтерфейс кінцевим автоматом для описування можливої часткової впорядкованості його операцій. *По-третє*, супроводження інтерфейсу коопераціями, визначить його очікувану поведінку за допомогою ряду діаграм взаємодії.

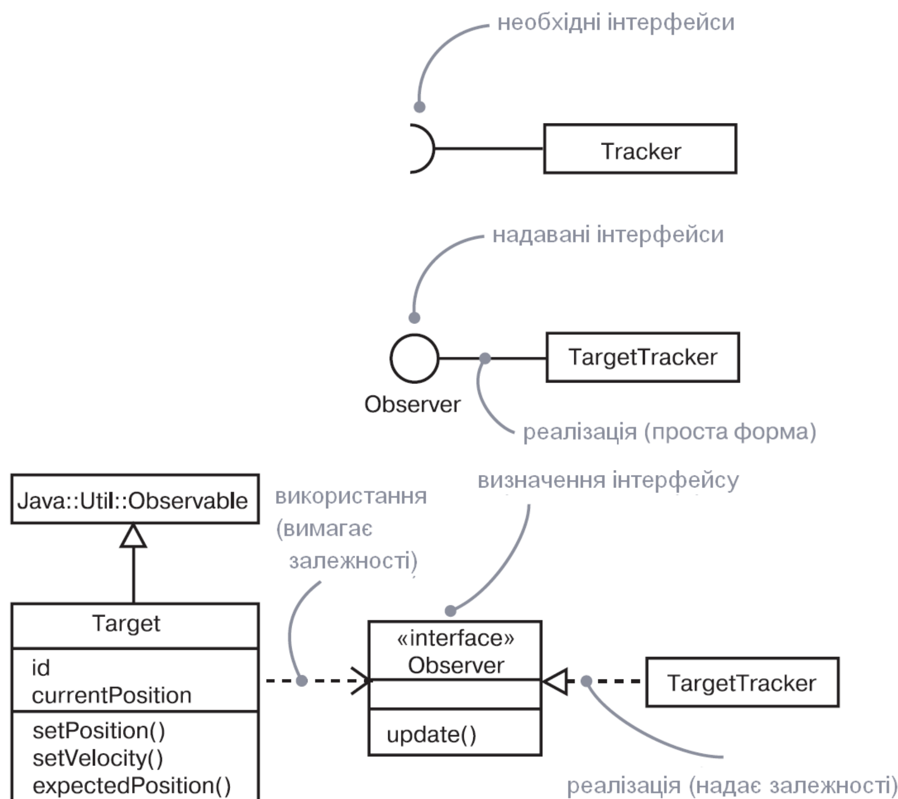


Рис. 2.2.25. Реалізація інтерфейсу

Практичні підходи моделювання інтерфейсів. Загальне призначення інтерфейсів – моделювати з'єднання в системах, що складаються із програмних компонентів (як Eclipse, .NET або Java Beans). Деякі компоненти доводиться запозичати з інших систем, інші – купувати, або створювати самостійно «з нуля». У кожному випадку доведеться писати сполучний код, призначений для їхнього з'єднання. А це вимагає розуміння інтерфейсів, надаваних і необхідних кожним компонентам.

Ідентифікація з'єднань у системі містить ідентифікацію чітких ліній поділу архітектури. З кожної від цих ліній будуть компоненти, які можуть змінюватися незалежно, не проявляючи впливу на протилежну компоненту (*поки компоненти на обох сторонах дотримуються вимог контракту, специфікованого інтерфейсом*).

При використанні певної компоненти, ймовірно, знайдеться в ній низка операцій з якимось мінімумом документації, де описується суть кожної з них. Це зручно, але

недостатньо. Набагато важливішим є розуміння порядку, в якому слід викликати операції, і механізм, який вони містять. Маючи малодокументовану компоненту, найкраще (*можливо, шляхом проб і помилок*) побудувати концептуальну модель роботи її інтерфейсу. Потім, моделюючи з'єднання в системі за допомогою інтерфейсів на UML, щоб згодом без проблем у ньому розбиратися. Точно так само, коли створюється власна компонента, то потрібно забезпечити розуміння контексту, у якому вона повинна застосовуватися, – специфікувати інтерфейси, на які вона опирається при виконанні своєї роботи, а також інтерфейси, які компонента надає контексту (*навколишньому середовищу*). Для моделювання з'єднань у системі слід виконати такі дії: провести межі між тими класами й компонентами системи, що тісніше взаємодіють один з одним, ніж інші набори класів і компонентів; перевірити правильність об'єднання елементів у ґрупи, розглянувши наслідки можливих змін. Класи або компоненти, які змінюються спільно, об'єднати в кооперації; розглянути операції й сиґнали, які перетинають границі, проведені на етапі 1, від екземплярів з одного набору класів або компонентів до екземплярів з іншого набору; об'єднати логічно зв'язані набори операцій і сигналів в інтерфейси; для кожної кооперації в системі визначити інтерфейси, які їй потрібні (*імпортовані*), і ті, які вона надає (*експортує*) іншим коопераціям. Імпорт інтерфейсів моделювати зв'язками залежності, а експорт – зв'язками реалізації. Для кожного такого інтерфейсу документувати його динаміку, використовуючи перед- і постумови кожної операції, а також ВВ й автомати для інтерфейсу в цілому.

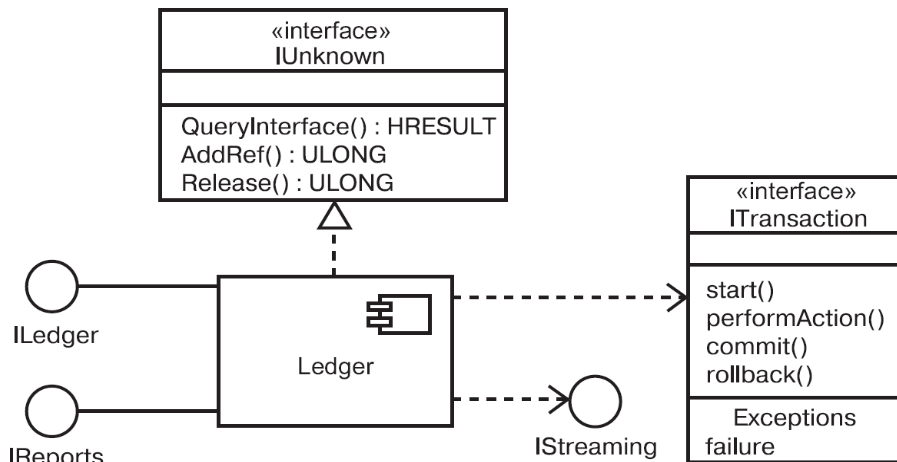


Рис. 2.2.26. Моделювання з'єднань у системі

На рис. 2.2.26 зображенно з'єднання, що оточують компоненту Ledger (Гросбух) системи керування фінансами. Ця компонента надає (реалізує) три інтерфейси: Iunknown (Невідомий), Ileclger (Гросбух) і Ireports (Звіти). На даній діаграмі Iunknown показано у розширеній формі інші два інтерфейси – у спрощеній, як «льодяники» (lollipops). Усі три інтерфейси є експортовані для використання іншими компонентами. Також на діаграмі показано, що Ledger вимагає (використовує) два інтерфейси: IStreaming (I-Потік) і Itransaction (I-Транзакція), причому другий показано у розширеній формі. Ці два інтерфейси необхідні компоненту Ledger для правильної роботи. Таким чином, у працюючій системі потрібно підставити компоненти, які реалізують обидва ці інтерфейси. Ідентифікуючи такі інтерфейси, як Itransaction, можна отримати ефективно розділені компоненти, що лежать з різних боків інтерфейсу, що дозволяє «наймати» будь-яку компоненту, що відповідає

інтерфейсу. Інтерфейс Itransaction може робити певні припущення про порядок виклику його операцій (*це більше, ніж набір операцій*).

Моделювання статичних і динамічних типів. Більшість ООП мов є статично типізовані. Це означає, що з об’єктом у момент його створення пов’язується певний тип (*навіть незважаючи на те, що об’єкт, імовірно, згодом відіграватиме різні ролі*). Клієнти, що використовують об’єкт, взаємодіють із ним через різні набори інтерфейсів, які представлють множини операцій, що їх цікавлять (які, можливо, перекриваються). При моделюванні бізнес-об’єктів, які змінюють свій тип у потоці робіт, іноді доцільно підкреслити динамічну природу їх типу, задавши її явно. За таких обставин протягом свого ЖЦ об’єкт може набувати і втрачати типи. Змоделювати ЖЦ об’єкта можна за допомогою машини станів (*кінцевого автомата*). Щоб змоделювати **динамічний тип**, необхідно: специфікувати можливі типи даного об’єкта, представляючи кожен з них у вигляді класу (*якщо абстракція вимагає структури й поведінки*) або інтерфейсу (*якщо абстракція вимагає тільки поведінки*); змоделювати всі ролі, які може відіграти клас об’єкта в будь-який момент часу. Можна позначити їхнім стереотипом dynamic (*не передбачений в UML, створюється самостійно*). На діаграмі взаємодії правильно зобразити кожен екземпляр динамічно типізованого класу. Відобразити тип екземпляра як стан, – у фігурних дужках прямо під іменем об’єкта.

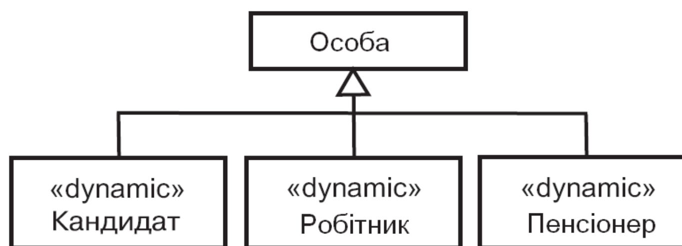


Рис. 2.2.27. Моделювання динамічних типів

На рис. 2.2.27 показано, які ролі виконують екземпляри класу Person (Особа) у системі керування персоналом. На діаграмі бачимо, що екземпляри класу Person можуть бути одного з таких типів: Candidate (Кандидат), Employee (Працівник) або Retiree (Пенсіонер). Інтерфейс повинен представляти з’єднання в системі, що відокремлює специфікацію від реалізації. Добре структурований інтерфейс є простий, але повний – представляє всі необхідні операції для описувння окремого сервісу, та зрозумілий – дає достатньо інформації як для використання, так і для реалізації, доступний для користувача за ключовими властивостями, без перевантаження деталями великої кількості операцій.

Щоб показати наявність з’єднання в системі (*для класів і компонентів*), використовують нотацію «*льодяника*» чи сокета (socket) і розширену форму для відображення деталей самого сервісу (*для описування з’єднань між пакетами й підсистемами*).

2.3. Діаґрами для моделювання статичних характеристик системи

2.3.1. Основні моделі статичних представлень системи

Основні питання:

- *Діаграми, зображення й моделі*
- *Моделювання різноманітних представлень системи*

- *Моделювання різних рівнів абстракції*
- *Моделювання складних представлень*
- *Організація діаґрам та інших артефактів*

Моделі ПЗ будуються з базових UML - блоків: класів, інтерфейсів, кооперацій, компонентів, вузлів, залежностей, узагальнень і асоціацій у вигляді діаґрам.

Діаґрама – графічне зображення набору елементів, що відображаються як зв'язний граф вершин (*сутностей*) і дуг (*зв'язків*). Діаґрами використовуються для візуалізації системи з різних точок зору. UML визначає множину діаґрам, щоб сфокусувати увагу на різних аспектах системи, розглянутих відокремлено, і полегшити розуміння розроблюваної системи. Вибір правильної множини діаґрам визначає якість, допомагає ухваленню рішень.

Працюючи над дизайном будинку, архітектор спочатку складає список побажань (у будинку повинно бути три ванних кімнати, замовник прагне укластися в таку-то суму…), робить низку простих креслень, що визначають ключові характеристики будинку та пропонує деякі загальні ідеї щодо стилю. Робота архітектора полягає у втіленні в дизайні усіх подібних і нечітких, мінливих та суперечливих побажань. Імовірно, почне він із плану першого поверху. Цей артефакт ляже в основу загального вигляду будинку, після чого уточнюватимуться деталі й документуватимуться ухвалені рішення. При кожному перегляді замовник захоче внести деякі зміни – перепроектувати кімнати, по-іншому розмістивши стіни, двері й вікна. На початковій стадії креслення міняються часто, але в міру розроблення дизайну вони все більше відповідають вимогам замовника щодо форм і функцій тих чи інших об'єктів,часу виконання і витратам. Нарешті, креслення стабілізуються настільки, що можна їх затвердити і почати будівництво. Але навіть після цього не виключено, що замовник змінить ряд діаґрам і побажає чогось нового. Крім плану першого поверху, потрібні будуть інші зображення будинку, зокрема його вигляд з різних сторін. Архітекторові також знадобиться скласти плани електромережі, систем опалення, водопроводу й каналізації. Якщо дизайн припускає нестандартні рішення (використання аркових прольотів, розміщення домашнього кінотеатру в безпосередній близькості від каміна), доведеться робити додаткові креслення цих «капризів», за гроші, звичайно.

Практика створення діаґрам для зображення систем із різних точок зору широко розповсюджена: без неї не обійтися в будь-якій інженерії, пов'язаній зі складними системами: будинки, літаки, мости й автомагістралі, ПЗ.

П'ять взаємодоповнюючих виглядів, особливо важливих для ВСКД програмної архітектури: *вигляд ВВ*, *зображення проектування*, *зображення взаємодії*, *зображення реалізації* та *зображення розгортання*. Кожен із цих виглядів (представлень) включає структурне моделювання (*моделювання статичних сутностей*), а також поведінкове моделювання (*моделювання динамічних сутностей*). Усі ці вигляди у своїй сукупності охоплюють найважливіші рішення, що стосуються системи; кожен окремо дозволяє сфокусувати увагу на одному її аспекті для чіткого оцінювання прийнятих проектних рішень.

Діаґрами UML можна використовувати двома основними способами: для специфікації моделей, на основі яких конструюється система (пряме проектування), і для реконструкції моделей на основі частин реалізованих систем (зворотне проектування).

Основні базові поняття моделювання

Система – набір підсистем, організованих для досягнення певної мети й описаних за допомогою набору моделей (*з різних точок зору*). [12]

Підсистема – ґрупа елементів, частина яких становить специфікацію поведінки, що здійснюється іншими її складовими. [1,12]

Модель – семантично завершена абстракція системи, створена за принципом повного й самодостатнього спрощення реальності, що ставить за мету краще розуміння системи. [2]

Зображення (вигляд) – проекція організації та структури системної моделі у контексті архітектури, сфокусована на одному з її аспектів. [1,12]

Діаґрама – графічне зображення набору елементів, що найчастіше зображуються у вигляді зв'язного графа вершин (*сутностей*) і дуг (*зв'язків*). [2]

Система є розробкою, представленою з різних точок зору різними моделями з використанням діаґрам. Діаґрама – графічна проекція елементів системи. Складна система (*проект системи керування людськими ресурсами*) може включати сотні класів. Неможливо візуалізувати структуру або поведінку системи на одній величезній діаґрамі, що містить всі ці класи та їх зв'язки. Потрібно створити кілька діаґрам, кожна з яких сфокусована на одному представленні. Діаґрама класів, що включає такі елементи, як Person (Людина), Department (Відділ) і Office (Офіс), відображає схему БД. Деякі з цих класів можуть бути показані API – програмним інтерфейсом для використання клієнтськими додатками. Ті ж класи можуть частково з'являться на діаґрамі взаємодій, що специфікує семантику транзакції, яка перепризначує співробітника (Person) у новий відділ (Department).

Одна і та ж сутність у системі (клас Person) – може неодноразово зустрічатися на одній або різних діаґрамах. При цьому кожна діаґрама забезпечує свій спосіб зображення елементів, з яких складається система.

При моделюванні статичних характеристик систем використовують діаґрами:

1) класів
2) компонентів;
3) складової структури;
4) об'єктів;
5) розміщення;
6) артефактів.

Для моделювання динамічних частин систем використовують діаґрами:

1) варіантів використання;
2) послідовності;
3) комунікації;
4) станів;
5) діяльності.

Кожній діаґрамі присвоюється ім'я, унікальне у своєму контексті, щоб можна було до неї звернутися і відрізнити від інших. При проектуванні реальних систем діаґрами об'єднують у пакети. Можна подати на одній діаґрамі будь-яку комбінацію елементів UML (класи і об'єкти, класи і компоненти тощо). Типи діаґрам UML визначаються елементами, які найчастіше на них зображуються.

Структурні діаґрами призначені для ВСКД статичних аспектів ПЗ. Подібно до того, як статичні аспекти будинку містять у собі наявність і місце розташування стін, дверей, вікон, труб, проводів, вентиляції, так і статичні аспекти програмної системи містять у собі *наявність* і *розміщення* **класів, інтерфейсів, кооперацій, компонентів** і **вузлів**.

Структурні діаґрами UML, що моделюють основні групи сутностей системи:

1. Діаґрама класів — класи, інтерфейси і кооперації
2. Діаґрама компонентів — компоненти
3. Діаґрама складової структури — внутрішня структура

4. Діаґрама об'єктів	об'єкти
5. Діаґрама розміщення	вузли
6. Діаґрама артефактів	артефакти

Діаґрама класів зображає набір класів, інтерфейсів і кооперацій, а також їх зв'язки при моделюванні об'єктно-орієнтованих систем, ілюструє статичні зображення проектування системи. Діаґрама класів, яка включає активні класи, визначає *статичний погляд на процеси* системи.

Діаґрама компонентів відображає внутрішні частини, коннектори і порти, що реалізують компоненту. При створенні екземпляра компоненти створюються екземпляри його внутрішніх частин.

Діаґрама складової структури описує *внутрішню структуру класу* або *кооперації*. Різниця між діаґрамою компонентів і діаґрамою складових структур невелика (*вони трактуються як діаґрами компонентів*).

Діаґрама об'єктів показує набір об'єктів і їх зв'язки. Використовується для описування структур даних, статичних знімків екземплярів сутностей, виведених на діаґрамі класів. Служить для статичного зображення дизайну або процесів системи, як і діаґрама класів, але, на відміну від останньої, – з погляду реальних або прототипних ситуацій.

Діаґрама артефактів показує набір артефактів, їх зв'язків з іншими артефактами, а також класами, які вони реалізують. Використовується для зображення фізичної реалізації елементів системи. Діаґрами артефактів є різновид діаґрам розміщення, але виділяються для зручності опису.

Діаґрама розміщення показує набір вузлів та їх зв'язків. Описує статичне зображення архітектури. Пов'язана з діаґрамою компонентів (*на кожному вузлі розміщується один або кілька компонентів*).

Практичні підходи моделювання діаґрам. Вибираючи правильний набір представлень, запускається процес, що допомагає ставити правильні запитання про систему і виявляти ризики, яких вона буде зазнавати. Якщо неграмотно підійти до вибору цих виглядів, зосередитися переважно на одному з них, то деякі важливі обставини можуть зникнути з поля зору і в подальшій роботі над проектом можна зіштовхнутися з серйозними проблемами. Щоб змоделювати систему з різних точок зору, необхідно вирішити, які вигляди потрібні для найкращого описування архітектури системи і зниження технічних ризиків проекту, які потрібні артефакти (UML діаґрами), щоб охопити найістотніші деталі кожного обраного зображення. Відзначити діаґрами, для яких слід передбачити формальний або напівформальний контроль (*які вимагають періодичного перегляду*) і зберегти їх у складі документації проекту.

При моделюванні простого монолітного додатка різного застосування, який виконується на одному комп'ютері, буде потрібно такі діаґрами: зображення варіантів діаґрами ВВ, зображення дизайну діаґрами класів (*для структурного моделювання*), зображення взаємодій діаґрами взаємодії (*для моделювання поведінки*), зображення реалізації діаґрами складової структури, зображення розгортання.

Якщо система зосереджена на потоці процесів, не виключено, що для моделювання її поведінки доцільно використовувати відповідно діаґрами станів і діаґрами діяльності. Для систем з архітектурою «клієнт/сервер» знадобляться діаґрами компонентів і діаґрами розгортання, щоб змоделювати її фізичні особливості.

При підготовці складної розподіленої системи необхідно задіяти повний набір діаграм UML, щоб виразити її архітектуру і виявити технічні ризики проекту: зображення варіантів діаграми ВВ і діаграми послідовності, зображення дизайну діаграми класів, діаграми взаємодії, діаграми станів (*для моделювання поведінки*), діаграми діяльності, зображення взаємодій діаграми взаємодії (*для моделювання поведінки*), зображення реалізації діаграми класів, діаграми складової структури, зображення діаграми розгортання.

Одна модель – різні рівні абстракції. [2] Потреба в оцінюванні системи з різних точок зору поряд з розробниками виникає і в користувачів, що потребують тих же виглядів системи, але на різних рівнях абстракції. Маючи набір класів, який охоплює *словник предметної області*, програміст може побажати уточнити вигляд аж до рівня атрибутів, операцій і зв'язків. Водночас аналітик, що досліджує сценарії ВВ разом із кінцевим користувачем, бажає бачити більш загальний вигляд тих же класів. У цьому контексті *програміст працює на нижчому рівні абстракції*, а аналітик і кінцевий користувач – на більш високому, хоча всі вони мають справу з однієї й тією ж моделлю.

Моделювання систем на різних рівнях абстракції. Діаграми – графічне зображення елементів моделі. Можна створювати кілька діаграм *однієї моделі* з різним ступенем деталізації. Другий шлях полягає у формуванні кількох моделей на різних рівнях абстракції з діаграмами, що відображають перехід від однієї моделі до іншої (*рівень абстракції протилежний рівню деталізації*).

Щоб змоделювати систему з різними рівнями деталізації, необхідно розглянути потреби майбутніх користувачів і почати з обраної моделі. Якщо модель буде використовуватися для конструювання реалізації, на діаграмах слід показати значне число деталей (*низький рівень абстракції*). Якщо модель є концепцію системи для користувачів, то потрібні діаграми, що перебувають на більш високому рівні абстракції з приховуванням більшості деталей. Потрібно створити діаграми з обраним рівнем абстракції, приховуючи або відображаючи *чотири категорії сутностей* моделі:

Будівельні блоки і зв'язки: приховати ті, що не відображають призначення діаграми і не відповідають запитам користувачів.

Доповнення: показувати їх тільки для тих будівельних блоків і зв'язків, які важливі для розуміння намірів розробника.

Потоки: на діаграмах поведінки слід розкривати тільки ті повідомлення й переходи, які прояснюють подальші наміри.

Стереотипи: при використовуванні для класифікації списків сутностей (атрибути й операції) показувати тільки ті елементи зі стереотипами, які розкривають наміри проектування.

Головна перевага такого підходу полягає в тому, що система моделюється на основі загального *семантичного репозиторію*. Недолік: зміни в діаграмі одного рівня абстракції можуть зробити недійсними діаграми інших рівнів абстракції.

Для моделювання системи на різних рівнях абстракції зі створенням різнорівневих моделей необхідно з урахуванням потреб користувачів визначити рівень абстракції для кожного вигляду і сформувати окрему модель для кожного рівня. Наповнити моделі, що перебувають на високому рівні абстракції, простими абстракціями, а моделі, що перебувають на низькому рівні абстракції, – деталізованими. Установити трасування залежностей між зв'язаними елементами різних моделей.

Якщо дотримуватися п'яти представлень архітектури, при моделюванні систем на різних рівнях абстракції можливі чотири загальні ситуації: ВВ у моделі ВВ будуть

пов'язані з коопераціями в моделі проектування; кооперації будуть пов'язані із сукупністю класів, що працюють разом у їхньому складі; компоненти в моделі реалізації будуть пов'язані з елементами моделі проектування; вузли в моделі розгортання будуть пов'язані з компонентами в моделі реалізації.

Головна перевага даного підходу в тому, що діаґрами різних рівнів абстракції залишаються слабко зв'язаними: зміни в одній моделі не виявлять значного впливу на інші моделі. Головний недолік: буде потрібно чимало зусиль, щоб синхронізувати всі моделі та їх діаґрами. Це особливо важливо, коли моделі відносяться до різних фаз ЖЦ розроблення ПЗ (*супровід аналітичної моделі окремо від моделі проектування тощо*).

Створення діаґрам. Важливим є якісна ВСКД розроблення ПЗ (*а не картинки*). Діаґрами – один із засобів підготовки працюючої системи; не всіх їх варто зберігати. Слід розглянути можливість побудови проміжних діаґрам, аналізуючи елементи моделей і використовувати в процесі побудови системи. Не слід перевантажувати моделі зайвими або надлишковими діаґрамами; показувати на кожній діаґрамі мінімум деталей, необхідний для виконання їх призначень (*другорядна інформація відволікає користувача від основної ідеї*). Не створювати надто спрощених діаґрам (*надмірне спрощення може сховати деталі, важливі для розуміння моделей*); потрібно зберігати баланс між структурними діаґрамами і діаґрамами поведінки в системі (*не всі системи є повністю статичними або повністю динамічними*). Не робити діаґрами надто великими (*не більше одного аркуша – важко читати*) або надто малими (*об'єднувати прості діаґрами в одну*); надавати кожній діаґрамі ім'я, яке чітко пояснює її призначення; організовувати діаґрами – групувати їх у пакети відповідно до зображення; не захоплюватися форматуванням діаґрам (*це краще виконають інструментальні засоби*).

Добре структурована діаґрама зосереджена на передаванні одного аспекту системи. Містить тільки ті елементи, які істотні для розуміння цього аспекту; деталізована у відповідності з рівнем абстракції (*показує тільки ті доповнення, які важливі для розуміння діаґрами на даному рівні*); не настільки лаконічна, щоб користувач випустив з уваги важливу семантику.

2.3.2. Діаґрами класів

Основні питання:

- *Моделювання простих кооперацій*
- *Моделювання логічної схеми бази даних*

Діаґрами класів показують набір класів, інтерфейсів і кооперацій, а також їх зв'язки. Застосовуються на практиці для моделювання статичного зображення об'єктно-орієнтованих систем (*це моделювання словника системи, кооперацій або схем*). Вони є основою для цілої ґрупи взаємозалежних діаґрам (*компонентів та розміщення*) і застосовуються для ВСКД систем із використанням прямого і зворотного проектування.

В будівництві будинку, які в будь-якій іншій справі, не обійтися без понятійного апарата – **словника***, що включає основні будівельні блоки: стіни, підлоги, вікна, двері…. Ці сутності значною мірою структуровані (стіни мають висоту, ширину, товщину). При цьому кожна з них певним чином поводиться (стіни різних типів витримують різні навантаження; двері можуть відчинятися і заинятися в різні боки …). Неможливо розглядати їх* **структурні** *й* **поведінкові** *параметри незалежно один від одного: потрібно враховувати, як вони* взаємодіють. *Процес проектування будинку у такий спосіб містить компонування всіх перерахованих вище сутностей на основі*

практики здорового глузду і покликані задовольнити всі функціональні й нефункціональні вимоги.

Розроблення ПЗ проводиться за тією ж схемою – за винятком того, що завдяки гнучкості ПЗ є можливість визначати «*з нуля*» власні базові будівельні блоки.

Діаґрама класів (class diagram) моделює статичні аспекти будівельних блоків – наборів класів, інтерфейсів, кооперацій та їх зв'язків (*графічне відображення набору вершин і з'єднувальних дуг*) та визначає деталі їх конструювання [1,12] (рис. 2.3.1).

Основні властивості. Діаґрама класів (ДК), як і будь-яка інша діаґрама, має ім'я і вміст, є проекцією моделі й відрізняється від інших конкретним наповненням. На діаґрамах класів показують:

- класи;
- інтерфейси;
- залежності, узагальнення й асоціації.

ДК можуть містити примітки й обмеження, пакети або підсистеми, що групують елементи моделі в більші утворення. Іноді в ДК потрібно додати екземпляри, особливо якщо необхідно візуалізувати тип екземпляра (*можливо, динамічний*).

Стандартне використання. ДК моделюють статичний вигляд системи, який описує послуги, які система повинна надавати її кінцевим користувачам насамперед (*підтримує функціональні вимоги*). Проробляючи статичне зображення системи, ДК використовуються для таких цілей:

1. *Для моделювання словника системи*. Моделювання словника системи припускає вирішення питання про те, які абстракції стануть предметом уваги системи, а які виходять за її границі. Діаґрами класів допоможуть специфікувати ці абстракції та їх призначення.

2. *Для моделювання простих кооперацій*. Кооперація – це співтовариство класів, інтерфейсів та інших елементів, що працюють у сукупності для формування деякої спільної поведінки, яка не може бути забезпечена всіма цими елементами, окремо взятими (при моделюванні семантики транзакції в розподіленій системі а, дивлячись на окремий клас, не зможете зрозуміти, що відбувається. Ця семантика забезпечується набором класів, які працюють разом). Діаґрама класів дозволить візуалізувати й специфікувати такий набір класів, їх зв'язків.

3. *Для моделювання логічної схеми БД*. Схему в даному контексті можна розглядати як проект концептуального проектування БД. У багатьох галузях потрібно зберігати інформацію в реляційній або об'єктно-орієнтованій БД. Їхні схеми зручно моделювати за допомогою ДК.

Моделювання кооперацій. Кожен клас працює в кооперації з іншими для реалізації ВВ. Проектуючи словник системи, слід звернути увагу на ВСКД різних способів взаємодії сутностей словника. Слід використовувати ДК для моделювання таких кооперацій та ідентифікувати механізм моделювання. Механізм є деякою функціональною або поведінковою частиною модельованої системи, отриманою в результаті взаємодії сукупності класів, інтерфейсів та інших сутностей. Для кожного механізму ідентифікувати класи, інтерфейси та інші кооперації, які беруть участь у даній кооперації, а також зв'язки між цими сутностями. Використовувати сценарії для тестування всіх цих сутностей. На даному етапі виявляються частини моделі, які були пропущені, а також ті, які семантично невірні. Переконатися, що всі елементи наповнені правильним вмістом. Що стосується класів, для початку необхідно забезпечити оптимальний баланс обов'язків, а потім проробити конкретні атрибути та операції.

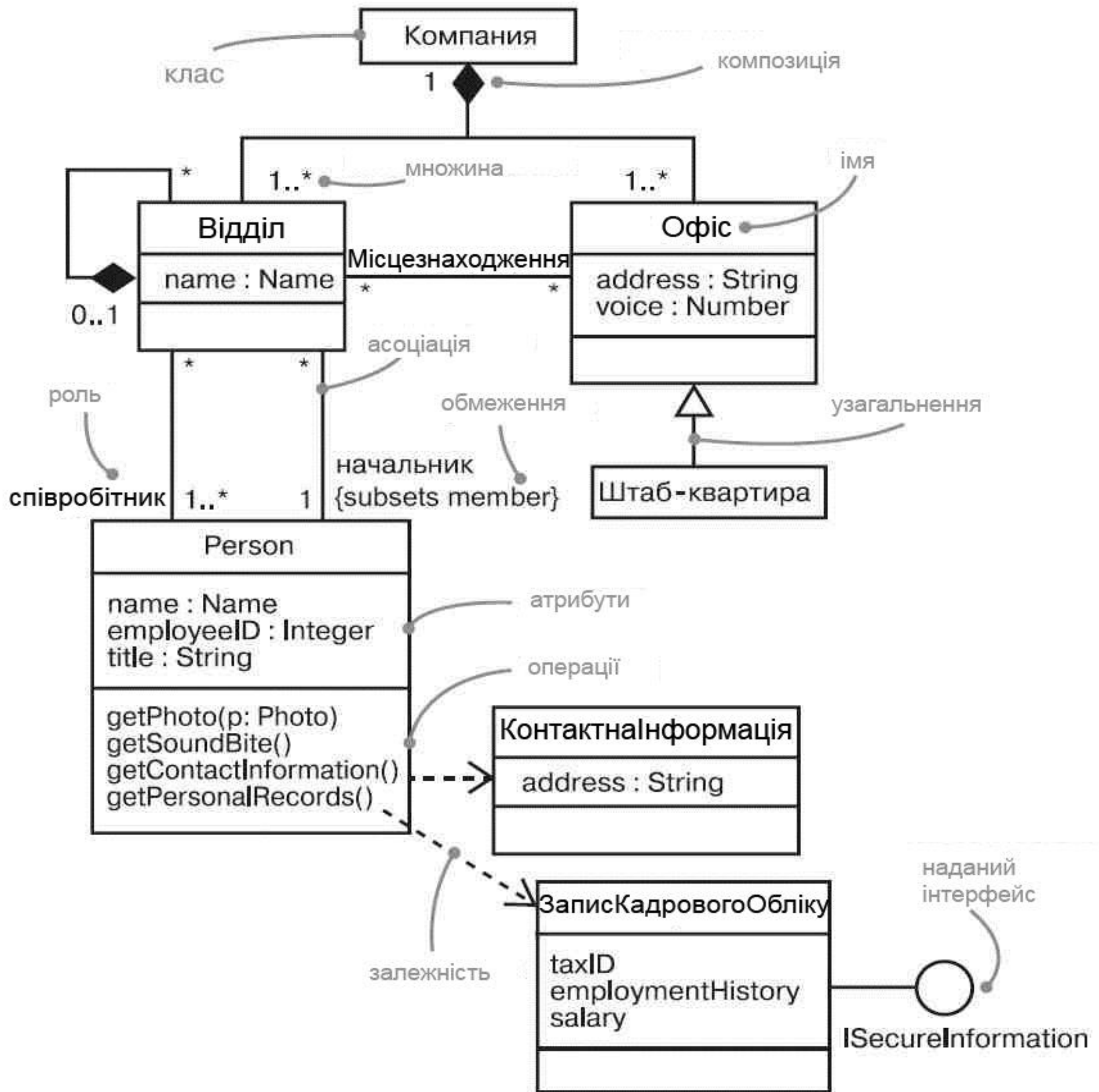


Рис. 2.3.1. Діаграма класів

Діаґрама класів описування реалізації автономного робота (рис. 2.3.2). На діаґрамі акцент зроблено на класи, пов'язані з механізмом переміщення робота за заданим маршрутом. Тут один абстрактний клас Motor (Двигун) із двома конкретними нащадками – Steeringmotor (Двигун поворотного механізму) і Mainmotor (Головний двигун). Обидва нащадки успадковують від свого батька Motor п'ять операцій. І обоє вони, у свою чергу, представлені як частина іншого класу – Driver (Привід). Клас Pathagent (Агент траєкторії) має асоціацію « один-до-одного» з Driver і «один-до-багатьох» – з Collisionsensor (Сенсор зіткнень). Ніяких атрибутів і операцій для Pathagent не показано, хоча його обов'язки й задані.

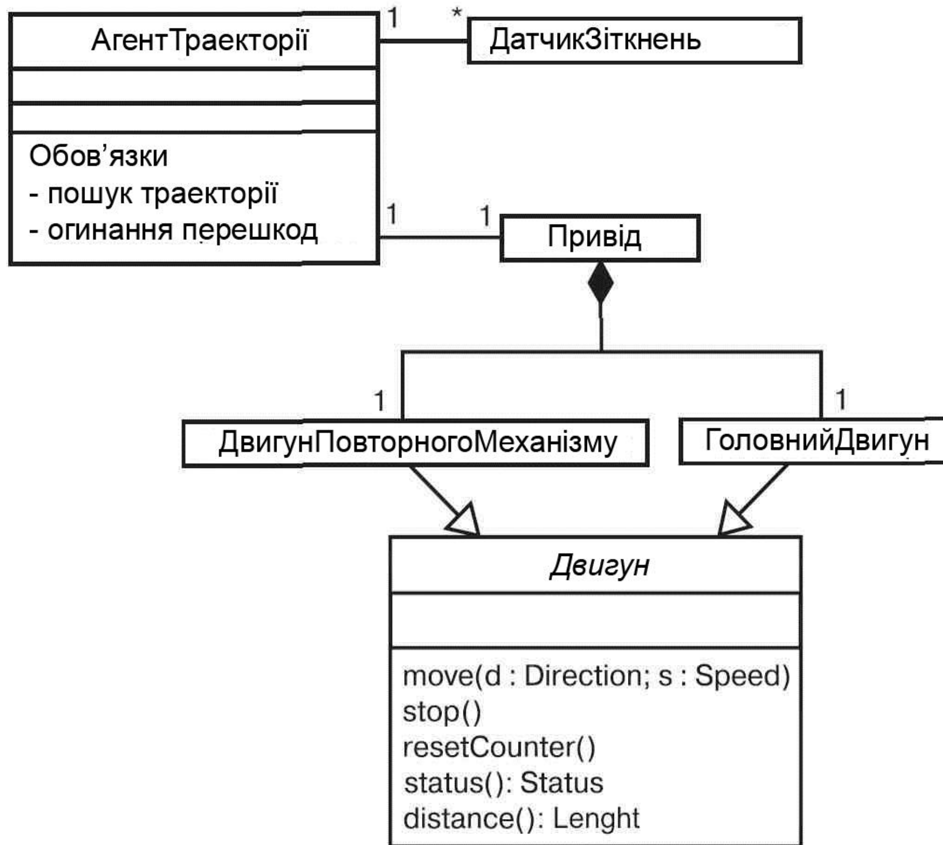


Рис. 2.3.2. Моделювання простої кооперації

У цій системі бере участь багато інших класів, але діаґрама зосереджена тільки на переміщенні робота. Водночас деякі класи, представлені на ній, можна знайти й на інших діаґрамах. Наприклад, клас Pathagent кооперується щонайменше із двома іншими: Environment (Оточення) і Goalagent (Аґент цілі) у механізмі вищого рівня, що управляє дозволом конфліктних ситуацій, у яких може опинитися робот (*хоча це не показано*). Класи Collisionsensor і Driver, а також їх частини кооперуються з класом Faultagent (Аґент відмови) у складі механізму, відповідального за безперервну діагностику устаткування робота на предмет різноманітних помилок. Зосереджуючи увагу на кожній із цих кооперацій у різних діаґрамах, створюється зрозуміле зображення про систему з різних точок зору.

Багато систем, які потрібно моделювати, будуть мати у своєму складі об'єкти, що зберігаються в базі даних для наступного використання. Найчастіше можуть використовуватися для зберігання реляційні й об'єктно-орієнтовані бази даних або гібридні об'єктно-реляційні бази. UML добре підходить для моделювання як логічних, так і фізичних схем баз даних.

Відмінніссть діаґрам класів UML і E-R **діаграм БД**. Діаґрами класів UML є надмножиною діаґрам «сутність-зв'язок» (entity-relationship, E-R) – загального інструмента моделювання, що використовується в логічному проектуванні баз даних. У той час, як *класичні* E-R *діаґрами* зосереджені тільки на даних, **діаґрами класів** ідуть на крок попереду, дозволяючи також **моделювати поведінку**. У фізичній базі даних ці логічні операції представлені у вигляді тригерів і збережених процедур.

Щоб змоделювати схему БД, необхідно: ідентифікувати ці класи в моделі, стани яких не повинні бути залежними від ЖЦ працюючого додатка; створити діаґраму класів,

що містить класи, виявлені на першому етапі. Можна визначити власний набір стереотипів і присвоєних значень, щоб представити специфічні для БД деталі; розкрити структурні подробиці цих класів. В основному це означає необхідність докладно специфікувати їх атрибути, а також асоціації із вказівкою множинності, які зв'язують дані класи. Виявити загальні зразки, які ускладнюють фізичне проектування БД, наприклад, циклічні асоціації й асоціації «один-до-одного». За необхідності створити проміжні абстракції для спрощення логічної структури. Розглянути поведінку класів, відзначених на першому етапі, розкривши операції, істотні для доступу до даних, і забезпечення їх цілісності. Для чіткого розмежування різних аспектів системи потрібна інкапсуляція бізнес-правил, що описують маніпуляції наборами цих об'єктів, у шарі, що перебуває над даними стійкими класами. Там, де можливо, слід використовувати інструментальні засоби, які допоможуть трансформувати логічний дизайн у фізичний. Це загальний підхід, за яким можна моделювати схеми за допомогою UML. На практиці найчастіше мають справу зі стереотипами, налаштованими під використовувану базу даних (*реляційну або об'єктно-орієнтовану*).

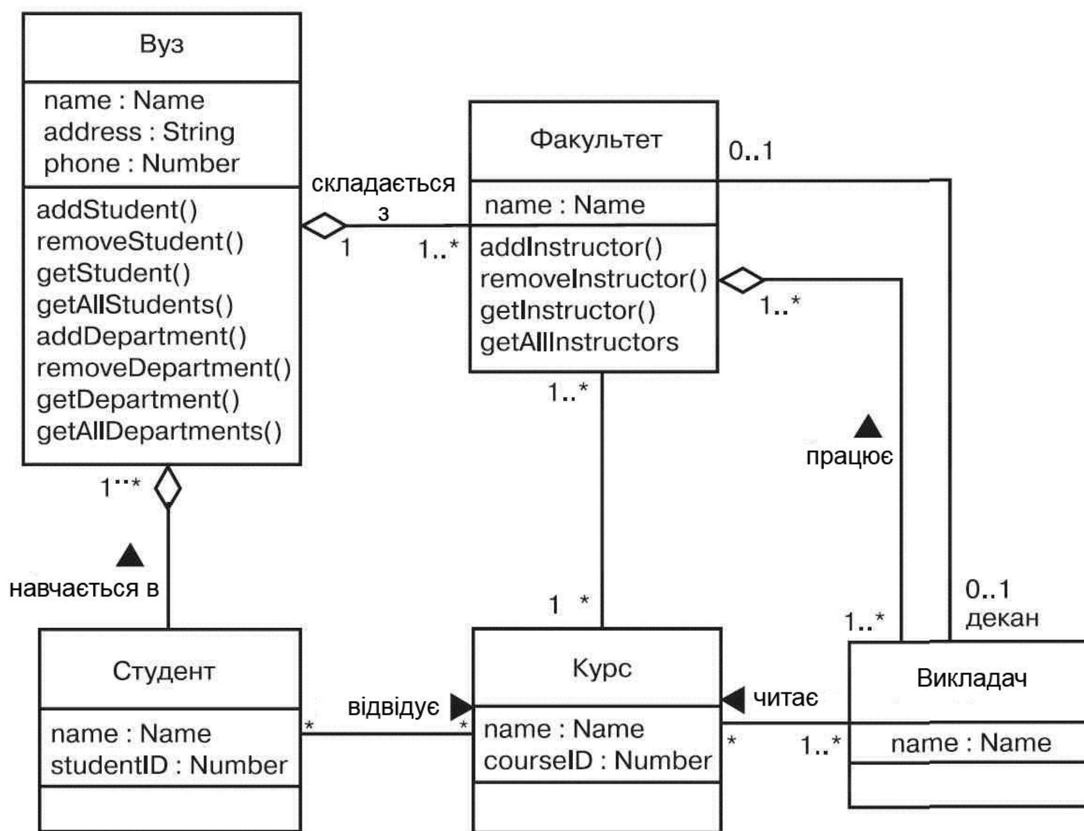


Рис. 2.3.3. Моделювання схеми бази даних

На рис. 2.3.3 показано набір класів, що описують розглядувану раніше інформаційну систему навчального закладу. Тут наведено деталізовану діаґраму класів, яка показує деталі класів, важливих для конструювання фізичної схеми БД, що містить класи Student (Студент), Course (Курс) і Instructor (Викладач). Існує асоціація між Student і Course, яка вказує, що студенти відвідують курси. Більше того, кожен студент може вибирати необмежене число курсів; кожен курс може відвідувати велика кількість студентів. Ця діаґрама розкриває атрибути всіх шести класів (всі атрибути є

примітивних типів). Коли моделюється схема реальної БД, зв'язки з непримітивними типами моделюються у вигляді явних асоціацій, а не атрибутів.

Два класи – School (Навчальний заклад) і Department (Факультет) – включають кілька операцій, призначених для маніпулювання їх частинами. Ці операції важливі для підтримування цілісності даних: наприклад, додавання і видалення факультетів може викликати деякі непорозуміння. Є багато інших операцій, які варто розглянути для цих та інших класів (*запит певної інформації перед записом студента на курс*). Такі функції можна трактувати скоріше як бізнес-правила, а не операції, покликані забезпечувати цілісність даних, тому їх краще помістити на вищий рівень абстракції.

Головний продукт розробників – ПЗ, а не діаґрами. Призначення моделі полягає в тому, щоб представити ПЗ, яке вчасно задовольняє мінливі потреби його користувачів і вимоги бізнесу. З цієї причини важливо, щоб створювані моделі та їх практичне втілення чітко відповідали один одному, причому з мінімумом витрат (а краще зовсім без таких) на підтримку їх у синхронізованому вигляді.

Пряме проектування (forward engineering) – це процес трансформації моделі в код мовою реалізації за допомогою відображення. [1] У результаті прямого проектування відбувається втрата інформації, оскільки моделі, описані на UML, семантично багатші, аніж будь-яка сучасна ООП мова. Фактично це головна причина існування моделей поряд із кодом. Структурні засоби, наприклад, кооперації, і поведінкові, наприклад, взаємодії можуть бути чітко візуалізовані за допомогою UML, але не так чітко – у вихідному коді. [10]

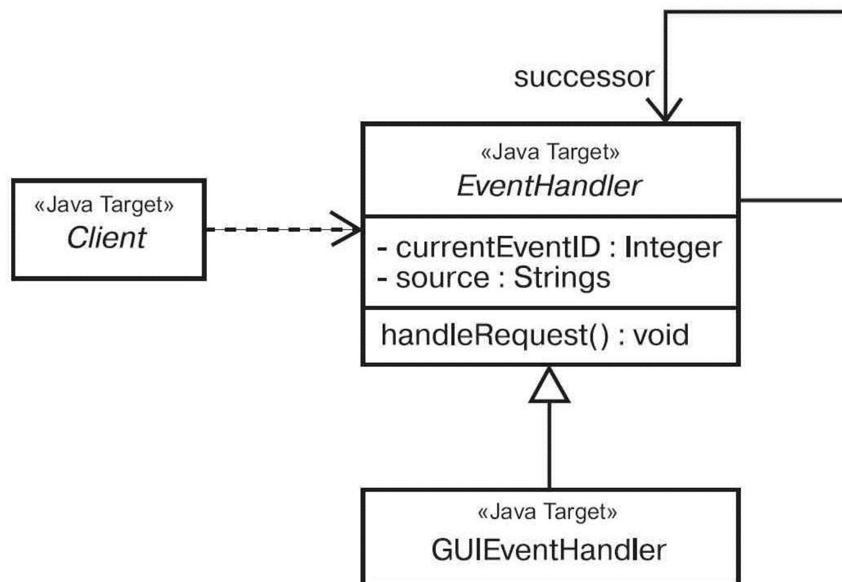


Рис. 2.3.4. Пряме проектування

Щоб здійснити пряме проектування діаґрами класів, необхідно: ідентифікувати конкретні правила відображення класів у код для обраної вами мови (або мов) реалізації. Ці правила будуть стосуватися проекту в цілому або всієї команди. Якщо потрібно, залежно від семантики обраних мов, можна встановити обмеження на деякі засоби UML (UML дозволяє моделювати множинне успадкування, а java його не допускає). Можна заборонити розробникам моделювати множинне успадкування або розробити ідіому, яка трансформує ці засоби в мову реалізації. Використовувати присвоєні значення для вказання параметрів реалізації цільовою мовою програмування.

Можна робити це на рівні індивідуальних класів, якщо потрібен тонкий контроль, або ж на більш високому рівні (*рівні кооперацій або пакетів*). Необхідно використовувати інструменти для генерації коду.

Зразок ланцюжка обов'язків (responsibility pattern). Діаґрама класів представляє реалізацію **зразка ланцюжка обов'язків** (responsibility pattern). На діаґрамі класів (рис. 2.3.4) ця конкретна реалізація включає три класи: Client (Клієнт), EventHandler (Обробник подій) і GuiEventHandler (Графічний інтерфейс обробника подій). Класи Client і EventHandler – абстрактні, GuiEventHandler – конкретний. До EventHandler ставиться звичайна операція, передбачена цим зразком, – handleRequest (опрацювати запит), хоча до його реалізації додано два закриті атрибути.

Усі ці класи припускають відображення мовою Java, як відзначено в їхньому стереотипі. Пряме проектування класів даної діаґрами мовою Java досить просте при використанні інструментальних засобів. Пряме проектування класу Eventhandler на Java генерує такий код:

```
public abstract class Eventhandler
 {
  Eventhandler successor;
  private Integer currenteventid;
  private String source;
  Eventhandler()
    { }
 public void handlerequest()
    { }
 }
```

Зворотне проектування (reverse engineering) – це процес трансформації коду в модель. Зворотне проектування породжує надлишок інформації, частина якої представлена на нижчому рівні деталізації, ніж потрібно для побудови зручної моделі. Водночас зворотне проектування неповне: при прямому проектуванні моделі в код відбувається втрата інформації, тому неможливо точно відновити модель із коду, якщо тільки використовуваний вами інструмент не кодує інформацію у вигляді коментарів до вихідного коду, які виходять за межі семантики мови реалізації.

Щоб здійснити зворотне проектування діаґрами класів, необхідно: ідентифікувати правила відображення для обраної мови або мов реалізації – це вам знадобиться зробити для всього проекту або всієї організації. Використовуючи який-небудь інструментальний засіб, указати код, який потрібно піддати зворотному проектуванню. Застосовувати інструмент для генерування нової моделі або модифікації існуючої, стосовно якої раніше застосовувалося пряме проектування. Не варто очікувати, що зворотне проектування породить єдину компактну модель на основі великого фрагмента коду. Доведеться вибирати невеликі фрагменти коду і будувати з них модель. Переглядаючи модель, необхідно створити діаґраму класів за допомогою якого-небудь інструментального засобу. Можна почати з одного або кількох класів, а потім розширювати діаґраму, проробляючи конкретні зв'язки або включаючи сусідні класи. Слід показувати або приховувати деталі діаґрами відповідно до намірів проектування та додати до моделі інформацію про проектування, щоб показати ті аспекти дизайну, які пропущені або сховані у вихідному коді. Створюючи діаґрами класів на UML, слід пам'ятати, що кожна з них – це лише графічне зображення статичного зображення

проекту системи. Жодна діаґрама класів не зобов’язана включати все, що стосується проекту системи. Однак діаґрами класів у сукупності повідомляють користувачеві повну інформацію, необхідну для статичного зображення системи, хоча кожна з них представляє тільки один її аспект.

Добре структурована діаґрама класів сфокусована на одному аспекті статичного вигляду дизайну системи; містить елементи, істотні для розуміння даного аспекту; забезпечує деталізацію, відповідну до рівня абстракції діаґрами, включаючи доповнення, важливі для розуміння; не настільки спрощена, щоб створити неправильне уявлення про важливу семантику. Створюючи діаґраму класів, слід надати їй ім’я, що відповідає її призначенню; уникати (мінімізувати) перетинання ліній; організувати елементи так, щоб семантично близькі сутності розташовувалися поруч; використовувати примітки і кольори як означки, що акцентують увагу на важливих деталях діаґрами; не показувати занадто багато видів зв’язків. На кожній діаґрамі класів повинен домінувати тільки один вид зв’язків.

2.3.3. Діаґрами об’єктів

Діаґрами об’єктів (object diagrams) дозволяють моделювати екземпляри сутностей, які містяться в діаґрамах класів. На діаґрамі об’єктів показано множини об’єктів і зв’язків між ними в певний момент часу. Діаґрами об’єктів застосовують при моделюванні статичних виглядів системи з погляду проектування й процесів. При цьому моделюється «знімок» системи в конкретний момент часу і відображається множина об’єктів, їх станів і зв’язків між ними. [4,12] Діаґрами об’єктів важливі для ВСКД структурних моделей і для конструювання статичних аспектів системи за допомогою прямого і зворотного проектування.

Для людини, не знайомої з правилами гри, футбол може здатися надзвичайно простим видом спорту: юрба народу хаотично носиться полем, ганяючи м’яч. Недосвідчений глядач навряд чи побачить у цьому русі який-небудь порядок або оцінить красу гри. Якщо перервати матч і показати глядачеві, які ролі виконують окремі гравці, перед ним постане зовсім інша картина. *У загальній масі він розрізнить нападаючих, захисників і півзахисників, а поспостерігавши за ними, зрозуміє, як вони взаємодіють за певною стратегією, спрямованою на те, щоб забити гол у ворота супротивника: ведуть м’яч по полю, відбирають його один в одного й атакують.* У досвідченій команді ніколи не знайдеться гравців, що безладно і безцільно переміщаються по полю. *Навпаки, у будь-який момент часу розташування гравців і їх взаємодії точно розраховані.*

Спостерігаючи за потоком керування в працюючій системі ПЗ, швидко втрачається загальна уява про організацію її складових, особливо якщо є кілька потоків. Вивчення стану одного об’єкта в конкретний момент часу так само не допоможе зрозуміти таку складну структуру. Потрібно *розглянути не тільки сам об’єкт, але і його найближчих сусідів та зв’язки між ними*. Об’єкти не існують автономно, а певним чином зв’язані з множиною інших. Більше того, неполадки в таких системах найчастіше пояснюються не логічними помилками, а саме порушеннями взаємозв’язків об’єктів або непередбаченими змінами їх стану.

Статичні аспекти UML – будівельних блоків системи – візуалізують за допомогою діаґрам класів. *Діаґрами взаємодії* дозволяють побачити динамічні аспекти системи, включаючи екземпляри цих будівельних блоків і повідомлення, якими вони обмінюються. *Діаґрама об’єктів* містить множини екземплярів сутностей, наведених на діаґрамі класів.

Діаграми об'єктів представляють *статичну складову взаємодії* й складаються із взаємодіючих об'єктів, однак повідомлення на них не показані. Їх представляють у вигляді графа, що складається з вершин і ребер. Діаграма об'єктів відображає *стан системи у фіксований момент часу* (рис. 3.6.1).

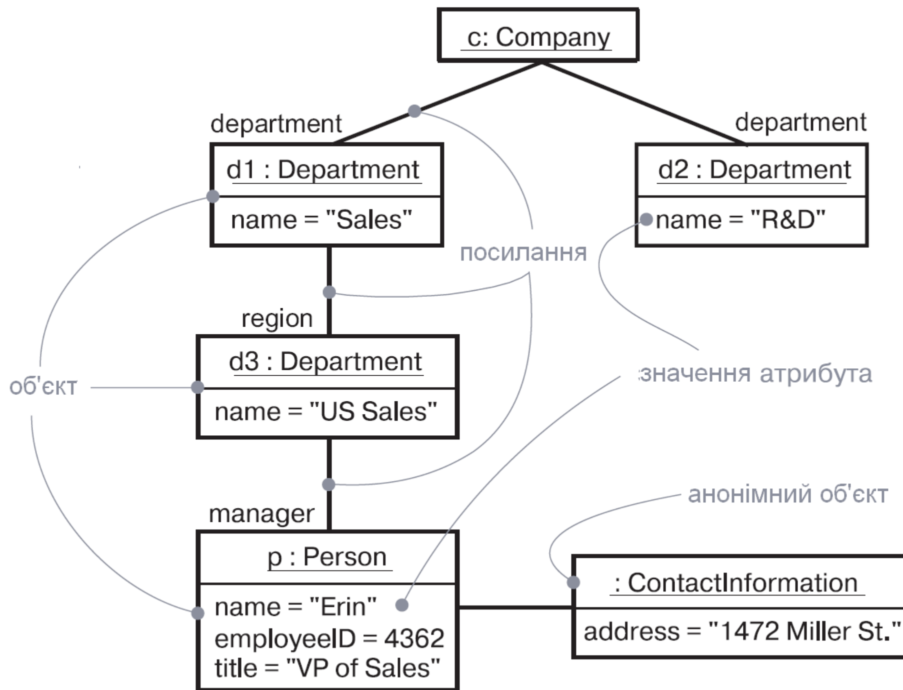


Рис. 2.3.5. Діаграма об'єктів

На діаграмі об'єктів показані об'єкти і зв'язок між ними в певний момент часу.

Загальні властивості. У діаграми об'єктів, як і в будь-якої діаграми, є ім'я і графічний зміст, що є проекцією моделі. Від інших діаграм відрізняється своїм конкретним наповненням. Зміст діаграми об'єктів, як правило, містить об'єкти й посилання та може містити в собі примітки та обмеження. Іноді в них вміщують і класи, особливо якщо треба візуалізувати класи, що визначають екземпляри.

Статичний кадр у динамічному сценарії. За допомогою діаграм об'єктів, як і за допомогою діаграм класів, моделюють статичний вигляд системи з погляду проектування або процесів, враховуючи реальні екземпляри чи прототипи. Цей вигляд відображає в основному функціональні вимоги до системи, тобто послуги, які вона повинна надавати кінцевим користувачам. Діаграми об'єктів дозволяють моделювати статичні структури даних.

При моделюванні статичних виглядів системи з погляду проектування або взаємодії діаграми об'єктів застосовують *для моделювання структури* об'єктів системи. Моделювання структури об'єктів дає «знімок» об'єктів системи в цей момент часу. Діаграма об'єктів є одним *статичним кадром у динамічному сценарії*, що описується діаграмою взаємодії. Вони застосовуються для ВСКД певних екземплярів у системі, а також зв'язків між цими екземплярами. Динаміку поведінки можна зобразити у вигляді послідовності кадрів.

Моделювання структур об'єктів. Конструюючи діаграму класів, компонентів або розміщення насправді описується необхідна група абстракцій і розкривається в даному контексті їх семантика та зв'язки з іншими абстракціями в групі. Ці діаграми

відображають тільки потенційні можливості. Наприклад, якщо клас A пов'язаний із класом B асоціацією типу «*один-до-багатьох*», то з одним екземпляром класу A може бути зв'язано п'ять екземплярів класу B, а з іншим – тільки один. Крім того, у будь-який конкретний момент часу екземпляр класу A і пов'язані з ним екземпляри класу B будуть мати цілком певні значення своїх атрибутів і стану автоматів.

«Заморозивши» працюючу систему або просто уявивши собі якусь мить у ЖЦ модельованої системи, виявляється сукупність об'єктів, кожен з яких перебуває в певному стані і має конкретні зв'язки з іншими об'єктами. При моделюванні виду системи з погляду проектування за допомогою набору діаґрам класів можна повністю визначити семантику абстракцій та їх зв'язків. Однак діаґрами об'єктів не дозволяють повністю описати об'єктну структуру системи. Клас може мати велику кількість різних екземплярів, а за наявності кількох класів, зв'язаних один з одним, число можливих конфігурацій об'єктів багаторазово зростає. Тому при використанні діаґрам об'єктів потрібно зосередитися на зображенні найважливіших наборів, конкретних об'єктів або *об'єктів-прототипів*. Саме це розуміється під моделюванням структури об'єктів – відображення на діаґрамі множини об'єктів і відношень між ними в деякий момент часу.

Для моделювання структури об'єктів потрібно ідентифікувати механізм моделювання, що є деякою функцією або поведінкою частини модельованої системи, що є результатом взаємодії співтовариства класів, інтерфейсів та інших сутностей. Для кожного виявленого механізму потрібно ідентифікувати класи, інтерфейси та інші елементи, що беруть участь у кооперації, а також зв'язки між ними. Розглянемо один зі сценаріїв роботи механізму. «Заморозити» цей сценарій у деякий момент часу і зобразити всі об'єкти, що беруть участь у механізмі. Показати стан і значення атрибутів кожного такого об'єкта, якщо це необхідно для розуміння сценарію. Також показати посилання між цими об'єктами, які представляють екземпляри асоціацій.

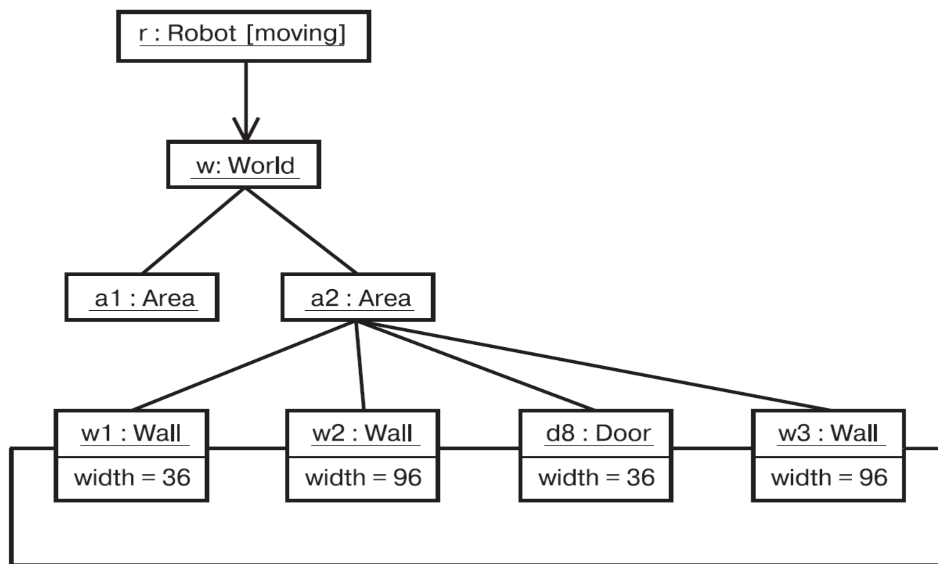


Рис. 2.3.6 Моделювання структур об'єктів

На на рис. 2.3.6 показана сукупність об'єктів, узята з реалізації автономного робота. Тут акцентована увага на кількох об'єктах, що є частиною механізму робота, призначеного для ідентифікації моделі навколишнього середовища, у якому той

переміщається. На даній діаґрамі розглядаються тільки абстракції, безпосередньо залучені до процесу формування представлення навколишнього середовища (*у роботі системи бере участь набагато більше об'єктів*). Як бачимо з рис. 2.3.6, один із об'єктів відповідає самому роботові (екземпляр класу Robot) і в даний момент перебуває в стані moving (рухається). Цей об'єкт пов'язаний з екземпляром w класу World (Світ), що є абстракцією моделі світу робота. У свою чергу об'єкт w пов'язаний з мультиоб'єктом, який складається з екземплярів класу Element (Елемент), що описує сутності, пізнані роботом, але ще не включені в його модель світу. Такі елементи позначені як частини глобального стану робота.

У даний момент часу екземпляр w пов'язаний із двома екземплярами класу Area. В одного з них (a2) показані його власні посилання на об'єкти класу Wall (Стіна) і об'єкт класу Door (Двері). Зазначена ширина кожної із трьох стін і відзначено, що кожна пов'язана із сусідніми. Як видно з діаґрами, робот розпізнав, що замкнене приміщення, у якому він перебуває, має із трьох сторін стіни, а із четвертої – двері.

Зворотне проектування діаґрами об'єктів здійснюється за такою схемою: вибрати, що необхідно реконструювати (*операція або екземпляр конкретного класу*); за допомогою інструментальних засобів, пройшовшись за сценарієм, зафіксувати роботу системи в деякий момент часу; ідентифікувати множину необхідних об'єктів, що взаємодіють у даному контексті, зобразити їх на діаґрамі об'єктів (*і показати стани об'єктів*); для забезпечення розуміння семантики ідентифікувати посилання, що існують між об'єктами; якщо діаґрама виявилася надто складною, спростити її, забравши об'єкти, несуттєві для прояснення даного сценарію. Якщо діаґрама надто проста, включити в неї оточення деяких об'єктів, що є цікавими і детальніше показати стан кожного об'єкта.

Жодна окремо взята діаґрама об'єктів не в змозі передати всю укладену в цих виглядах інформацію. В складних існують сотні й тисячі об'єктів, більша частина яких анонімна. Повністю специфікувати усі об'єкти системи і всі способи, якими вони можуть бути асоційовані, неможливо. Отже, *діаґрами об'єктів повинні відображати тільки деякі конкретні об'єкти або прототипи, що входять до складу працюючої системи*.

Добре структурована діаґрама об'єктів акцентує увагу на одному аспекті статичного вигляду системи з погляду проектування або процесів; представляє лише один з кадрів динамічного сценарію, показаного на діаґрамі взаємодії; містить тільки істотні для розуміння даного аспекту елементи; рівень її деталізації відповідає рівню абстракції системи (показують тільки значення атрибутів, істотні для розуміння).

Створюючи діаґраму об'єктів, слід присвоїти їй ім'я, відповідне до призначення; розташувати її елементи так, щоб мінімізувати перетинання ліній; організувати її елементи так, щоб семантично близькі сутності виявлялися поруч; використовувати примітки і колір для привернення уваги до важливих особливостей діаґрами; включати в описи кожного об'єкта значення і стани, необхідні для розуміння намірів проектування.

2.3.4. Компоненти та інтерфейси. Діаґрами компонентів

Основні питання:

- *Компоненти, інтерфейси і реалізація*
- *Внутрішні структури, порти, частини і конектор*
- *З'єднання підкомпонентів*
- *Моделювання API*

Компонента – це логічна частина системи, що заміщається та відповідає деякому набору інтерфейсів і забезпечує їхню реалізацію. [2] Якісні компоненти визначають чіткі абстракції з чітко визначеними інтерфейсами, що дає можливість легко замінювати старі компоненти на сумісні з ними нові. Інтерфейси з'єднують логічні моделі з моделями проектування. Можна специфікувати інтерфейс класу в логічній моделі, і той же інтерфейс буде підтримуватися деяким компонентом дизайну, що реалізує його. Інтерфейси дозволяють реалізовувати компоненти із використанням дрібніших компонентів шляхом з'єднання їх портів.

Якщо планується побудувати дім, то можна, мабуть, виділити трохи коштів і на домашній кінотеатр. Можна купити єдиний блок, який включає телевізійний екран, тюнер, VCR/DVD-плеєр і аудіоколонки. Таку систему легко встановити. Вона буде добре працювати, якщо відповідає потребам. Однак система, що складається з єдиного блока, не надто гнучка. Більш гнучкий підхід до установки домашнього кінотеатру – використання окремих компонентів з різною функціональністю. На моніторі видно зображення; колонки відтворюють звук, причому розмістити їх можна в будь-якому місці кімнати, домагаючись доброї акустики… Не будучи жорстко зв'язаними один з одним, ці компоненти дозволяють розмістити їх за своїм розсудом і з'єднати кабелями. Кожен кабель має особливий тип роз'єму, що підходить до відповідного порту пристрою (неможливо під'єднати ти колонку до відеовиходу). Зате при бажанні можна легко додати до цієї системи програвач. Якщо ж виникне ідея її обновити, то потрібно буде заміняти одну компоненту, не втрачаючи кошти на інші. Отже, з одного боку, забезпечена більша гнучкість, а з іншого – висока якість, яку можна поступово «нарощувати». [1,12]

Розробка ПЗ у вигляді великого монолітного і «твердого» вузла є важко-модифікованою в подальшому. Якщо навіть існуюча система має здебільшого необхідні функціональності, вона, швидше за все, містить у собі багато зайвих деталей, які складно або взагалі неможливо вилучити. Тому слід будувати системи з окремих компонентів, які гнучко взаємодіють один з одним і при зміні вимог можуть відокремлюватися і додаватися без шкоди для цілого.

Інтерфейс – набір операцій, що специфікують сервіс, представлений класом або компонентою. [12]

Компонента – змінна частина системи, яка відповідає набору інтерфейсів і забезпечує його реалізацію. [1]

Порт – специфічне «вікно» в інкапсулюючу компоненту, що приймає повідомлення для компоненти і від неї відповідно до заданого інтерфейсу. [2]

Внутрішня структура – реалізація компоненти, представлена набором частин, з'єднаних одна з іншою конкретним способом. [5]

Частина – специфікація ролі, що становить частину реалізації компоненти. В екземплярі компоненти присутній екземпляр, що відповідає частині. [2]

Коннектор – зв'язок комунікації між двома частинами або портами в контексті компоненти. [4]

Компоненти та інтерфейси. Зв'язок між компонентою (*що надає сервіс*) і інтерфейсом має важливе значення. Усі інструментальні та інші програмні системи (COM+, CORBA і Enterprise Java Beans, …) побудовані на компонентах, які використовують інтерфейси компоненти один з одним.

Проектування системи на основі компонентів полягає в її декомпозиції, специфікуючи інтерфейси, що представляють основні з'єднання. Наступним кроком є визначення компонентів, які реалізують інтерфейси, разом з іншими компонентами, які мають доступ до сервісів за допомогою своїх інтерфейсів. Цей механізм дозволяє розгорнути систему, сервіси якої певною мірою незалежні від місця розташування.

Інтерфейс, реалізований компонентою, називають **надаваним** (*компонента надає інтерфейс у вигляді сервісу іншим компонентам*). Компонента може декларувати багато надаваних інтерфейсів. Інтерфейс, який вона використовує, називають **необхідним** – йому відповідає дана компонента, коли запитує сервіси від інших компонентів. Компонента може відповідати багатьом необхідним інтерфейсам. Бувають компоненти, які одночасно надають і вимагають інтерфейси.

Компонента зображується у вигляді прямокутника із двозубчастою піктограмою в правому верхньому куті (рис. 2.3.7). В середині прямокутника вказується ім’я компоненти. У нього можуть бути атрибути й операції, які на діаґрамах часто опускаються. Компонента може показувати мережу внутрішньої структури.

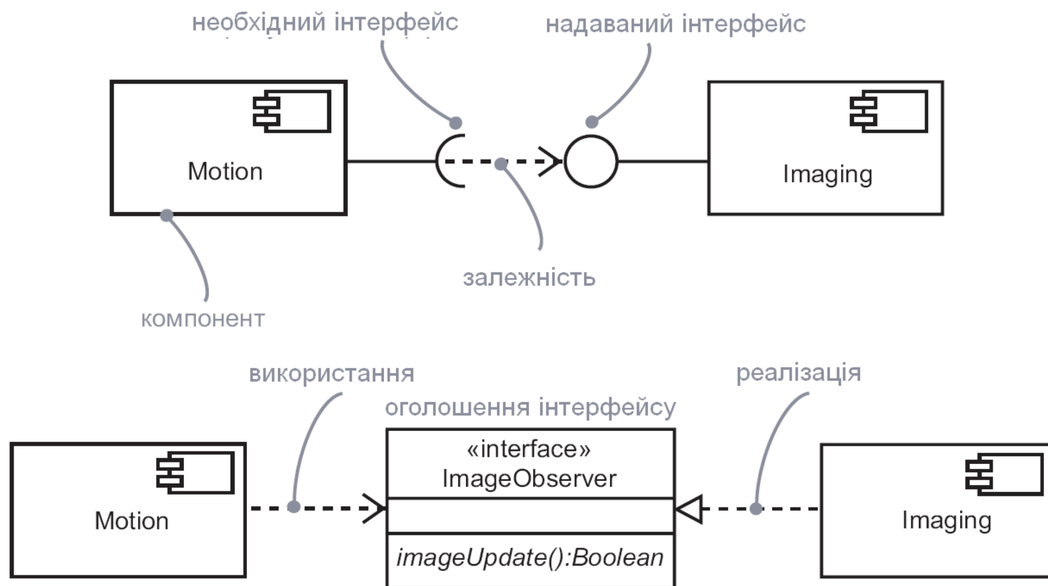


Рис. 2.3.7. Компоненти та інтерфейси

Зв’язок між компонентою та її інтерфейсами виражається одним із двох способів. У піктографічній формі надаваний інтерфейс має вигляд колоа, з’єднаного лінією з компонентою (*«льодяник на паличці», як і для класів*). Необхідний інтерфейс – півколо, з’єднаний з компонентою (*«гніздо»*). Ім’я інтерфейсу зазначене поруч із символом. У розширеній формі зображення інтерфейсу можливе, із вказівкою його операцій: компонента, що реалізує інтерфейс, з’єднується з ним за допомогою повного зв’язку реалізації. Компонента, що має доступ до сервісів іншої компоненти через інтерфейс, з’єднується з інтерфейсом зв’язком залежності. Деякий інтерфейс може бути наданий однією компонентою і вимагатися іншою. Оскільки цей інтерфейс перебуває між двома компонентами, їх пряма залежність одна від одного руйнується. Компонента, що використовує даний інтерфейс, буде працювати правильно незалежно від того, яка компонента його реалізує. Компонента може бути використана у цьому контексті тоді й тільки тоді, коли всі її необхідні інтерфейси реалізовані як надавані в інших компонентах.

Замінність. Основне призначення засобу розроблення системи на основі компонентів – забезпечити складання системи з бінарних замінних артефактів. Таку систему можна проектувати, використовуючи компоненти і потім реалізуючи їх у вигляді артефактів. Можна навіть розбудовувати систему, додаючи нові компоненти й заміняючи старі, не перебудовуючи її всю цілком. Інтерфейси – ключовий засіб, що

забезпечує такі можливості. У працюючій системі допускається застосування будь-яких артефактів, що реалізують компоненти, які погоджуються і надають потрібний інтерфейс. Розширення системи можливе за рахунок створення компонентів, що надають нові сервіси через інші інтерфейси, які інші компоненти, у свою чергу, можуть виявити й використовувати. Ця семантика прояснює мету визначення компонентів у UML. Компонента відповідає набору інтерфейсів і забезпечує його реалізацію, що дозволяє заміщувати його – як у логічному дизайні, так і в заснованій на ньому фізичній реалізації.

Заміщувана компонента — це та, яку у процесі проектування можна замінити іншою, що відповідає тим же інтерфейсам. Механізм вставки заміни артефакту в здійсненній системі прозорий для користувача компоненти і допускається об'єктними моделями (такими, як COM+ і Enterprise Java Beans), які вимагають невеликої проміжної трансформації, або здійснюється інструментами, що автоматизують цей механізм. [13]

Компонента є частиною системи і не часто використовується сама по собі. Частіше вона об'єднана з іншими компонентами, тобто залучена в архітектурний або технологічний контекст, де передбачається її використання. Компонента логічно й фізично погоджена та складає і/або поведінковий фраґмент більшої системи. Вона може бути повторно використана. Це є фундаментальний будівельний блок, на основі якого може бути спроектована й складена система. Компонента відповідає наборові інтерфейсів і забезпечує його реалізацію.

Організація компонентів. Компоненти можна організувати тим же способом, що й класи, ґрупуючи їх у пакети. Допускається організація компонентів шляхом встановлення між ними зв'язків залежності, узагальнення, асоціації (включаючи аґреґацію) та реалізації. Одні компоненти можуть бути побудовані з інших.

Порт (port) **–** це своєрідне «вікно» в інкапсульовану компоненту. Інтерфейси зручні для описування загальної поведінки компоненти, але їм не властива «*індивідуальність*»: реалізація компоненти повинна лише ґарантувати, що всі операції в усіх надаваних інтерфейсах реалізовані. Для повнішого контролю над реалізацією можна використовувати порти.

Уся взаємодія з такою компонентою на вході й на виході відбувається через порти. Поведінка, що виражається ззовні компоненти, складає суму її портів. Кожен порт є унікальним. Одна компонента може взаємодіяти з іншою через певний порт. При цьому їх комунікації повністю описуються інтерфейсами, що підтримує порт, навіть якщо компонент підтримує інші інтерфейси. В реалізації внутрішні частини компоненти можуть взаємодіяти одна з одною через специфічний зовнішній порт, тому кожна частина може бути незалежна від вимог інших. Порти дозволяють розділяти інтерфейси компоненти на дискретні пакети й використовувати відокремлено. Інкапсуляція і незалежність, що забезпечується портами, підвищують ступінь заміщуваності компоненти.

Порт схематично представлений маленьким квадратом на бічній грані компоненти – це отвір на межі інкапсуляції компоненти. Як надаваний, так і необхідний інтерфейс може бути з'єднаний із символом порту. Надаваний інтерфейс зображує сервіс, який може бути запитаний ззовні через даний порт, а необхідний інтерфейс — сервіс, який порт повинен отримати від якого-небудь іншого компонента. У кожного порту є **ім'я**, а отже, він може бути ідентифікований через компоненту та ім'я. Останнє можуть використовувати внутрішні частини компоненти для ідентифікації порту, через який

слід відправляти й отримувати повідомлення. Ім'я компоненти разом з іменем порту ідентифікує порт для використання його іншими компонентами.

Порти є частиною компоненти. Екземпляри портів створюються й знищуються разом з екземпляром компоненти, якій вони належать. Порти також можуть мати множинність. Це означає можливість існування кількох екземплярів порту всередині екземпляра компоненти. Кожен порт компоненти має відповідний масив екземплярів. Хоча всі екземпляри портів у масиві задовольняють один і той же інтерфейс і приймають запити одних і тих самих видів, вони можуть перебувати у різних станах і мати різні значення даних. Наприклад, кожен екземпляр у масиві може мати свій рівень пріоритету (*екземпляр порту з найбільшим рівнем пріоритету обслуговується першим*).

На рис. 2.3.8 представлена модель компоненти Ticket Seller (Продавець квитків) з портами. У кожного порту є ім'я й необов'язковий тип, що показує, яке призначення даного порту. Компонента має порти для продажу квитків, оголошень і обслуговування кредитних карт. Є два порти для продажу – один для звичайних покупців і один для привілейованих. Обоє надають один і той самий інтерфейс типу Ticket Sales (Продаж квитків). Порт обслуговування кредитних карт має необхідний інтерфейс; будь-яка компонента, що його надає, може задовольнити його. Порт оголошень має як надаваний, так і необхідний інтерфейси. Використовуючи інтерфейс Load Attractions (Інформація про розваги), театр може передавати афіші й іншу інформацію про спектаклі в базу даних, яка використовується для продажу квитків. За допомогою інтерфейсу Booking (Замовлення) компонентів продавець квитків може запитувати в театрах відомості про наявність квитків і здобувати їх.

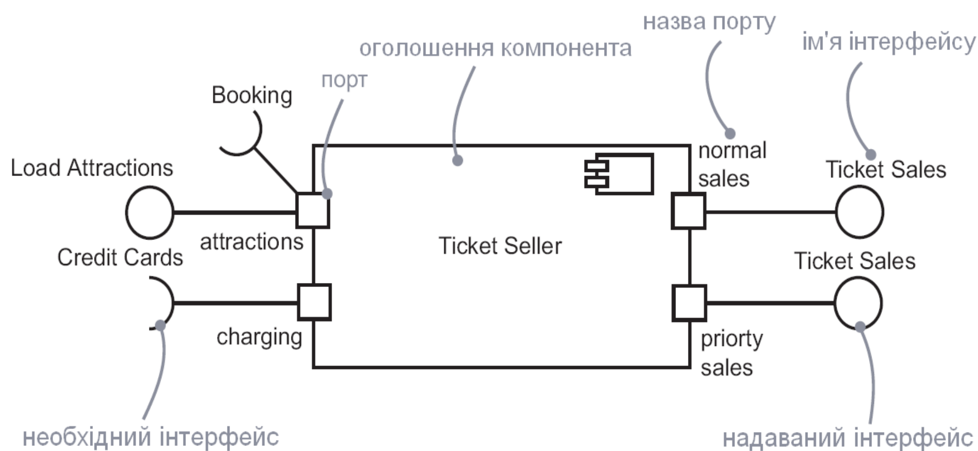


Рис. 2.3.8. Порти компоненти

Внутрішня структура. Компонента може бути реалізована як єдиний фрагмент коду, але для великих систем бажано мати можливість будувати великі компоненти з малих, які використовуються в якості будівельних блоків. Внутрішня структура компоненти містить частини, які, разом із з'єднаннями між ними, становлять його реалізацію. У багатьох випадках внутрішні частини можуть бути екземплярами дрібніших компонентів, статично зв'язаних через порти для забезпечення необхідної поведінки, без необхідності для автора моделі специфікувати додаткову логіку.

Частина — це одиниця реалізації компоненти, якій присвоєно ім'я і тип. В екземплярі компоненти утримується по одному або кілька екземплярів кожної частини певного типу. [2] Частина має множинність у межах компоненти. Якщо ця

множинність більше одиниці, то в екземплярі компоненти може бути ряд екземплярів компоненти даного типу. Якщо множинність представлена не одним цілим числом, то кількість екземплярів частини може варіюватися в різних екземплярах компоненти. Екземпляр компоненти створюється з мінімальною кількістю частин (інші за необхідності додаються пізніше). Атрибут класу – це різновид частини: він має тип і множинність. У кожного екземпляра класу є один або кілька екземплярів атрибуту даного типу.

На рис. 2.3.9 показано компоненту-компілятор, що складається з частин чотирьох видів. У їхньому числі – лексичний аналізатор, синтаксичний аналізатор (parser), генератор коду й від одного до трьох оптимізаторів. Повніша версія компілятора може бути сконфігурована з різними рівнями оптимізації. У даній версії необхідний оптимізатор може вибиратися під час виконання.

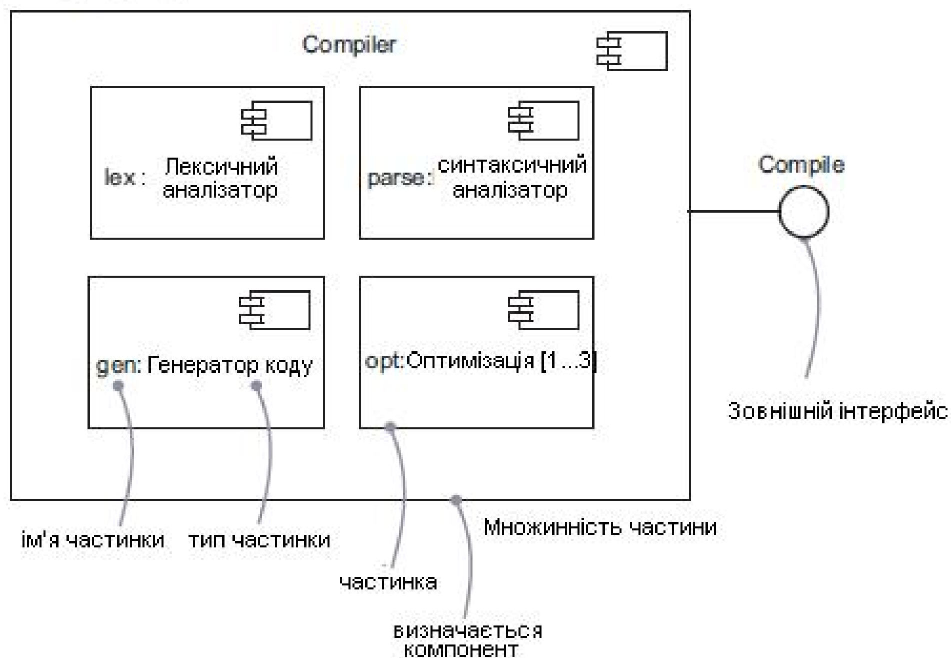


Рис. 2.3.9. Частини компоненти

Частина – це не те саме, що клас. Кожна частина ідентифікується за її іменем так само, як у класі різняться атрибути. Допустима присутність кількох частин того самого типу, що виконують різні функції усередині компоненти. Їх можна розрізняти за іменами. На рис. 2.3.10 компонента Airticket Sales (Продаж авіаквитків) може включати окремі частини Sales для постійних і звичайних клієнтів. Обидві вони працюють однаково, але перша обслуговує тільки привілейованих клієнтів, дає більше шансів уникнути черг і надає деякі пільги. Оскільки це компоненти однакового типу, їх потрібно буде розрізняти за іменами. Інші дві компоненти типів Seat Assignment (Вказання місць) і Inventorymanagement (Управління списками) не вимагають імен, оскільки присутні в одному екземплярі всередині компоненти Air Ticket Sales.

Якщо частини є компонентами з портами, то можна зв'язати їх одна з одною через ці порти. Два порти можуть бути під'єднана один до одного, якщо один з них надає даний інтерфейс, а інший вимагає його. Під'єднання портів означає, що для отримання сервісу порт, що потребує, викликає порт, що надає інтерфейс. Переваги портів та інтерфейсів у тому, що крім них більше нічого не важливо; якщо інтерфейси сумісні, то порти можуть бути під'єднані один до одного. Інструментальні засоби здатні

автоматично ґенерувати код виклику однієї компоненти від іншої. Також їх можна під'єднати до інших компонентів, що надають ті ж інтерфейси, коли такі з'являться і стануть доступні.

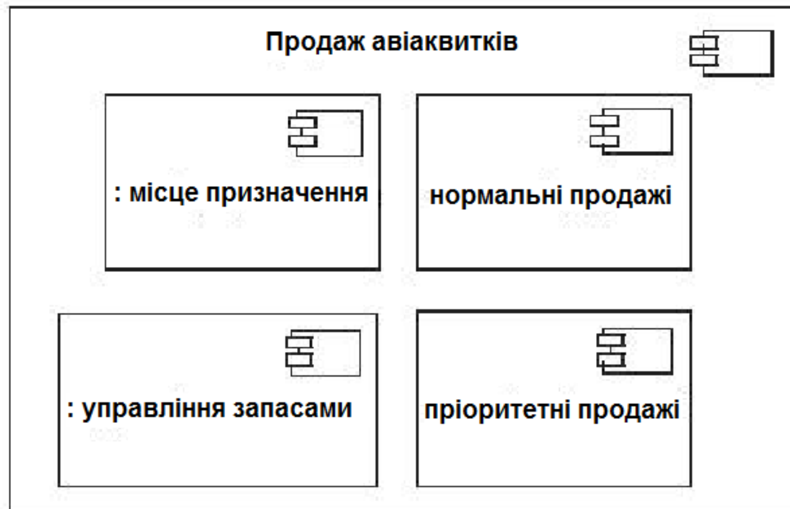


Рис. 2.3.10. Частини одного типу

Коннектором є «провід» між двома портами. В екземплярі компоненти, яку вона охоплює, він являє собою просте або тимчасове посилання.

Просте посилання (link) – це екземпляр звичайної асоціації. [2]

Тимчасове посилання (transient link) – являє собою зв'язок використання між двома компонентами. Замість звичайної асоціації вона може бути забезпечена параметром процедури або локальної змінної, яка служить метою операції. Перевага портів та інтерфейсів у тому, що ці дві компоненти не зобов'язані «*знати*» одна про одну на етапі проектування, поки їхні інтерфейси сумісні. [2,8]

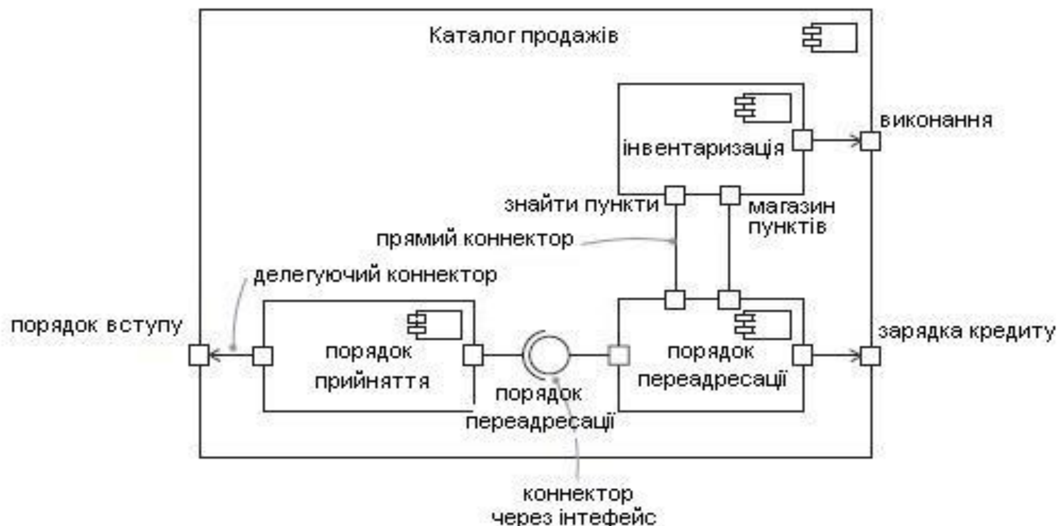


Рис. 2.3.11. Діаграма компонентів з використанням конекторів

Конектори зображуються двома способами (рис. 2.3.11). Якщо дві компоненти явно зв'язані між собою (*прямо або через порти*), досить провести лінію між ними або їх портами. Якщо ж дві компоненти під'єднані одна до одної тому що мають сумісні інтерфейси, то можна використовувати нотацію «*кулька-гніздо*», щоб показати, що між цими компонентами не існує постійного зв'язку, хоча вони з'єднані всередині утримуючої їх компоненти. В будь-який час можна підставити замість кожної з них будь-яку іншу компоненту, якщо вона задовольняє інтерфейс.

Також можна зв'язати внутрішні порти із зовнішніми портами компоненти. У такому випадку мова йде про **делегуючий конектор**, оскільки повідомлення із зовнішнього порту делегуються внутрішньому. Подібний зв'язок зображається стрілкою, спрямованою від внутрішнього порту до зовнішнього. Можна показати її подвійно. По-перше, можна вважати, що внутрішній порт – те ж саме, що й зовнішній; він винесений на межу і допускає під'єднання ззовні. По-друге, слід взяти до уваги, що будь-яке повідомлення, яке прийшло на зовнішній порт, негайно передається на внутрішній, і навпаки. Це неважливо – поведінка буде однаковою в кожному разі.

На рис. 2.3.11 показано використання внутрішніх портів і різних видів коннекторів. Зовнішні запити, що приходять на порт OrderEntry (Передача Замовлення), делегуються на внутрішній порт підкомпоненти Ordertaking (Прийом Замовлення). Ця компонента, у свою чергу, відправляє свій вивід на свій порт Orderhandoff (Переказ Замовлення). Останній під'єднаний за схемою «кулька-гніздо» до підкомпоненти Orderhandling (Управління Замовленнями). Даний вид підєднання припускає, що в компонентів не існує знань одна про одну; вивід може бути приєднаний до будь-якого іншого компонента, який відповідає інтерфейсу Orderhandoff. Компонент Orderhandling взаємодіє з компонентою Inventory (Опис) для пошуку елементів на складі. Ця взаємодія виражена прямим конвектором. Оскільки ніяких інтерфейсів не показано, можна припустити, що дане під'єднання щільніше, тобто забезпечує сильніший зв'язок. Як тільки елемент знайдений на складі, компонент Orderhandling звертається до зовнішнього сервісу Credit (Кредит) – про це свідчить, що делегує коннектор до зовнішнього порту changing (зміна).

Як тільки зовнішній сервіс Credit дає відповідь, компонент Orderhandling Orclerhanclling зв'язується з іншим портом ShipItems (Деталі доставки) компонента Inventory, щоб підготувати пересилання замовлення. Компонента Inventory звертається до зовнішнього сервісу Fullfillment (Виконання) для здійснення доставки.

Діаґрама компонентів показує структуру і можливі способи доставки повідомлень компонента. Послідовність повідомлень у компоненті вона, однак, не відбражає. Послідовності та інші види динамічної інформації можуть бути представлені на діаґрамі взаємодії.

Моделювання структурованих класів. Структурований клас може бути використаний для моделювання структур даних, у яких частини мають контекстно-залежні з'єднання, що існують тільки в межах класу. Звичайні атрибути або асоціації можуть визначати складові класу, але частини не можуть бути зв'язані одна з одною на простій діаґрамі класів. Клас, внутрішня структура якого показана за допомогою частин і коннекторів, дозволяє уникнути цієї проблеми.

Щоб змоделювати структурований клас, необхідно: ідентифікувати внутрішні частини класу і їх типи; дати кожній частині ім'я, що відображає її призначення в структурованому класі, а не її тип; зобразити коннектори між частинами, які забезпечують комунікацію або мають контекстно-залежні зв'язки; використовувати за

необхідності інші структуровані класи як типи частин, але враховуючи, що під'єднання до частин не можна здійснювати всередині іншого структурованого класу – можна під'єднуватися тільки до їхніх зовнішніх портів. На рис. 3.7.6 продемонстровано дизайн структурованого класу Ticketorder (Замовлення білетів). Цей клас має чотири частини й один звичайний атрибут price (ціна). Далі, customer (покупець) – це об'єкт Person (Людина); він може мати або не мати статус priority (пріоритет), тому частина priority показана із множинністю 0..1. Коннектор від customer до priority має ту ж множинність. Оскільки кожен клієнт має право бронювати одне або кілька посадкових місць, в об'єкта seat (місце) теж є значення множинності. Немає необхідності показувати коннектор від customer до seats, тому що ці об'єкти й так перебувають в одному структурованому класі. Відзначимо, що клас Attraction (Категорія) обведений пунктирною рамкою. Це означає, що дана частина являє собою посилання на об'єкт, яким не володіє структурований клас. Посилання створюється й знищується разом з екземпляром класу Ticketorder, але екземпляри Attraction незалежні від класу Ticketorder. Частина seat під'єднана до посилання attraction, оскільки замовлення може включати посадкові місця різних категорій і кожне бронювання квитка повинно бути зіставлене з певною категорією посадкового місця. За значенням множинності бачимо, що кожне резервування Seat під'єднане строго до одного об'єкта Attraction.

Моделювання програмного інтерфейсу API. При розробленні системи, що складається із частин-компонентів, часто потрібно бачити інтерфейси прикладного програмування (application programming interfaces, API), за допомогою яких ці частини зв'язуються один з одним. API представляють програмні з'єднання в системі, які можна змоделювати, використовуючи інтерфейси і компоненти. API це інтерфейс, який реалізується одним або кількома компонентами. Важливим для розробника є які саме компоненти реалізують операції інтерфейсу. Однак з погляду керування конфігурацією системи, ці реалізації, асоційовані з будь-яким семантично насиченим API, будуть зустрічатися досить часто, тому в більшості випадків не має потреби візуалізовувати всіх їх відразу. Потрібно прагнути до того, щоб залишати операції на периферії розроблюваних моделей і використовувати інтерфейси як дескриптори, за допомогою яких можна буде знайти всі ці набори операцій. При створенні реальних систем на основі таких API, потрібно деталізувати моделі настільки, щоб інструменти розроблення були здатні робити компіляцію у відповідності з властивостями інтерфейсів. Поряд із сигнатурою кожної операції може виникнути необхідність відобразити ВВ, що пояснюють, як потрібно застосовувати кожний інтерфейс. Щоб змоделювати API, необхідно: ідентифікувати з'єднання в системі і змоделювати кожне з них у вигляді інтерфейсу, збираючи разом атрибути й операції, що його утворюють, показати тільки ті властивості інтерфейсу, які важливі для візуалізації в даному контексті. Я якщо ні, то сховати їх, зберігаючи тільки в специфікації інтерфейсу для наступного посилання. На рис. 2.3.12 представлений API компоненти анімації. Тут чотири інтерфейси, що становлять API:Iapplication.

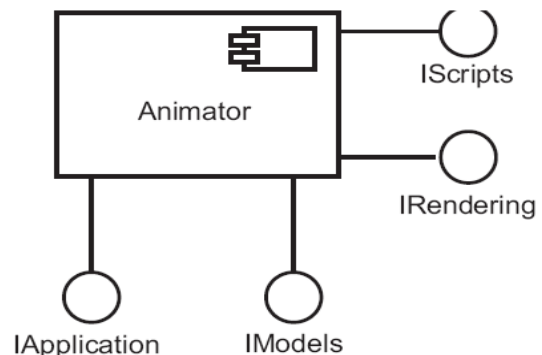


Рис. 2.3.12. Моделювання API

Компоненти дозволяють інкапсулювати частини системи, щоб зменшити кількість залежностей, зробити їх явними, а також підвищити взаємозамінність і гнучкість на

випадок, якщо система повинна буде змінюватися в майбутньому. Добра компонента наділена такими характеристиками:

- інкапсулює сервіс із добре окресленим інтерфейсом і межами;
- має внутрішню структуру, яка допускає можливість її описування;
- не комбінує незв'язаної функціональності в межах однієї одиниці;
- організовує зовнішню поведінку, використовуючи інтерфейси й порти в невеликій кількості;
- взаємодіє тільки через оголошені порти.

Якщо необхідно показати реалізацію компоненти, використовуючи вкладені підкомпоненти, слід не зловживати використанням підкомпонентів. Якщо їх надто багато, щоб вони легко вмістилися на одній сторінці, застосовувати додаткові рівні декомпозиції до деяких із них. Переконайтеся, що підкомпоненти взаємодіють тільки через певні порти і коннектори; визначте, які підкомпоненти безпосередньо взаємодіють із зовнішнім світом, і моделюйте їх делегуючими коннекторами.

При зображенні компонент у UML необхідно: дати їй ім'я, що зрозуміло описує її призначення, у такий же спосіб іменувати інтерфейси; присвоювати імена підкомпонентам і портам у випадку, якщо їх значення не можна визначити за їхніми типами або ж у моделі присутні багато частин одного типу; приховувати зайві деталі реалізації на діаґрамі компонентів; показувати динаміку компонентів, використовуючи діаґрами взаємодії.

3. Основи моделювання поведінки ПЗ

3.1. Моделювання варіантів використання

3.1.1. Стратегія розроблення варіантів використання як кооперації класів. Поділ специфікації поведінки й реалізації

Основні питання:

- *Варіанти використання, дійові особи, включення й розширення*
- *Моделювання поведінки елемента*
- *Реалізація варіантів використання за допомогою кооперацій*

Об'єктно-орієнтована система не існує в ізоляції, а взаємодіє з дійовими особами (актантами: людьми, системами), які використовують її для досягнення деякої мети, очікуючи від неї певної поведінки. Варіанти використання (ВВ) специфікують цю очікувану поведінку системи, описують послідовності дій, які вона здійснює для досягнення дійовою особою певного результату. ВВ застосовуються для вираження необхідної поведінки розроблювальної системи, *без описування реалізації* цієї поведінки. Вони дозволяють розробникам, кінцевим користувачам і експертам у предметній області досягти консенсусу, допомагають впевнитися в правильності архітектурних рішень і перевірити систему в ході її розроблення. У процесі створення системи ВВ реалізуються за допомогою кооперацій, елементи яких працюють спільно для досягнення цілей кожного з них. Добре структуровані ВВ описують тільки істотні аспекти поведінки і не є ні надто узагальненими, ні надто деталізованими.

Правильно спроектований будинок — це більше, ніж стіни, що підпирають дах, який захищає мешканців від дощу. Працюючи разом з архітектором над проектом, продумують план використання приміщень: для вітальні, щоб приймати гостей і зручно спілкуватися; кухні – для приготування їжі, розміщення меблів, побутової техніки…. Навіть маршрут транспортування продуктів з машини на кухню вплине на розташування кімнат. Є потреба у

визначенні оптимальної кількості ванних кімнат і їх розміщення, що дозволило б уникнути «черг», коли всі одночасно збираються на роботу, до школи, університету… Це приклад аналізу ВВ. *Потрібно розглядати різні ВВ будинку, які в підсумку зумовлюють його архітектуру. Для багатьох родин ВВ є подібними: в усіх будинках їдять, сплять, виховують дітей, обговорюють витвори провінційних брендів за бокалом шамбертен. Але в кожному випадку висуваються індивідуальні вимоги до житла. Потреби великої родини відрізняються від вимог самотнього холостяка…, що вплине на майбутній вигляд будинку.*

Спочатку найістотніші проблеми – подробиці далі. Найважливіша особливість розроблення ВВ полягає в тому, що при цьому *не специфікується* конкретний спосіб їх реалізації (*поведінку банкомата можна описати за допомогою варіантів взаємодії з ним користувачів, але не обов'язково знати, що в ньому всередині*). ВВ специфікують **зовнішню поведінку**, нічого не кажучи про те, як цього досягти, що дозволяє як експертові або кінцевому користувачеві спілкуватися з розробниками ПЗ відповідно до вимог, **не заглиблюючись у деталі реалізації**. Поведінка системи в UML моделюється за допомогою ВВ, специфікованих незалежно від реалізації.

Варіант використання (use case) – це опис багатьох послідовних дій (*включаючи варіації*), які виконуються системою з метою отримання результату, важливого для деякої дійової особи. [12]

Важливі позиції змісту ВВ:

1). На системному рівні ВВ описує *набір послідовностей*, кожна з яких являє собою взаємодію сутностей, що перебувають поза системою (*дійових осіб - актантів*), із самою системою та її ключовими абстракціями. Такі взаємодії є функціями системи, якими користуються для ВСКД її очікуваної поведінки на етапах збирання та аналізу вимог до системи в цілому. *Один з основних ВВ у роботі банку є опрацювання позик.*

2). **Дійова особа** являє собою логічно зв'язану **множину ролей**, які виконують користувачі системи під час взаємодії з нею. Дійовими особами можуть бути як люди, так і автоматизовані системи. *Процес опрацювання позик містить взаємодію клієнта зі співробітником кредитного відділу.*

3). ВВ можуть мати різновиди. У будь-якій системі існують ВВ, які або є спеціалізованими версіями інших, або входять до складу інших ВВ, або розширюють їхню поведінку. Можна виділити загальну, повторно застосовувану поведінку з усієї множини ВВ, організувати їх відповідно до цих трьох видів зв'язків. *Базовий ВВ опрацювання позик має кілька різновидів — від оформлення застави до видавання маленької позики. Всі ці ВВ мають щось загальне в поведінці (оцінювання платоспроможності клієнта).*

4). Кожен ВВ повинен виконувати деякий обсяг роботи, що з погляду дійової особи він робить щось, що має для неї певну цінність (*обчислює результат, створює новий об'єкт, змінює стан іншого*). *Процес опрацювання заявки на позику призводить до підписання видаткового документа й матеріалізується у вигляді деякої суми грошей для клієнта.*

Рис. 3.1.1. Дійові особи та варіанти використання

UML-нотація ВВ зображується у вигляді *еліпса*, а дійових осіб – «*чоловічками*», що дозволяє візуалізувати певний ВВ у контексті інших і окремо від його реалізації (рис. 3.1.1).

Суб'єкт – це клас (*мегаклас*), описаний повним набором ВВ, що представляє систему або підсистему. ВВ представляють аспекти поведінки цього класу. Дійові особи представляють аспекти інших класів, що взаємодіють з суб'єктом. [8.12] Взяті разом, ВВ описують повну поведінку суб'єкта. Кожен ВВ повинен мати ім'я (*текстовий рядок*), що відрізняє його від інших. **Ім'я ВВ** може бути як *простим,* так і *кваліфікованим*, до якого додається префікс – ім'я пакета, у якому перебуває ВВ. При зображенні ВВ вказується тільки його ім'я (рис. 3.1.2).

Рис. 3.1.2. Просте і кваліфіковане ім'я ВВ

ВВ і дійові особи (актанти). Дійова особа є зв'язаною множиною ролей, які виконують користувачі ВВ під час взаємодії з ними. Дійова особа це та роль, яку в даній системі відіграє людина, апаратний пристрій чи інша система. *Працюючи в банку, можна виконувати роль* LoanOfficer *(Співробітник кредитного відділу).*

Екземпляр дійової особи є конкретною особою, що певним чином взаємодіє з системою і являє собою контекст, що існує поза нею. У працюючій системі дійові особи не зобов'язані бути присутніми як окремі сутності. Один об'єкт не обмежується однією роллю дійової особи. Одна і та ж особа (Person) може бути і співробітником кредитного відділу (LoanOfficer), і клієнтом (Customer). Можна вводити загальні типи дійових осіб, такі, як Customer, і специфікувати їх (CommercialCustomer — *комерційний клієнт*), визначивши зв'язки узагальнення (рис. 3.1.3).

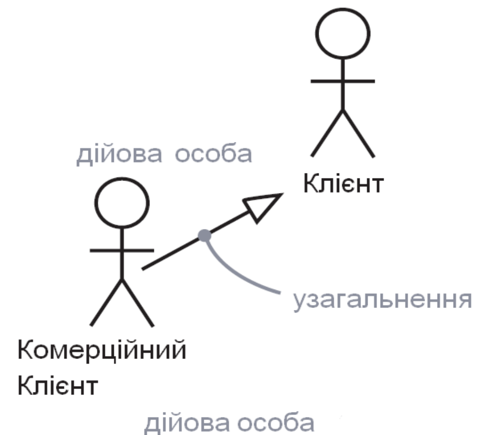


Рис. 3.1.3 Дійові особи-актанти

Дійові особи можна зв'язувати з ВВ за допомогою асоціацій. Асоціація між діючою особою і ВВ показує, що вони спілкуються між собою, можливо, посилаючи або приймаючи повідомлення.

Опис ВВ як потоку подій. ВВ описує, що робить система (*підсистема, клас, або інтерфейс*), але не вказує, як вона це робить. У процесі моделювання завжди важливо розділяти зовнішнє і внутрішнє представлення. Необхідно специфікувати поведінку ВВ, описавши потік подій у *доступно зрозумілій текстовій формі*. В описі повинна бути присутня вказівка на те, як і коли ВВ починається і закінчується, коли він взаємодіє з дійовими особами, якими об'єктами вони обмінюються, а також описами основного й альтернативних (*виняткових*) потоків поведінки.

Приклад опису ВВ у контексті системи банкомата: ВВ ValidateUser (Перевірка Користувача):

- **основний потік подій***:* ВВ починається, коли система запитує в клієнта його персональний ідентифікаційний номер (PIN). Клієнт (Customer) вводить його з клавіатури. Завершується введення натисканням клавіші Enter. Потім система перевіряє введений PIN і, якщо він правильний, підтверджує введення. Цим ВВ закінчується;
- **винятковий потік подій 1***:* клієнт може скасувати транзакцію в будь-який момент, натиснувши клавішу Cancel. Ця дія запускає ВВ заново. Ніяких змін з рахунком клієнта не проводиться;
- **винятковий потік подій 2***:* клієнт може в будь-який момент до натискання клавіші Enter стерти свій PIN і ввести новий;
- **винятковий потік подій 3***:* якщо клієнт вводить неправильний PIN, ВВ перезапускається. Якщо це відбувається три рази підряд, система скасовує всю транзакцію і не дозволяє працювати клієнтові з банкоматом впродовж 60 с.

ВВ і сценарії (*екземпляри***) та діаґрами взаємодій.** На початку роботи потоки подій ВВ описуються в текстовій формі. В міру уточнення вимог до системи для зображення потоків подій знадобляться діаґрами взаємодій, що дозволяють показати їх графічно. Для цього застосовуються **діаґрами послідовності** (*стандартна – для описування основного потоку ВВ, а її варіації – для виняткових потоків*).

Основний потік повідомлень є відокремлений від альтернативних, оскільки ВВ описує не одну, а багато послідовностей, і виразити всі необхідні деталі ВВ у вигляді однієї послідовності неможливо. В системі керування персоналом присутній ВВ HireEmployee (Найняти працівника). Існує багато різновидів цієї основної бізнес-функції. Можна переманити працівника з іншої компанії (*найзагальніший сценарій*), перевести співробітника з одного підрозділу в інший (*типовий сценарій для компанії*) або найняти іноземця (*особливий випадок, регульований спеціальними правилами*). Кожен із цих ВВ описується своєю послідовністю. HireEmployee описує набір послідовностей, кожна з яких представляє один потік із усіх можливих варіацій. Таку послідовність називають *сценарієм*.

Сценарій (scenario) **як екземпляр ВВ** – це конкретна послідовність дій, що ілюструє поведінку. [2] Сценарії є екземплярами ВВ (*аналогія з класами*). Сценарій – це в основному один екземпляр ВВ (*на відміну від класу*).

ВВ і коопeрції. ВВ визначає поведінку створюваної системи (*підсистеми, класу чи інтерфейсу*), не вказуючи того, як ця поведінка реалізована. Поділ специфікації

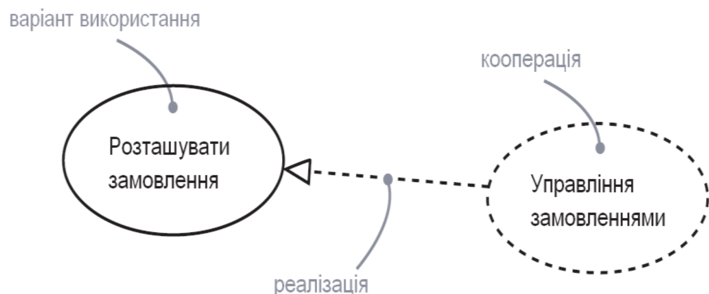


Рис. 3.1.4. Варіанти використання і кооперації

поведінки і її реалізації забезпечує незалежність проектування специфікації поведінки від рішень її реалізації.

Кооперація класів. Реалізація ВВ здійснюється шляхом створення співтовариств класів та інших елементів, що працюють разом для реалізації поведінки, описаної ВВ. Таке співтовариство елементів, що володіє як статичною, так і динамічною структурою, моделюється в UML як **кооперація**.

Реалізацію ВВ можна специфікувати явно через кооперацію (*пунктирний овал).* Зв'язок кооперації з ВВ визначається залежністю **реалізації** (рис. 3.1.4).

Від ВВ до варіантів тестування. Можна застосовувати ВВ до всієї системи або до її частин (*підсистем, класів та інтерфейсів*). ВВ не тільки описують бажану поведінку цих елементів, а є основою **сценаріїв тестування** на різних етапах розроблення. У застосуванні до підсистем – це ефективне джерело реґресійних тестів, а до системи в цілому – комплексних і системних тестів.

Організація ВВ. Для організації ВВ їх ґрупують у пакети так само, як класи. Крім того, можна організувати ВВ, визначивши між ними зв’язки узагальнення, включення й розширення. Ці зв’язки застосовуються для того, щоб виділити деяку загальну поведінку (*вилучивши її з інших ВВ*), а також різновиди (*вміщуючи таку поведінку в інші ВВ, які розширюють даний*).

Узагальнення між ВВ (*батьківські й дочірні ВВ***)** є подібні узагальненням між класами. Дочірній ВВ *успадковує поведінку* і *суть* батьківського ВВ; нащадок може додати або перевизначити поведінку батьківського ВВ, бути підставленим замість нього в будь-якому місці, де той з’являється (*як батьківський, так і дочірній ВВ можуть мати конкретні екземпляри*). У банківській системі може існувати ВВ ValidateUser (Перевірити користувача), який відповідає за ідентифікацію клієнта. У нього можуть бути два спеціалізовані дочірні ВВ – CheckPassword (Перевірка пароля) і RetinalScan (Сканування сітківки), кожен з яких поводиться так само, як ValidateUser і може бути застосований у будь-якому місці, де з’являється останній. Обидва нащадки додають свою власну поведінку: перший перевіряє текстовий пароль, а другий – унікальний рисунок сітківки ока користувача.

Нотацією узагальнення між ВВ є суцільна лінія з трикутною стрілкою (рис. 3.1.5).

Зв’язок включення між ВВ означає, що базовий ВВ у певному місці явно містить у собі поведінку іншого. Включений ВВ не існує окремо: він *є екземпляром тільки всередині базового ВВ*, який його містить. [1,2] Базовий ВВ запозичає поведінку ВВ, що включається. Завдяки наявності зв’язків включення вдається уникнути багаторазового описування потоку подій, оскільки загальну поведінку можна показати у вигляді окремого ВВ, що включається в базовий. Зв’язок включення є прикладом делегування, коли ряд обов’язків системи описується в одному місці (*у ВВ, що включається*), а інші ВВ за необхідності вводять ці обов’язки у свій набір.

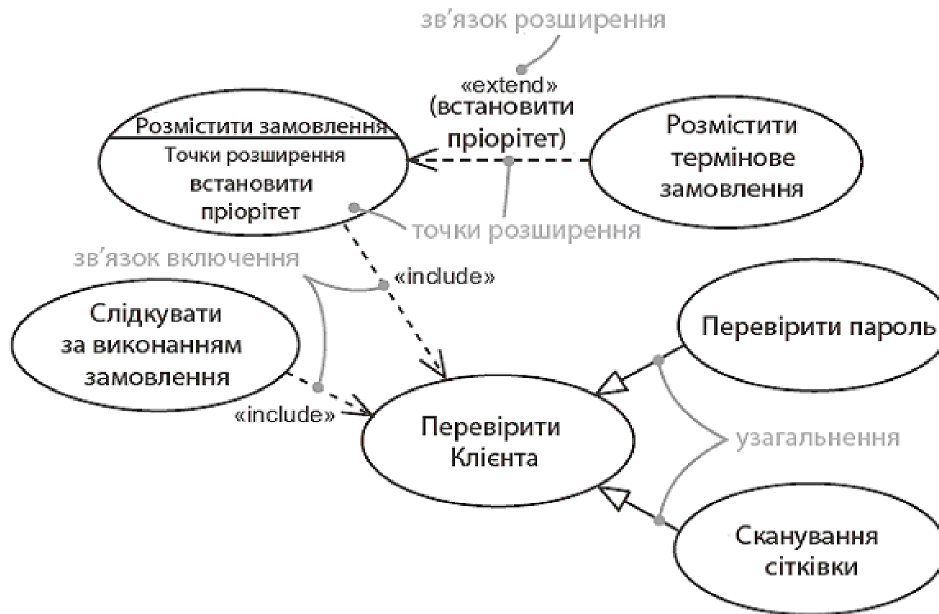


Рис. 3.1.5. Узагальнення, включення, розширення

Зв'язок включення зображується як залежність зі стереотипом «include» (рис. 3.1.5). Щоб показати місце в текстовому описі *потоку подій* базового ВВ (*одна із неформальних текстових нотацій описування ВВ*), де включається поведінка іншого ВВ, слід написати ключове слово include з наступним іменем ВВ використання, як це зроблено у потоці Track order (Відстежити замовлення):

Track order:
Отримати й перевірити номер замовлення;
include "ValidateUser";
для кожної частини замовлення
запросити стан;
повідомити загальний підсумковий стан користувачеві.

Зв'язок розширення між ВВ означає, що базовий ВВ неявно включає поведінку деякого іншого в побічно зазначеному місці. Базовий ВВ здатний існувати окремо, але за деяких умов його поведінка може бути розширена поведінкою іншого ВВ. Базовий ВВ можна розширити тільки викликом із певної точки – **точки розширення** (extension point). [1,2] Розширюючий ВВ «*виштовхує*» поведінку у базовий. Зв'язок розширення використовується при моделюванні тих частин ВВ, які користувач бачить як необов'язкові. У такий спосіб *обов'язкова поведінка* відділяється *від необов'язкової (виділення частин системи з різними розширеннями).*

Зв'язок розширення зображується як **залежність** зі стереотипом «extend». У додатковій секції можна перелічити точки розширення базового ВВ. Ці точки розширення – **прості позначки**, які можуть з'являтися в потоці подій базового ВВ, як у потоці подій PlaceOrder (Розмістити замовлення):

Place order:
include ValidateUser;
зібрати компоненти користувацького замовлення;
встановити пріоритет: точка розширення;

підтвердити замовлення для опрацювання.

Тут позначка встановити пріоритет є *точкою розширення*. ВВ може мати кілька точок розширення, кожна з яких може зустрічатися неодноразово з обов’язковим вказуванням їх імені. У звичайних умовах цей базовий ВВ буде виконуватися без урахування пріоритетності замовлення. Якщо це буде екземпляр ВВ **пріоритетного замовлення**, то потік іде, як описано вище, але в точці розширення встановити пріоритет буде вставлено розширюючий ВВ PlaceRushOrder (Розмістити термінове замовлення), після якого потік продовжує виконання. Якщо є багато точок розширення, то в кожну просто вставляється розширюючий ВВ.

ВВ є класифікаторами, що можуть мати атрибути й операції, які можна зображувати так само, як у класах. Можна показати атрибути у вигляді об’єктів, що перебувають всередині ВВ і необхідних для описування його зовнішньої поведінки, а операції — як дії системи, які потрібні для описування потоку подій. Ці об’єкти й операції можуть використовуватися в діаґрамах взаємодії для специфікації поведінки ВВ. Як класифікатори, ВВ допускають приєднання до них автоматів. Можна застосовувати автомати як додатковий засіб описування поведінки, отриманої ВВ. [4]

ВВ у моделюванні поведінки системи. Важливим і найзагальнішим випадком застосування ВВ є моделювання поведінки елемента (*системи, підсистеми, класу*), коли необхідно зосередити увагу на тому, **що елемент робить**, а не на тому, **як він це робить**. Це дозволяє, *по-перше*, експертам у предметній області специфікувати зовнішнє представлення елемента, достатнє для конструювання розробникам його внутрішнього представлення та забезпечення можливості спілкування експертів з користувачами й розробниками. *По-друге*, розробники отримують можливість виробити певний підхід до елемента (*як складної системи*) і зрозуміти його відповідно до їх застосування. *По-третє*, ВВ служать базою для тестування кожного елемента в процесі його розроблення, перевіряючи працездатність реалізації. Додаючи новий ВВ елемента, виникає потреба в контролі й перегляді реалізації, щоб ґарантувати достатню його гнучкість і стійкість до змін (*якщо не так, доведеться вносити зміни в архітектуру*).

Для моделювання поведінки елемента необхідно: ідентифікувати актанти, що взаємодіють з елементом (*кандидати на включення в цю ґрупу є ті, хто потребує певної поведінки елемента для виконання своїх завдань, або ті, хто залучений у функціонуванні елемента*); організувати актанти, визначивши загальні й більш спеціалізовані ролі; розглянути основні та винятковi шляхи взаємодії (поведінки) кожнго актанта з елементом та самі взаємодії, що змінюють стан елемента або його оточення чи забезпечують реакцію на якусь подію; організувати виявлену поведінку у вигляді ВВ, застосовуючи зв’язки включення й розширення, щоб виділити загальну поведінку й відокремити виняткову.

Рис. 3.1.6. Моделювання поведінки елемента

Приклад моделі поведінки системи. Система роздрібної торгівлі взаємодіє із замовниками, які розміщують замовлення і відслідковують їхнє виконання. Система, у свою чергу, відвантажує замовлені товари та виставляє клієнтам рахунки до оплати (рис. 3.1.6). Моделювання поведінки такої системи здійснюється шляхом оголошення кількох ВВ: PlaceOrder (Розмістити замовлення), TrackOrder (Відстежити замовлення), ShipOrder (Відвантажити Замовлення) і BillCustomer (ВиставитиРахунок). Можна виділити загальну поведінку ValidateCustomer (ПеревіритиКлієнта) і ВВ типу ShipPartialOrder (ВідвантажитиЧастковоВиконане Замовлення). Для кожного із цих ВВ передбачено специфікацію поведінки, виражену текстом, автоматом або взаємодією.

У міру розвитку моделі виявиться необхідність об'єднання ВВ у концептуально й семантично близькі ґрупи. Такі ґрупи в UML можна моделювати у вигляді пакетів. При моделюванні ВВ кожен з них повинен описувати деяку окрему ідентифіковану поведінку системи (частини). Добре структурований ВВ іменує просту, ідентифіковану поведінку системи; виділяє загальну поведінку, вилучаючи його з інших ВВ; виділяє варіації, вміщуючи деяку поведінку в інші ВВ, що розширюють його; описує потік подій у зрозумілій формі; описується мінімальним набором сценаріїв, що специфікують його основну і варіативну семантику. Слід показувати тільки ВВ, важливі для розуміння поведінки системи в даному контексті та актанти, пов'язані з цими ВВ.

3.1.2. Діаґрами варіантів використання

Основні питання:

- *Моделювання контексту системи*
- *Моделювання вимог до системи*
- *Пряме й зворотне проектування*

Діаґрами ВВ є одним з видів діаґрам UML для моделювання динамічних аспектів систем (*інші чотири — це діаґрами діяльності, станів, послідовності й комунікації*).

Діаґрами ВВ є основним видом діаґрам при моделюванні поведінки системи. Кожна з них показує набір ВВ і актантів у їхній взаємодії (*їх зв'язки*).

Діаґрами ВВ важливі для ВСКД поведінки елемента, забезпечуючи доступність і зрозумілість системи за рахунок зовнішнього представлення щодо їх використаня в контексті. Вони важливі для тестування працюючих систем за допомогою прямого і зворотного проектування. *Маленький технічний пристрій без будь-якого опису про його використання включає невеликий корпус, з одного боку якого розташовані якісь кнопки і рідкокристалічна панель. Можна у будь-якому порядку натискати кнопки і дивитися, що станеться, але для цього потрібно витратити багато часу на метод проб і помилок, а чи досягнеться результ – невідомо.*

Те ж саме із ПЗ. Користувачеві, що приступає до роботи з новим додатком, потрібні чіткі інструкції. Якщо додаток дотримується стандартних угод, прийнятних для використовуваної операційної системи, набагато простіше буде освоїти інтерфейс. Однак тільки на підставі цього навряд чи користувач зрозуміє більш тонкі й складні нюанси поведінки програми. Аналогічна ситуація буде, якщо розробникові передали додаток, який робив інший розробник і попросити в ньому розібратися. Виконати це завдання, поки не буде сформована концептуальна модель його застосування, буде практично неможливо.

В UML діаґрами ВВ застосовуються, щоб візуалізувати поведінку системи, аби користувач міг зрозуміти, як застосовувати цей елемент, а розробник – як реалізувати його. Діаґрама ВВ допомагає змоделювати поведінку того самого «*загадкового*» пристрою із кнопками і дисплеєм, у якому більшість людей без проблем впізнає мобільний телефон (рис. 3.1.7).

Зміст діаґрами ВВ. Діаґрама ВВ має ім'я й графічне наповнення, що є певною проекцією моделі ПЗ. Діаґрами ВВ містять суб'єкт, дійові особи, ВВ, а також зв'язки залежності, узагальнення й асоціації. Можуть містити примітки і обмеження, а також включати пакети (*для об'єднання елементів моделі в більші ґрупи*), екземпляри ВВ.

Суб'єкт зображується у вигляді прямокутника, що **містить набір еліпсів** (ВВ). Ім'я суб'єкта зазначене всередині прямокутника. Актанти (дійові особи) представлені фігурками поруч із прямокутником. Імена актантів розташовуються під фігурками. Останні з'єднуються лініями з еліпсами ВВ, з якими вони взаємодіють. Зв'язки між ВВ, такі, як розширення і включення, зображуються всередині прямокутника.

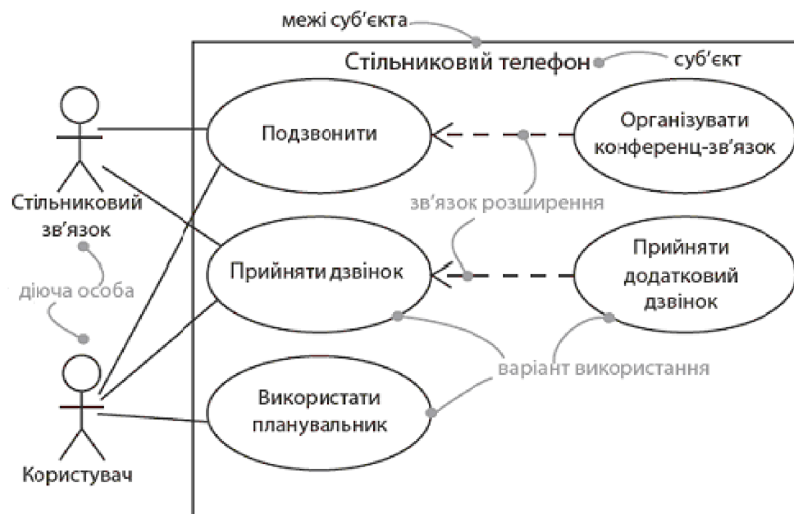


Рис. 3.1.7. Діаґрама варіантів використання

Стандартне використання. Діаґрами ВВ застосовуються для **моделювання вигляду суб'єкта** (системи) з **погляду ВВ**. Цей вигляд моделює **зовнішню поведінку** суб'єкта: **видимі ззовні послуги**, які суб'єкт надає в контексті його оточення. Під цим розуміється моделювання контексту системи або **моделювання вимог**. [1,2,11,12]

Діаґрами ВВ у задачах проектування. Моделюючи вигляд суб'єкта з погляду ВВ, діаґрами ВВ застосовуються для таких завдань проектування:

- **моделювання контексту суб'єкта,** шо має на меті окреслення границь навколо всієї системи і визначення актантів, які перебувають поза її межами і ввзаємодіють з нею. Діаґрами потрібні для специфікації актантів і визначення їх ролей;
- **моделювання вимог до суб'єкта,** що має на меті специфікацію того, що він повинен робити (*з зовнішнього погляду стосовно суб'єкта*), незалежно від того, як він повинен це робити. Діаґрами ВВ застосовуються тут для специфікації необхідної поведінки суб'єкта (системи). У концептуальному змісті діаґрама ВВ дозволяє показати суб'єкт як «*чорний ящик*»: можна бачити, що відбувається поза ним і як він реагує на зовнішні впливи, але не видно, як він працює всередині.

Моделювання контексту системи (context)**.** Внутрішні елементи системи відповідають за її очікувану *поведінку*. Все, що поза системою і взаємодіє з нею, становить її **контекст**, визначає середовище, у якому живе система. У системі перевірки кредитних карт є такі сутності, як рахунок, транзакції й аґенти запобігання шахрайств, а за її межами – власники кредитних карт і системи роздрібної торгівлі, що передбачають оплату по картах.

В UML можна моделювати контекст системи за допомогою діаґрами ВВ, що зображує оточуючі її актанти. Рішення про те, що включити (*не включати*) в діаґраму в якості актантів, є важливим, оскільки специфікується клас сутностей, що взаємодіють з системою (*обмежує навколишнє середовище системи для включення дійових осіб, необхідних для її життя*).

Для моделювання контексту системи необхідно: ідентифікувати границі системи, прийнявши рішення, яка поведінка є її частиною, а яку здійснюють зовнішні сутності (*цим визначається суб'єкт*); ідентифікувати актанти, визначивши ґрупи, що вимагають допомоги з боку системи у виконанні своїх завдань для забезпечення функціонування системи, які взаємодіють із зовнішніми системами і які здійснюють вторинні функції (*адміністрування, підтримка*); організувати ієрахію актантів (*шляхом узагальнення-спеціалізації*, *використання стереотипів*); наповнити діаґраму цими актантами й описати шляхи взаємодії кожного з ВВ системи.

На рис. 3.1.8 показано контекст системи перевірки кредитних карт (Credit Card Validation System) з описом дійових осіб, що оточують її. Тут клієнти (Customers) поділяються на дві ґрупи: IndividualCustomer (Приватний клієнт) і CorporateCustomer (Корпоративний клієнт). Ці дійові особи представляють ролі, які виконують люди при взаємодії з системою. У даному контексті показані також актанти, що представляють інші установи: Retail Institution (Підприємство роздрібної торгівлі), з яким Customer взаємодіє за допомогою карткової транзакції, здобуваючи товар або послугу, і Sponsoring Financial Institution (Інститут фінансового спонсорування) - депозитний центр для карткових рахунків. Останні два будуть представлені програмними системами. Моделювання контексту підсистем є також корисним, коли розробляється система з «включеннями».

Моделювання вимог до програмної системи. Вимога визначає функцію, властивість або поведінку системи. Формулюючи вимоги до системи, встановлюється

контракт між нею і тими сутностями, які перебувають поза нею. Контракт декларує те, що система повинна робити. Важливішим є не як система це буде робити, а щоб вона **відповідала заявленим вимогам** (*для конкретного користувача*). [1,12]

Діаґрами ВВ керують вимогами. Вимоги можуть бути виражені в різних формах – від неструктурованого тексту до виразів формальної мови. Більшість функціональних вимог до системи (*якщо не всі*), можуть бути виражені у вигляді ВВ.

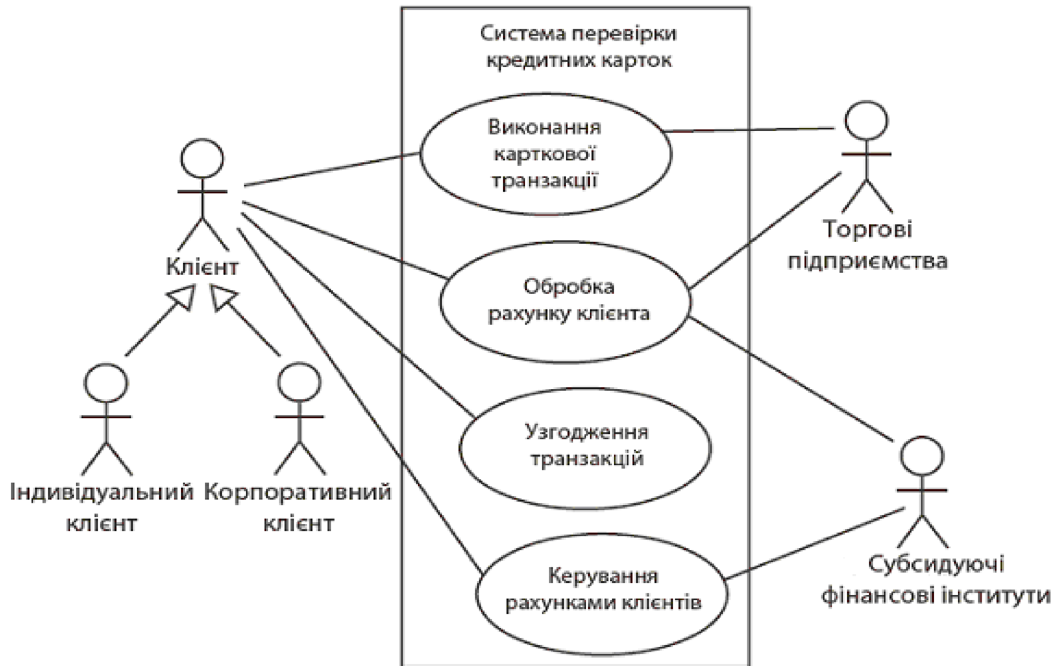


Рис. 3.1.8. Моделювання контексту системи

Для моделювання вимог до системи необхідно: встановити контекст системи, ідентифікуючи актанти; розглянути поведінку системи, яку від неї очікує/вимагає кожен актант; узагальнити цю поведінку у вигляді ВВ; виділити загальну поведінку в нові ВВ, на які посилаються інші; виділити різновиди поведінки в нові ВВ, що розширюють основні потоки; змоделювати ці ВВ, актанти і їх зв’язки на діаґрамі ВВ; доповнити ВВ примітками, обмеженнями, що задають нефункціональні вимоги..

Рис. 3.1.9. Моделювання вимог до системи

Рис. 3.1.9 доповнює діаґраму ВВ, показану вище. Тут приховані зв'язки між актантами і ВВ, але вводяться додаткові ВВ, не видимі звичайному користувачеві, хоча представляють важливі аспекти поведінки системи.

Діаґрама є корисною для користувачів, експертів і розробників для ВСКД їх спільних рішень щодо функціональних вимог до системи. Наприклад, ВВ Detect card fraud (Виявлення шахрайств) є важливою поведінкою як для підприємств роздрібної торгівлі, так і для інститутів фінансового спонсорування. Report on Account Status (Звіт про стан рахунку) – ще одна поведінка, необхідна від системи різними установами в її контексті. Вимога, яка моделюється ВВ ManageNetworkOutage (Керування мережними потоками), є дещо відмінною від усіх інших, оскільки описує вторинну поведінку системи, необхідну для забезпечення її надійної безперервної роботи.

Визначивши структуру ВВ, потрібно описати поведінку кожного з них. Слід скласти одну або кілька *діаґрам послідовності для кожного основного сценарію*, потім для діаґрами послідовності для варіацій сценаріїв і, нарешті, хоча б одну діаґраму послідовності, що ілюструє кожен з *можливих типів помилок або виключень*. Опрацювання помилок є частиною ВВ, тому вона повинна бути запланована поряд з нормальною поведінкою.

ВВ тестування. Більшість інших UML діаграм (діаґрами класів, компонентів, станів) є явними кандидатами на пряме і зворотне проектування, тому що кожна з них має якийсь аналог у системі, що виконується. ВВ описують те, як поводиться елемент, а не те, як він реалізований. Тому вони не можуть бути безпосередньо використані в прямому і зворотному проектуваннях, а через формування тестів – для елемента, що його описує. Кожен ВВ на такій діаґрамі специфікує потік подій з його варіаціями і всі ці потоки описують очікувану поведінку елемента, що підлягає тестуванню. Добре структурований ВВ специфікує перед- і післяумови, які можуть застосовуватися з метою визначення початкового стану тесту й критеріїв його успішності. Для кожного

ВВ з діаґрами можна створити **сценарій тестування** (test case), який буде виконуватися при випуску кожної нової версії даного елемента, у такий спосіб підтверджуючи, що він працює належним чином, перш ніж інші елементи зможуть опиратися на нього у своїй роботі. [1,2,12]

Щоб здійснити пряме проектування діаґрами ВВ, необхідно: ідентифікувати об'єкти, що взаємодіють із системою; ідентифікувати різні ролі, які може виконувати кожен зовнішній об'єкт; створити актант, що представляє кожну з окремих взаємодіючих ролей; для кожного ВВ на діаґрамі ідентифікувати основний і альтернативний потоки подій; зґенерувати відповідний сценарій для кожного потоку, використовуючи предумову потоку як початковий стан тесту, і постумову – як критерій успішності; за необхідності зґенерувати тестову інфраструктуру, яка буде представляти кожну з дійових осіб, що взаємодіють з ВВ; використовувати інструментальні засоби для запуску кожного із тестів при реалізації елемента, до якого відноситься діаґрама ВВ.

Добре структурована діаґрама ВВ: сконцентрована на передаванні одного аспекта вигляду системи з погляду ВВ; містить тільки ті ВВ і дійові особи, які важливі для розуміння даного аспектука; забезпечує вигляд, що відповідає її рівню абстракції: вказуються тільки важливі для розуміння діаґрами доповнення (зокрема, точки розширення). Ім'я діаґрами ВВ має відповідати її призначенню. Слід організовувати елементи так, щоб семантично близькі за поведінкою і ролями розташовувалися поруч; акцентувати увагу на важливих особливостях; не показувати надто багато видів зв'язків, використовуючи примітки й колірне виділення. Складні зв'язки включення й розширення слід виносити в окрему діаґраму.

3.2. Моделювання взаємодій об'єктів

3.2.1. Стратегія моделювання взаємодій як обміну повідомленнями між об'єктами

Основні питання:

- *Ролі, посилання, повідомлення, дії та послідовності*
- *Моделювання потоків керування*
- *Створення добре структурованих алгоритмів*

У побудованій об'єктно-орієнтованій системі об'єкти є не обособлені, а взаємодіють між собою шляхом передавання повідомлень.

Взаємодія (interaction) – поведінка, що визначається *обміном повідомленнями* між множиною об'єктів, що відіграють певні ролі в межах деякого контексту для досягнення заданої мети. [2] Взаємодії використовуються для моделювання динамічних аспектів кооперацій об'єктів (*за реалізацією ВВ*), які відіграють специфічні ролі й працюють спільно для забезпечення поведінки, більшої ніж поведінка простої суми елементів. Ці ролі представляють **прототипні екземпляри** класів, інтерфейсів, компонентів, вузлів і ВВ. Їхні динамічні аспекти моделюються у вигляді **потоків керування**, які можуть бути представлені в системі як прості послідовні потоки або включати *розгалуження, цикли, рекурсії* і *паралелізм*. Взаємодія моделюється шляхом виділення тимчасової послідовності повідомлень у контексті деякої структурної організації об'єктів. Добре структуровані взаємодії (*подібні до добре структурованих алґоритмів*) є ефективні, адаптовувані, зрозумілі.

Будинок — це жива сутність. Хоча кожен будинок і складається зі статичних об'єктів (цеглини, балки, панелі, скло ...), усі ці речі в сукупності працюють динамічно для досягнення поведінки, зручної для жителів будинку. Двері й вікна відчиняються і зачиняються, світло

вмикається й вимикається. Системи опалення й кондиціювання спільно забезпечують температурний режим. В «інтелектуальних» будинках сенсори визначають наявність чи відсутність діяльності жителів і регулюють освітлення, опалення тощо. Добре структуровані будинки проектують з урахуванням протистояння динамічним впливам: вітру, землетрусам і переміщенню жителів, забезпечуючи адаптивність до змін температури впродовж доби і її сезонних коливань.

Аналогічно для ПЗ. Система управління польотами взаємодіє з великою множиною інформаційних об’єктів, які викликаються до життя і виконують свою роботу тільки за певних обставин, коли система запускає в хід певну подію: продаж квитків, виліт літака, планування розкладу польотів. *«Аеропорт – кожен аеропорт – це складний механізм, керувати ним не легко. Немає такої людини, яка б відповідала за все, але й самостійно функціонуючих ділянок також немає: все переплетено, все взаємозв'язано…» [Артур Гейлі, «Аеропорт»].*

Діаґрами взаємодій як і діаґрами об’єктів статично визначають **фази поведінки**, представляючи всі об’єкти, що корпоративно виконують спільну роботу для досягнення певної мети (*для певних сценаріїв поведінки*). На відміну від діаґрам об’єктів, діаґрами взаємодій представляють **повідомлення**, передані від об’єкта до об’єкта.

Діаґрами взаємодій використовуються для моделювання **потоків керування** (*в межах операції, класу, компоненти, ВВ чи системи в цілому*), визначивши, як повідомлення передаються в часі, якими є структурні зв’язки між об’єктами у взаємодії, як повідомлення передаються в контексті структури. [1,2,4]

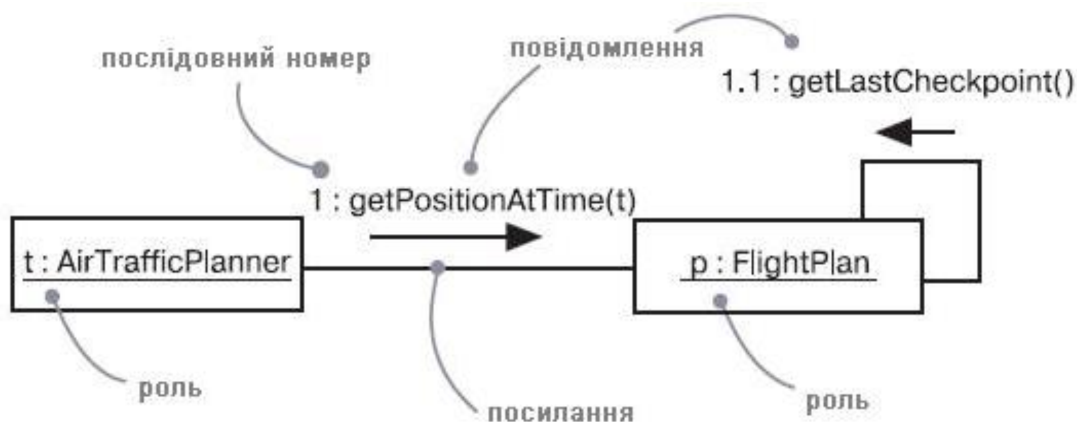


Рис. 3.2.1. Повідомлення, посилання і послідовність

Повідомлення (message) є *специфікацією взаємодії об’єктів*, що передає інформацію й очікує наступних дій. [2] Повідомлення можуть супроводжуватися викликом операції або передавання сиґналу та можуть супроводжуватися створенням і знищенням інших об’єктів. На концептуальному рівні повідомлення специфікуються від взаємодії між класами об’єктів. Конкретна взаємодія між двома об’єктами визначатиметься конкретним **екземпляром повідомлення**. Графічна нотація повідомлень в UML є *лінією зі стрілкою* і специфікує його найважливіші властивості: номер послідовності, ім’я його операції, параметри (якщо є) (рис. 3.2.1).

Контекст взаємодії. Взаємодія має місце, коли об’єкти з’єднані один з одним. Ввзаємодії проявляються в коопераціях об’єктів, що існують у контексті системи (*підсистем, класів із реалізації ВВ та окремих операцій*). У контексті взаємодії можна виявити екземпляри класів, компонентів, вузлів і ВВ. [1,2,12]

У системі Web-Комерції можна виявити такі клієнтські об’єкти, як екземпляри класів BookOrder (Замовлення книги) і OrderForm (Анкета замовлення), що

взаємодіють один з одним. Там же присутні клієнти (екземпляри BookOrder), взаємодіючі з об'єктами на сервері (екземплярами BookOrderManager (Менеджер замовлень). Взаємодії не тільки включають локалізовану кооперацію об'єктів, а й поширюються через багато концептуальних рівнів системи.

Взаємодії мають місце між об'єктами в реалізації операцій. Параметри операцій, їх локальні змінні, глобальні стосовно операції об'єкти (*видимі в ній*) можуть взаємодіяти один з одним для забезпечення роботи алґоритму, що реалізує операцію. *Операція* MovetoPosition(p:Position) (Перейтина позицію p:Позиція), *визначена в класі рухомого робота, включає взаємодію з параметром* p, *глобальними стосовно операції об'єктами* (CurrentPosition (Поточна позиція) та *кількома локальними об'єктами: локальними змінними, використовуваною операцією для обчислення проміжних точок на шляху до нової позиції.*

Взаємодії в контексті класу можна застосовувати для ВСКД його семантики. Щоб зрозуміти зміст класу RaytraceAgent, можна описати взаємодію кооперації його атрибутів (з глобальними об'єктами стосовно екземплярів класу та параметрами його операцій).

Об'єкти і ролі у взаємодії. Об'єкти, які беруть участь у взаємодії, можуть бути або конкретними сутностями (*певні реалії*), або прототипами (*будь-який екземпляр класу*). p екземпляр класу Person описує конкретну особу. З іншого боку, як прототипна сутність p може представляти будь-який екземпляр класу Person.

Хоча абстрактні класи й інтерфейси за визначенням не можуть мати *прямих екземплярів*, їх екземпляри можна показати у взаємодії. Останні не є прямими екземплярами абстрактних класів або інтерфейсів, але можуть представляти **непрямі** або **прототипні екземпляри** будь-яких конкретних нащадків абстрактного класу чи конкретного класу, що реалізує інтерфейс.

Посилання та коннектори. Взаємодія (*на діаґрамі кооперації об'єктів*) представляє *динамічну послідовність повідомлень*, які можуть передаватися через **посилання**, *що з'єднують дані об'єкти*.

Посилання (link) – це *семантичне з'єднання* об'єктів один з одним. Посилання є **екземпляром асоціації**. [2] Якщо між класами встановлений *зв'язок асоціації*, то *між екземплярами цих класів (об'єктів)* обов'язково має існувати *посилання* (рис. 3.2.2).

Необхідна умова для обміну повідомлень між об'єктами. За наявності посилання між двома об'єктами один з них може посилати повідомлення іншому. **Посилання вказує шлях**, по якому один об'єкт може посилати повідомлення іншому (*або самому собі*). Для точнішого описування посилання як *екземпляра асоціації* можна позначити відповідний його кінець одним із обмежень:

- association – вказує, що відповідний об'єкт є видимий асоціації;
- self – вказує, що об'єкт є видимий, оскільки він є диспетчером операції;
- global – об'єкт видимий як об'єкт, що перебуває в гдобальній області дії;
- local – об'єкт видимий, оскільки він перебуває в локальній області дії;
- parameter – відповідний об'єкт видимий, тому що він є параметром.

Прототипні об'єкти (*ролі*) **та прототипні зв'язки або посилання** (*коннектори*). При розробленні моделей більше використовуються прототипні об'єкти та їх прототипні посилання ніж конкретні індивідуальні об'єкти з їхніми посиланнями. Множинність **ролей** (*прототипних об'єктів*) і **коннекторів** (*прототипних посилань*) визначена щодо контексту, що їх включає (*контекстом тут є кооперація або внутрішня структура класифікатора*). [1,2] Якщо множинність ролі дорівнє 1, то на один об'єкт, що представляє контекст, припадає один об'єкт, що представляє роль. Неідентифікована

роль візуалізується як анонімний чи прототипний об’єкт. Ролі (прототипні об’єкти) зв’язані *коннекторами* (прототипними посиланнями /зв’язками/) із вказаними повідомленнями (*концептуальними*), а об’єкти – посиланнями (зв’язками як *екземплярами асоціацій*), позначеними реальними *екземплярами повідомлень*.

У верхній частині рис. 3.2.2 показана діаґрама класів, яка оголошує класи Person (Особа) і Company (Компанія) та існуючу між ними асоціацію «*багато до багатьох*» employee-employer (наймач-працівник). Середня представляє зміст *кооперації* WorkAssignment (Призначення на посаду), що містить у собі дві ролі з коннектором між ними. Внизу поданий **екземпляр кооперації**, у якому є об’єкти і посилання, що представляють ролі й коннектори. Конкретне повідомлення є *прототипним оголошенням повідомлення* в даній кооперації.

Типи та екземпляри повідомлень. Діаґрама об’єктів, що включає набір об’єктів і посилань, які з’єднують ці об’єкти, є статичною моделлю, що описує стан співтовариства об’єктів у заданий момент часу. При моделюванні *змінного стану* набору об’єктів за деякий період часу потрібно наочно продемонструвати їх переміщення в наступні моменти часу, включаючи *передавання повідомлень між об’єктами*, *події* та *виклики операцій*. У кожному такому «кадрі» можна явно візуалізувати поточний стан і роль індивідуальних екземплярів. [1]

Повідомлення є специфікацією взаємодії об’єктів, що забезпечує передавання інформації, яка повинна ініціювати певні дії. Момент прийому **екземпляра повідомлення** розглядається як *випадок ініціації події*.

Випадок ініціації події (occurrence) – це UML-термін, що описує екземпляр події. При передаванні повідомлення в результаті його прийому відбувається деяка дія, що полягає в зміні стану цільового об’єкта і доступних йому об’єктів. [1]

Типи повідомлень, моделювання яких забезпечує UML:

- call (*викликати*) – об’єкт викликає операцію іншого об’єкта або свою власну, посилаючи повідомлення самому собі (*локальний виклик операції*);
- return (*повернути*) – повертає значення викликаючому об’єкту;
- send (*послати*) – посилає сиґнал об’єкту;
- create (*створити*) – створює об’єкт;
- destroy (*знищити*) – знищує об’єкт. Об’єкт може знищити сам себе.

В UML ці типи повідомлень візуально відрізняються (рис. 3.2.3).

Об’єкт не може викликати будь-яку довільну операцію. Якщо об’єкт (рис. 3.2.3) викликає, операцію setItinerary (вказати напрямок) на екземплярі класу TicketAgent (Агент продажі білетів), то вона повинна бути оголошена в класі TicketAgent або в одному з його батьків і бути видима для викликаючого об’єкта c.

Коли об’єкт викликає операцію чи *посилає сиґнал* іншому об’єкту, то *повідомлення може бути оснащене конкретним параметром*. Аналогічно, коли один об’єкт повертає керування іншому, при цьому також можна змоделювати значення, що повертається.

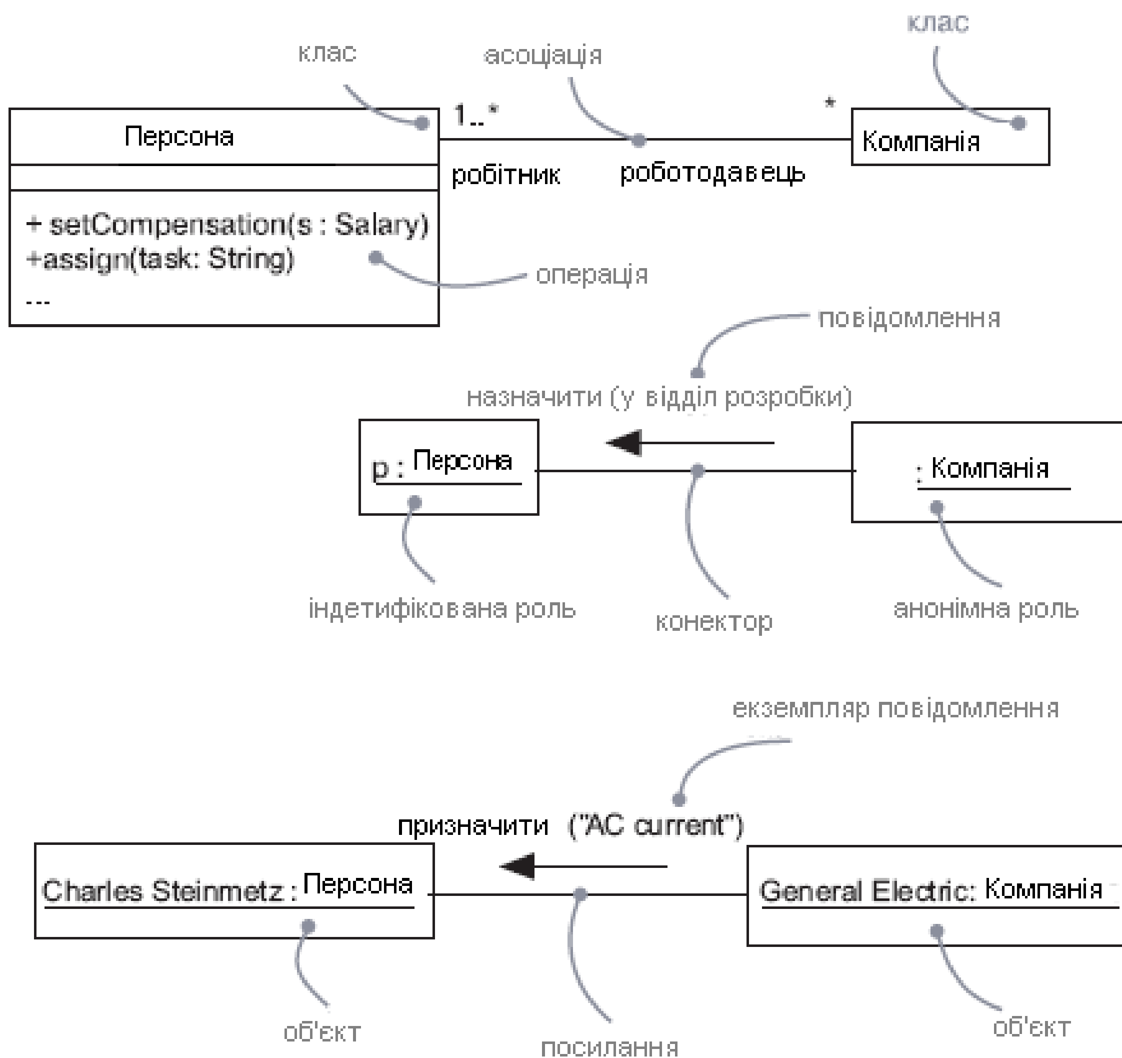


Рис. 3.2.2. Асоціації, посилання (екземпляри асоціацій), коннектори (прототипні посилання)

Посилання сиґналів. Повідомлення можуть мати відношення й до посилання сиґналів. **Сиґнал** (signal) – це значення об'єкта, *передане цільовому об'єкту асинхронно*. Після відправлення сиґналу об'єкт, який його відправив, продовжує свою діяльність. Отримуючи повідомлення сиґналу, цільовий об'єкт незалежно приймає рішення стосовно того, що з ним потрібно робити. [2]

Сиґнали ініціюють перехід автомата цільового об'єкта в інший стан. Ініціалізація такого переходу змушує цільовий об'єкт виконати якісь дії й змінити свій стан. У системах з асинхронним передаванням подій взаємодіючі об'єкти виконуються паралельно й незалежно. Вони розділяють інформацію тільки за допомогою передавання повідомлень, тому не виникає небезпеки конфлікту за використовувані ресурси.

Потоки та **послідовності** (sequencing) **повідомлень.** Коли один об'єкт передає повідомлення іншому (*делегуючи йому деяку дію*), то об'єкт, що приймає, може, в свою чергу, послати повідомлення ще одному об'єкту, а той – наступному й т.д у вигляді **потоку повідомлень**, що формує **послідовність.** [12,17] Усяка послідовність повідомлень повинна мати початок. Її запуск виконується певним *процесом* або *потоком керування*. Послідовність триває доти, поки існує *процес* або *потік,* що володіє

нею. Системи «нон-стоп», що використовуються для моделювання СРЧ, працюють доти, поки працює вузол, на якому вони запущені.

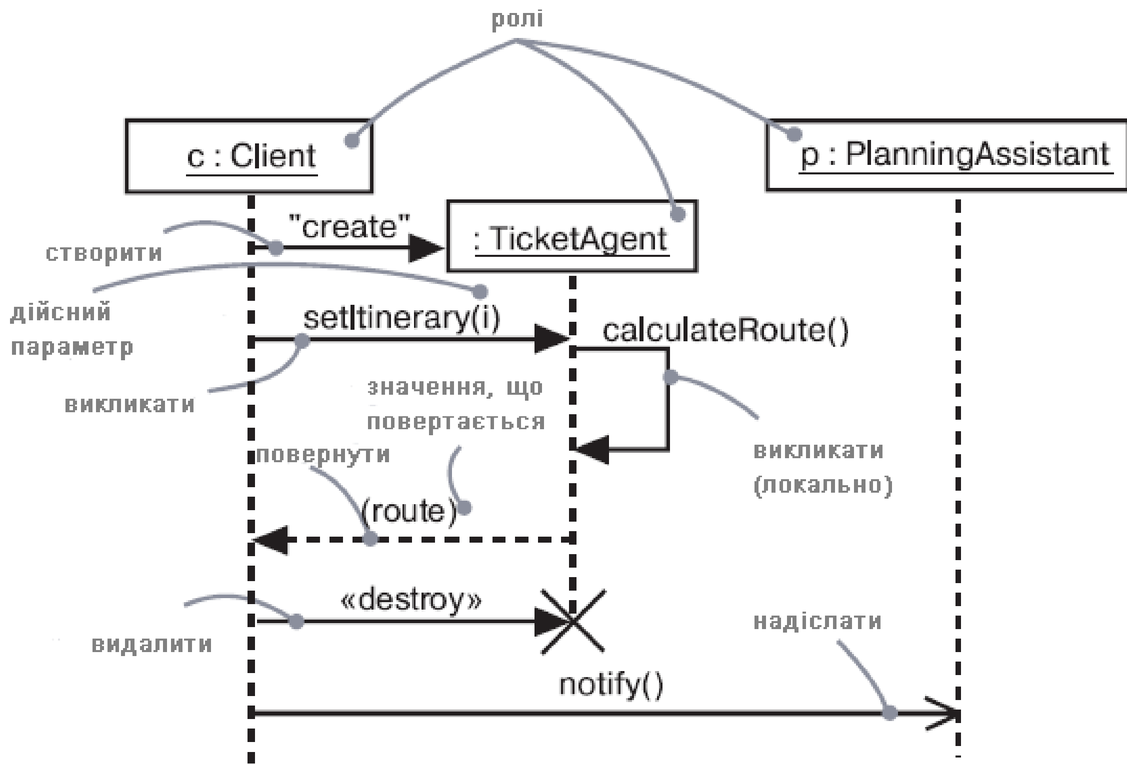


Рис. 3.2.3. Типи повідомлень на діаґрамі кооперації

Потоки керування *або* **процеси** визначають упорядковані в часі (*динамічно*) послідовністі повідомлень у системі й характеризують її поведінку в цілому. Потік керування (процес) **запускає** (*ініціалізує*) послідовність повідомлень. Щоб краще візуалізувати послідовність повідомлень, можна явно змоделювати їхній порядок відносно початку послідовності, надавши повідомленню префікс, що вказує його номер з двокрапкою в якості роздільника.

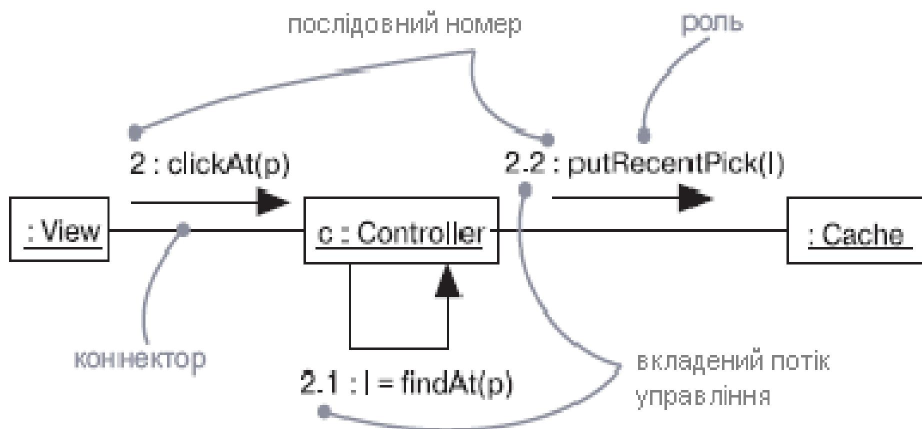


Рис. 3.2.4. Процедурний (вкладений) потік керування

Діаґрама комунікації відображає *потік повідомлень між ролями* в межах кооперації. Потік повідомлень передається через **з'єднання** у коопераціях об'єктів (рис. 3.2.4).

Процедурний потік керування. Повідомлення процедурного (*вкладеного*) потоку керування специфікється *зафарбованою стрілкою* з вказанням рівнів їх вкладеності (рис. 3.2.4). Повідомлення find At (знайти у) специфікується як перше повідомлення, вкладене в друге в послідовності (2.1).

Послідовний (*плоский*) потік керування специфікується послідовною нумерацією повідомлень (*у вигляді загостреної або зафарбованої стрілки*), моделюючи послідовність керування «*крок за кроком*» (рис. 3.2.5). Повідомлення acceptCall (Прийняти дзвінок) вказується як друге в послідовності. [1,2]

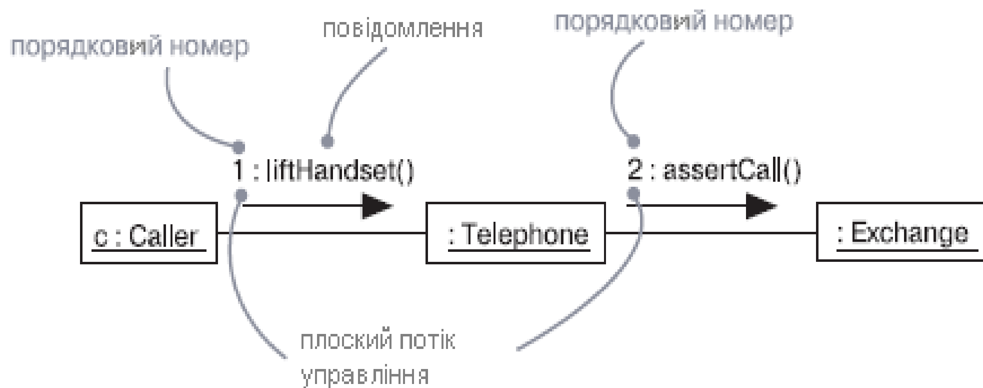


Рис. 3.2.5. Послідовний (плоский) потік керування

Ідентифікація процесів і потоків керування. При моделюванні взаємодій, що включають багато потоків керування, важливим є *ідентифікувати процес або потік, який відправив конкретне повідомлення*. В UML можна відрізнити один потік керування від іншого, доповнивши послідовний номер повідомлення **префіксом**, у якості якого виступає *ім'я процесу або потоку, що стоїть на початку послідовності*. Вираз D5:ejectHatch(3) вказує, що операція ejecthatch (*відкрити затвор*) реалізується з переданим їй фактичним параметром 3 як п'яте повідомлення послідовності, запущеної в *процесі або потоці* з іменем D. Можна показати значення, що повертає операція: 1.3.2 : p := find("Rachelle"). Тут значення p, що повертається з операції find, «Rachelle» – *фактичний параметр*. Послідовність є вкладена: операція виконується в результаті другого повідомлення, вкладеного в третє, яке, у свою чергу, вкладене в перше повідомлення послідовності. На тій же діаґрамі p можна використовувати в якості фактичного параметра для інших повідомлень.

Створення, модифікація та знищення об'єктів і посилань. Модельовані об'єкти, що беруть участь у взаємодіях, як правило, існують протягом усього часу взаємодії. Однак у деяких взаємодіях об'єкти можуть створюватися (*повідомлення* create) і знищуватися (*повідомлення* destroy). Те ж саме стосується і посилань: *зв'язки між об'єктами можуть виникати і зникати*. Щоб вказати, що об'єкт або посилання з'являється або зникає в процесі взаємодії, можна *приєднати примітку до її ролі* в діаґрамі комунікації.

При взаємодії об'єкт змінює значення своїх атрибутів, свій стан або свої ролі. Можна показати *модифікації об'єкта* на діаґрамі послідовності, *вказавши стан або значення на лінії життя*.

ЖЦ, створення і знищення об'єктів або ролей всередині діаґрами послідовності зображуються у вигляді *вертикальних ділянок* їх ліній життя. На діаґрамі комунікації створення і знищення позначаються примітками. Слід використовувати діаґрами послідовності, коли важливо показати ЖЦ об'єктів.

Представлення. При моделюванні взаємодії візуалізуються **ролі** (*об'єкти в екземплярі взаємодії*) і **повідомлення** (*комунікації між об'єктами разом з деякою результуючою дією*) з допомогою двох способів: вказуючи часовий *порядок повідомлень* або *структурну організацію ролей*, що відправляють і приймають повідомлення. В UML перший тип називаються **діаґрамою послідовності**, а другий – **діаґрамою комунікації**, що є різновидами діаґрам взаємодії. UML передбачає також більш спеціалізований різновид діаґрам взаємодії – **тимчасову діаґраму**, що показує конкретний час передавання повідомлень між ролями. [1]

Діаґрами послідовності й комунікації є семантично еквівалентними: одну із них можна перетворити в іншу. Є окремі візуальні відмінності. Діаґрами послідовності дозволяють моделювати лінію життя об'єкта, яка описує його існування в певний період часу (*можливо, включаючи його створення і ліквідацію*). Діаґрами комунікації дозволяють моделювати структурні посилання, що виникають між об'єктами при їхній взаємодії.

Моделювання потоку керування. Найчастіше взаємодії використовуються для моделювання потоку керування, що характеризує поведінку системи в цілому (*в т.ч. ВВ, зразки, механізми і каркаси*) або поведінку класу чи окремої операції. Моделюючи взаємодію створюється **сценарій** дій, що реалізовується *кооперацією* об'єктів певних класів.

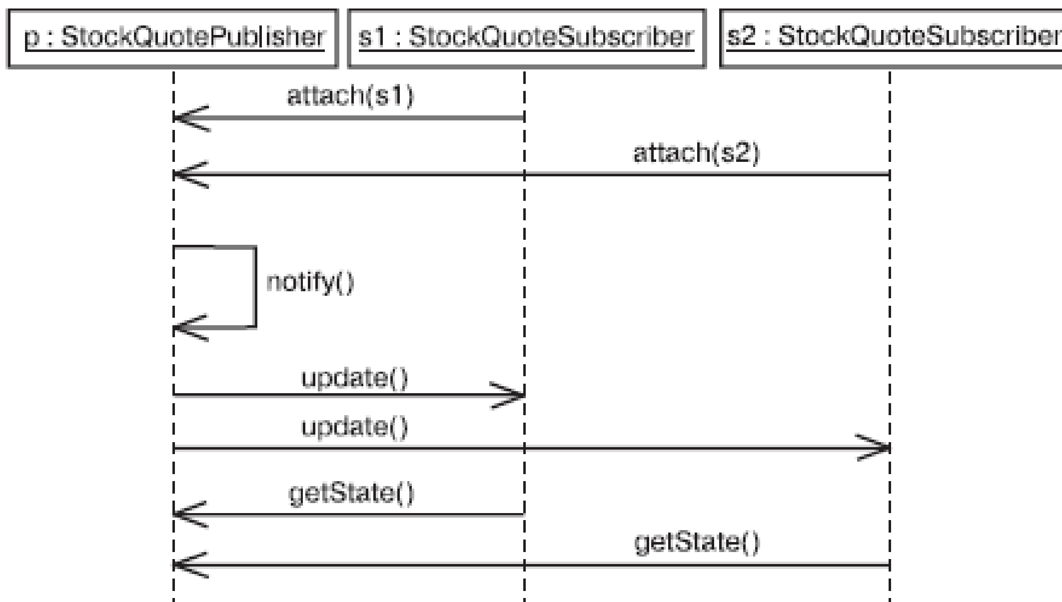


Рис. 3.2.6. Потік керування в часі (діаграма послідовності)

Для моделювання потоку керування необхідно: встановити контекст взаємодії (чи це система, клас, операція); визначити **фазу взаємодії**, виявивши, які об'єкти відіграють ті чи інші ролі; встановити початкові характеристики об'єктів (значення

атрибутів, стан і роль та дати ролям імена). Якщо модель описує структурну організацію об'єктів, ідентифікувати з'єднуючі їх посилання, важливі для здійснення комунікацій; специфікувати їх природу, використовуючи необхідні засоби UML. Відповідно до послідовності слід вказати повідомлення, передані від об'єкта до об'єкта; виділити різні їх види; описати параметри і значення, що повертаються. Для деталізації взаємодії слід наділити кожен об'єкт, представлений у різні моменти часу, інформацією про його стан і ролі.

Діаграма послідовності (рис. 3.2.6) демонструє набір ролей, що взаємодіють у контексті механізму публікації і підписки (*екземпляр патерну проектування* observer *(«оглядач»)*) і показує часовий порядок повідомлень між ними. Тут введено три ідентифіковані ролі: p: StockQuotePublisher (екземпляр класу Видавець таблиць квотування акцій) і два екземпляри s1 і s2 класу StockQuoteSubscriber (підписник таблиць квотування акцій).

Семантично еквівалентна попередній діаграма комунікації (рис. 3.2.7) розкриває структурну організацію об'єктів. Тут показано той же потік керування візуалізованим посиланням між об'єктами.

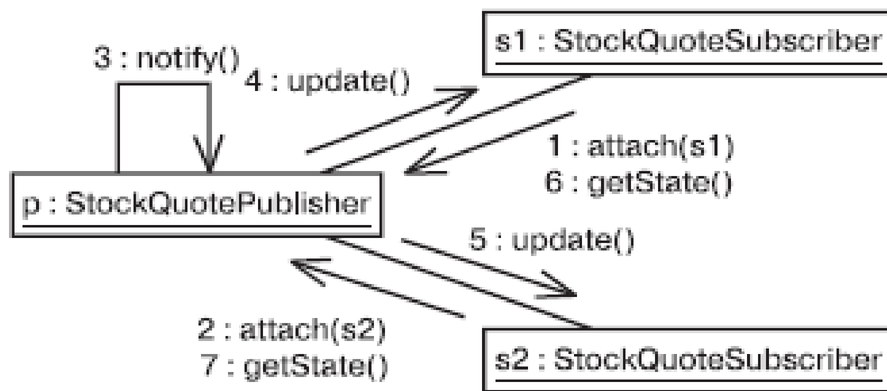


Рис. 3.2.7. Організація потоку управління

Модель взаємодії в UML описує динамічний аспект *кооперації* об'єктів. Добре структурована взаємодія доступно описує кооперацію об'єктів, що працюють разом для забезпечення поведінки, більшої за суму поведінок цих об'єктів; має чіткий контекст, ефективно забезпечує потрібну поведінку з оптимальним балансом часу і ресурсів. При моделюванні взаємодії в UML слід вибрати спосіб відображення взаємодії за хронологічним порядком повідомлень або структурної організації об'єктів. Слід звернути увагу на часткову впорядкованість подій в субпослідовностях (*кожна з них є впорядкована, але відносний час настання подій у різних субпослідовностях не фіксований*); показувати тільки важливі для розуміння взаємодії властивості кожного об'єкта (*значення атрибутів, ролі і стану*) й кожного повідомлення (*параметри, семантику паралельності значення, що повертається*).

3.2.2. Діаграми взаємодії

Основні питання:

- *Моделювання потоків керування за часом*
- *Моделювання потоків керування по організації*

Діаграми взаємодії показують взаємодію *кооперації об'єктів* та їх *зв'язки і повідомлення*, що можуть передаватися між ними. Діаграми послідовності й комунікації

– це діаґрами взаємодії, перша з яких відображає **часовий порядок** повідомлень, а друга – **структурну організацію об'єктів**, що відправляють і приймають повідомлення.

Під час зйомки фільму застосовується техніка послідовних зрізів подій, створюючи розкадрування основних сцен. Будується модель кожної сцени, щоб передати свій задум колегам по знімальній ґрупі. Створення розкадрування — це основна мета виробничого процесу зйомок, що дозволяє команді ВСКД модель кінофільму в її розвитку – від початкової точки до випуску в прокат.

При розробленні ПЗ постає та ж проблема: як змоделювати динамічну поведінку системи? Як можна візуалізувати працюючу систему? Використовуючи інтерактивний відладчик, з'єднаний із системою, можна переглянути ділянку пам'яті й побачити, як його зміст з часом змінюється, спостерігати за окремими об'єктами тощо. Можна побачити створення деяких об'єктів, зміни значень їх атрибутів, а потім – знищення деяких з них.

Стратеґії розкадрування сценаріїв. Можливості подібної візуалізації динамічних аспектів системи є обмежені, особливо якщо мова йде про розподілену систему з множиною паралельних потоків керування. З таким же успіхом можна спробувати зрозуміти систему кровообігу людини, зосередивши увагу на порції крові, яка перекачується через одну ділянку однієї артерії за одиницю часу. Раціональніший спосіб моделювання динамічних аспектів системи полягає в побудові «розкадрування сценаріїв» із *включенням взаємодії певних об'єктів і повідомлень*, які можуть передаватися між ними.

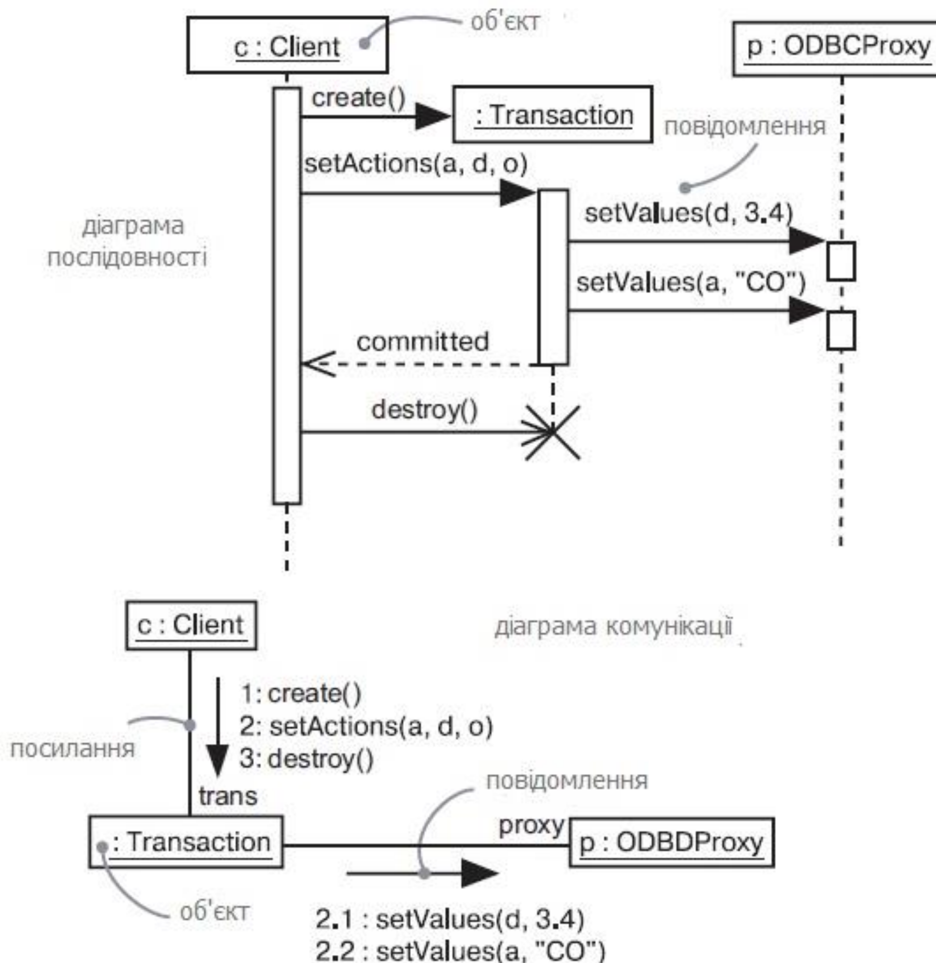


Рис. 3.2.8. Діаґрама взаємодії

В UML такі «розкадрування» моделюються діаграмами взаємодії. Останні можна будувати одним із двох способів: виділяючи часовий порядок повідомлень і виділяючи структурні зв’язки між взаємодіючими об’єктами кооперацій. Ці семантично еквівалентні діаграми можна конвертувати одну в іншу без втрати інформації (рис. 3.2.8).

Діаграма взаємодії показує взаємодію, що бере за основу набір об’єктів і їх зв’язків, включаючи передані між ними повідомлення. [2] Діаграма взаємодії характеризується іменем і змістом, що представляють проекцію моделі, містять ролі або об’єкти, зв’язки або посилання, повідомлення, можуть містити примітки і обмеження.

Діаграма послідовності підкреслює часовий порядок повідомлень і *зображується як таблиця, у якій представлені об’єкти*, розташовані вздовж осі *X* і повідомлення, упорядковані по ходу часу – вздовж осі *У*. [1,2] Діаграми послідовності виділяють *часовий порядок повідомлень*. Формування діаграми послідовності починається з розміщення об’єктів або ролей, що беруть участь у взаємодії, у верхній частині діаграми, по горизонтальній осі (рис. 4.4.2). Об’єкт або роль, що *ініціюють взаємодію*, розміщуються зліва, а *залежні об’єкти* або ролі – справа. Далі по вертикальній осі розставляються повідомлення, які відправляють і приймають ці об’єкти, згори до низу у хронологічному порядку, що створює асоціацію *потоку керування в часі*.

Діаграми послідовності відрізняються від інших (діаграм комунікації) двома ознаками:

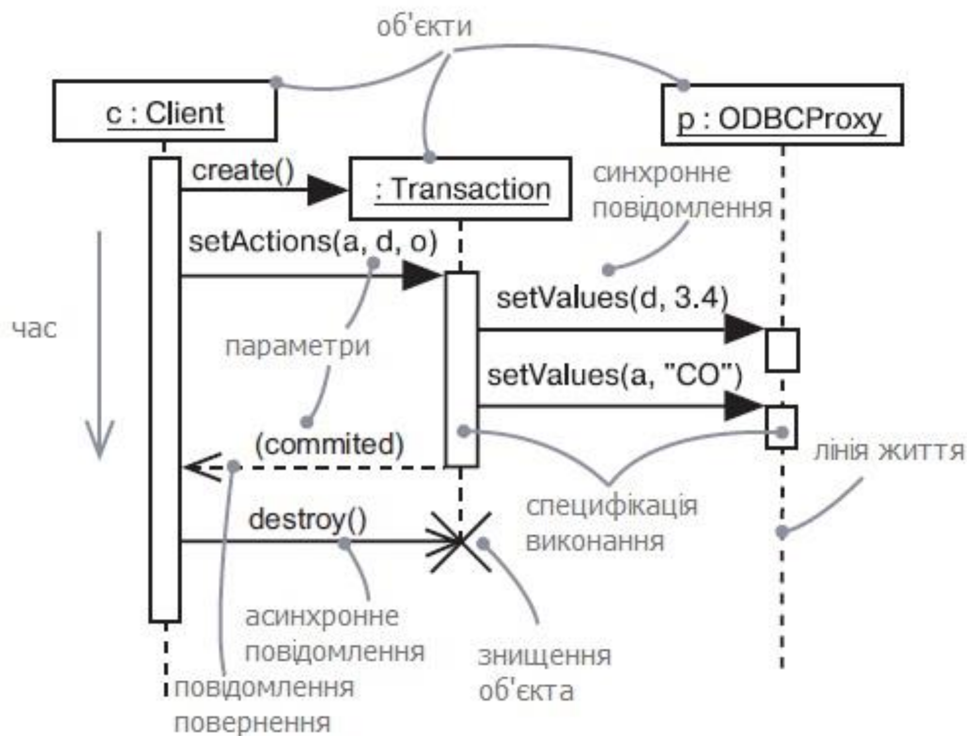


Рис. 3.2.9. Діаграма послідовності

По-перше, це наявність лінії життя об’єкта. **Лінія життя об’єкта** (lifelines) – це вертикальна пунктирна лінія, що символізує існування об’єкта впродовж деякого періоду часу. [1] Більшість об’єктів на діаграмі взаємодії існують протягом усього часу взаємодії, тому всі вони *вирівняні по верхній границі діаграми*, а лінії їх життя проведені зверху вниз.

Об'єкти можуть бути створені в процесі взаємодії. Їхній час життя починається з отримання повідомлення create, *спрямованого до прямокутника об'єкта* на початку ЖЦ. Так само в процесі взаємодії об'єкти можуть знищуватися. Їхня лінія життя закінчується при отриманні повідомлення destroy, що графічно позначене великим символом X. Якщо взаємодія відображає історію конкретних об'єктів, то *символ об'єкта з підкресленим іменем* розміщується на початку лінії життя. В основному показують **взаємодії прототипів** (*ролей*). При цьому лінії життя не представляють конкретних об'єктів – вони специфікують **прототипні** *(неідентифіковані)* **ролі**, що представляють різні об'єкти в кожному екземплярі взаємодії. В цьому випадку **не потрібно підкреслювати** *імена ролей*: вони не символізують конкретних об'єктів.

По-друге, це наявність фокуса керування. **Фокус керування** (focus of control) – високий вузький прямокутник, що показує період часу виконання дії об'єктом як безпосередньо, так і за допомогою залежної процедури. [1] Верхня грань прямокутника вирівняна за початком дії, а нижня – після її завершення і може бути позначена **повідомленням повернення**. Можна показати **вкладеність** фокуса керування, викликану рекурсією, викликом власної операції чи поверненням виклику з іншого об'єкта, наклавши інший фокус керування ненабагато правіше батьківського (*допускається довільна кількість рівнів вкладення*). Якщо потрібно особливо точно показати розташування фокуса керування, то слід *відтінити* частину прямокутника, що визначає період часу, протягом якого насправді працює метод об'єкта і керування не передається іншому об'єкту.

Основний зміст діаграми послідовності – повідомлення, що зображуються *стрілками*, спрямованими від однієї *лінії життя* до іншої. Стрілка вказує на *отримувача* повідомлення. Повідомлення можуть бути синхронними й асинхронними. **Синхронне** повідомлення (виклик) зображується *зафарбованим трикутником*. Якщо воно **асинхронне** (сигнал), то стрілка малюється *«кутиком»* (напівстрілкою).

Відповідь на синхронне повідомлення (повернення з виклику) показується **пунктирною** *стрілкою* «*кутиком*». **Повідомлення повернення** може бути опущене, оскільки кожен виклик неявно має на увазі повернення, але іноді зручно в такий спосіб продемонструвати *значення, що повертається*.

Упорядкування за часом уздовж єдиної лінії життя досить важливе. Точна відстань не має значення: лінії життя показують лише відносні послідовності, не забезпечуючи масштабного відображення часу. Крім того, позиції повідомлень на окремих парах ліній життя, як правило, не впливають на хронологію передавання інформації; **повідомлення можуть надходити в будь-якому порядку**. Повні набори повідомлень на окремих лініях життя формують *часткове впорядкування*. Серії повідомлень встановлюють **ланцюг причинних зв'язків**, тому *будь-яка точка на іншій лінії життя наприкінці ланцюга повинна завжди іти за точкою початку ланцюга на вихідній лінії*.

Структуроване високорівневе керування на діаґрамах послідовності. Послідовність повідомлень добре підходить для відображення окремих **лінійних послідовностей**, але часто виникає необхідність показати **умовні оператори** і **цикли**. Потрібно іноді відобразити **паралельне виконання** багатьох послідовностей. Ці різновиди *високорівневого керування* можуть бути показані на діаґрамах послідовності за допомогою операторів структурованого керування.

Оператор керування (control operator) зображується на діаґрамі послідовності у вигляді **прямокутної області**. [2,12] Він супроводжується **тегом** — текстовою позначкою, укладеною в маленький *п'ятикутник* у верхньому лівому куті. Вона показує, який це оператор керування. Оператор *застосовується до ліній життя, які*

його перетинають. Цю частину називають **тілом оператора**. Якщо лінія життя не порушується оператором, вона може бути перервана в його початку (*вгорі*) і продовжена наприкінці (*внизу*).

Види керування при моделюванні динамічних характеристик систем:

- **необов'язкове виконання** (тег opt). Тіло цього керуючого оператора **виконується**, якщо *захисна умова дійсна* на його вході. Захисна умова — це булевий вираз, вказаний у *квадратних дужках* у верхній частині однієї з ліній життя всередині тіла і може посилатися на атрибути об'єкта;
- **умовне виконання** (тег alt). Тіло керуючого оператора розділяється на кілька підобластей *горизонтальними пунктирними лініями*. Кожна підобласть представляє одну гілку умови і має власну захисну умову. Якщо захисна умова підобласті істинна, то вона виконується. Разом з тим може виконатися не більше однієї підобласті. *Якщо істинні кілька захисних умов, то вибір підобласті для виконання не визначений* і може варіюватися від запуску до запуску. Якщо жодна з умов не істинна, то керування переходить за межі оператора. Одна підобласть може мати особливу захисну умову else. Така підобласть виконується, якщо недійсні ніякі інші захисні умови;
- **паралельне виконання** (тег par). Тіло керуючого оператора ділиться на кілька підобластей *горизонтальними пунктирними лініями*. Кожна підобласть представляє **паралельний (***конкуруючий***) потік обчислень**. При цьому в більшості випадків кожна підобласть включає різні лінії життя. Коли керування переходить до даного оператора, усі його підобласті починають *виконуватися паралельно*. Виконання повідомлень у кожній з них послідовне, але відносний порядок повідомлень із паралельних підобластей зовсім довільний. Ця конструкція не повинна застосовуватися, якщо різні обчислення взаємодіють. Це дуже зручний оператор, тому що існує дуже багато ситуацій в реальному світі, у яких можна *вичленувати* **незалежні паралельні потоки діяльності**;
- **циклічне (ітераційне) виконання** (тег loop). **Захисна умова** з'являється у вершині однієї лінії життя всередині тіла. Тіло оператора циклу виконується неодноразово – *поки захисна умова дійсна перед кожною ітерацією*. Коли вона приймає значення false у вершині тіла, керування передається за межі оператора. Існує багато інших операторів, але ці використовуються найчастіше.

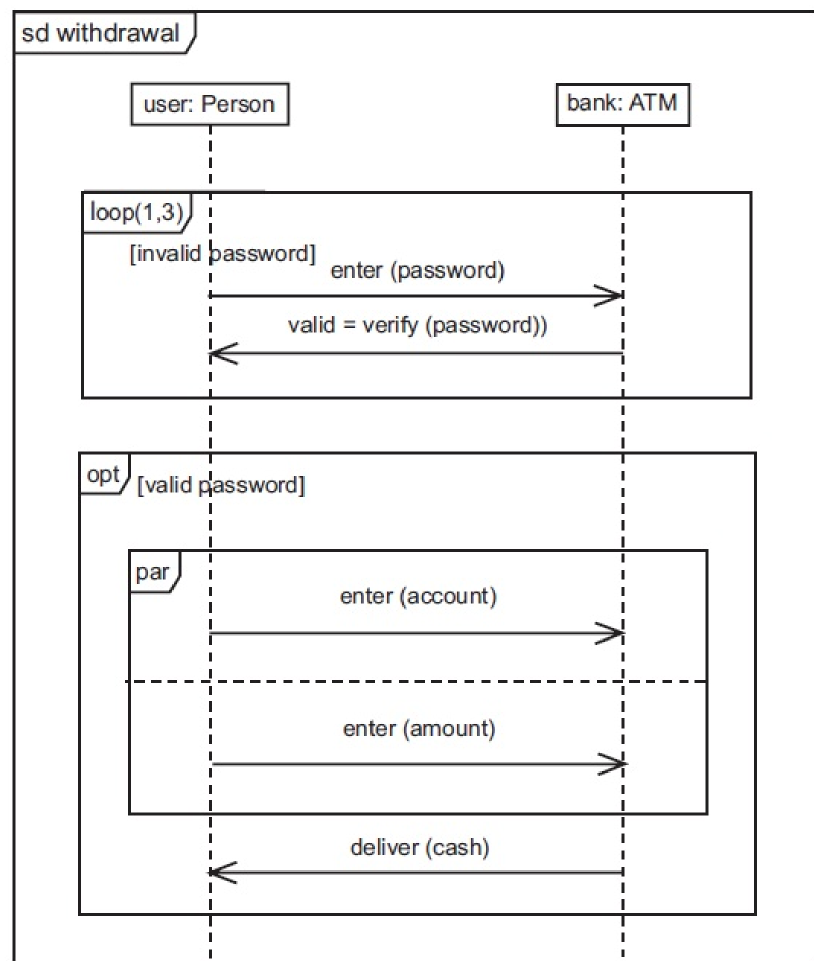


Рис. 3.2.10. Оператори структурованого управління

Для чіткого позначення границь діаґрама

послідовності може входити в прямокутник з тегом sd у верхньому лівому куті. За цим тегом може йти *ім'я діаґрами*. На рис. 3.2.10 представлений спрощений приклад, який ілюструє використання деяких керуючих операторів. Користувач (User) ініціює послідовність. Перший оператор є оператор циклу.

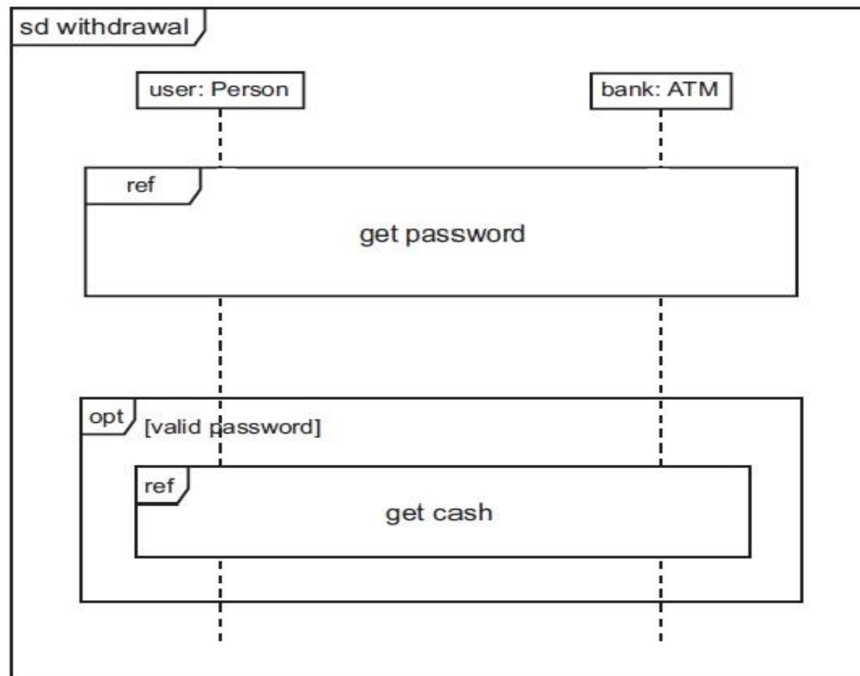


Рис. 3.2.11. Вкладена діаґрама діяльності

Цифри в дужках (1,3) означають мінімальну і максимальну кількість виконань тіла циклу. Оскільки мінімальне число – одиниця, це тіло виконається як мінімум один раз, перш ніж буде перевірена захисна умова. У циклі користувач вводить пароль, і система перевіряє його. Цикл переривається після трьох спроб, хоча може завершитися і раніше у випадку введення неправильного пароля. Наступний оператор необов'язковий. Його тіло виконується, якщо введений правильний пароль, а якщо ні, то залишок діаґрами послідовності пропускається. Тіло необов'язкового оператора містить у собі паралельний оператор. Оператори можуть бути *вкладені* (рис. 3.2.10).

Паралельний оператор має дві підобласті: одна дозволяє користувачеві ввести *номер* рахунку, а інша – *суму*. Оскільки вони паралельні, для цих двох елементів не передбачається певний порядок введення (*паралельність не завжди означає одночасне виконання*). Це означає, що ці дві дії не є скоординовані й можуть відбуватися в будь-якому порядку. Якщо вони дійсно незалежні, то можуть перекривати одна одну; якщо ж послідовні, те одна починається після завершення іншої в довільній послідовності.

Після того, як обидві дії виконані, паралельний оператор завершений. Далі всередині необов'язкового оператора банк видає готівку користувачеві. Цим вичерпується роль діаграми послідовності.

Вкладені діаґрами послідовності й діяльності (ref). Дуже великі діаґрами складні для розуміння. Тому їх структуровані розділи можуть бути організовані у вигляді підлеглих діяльностей (*особливо, коли такі виконуються не один раз у межах головної*). При цьому головна й підлеглі діяльності зображуються на окремих діаґрамах. Всередині головної діаґрами діяльності підлегла діяльність представлена у вигляді прямокутника з тегом ref у лівому верхньому куті. Ім'я підлеглої поведінки вказується в центрі рамки.

Підлегла поведінка не обмежується діаґрамою діяльності; вона може бути автоматом, діаґрамою послідовності чи іншою поведінковою специфікацією. На рис. 3.2.11 показана діаґрама з рис. 3.2.10, перероблена таким чином, що дві секції переміщені в окремі діаґрами діяльності, а на головній зазначені посилання на них.

Діаґрами комунікацій. Діаґрама комунікацій описує організацію об'єктів взаємодії. Вона виділяє *структурну організацію об'єктів*, що відправляють і приймають повідомлення і графічно є набір дуг і вершин. Діаґрама комунікації (рис. 3.2.12) формується, починаючи з розміщення об'єктів, що брали участь у взаємодії, у вершинах графів. Далі у *вигляді дуг графа* зображуються *посилання*, які з'єднують ці об'єкти. Зв'язкам можуть бути присвоєні імена ролей, що ідентифікують їх. Зв'язки доповнюються повідомленнями, якими обмінюються об'єкти, що дає чітке візуальне представлення потоку керування в контексті структурної організації взаємодіючих об'єктів.

Діаґрами комунікації мають дві ознаки, які відрізняють їх від діаґрам послідовності. *По-перше*, тут визначений **шлях** (path), який відображається відповідно до асоціації. Втім, можна також показати його відповідно до локальних змінних параметрів, глобальних змінних і звертання до самого себе. Шлях представляє джерело інформації для об'єкта.

По-друге, тут є порядковий номер. Щоб вказати порядок повідомлення в часі, воно ініціалізується номером, починаючи з 1 і далі – в арифметичній прогресії для кожного нового повідомлення в потоці керування (2, 3, ...). Щоб показати вкладеність, застосовується десяткова система класифікації Дьюі (1 – перше повідомлення, яке містить у собі повідомлення 1.1, 1.2 і т.д.). Кількість рівнів вкладення не обмежується. Крім того, на лінії одного і того ж посилання можна показати багато повідомлень (*можливо, що перетинаються в різних напрямках*) і кожне буде мати унікальний порядковий номер. Більшу частину часу доведеться моделювати прямі послідовні потоки керування. Є можливість моделювання складніших потоків, що включають *ітерації* й *розгалуження*.

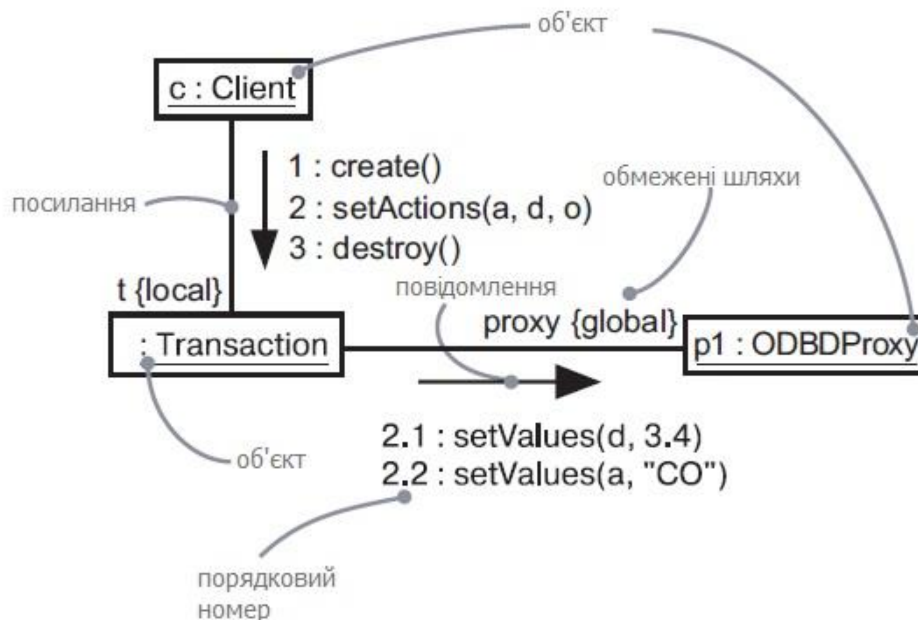


Рис. 3.2.12. Діаґрама комунікації

Ітерація (iteration) є *повторюваною послідовністю повідомлень*. Щоб змоделювати її, треба описати *порядковий номер повідомлення виразом ітерації* у форматі *[i:=1..n] (або просто *, якщо потрібно показати наявність ітерації, не вдаючись у деталі). [11,12] Ітерація свідчить про те, що *повідомлення і всі його вкладення повинні повторитися відповідно до заданого виразу*. Аналогічно можна описати ітераційність повідомлення умовою, виконання якої полягає в обчисленні булевого виразу. Щоб змоделювати умову, *порядковий номер повідомлення ініціалізують умовним виразом*, наприклад, [i<5].

Розгалуження (branching) – альтернативний шлях, що має такий самий порядковий номер, але кожен шлях повинен унікально відрізнятися умовами, що не перекриваються. Як для ітерацій, так і для розгалуження UML не регламентує формат виразу всередині дужок; можна використовувати псевдокод або синтаксис певної мови програмування. [2,3,7]

Семантична еквівалентність та відмінність моделей взаємодії. Оскільки діаґрами послідовності й діаґрами комунікації успадковують ту саму інформацію метамоделі UML, вони семантично еквівалентні й одну діаґраму можна перетворити в іншу без особливих втрат інформації. Однак обидві діаґрами явно не візуалізують ту саму інформацію. Діаґрама комунікації на рис. 3.2.12 показує, як пов’язані об’єкти (*анотації* {local} і {global}), а відповідна діаґрама послідовності (рис. 3.2.9) – ні. Проте діаґрама послідовності показує повернення повідомлення (*значення, що повертається*, committed), а відповідна діаґрама комунікації цього не робить. Оскільки модель, описана в одному форматі, не містить інформації, яка присутня в іншому, то ці діаґрами створюють дві принципово різні моделі, хоча використовують одну загальну базову модель.

Прикладні аспекти застосування. Діаґрами взаємодії можна використовувати до ВВ для моделювання сценарів і кооперацій. Вони також застосовуються для моделювання взаємодії екземплярів у контексті системи в цілому (*підсистеми, класу, операції*) на будь-якому представленні архітектури системи, включаючи класи (в тому числі активні), інтерфейси, компоненти і вузли. Можна моделювати потоки керування, упорядковані за часом із використанням діаґрами послідовності та потоки керування по організації, застосовуючи діаґрами комунікацій.

Моделювання потоків керування, упорядкованих за часом. Моделювання потоку керування, упорядкованого за часом, виділяє передавання повідомлень у хронологічному порядку, що, зокрема, зручно для візуалізації динамічної поведінки в контексті сценаріїв ВВ. Діаґрами послідовності (ДП) краще виконують завдання візуалізації простих ітерацій і розгалуження, ніж діаґрами комунікацій. Розглядаються об’єкти, які перебувають у контексті системи (*підсистеми, операції, класу*), а також об’єкти й ролі, які беруть участь у ВВ або кооперації. Для моделювання потоку керування, що проходить через ці об’єкти і ролі, слід застосовувати ДП, підкреслюючи часовий порядок повідомлень.

Загальна схема моделювання: встановити контекст взаємодії (*системи, підсистеми, операції чи класу, або одного зі сценаріїв ВВ, або кооперації*); встановити основу взаємодії, ідентифікувавши об’єкти, які відіграють роль у взаємодії, розташувати їх на ДП зліва направо – від важливіших до залежних; зобразити лінію життя (ЛЖ) кожного об’єкта (більшість із них існує протягом усієї взаємодії. Для тих, які створюються і знищуються під час взаємодії, встановити ЛЖ, явно позначивши «створення» і «ліквідацію» об’єкта повідомленнями зі стереотипами); починаючи з повідомлення, яке ініціює взаємодію, розташувати всі наступні згори донизу між ЛЖ,

показуючи властивості кожного повідомлення (*зокрема, його параметри*), наскільки це необхідно для пояснення семантики взаємодії.

Якщо потрібно візуалізувати вкладеність повідомлень або моменту часу, у який виконується конкретне обчислення, доповнити ЛЖ кожного об'єкта його фокусом керування. Якщо потрібно специфікувати тимчасові або просторові обмеження, то слід пристосувати до кожного повідомлення тимчасову позначку і приєднати відповідні обмеження часу і простору. Якщо необхідно специфікувати потік керування більш формально, під'єднати до кожного повідомлення перед- і післяумови.

Окрема діаграма послідовності може показати тільки один потік керування (*допускається вираз структурної паралельності засобами керуючих структурних операторів*). Тому використовують багато діаграм взаємодії, одні з яких первинні, а інші показують альтернативні шляхи або виняткові умови. Для організації цих наборів ДП можна використовувати пакети, привсоюючи кожній діаграмі відповідне унікальне ім'я.

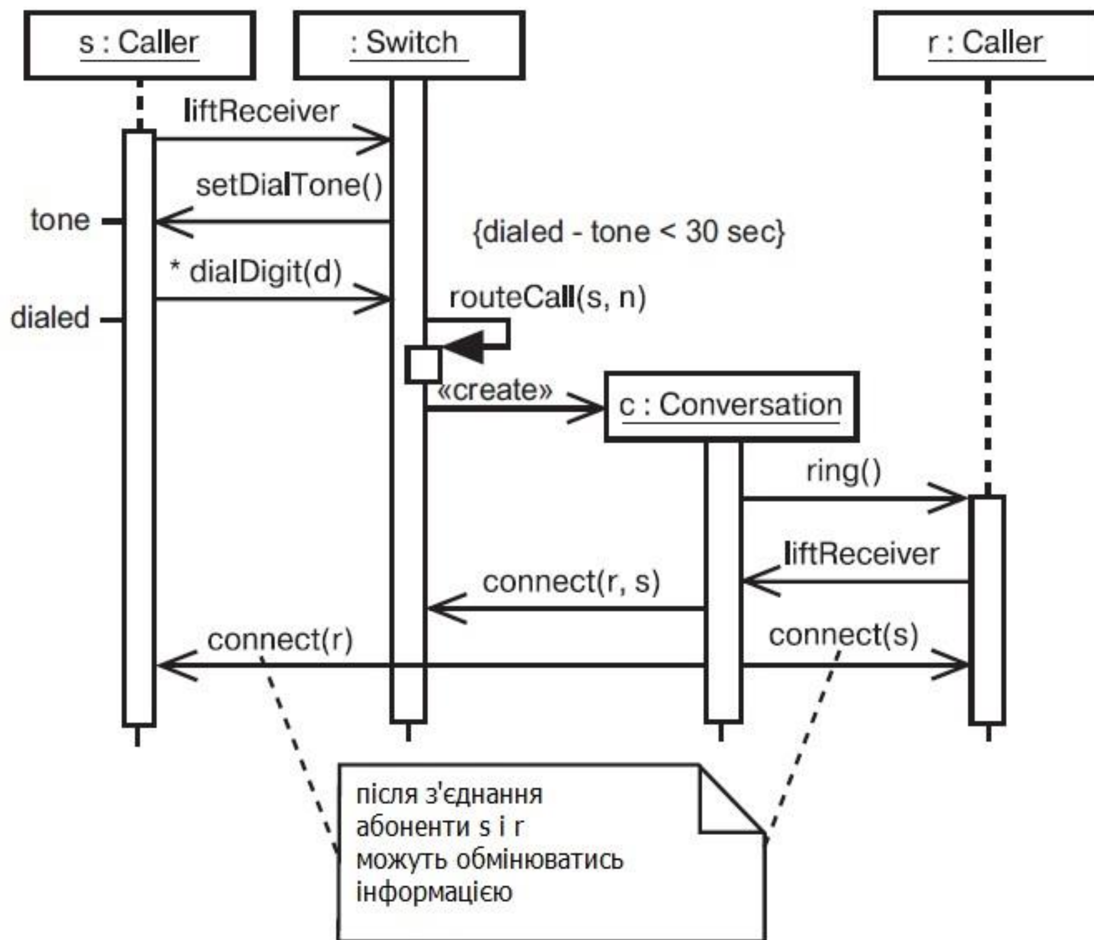


Рис. 3.2.13. Моделювання потоку управління в часі

На рис. 3.2.13 показана ДП, що описує потік керування, який ініціює простий телефонний дзвінок. На цьому рівні абстракції існують чотири ролі, залучені в процес: два абоненти (Callers) – s і r, безіменний телефон Switch і екземпляр із класу Conversation (Розмова) між двома абонентами. При тому, що діаграма моделює чотири ролі, кожен екземпляр діаграми має конкретні об'єкти, пов'язані з кожною із ролей. Той самий зразок взаємодії застосовується до кожного екземпляра діаграми. Послідовність

починається з того, що один Caller (абонент s) посилає сиґнал (liftreceiver – Зняти трубку) об'єкту Switch. У свою чергу, Switch посилає setdialtone (Довгий гудок) об'єкту Caller, який починає ітерацію повідомлень dialdigit (Набір номера). Обмеження показує, що ця послідовність повинна займати не більше 30 с. Діаґрама нічого не повідомляє при порушенні тимчасового обмеження.

Щоб показати наслідок цієї події, можна включити цілу гілку або ж окрему діаґраму послідовності. Потім об'єкт Switch викликає самого себе для виконання операції routecall (Передати виклик). Далі він створює об'єкт Conversation (c), якому доручає всю іншу роботу. Хоча це не показано, об'єкт c повинен виконувати додаткові обов'язки як частина механізму, що враховує вартість розмов рахунку і видає рахунки абонентам (даний механізм повинен бути описаний в іншій діаґрамі взаємодії). Об'єкт Conversation (c) дзвонить Caller (r), який асинхронно посилає повідомлення liftreceiver. Потім об'єкт Conversation говорить Switch, щоб той викликав операцію connect (з'єднати), і повідомляє обом абонентам, що потрібно виконати connect, після чого вони зможуть обмінюватися інформацією, як показано в примітці.

Діаґрама взаємодії може починатися або закінчуватися в будь-якій точці послідовності. Повне трасування потоку керування може бути надзвичайно складним, тому правильно виділити об'ємні потоки в окремі діаґрами.

Моделювання потоків керування по організації – виділяє структурні зв'язки між екземплярами у взаємодії поряд з повідомленнями, що між ними передаються. Для цього зручніше використати діаґраму комунікаціїї, показуючи передачу повідомлень у контексті структури. Необхідно: встановити контекст взаємодії (*системи, підсистеми, операції, класу або одного зі сценаріїв ВВ чи кооперації*); встановити основу взаємодії, ідентифікувавши об'єкти взаємодії, розташувати їх на діаґрамі комунікацій у вершинах графа, вміщуючи найважливіші об'єкти в центрі діаґрами, а вторинні – ближче до країв; специфікувати посилання між об'єктами поряд з повідомленнями, що пересилаються по них: спочатку розташувати посилання асоціацій (*вони найважливіші, тому що представляють структурні з'єднання*); зобразити інші зв'язки й надати їм відповідні **шляхові анотації** (такі, як global або local), показати у якому відношенні ці об'єкти; починаючи з повідомлення, що ініціює взаємодію, приєднати кожне наступне повідомлення до відповідного посилання, вказуючи відповідний порядковий номер, показати вкладеність повідомлень у нотації системи класифікації Дьюі. Якщо необхідно, вказати тимчасові й просторові обмеження, надати кожному повідомленню тимчасову позначку і приєднати потрібне тимчасове або просторове обмеження. Якщо потрібно, описати потік керування формальніше, приєднати до кожного повідомлення перед-і постумову.

Як і діаґрами послідовності, діаґрама комунікації може показати тільки один потік керування (*хоча деякі прості варіації можна відобразити в нотації взаємодій і розгалуження* UML). При використанні великої кількості взаємодій слід використовувати пакети.

На рис. 3.2.14 показана діаґрама комунікації, що описує потік керування в процесі реєстрації нового студента в навчальному закладі, з підкресленням структурних зв'язків між цими об'єктами. Тут є чотири ролі: Registraragent (Агент реєстрації – r), Student (Студент – s), Course (Курс – c) і безіменну роль School (Навчальний заклад). Потік керування явно пронумерований. Дія починається з того, що Registraragent створює об'єкт Student, додаючи його до навчального закладу (повідомлення addStudent – Додати студента), а потім, сповіщаючи об'єкт Student про те, що йому потрібно зареєструватися. Останній викликає операцію getSchеclule (Взнати розклад) і отримує

перелік об'єктів Course – курсів, на яких він повинен зареєструватися. Об'єкт Student додає себе до кожного об'єкта Course у цьому наборі.

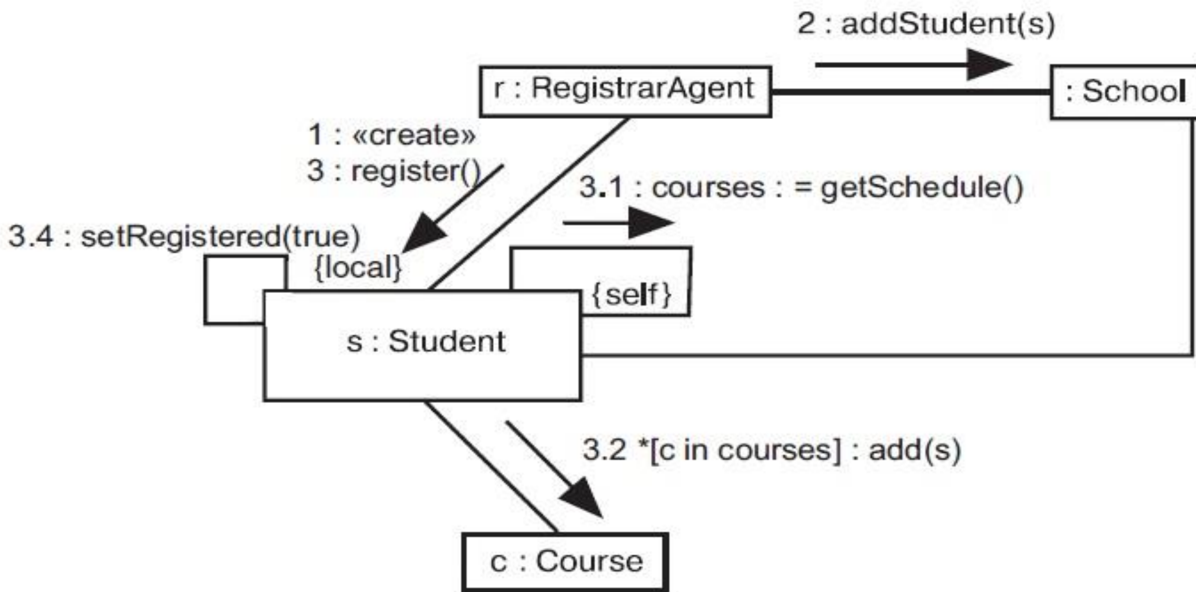


Рис. 3.2.14. Моделювання потоку управління по організації

Пряме і зворотне проектування. Застосовуючи попередню діаґраму комунікації, інструментальний засіб може зґенерувати наступний код операції register (реєструвати), включеної в клас Student мовою Java:

```
public void register()
{
    CourseCollection courses = getSchedule();
    for(int i = 0; i< courses.size(); i++)
           courses.item (i). add (t hiis);
    this.registered = true;
}
```

Інструментальний засіб зможе реалізувати метод getscheclule, що повертає об'єкт CourseCollection (Список курсів), обумовлений на основі сиґнатури операції. Проходячи по змісту об'єкта із застосуванням стандартної ідіоми ітерацій (*яка має бути відомою інструменту*), код можна узагальнити для будь-якої кількості запропонованих курсів.

Для ДП і діаґрам комунікації можливе і зворотне проектування, особливо контекст операції. Сегменти попередньої діаґрами могли бути відновлені інструментом із прототипу операції register. Цікавою задачею є анімація моделі на основі працюючої системи, зокрема анімація повідомлень, як вони передаються в системі. Можна керувати швидкістю виконання, встановлюючи точки переривання, призупиняючи процес у цікавих місцях і переглянути значення атрибутів конкретних об'єктів.

Добре структурована діаґрама взаємодії зосереджена на передаванні певного аспекту динаміки системи; містить у собі лише ті елементи, рівень деталізації, відповідні рівню абстракції та істотні для розуміння даного аспекту. Слід використовувати ДП, якщо потрібно підкреслити порядок повідомлень у часі. Діаґраму

комунікацій потрібно використовувати для організації об’єктів взаємодії; складні розгалуження виконувати на діаґрамах діяльності. [10,12,13]

3.2.3. Діаґрами діяльності

Основні питання:
- *Моделювання потоку робіт*
- *Моделювання операції*

Діаґрами діяльності – це один з п'яти видів діаґрам, які застосовуються в UML для моделювання динамічних аспектів систем, що показуть, як потік керування переходить від однієї діяльності до іншої (*подібно блок-схемам*). На відміну від традиційної блок-схеми діаґрама діяльності *показує паралелізм* так само добре, як і *розгалуження потоку керування*.

Метою моделювання динамічних аспектів систем за допомогою діаґрам діяльності є моделювання **послідовних** і **паралельних** кроків обчислювального процесу. Крім того, за допомогою діаґрам діяльності можна моделювати **потік передавання даних** від одного кроку процесу до іншого. Діаґрами діяльності можуть використовуватися окремо для моделювання динаміки кооперації об'єктів або потоку керування в операції.

Від одного кроку процесу до іншого. Якщо в діаґрамах взаємодії акцент робиться на *переходи потоку керування від одного об’єкта до іншого*, то діаґрами діяльності описують переходи потоку керування *від одного кроку процесу до іншого*. Діаґрами діяльності використовуються також в системах прямого і зворотного проектування.

Розглянемо потік робіт (workflow) *при будівництві будинку. Спочатку вибирають місце під забудову. Потім наймають архітектора, який проектує будинок. Після узгодження проекту забудовник складає кошторис для оцінювання загальних витрат. Після затвердження плану і ціни, починається будівництво. Отримуть дозвіл, риється котлован, заливають фундамент, зводять каркас і т.д., поки вся робота не буде завершена. Нарешті, замовник отримує ключі і документи на право власності і проживання.*

У реальних проектах багато різних процесів виконуються паралельно: електрики працюють одночасно з водопровідниками. В цих процесах є умови і розгалуження (залежно від якості ґрунту доведеться або виконувати підривні роботи, копат чи розмивати ґрунт). Можуть також бути присутні ітерації (будівельна інспекція виявила серйозні порушення, у результаті чого доводиться ламати і переробляти частину будинку). У будівельній промисловості для ВСКД робочого процесу застосовують діаґрами (графіки) Гантта і Перта.

При розробленні ПЗ виникає подібна проблема: яким чином найкраще змоделювати такі динамічні аспекти системи, як потік робіт або операцію? Тут є два основні шляхи (*аналогічні використанню діаґрам Гантта й Перта*).

З одного боку, можна побудувати розкадрування сценаріїв, які включають взаємодію ряду необхідних об'єктів і переданих між ними повідомлень. Ці розкадрування в UML можна моделювати двома способами: підкреслюючи часовий порядок повідомлень на діаґрамах послідовності або виділяючи структурні звязки між взаємодіючими об'єктами на діаґрамах комунікації. *Такі діаґрами взаємодії чимось подібні до графіків Гантта, які зосереджені на об’єктах (ресурсах), що забезпечують деяку діяльність протягом певного часу.* З іншого боку, ці ж динамічні аспекти можна моделювати за допомогою діаґрам діяльності, які зосереджують увагу, в першу чергу, на діяльності, до якої залучені об'єкти (рис. 3.2.15) (*такі діаґрами подібні до графіка Перта*).

Діаґрама діяльності є блок-схемою, яка відображає діяльність, що розгортається в часі, яку можна показати як діаґраму взаємодії, «*навиворіт*». Діаґрама взаємодії

розглядає об'єкти, через які проходять повідомлення; діаґрама діяльності *розглядає операції*, що виконуються між об'єктами. Семантична відмінність досить тонка, але в результаті отримуються принципово різні погляди на світ. Діаґрама діяльності показує *потік керування – від однієї діяльності до іншої*.

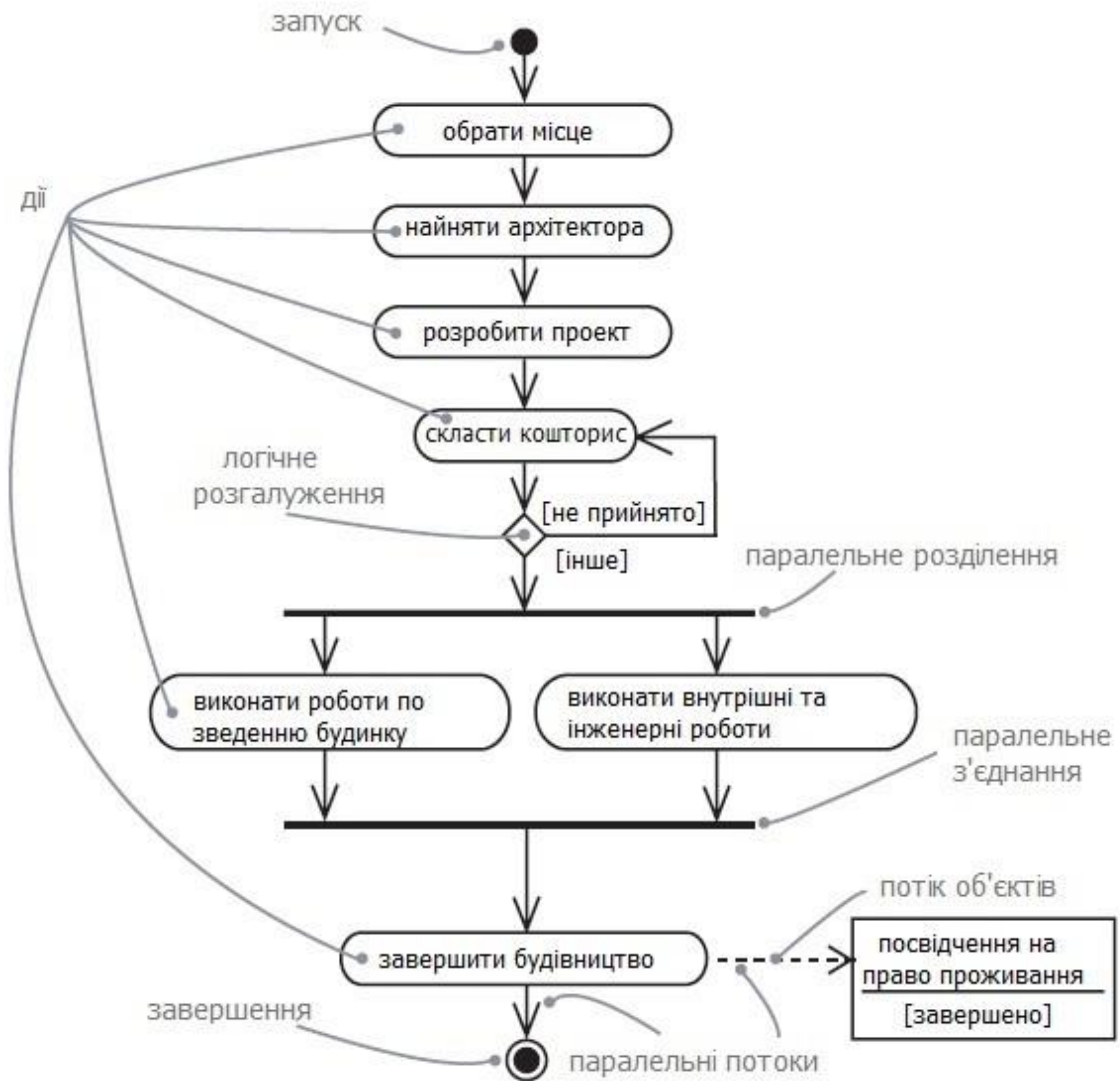


Рис. 3.2.15. Діаґрама діяльності

Діяльність (activity) – структурований опис поточної поведінки. Він представляє неатомарний набір дій всередині машини станів (автомата), що виконуються в цей момент. Виконання деякої діяльності в остаточному підсумку розкривається у вигляді виконання окремих **дій** (actions), кожна з яких може змінювати стан системи або передавати повідомлення. Дії полягають у виклику іншої операції, посилці сиґналу, створенні або знищенні об'єкта або у виконанні простих обчислень (*обчислення виразів*). Діаґрама діяльності є набором вузлів і дуг. [2,13]

Вміст діаґрами діяльності. Діаґрама діяльності має ряд властивостей, загальних для всіх діаграм (ім'я і графічне наповнення, що є проекцією моделі) та містить дії, вузли діяльності, потоки і значення об'єктів, примітки й обмеження.

Дії і вузли діяльності. У потоці керування, що моделюється діаґрамою діяльності, відбуваються певні дії: можна обчислити вираз, що встановлює значення атрибута або повертає деяке значення, викликати операцію об'єкта, послати йому сиґнал або створити чи знищити об'єкт. Ці атомарні обчислення (дії) зображується у вигляді овалу, всередині якого можна написати вираз (рис. 3.2.16).

Рис. 3.2.16. Дії

Дії не можуть бути піддані декомпозиції. Вони є атомарні (*події можуть відбуватися, але внутрішня поведінка дії невідома*). Не можна виконати частину дії: вона або виконується цілком, або не виконується взагалі. Часто передбачається, що час виконання дії несуттєвий, хоча деякі з них можуть вимагати відчутних витрат часу.

Вузол діяльності (activity node) – це *організаційна одиниця діяльності*. В загальному він є **вкладеною ґрупою дій** або інших вузлів. Вузли діяльності мають видиму підструктуру і, як правило, вимагають деякого часу на завершення. Дію можна зобразити як окремий випадок вузла діяльності, який не може бути підданий композиції. Аналогічно, вузол діяльності можна розглядати як *композицію*, потік керування якої *складається з інших дій і вузлів*. [2,10] Нотація дії і вузла діяльності однакова, за винятком того, що останній може мати додаткові частини, які підтримуються інструментом редагування за межами діаґрами (рис. 3.2.17).

Рис. 3.2.17. Вузли діяльності

Потоки керування. Коли деяка дія або вузол діяльності завершує виконання, потік керування негайно переходить до наступної дії або вузла діяльності. Цей потік зображується за допомогою стрілок, що показують його шлях від однієї дії або вузла до іншої. В UML потік керування представлений у вигляді *простої стрілки*, спрямованої від попередника до наступника, без **позначки події** (рис 3.2.18). Насправді потік керування повинен десь починатися і завершуватися, якщо тільки він не нескінченний. Можна виразити спеціальними символами **ініціалізацію** (*круг*) і **завершення** (*круг, вкладений в коло*).

Рис. 3.2.18. Завершення переходів

Розгалуження (branching). Крім послідовних потоків управління в діаґрами можна включати розгалуження, що специфікують альтернативні шляхи, обрані на основі булевих виразів. [2] Розгалуження може мати один вхідний потік і кілька вихідних. На кожному вихідному потоці міститься *булевий вираз умови*, яка обчислюється на вході з розгалуження. Умови вихідних потоків не повинні перекриватися (*якщо ні, то потік керування буде неоднозначним*), але при цьому повинні враховуватися всі можливі варіанти (*якщо ні, то потік керування виявиться «замороженим»*). Розгалуження зображується ромбиком (рис. 3.2.19).

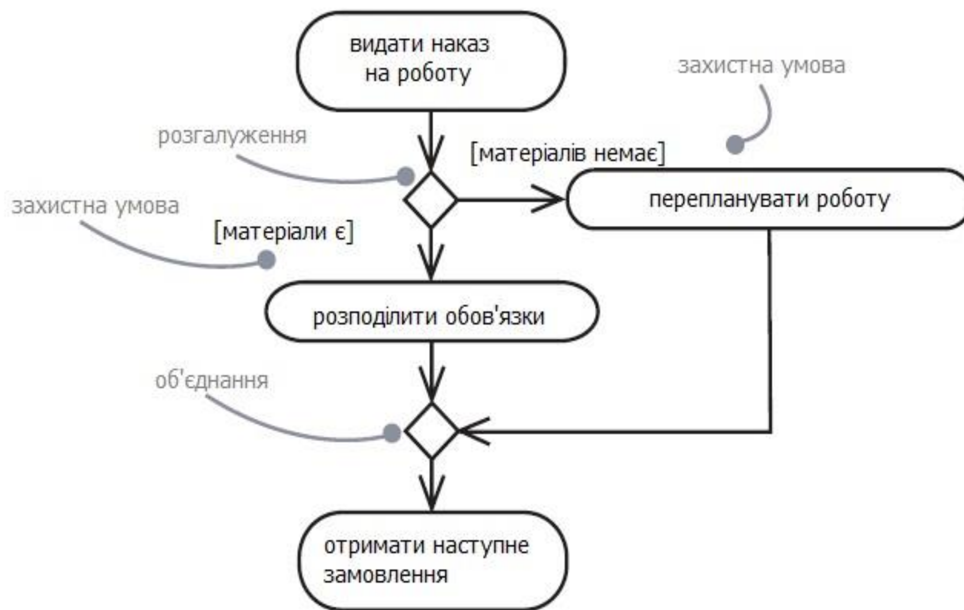


Рис. 3.2.19. Розгалуження

Для зручності можна використовувати ключове слово else, щоб позначити вихідне відгалуження потоку, яке виконується, якщо не істинний жоден з інших виразів. Коли два шляхи потоку керування знову сходяться разом, можна задіяти спеціальний символ – *ромбик із двома вхідними стрілками і однією вихідною*. Для об'єднання потоків ніяких умов не потрібно. Можна досягти ефекту ітерації, використовуючи одну дію, що встановлює значення ітератора і розгалуження, яке обчислює ознаку завершення ітерації. UML передбачає типи вузлів для циклів, але це найчастіше легше виразити текстовими, а не графічними засобами.

Поділ і з'єднання. Прості й розгалужені послідовні переходи зустрічаються на діаґрамах діяльності найчастіше. Однак іноді, особливо при моделюванні бізнес-процесів, можуть зустрітися і паралельні потоки. В UML для описування поділу і з'єднання паралельних потоків керування застосовують **лінійку синхронізації** (synchronization bar). Вона зображується у вигляді жирної горизонтальної або вертикальної лінії.

Моделювання паралельних потоків у системі. Розглядаються паралельні потоки в системі, яка імітує людську мову і жестикуляцію (рис. 3.2.20). **Поділ** (forking) є розщепленням одного потоку керування на паралельні. У цьому процесі присутні один вхідний і ряд незалежних вихідних потоків керування. Після поділу діяльність, асоційована з кожним з них, виконується паралельно. [12] Концептуально діяльності в кожному потоці паралельні, хоча в реальній працюючій системі вони можуть бути не

тільки паралельними (*якщо система розгорнута на кількох вузлах*), але й складатись із переривчастих послідовностей (*коли вона розгорнута на одному вузлі*), що створює ілюзію паралельності.

З'єднання (joining) є синхронізацією кількох паралельних потоків керування (рис. 3.2.20). З'єднання може мати ряд вхідних потоків і один вихідний. Перед з'єднанням потоків діяльності, які асоційовані з кожним з них, виконуються паралельно; у момент з'єднання потоки синхронізуються, тобто тепер кожен з них чекає, коли всі інші досягнуть точки з'єднання, починаючи з якої буде виконуватися один потік керування.

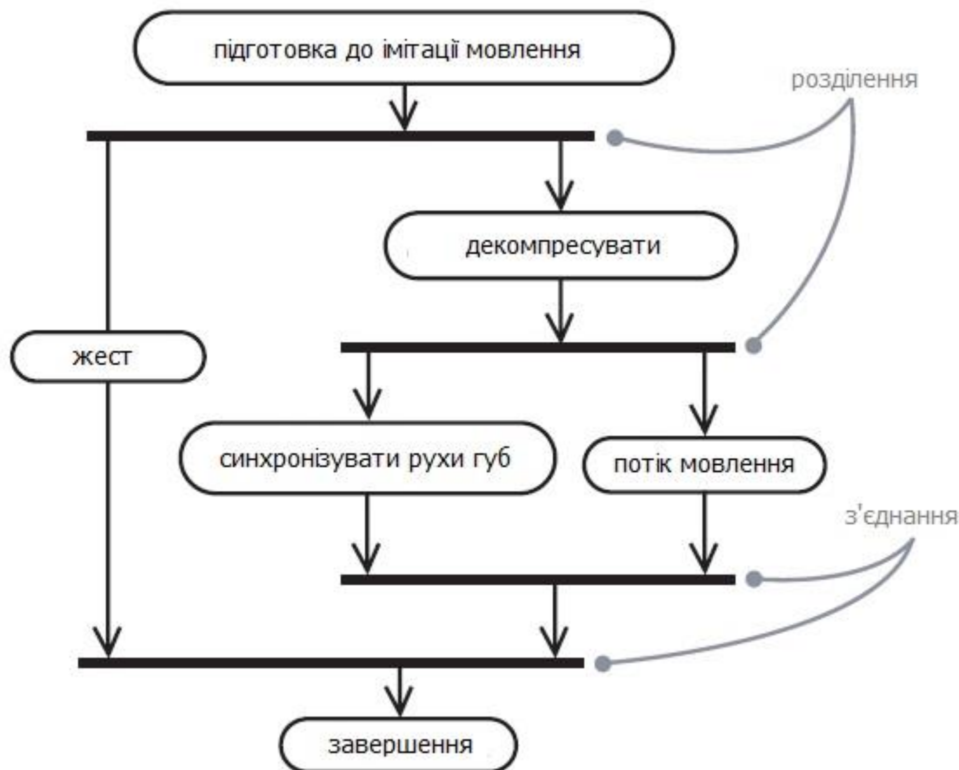


Рис. 3.2.20. Розділення і з'єднання

«Плаваючі доріжки». При моделюванні бізнес-процесів може виявитися, що на діаґрамах діяльності іноді необхідно розбивати стан діяльності на ґрупи, кожна з яких представляє відділ, що займається певною діяльністю. В UML такі ґрупи називаються *«плаваючими доріжками»* (swimlanes), де кожна ґрупа відділяється від сусідніх вертикальною лінією (рис. 3.2.21). Доріжки визначають набори діятельностей, яким притаманна деяка загальна організаційна властивість. Кожній доріжці присвоюється унікальне ім'я. Насправді доріжка не несе ніякої глибокої семантики – хіба що може відображати таку сутність реального світу, як, наприклад, організаційний підрозділ компанії.

Кожна доріжка – це **високорівневий обов'язок частини діяльності**, відображеної на діаграмі. Вона може бути реалізована у вигляді одного або кількох класів. На діаґрамі діяльності, розбитої на доріжки, кожна діяльність належить тільки одній з них, але переходи можуть перетинати границі доріжок.

Потік об'єктів. Об'єкти можуть брати участь у потоці керування, асоційованому з діаґрамою діяльності. Для послідовності операцій опрацювання замовлення (рис. 3.2.21), словник проблемної області, ймовірно, буде включати класи Order (Замовлення)

і Bill (Рахунок). Екземпляри цих двох класів будуть створені в результаті деякої діяльності – наприклад, діяльність Process order (Обробити замовлення) створить об'єкт Order), тоді як інші види діяльності можуть використовувати або модифікувати ці об'єкти.

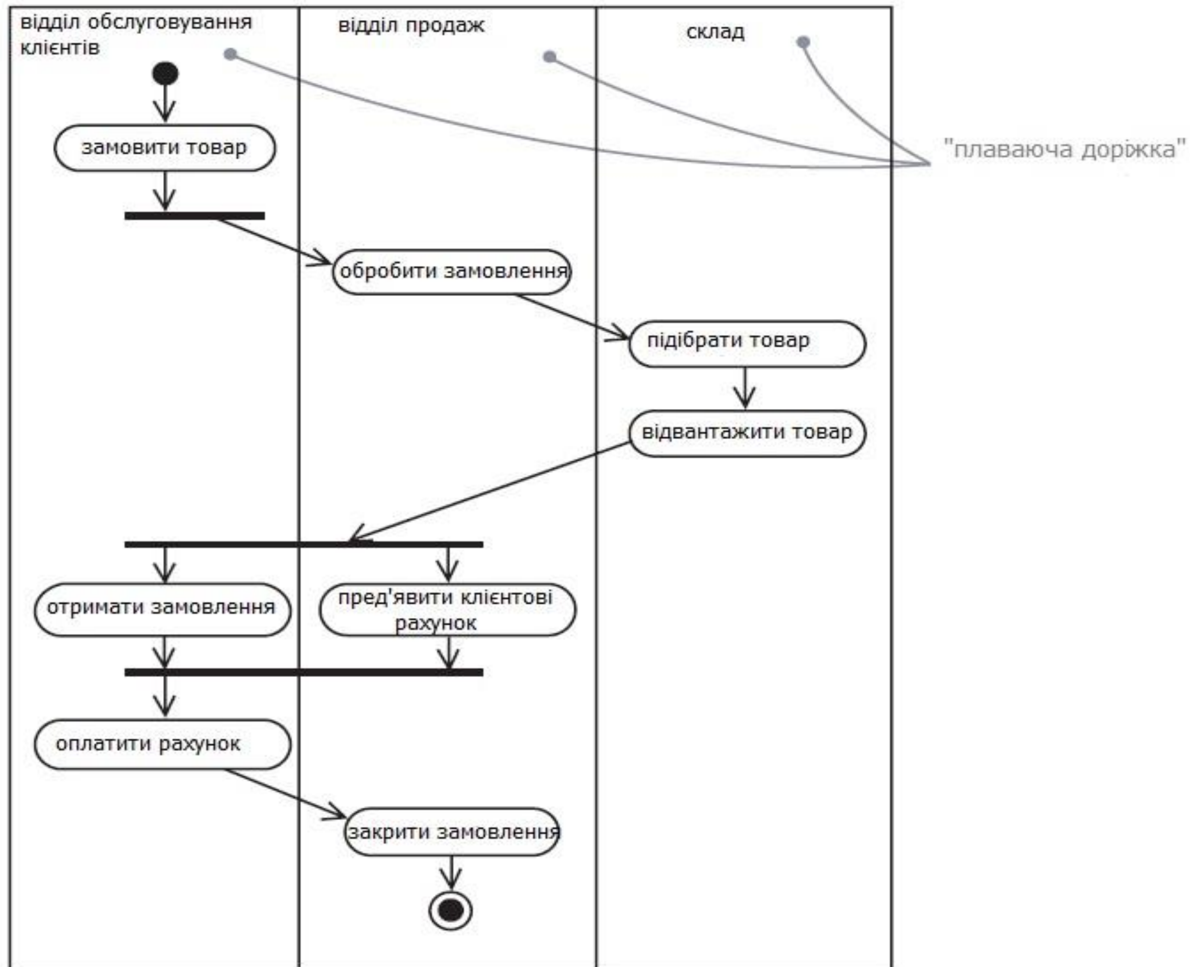


Рис. 3.2.21. Плаваючі доріжки

Наприклад, діяльність Ship order (Відвантажити замовлення) може змінити стан Order на filled (виконаний). Можна специфікувати сутності, які мають відношення до діаґрами діяльності, розмістивши їх на ній і з'єднавши стрілками з діяльностями, які їх створюють або використовують (рис. 3.2.22). Таке явище називають **потоком об'єктів** (object flow), тому що тут має місце *потік значень об'єкта від однієї діяльності до іншої*. Під потоком об'єктів, по суті, йдеться про наявність потоку керування (*неможливо виконати діяльність, яка вимагає значення, не маючи цього значення*), тому немає необхідності показувати потік керування між діяльностями, з'єднаними потоками об'єктів. Крім самого потоку об'єкта на діаґрамі діяльності можна показати, як змінюється його роль, стан і значення атрибутів. Для зображення стану об'єкта ім'я цього стану береться в дужки й міститься під іменем об'єкта (рис. 3.2.22).

Області розширення. Часто ту саму операцію потрібно виконувати для всіх елементів набору. Якщо замовлення містить у собі список позицій, оброблювач замовлення повинен виконати ту саму операцію для кожного рядка цього списку:

перевірити наявність товару, подивитися ціну, довідатися, чи обкладається дана позиція податком і т.д.

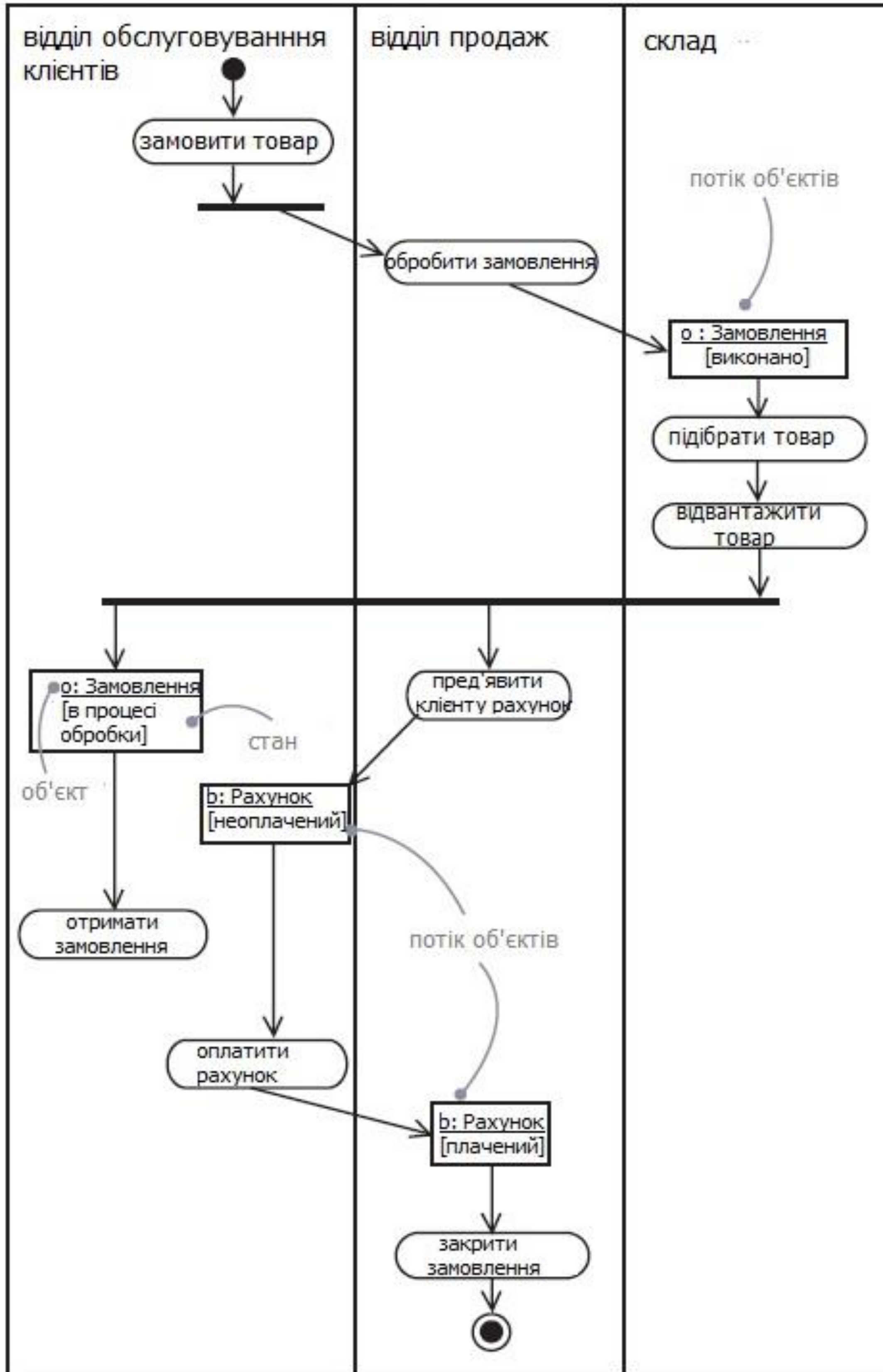


Рис. 3.2.22. Потік об’єктів

Операції над списками часто моделюються у вигляді циклів, але при цьому проектувальник повинен передбачити прохід по всіх позиціях, витягти їх усіх одну за іншою, виконати операцію, додати результат у вихідний масив, збільшити індекс і перевірити цикл на завершеність. Механіка виконання циклу приховує дійсний зміст операції. Цей зразок досить часто застосовується. Його можна змоделювати, застосовуючи **область розширення** (expansion région). [12,13]

Область розширення являє собою фрагмент моделі діяльності, який виконується для списку або набору елементів. На діаґрамі діяльності зображується пунктирною лінією, проведеної навколо області діаґрами. Введення в область розширення і виведення з неї є наборами значень (таких, як рядок списку елементів замовлення). Вхідні й вихідні набори зображуються у вигляді ряду маленьких квадратиків, з'єднаних один з одним; вони символізують масив значень. Коли значення масиву надходить у вхідну колекцію області розширення з попередньої частини моделі діяльності, воно розбивається на індивідуальні елементи.

Потім область розширення виконується для кожного елемента масиву. Немає необхідності моделювати ітерацію, – в області розширення вона вважається неявно. Притому опрацювання кількох елементів набору в міру можливості відбувається паралельно. Коли завершується опрацювання елемента, його вихідне значення, якщо воно є, вміщується у вихідний масив у тому ж порядку, в якому елементи розташовані у вхідному. Область розширення виконує операцію forall (для всіх) над елементами існуючого масиву, щоб створити новий.

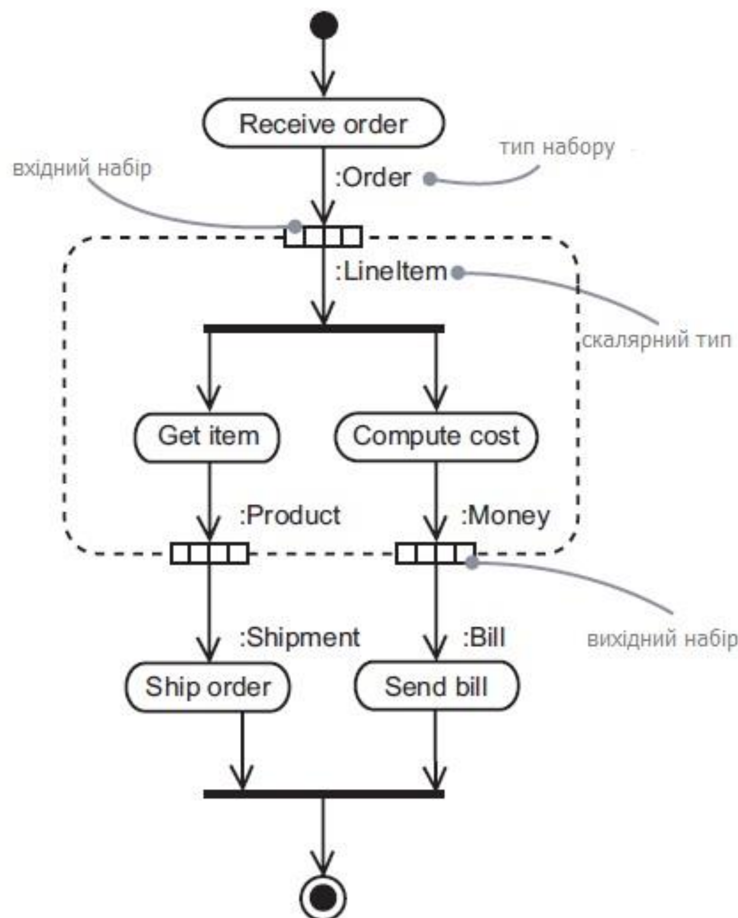


Рис. 3.2.23. Область розширення

У найпростішому випадку область розширення має один вхідний і один вихідний масиви, але в принципі в неї можуть бути один або кілька вхідних масивів і нуль або кілька вихідних. Усі ці масиви повинні бути одного розміру, але не зобов'язані включати тільки однотипні значення. Значення з відповідних позицій виконуються разом для генерації вихідних значень у тих же позиціях. Область розширення може мати нульове число виходів, якщо всі її операції дають якийсь побічний ефект відносно кожного елемента масиву. Області розширення дозволяють зобразити операції над індивідуальними елементами й над їхніми наборами в межах однієї діаґрами без необхідності показувати всі деталі пристрою механізму ітерацій.

Рис. 3.2.23 демонструє приклад області розширення. В основному тілі діаґрами ухвалюється замовлення. Ця подія генерує значення типу Order (Замовлення), яке складається зі значень типу LineItem (Стрічка замовлення). Значення Order служить вхідним для області розширення.

Кожне виконання області розширення працює з одним елементом з набору Order. Тому усередині області тип вхідного значення відповідає одному елементу масиву Order, а саме LineItem. Діяльність у межах області розширення розділяється на дві: одна знаходить Product (Продукт) і включає його в поставку, а інша обчислює вартість цієї позиції. Необов'язково обробляти елементи LineItem по-порядку; різні виконання області розширення можуть проводитись паралельно.

Динамічні аспекти поведінки системи можуть включати діяльність на будь-якому рівні абстракції в будь-якому представленні системної архітектури, включаючи класи (*у тому числі активні*), інтерфейси, компоненти і вузли. Діаґраму діяльності можна застосувати до ВВ (моделювання сценаріїв) та кооперацій об'єктів. При моделюванні динамічних аспектів системи діаґрами діяльності використовуються для:

- моделювання потоку робіт, де увага зосереджена на тому, як виглядає діяльність із погляду діючих осіб, що взаємодіють із системою. Потоки робіт перебувають на периферії програмних систем і застосовуються для ВСКД бізнес-процесів, які включає розроблювальна система. При такому використанні діаґрам діяльності особливе значення набуває моделювання потоків об’єктів.
- моделювання операції, де діаґрами діяльності виступають як схеми, що моделюють подробиці обчислювального процесу. При цьому особливо важливе моделювання розгалужень, поділу і з’єднання потоків керування. Контекст використовуваної в такий спосіб діаґрами діяльності включає параметри операцій і їх локальні об’єкти.

Моделювання потоку робіт. Програмна система функціонує в певному контексті, який включає актанти, що працюють нею. Особливо для критично важливого програмного забезпечення масштабу підприємства завжди можна виявити автоматизовані системи, що працюють у контексті більш високорівневих бізнес-процесів. Це є різновид **потоку робіт** (workflow), оскільки вони за своєю суттю являють потік робіт і об'єктів, що проходять через весь бізнес. У роздрібній торгівлі існують деякі автоматизовані системи: мережа касових терміналів взаємодіє із системами маркетингу, складськими системами і т.д. так само як і з людськими системами – продавцями, співробітниками відділів маркетингу, закупівель і поставок. Бізнес-процеси можна моделювати як спільну роботу всіх цих автоматизованих і людських систем, використовуючи діаґрами діяльності. Щоб змоделювати потік робіт, необхідно: встановити фокус потоку робіт (неможливо показати всі важливі потоки робіт нетривіальної системи в межах однієї діаграми); вибрати бізнес-об’єкти, що мають обов’язки найвищого рівня у відношенні всього потоку робіт або його частин (це

можуть бути реальні сутності зі словника системи або якісь абстрактніші. У кожному разі необхідно створити «плаваючу доріжку» для кожного важливого бізнес-об'єкта або підрозділу.); ідентифікувати передумови початкового стану робочого потоку й післяумови його кінцевого стану. Це досить суттєво для встановлення меж потоку робіт: починаючи зі стартового стану потоку робіт, специфікувати діяльності, що розгортаються в часі, і відобразити їх на діаґрамі. Складні дії або їх набори, які застосовуються багаторазово, показати у вигляді викликів окремих діаґрам діяльності. Зобразити потоки, які з'єднуються із цими діями і вузлами діяльності: почати з послідовних потоків, потім розглянути розгалуження й тільки в останню чергу – поділ і з'єднання. Якщо існують важливі значення об'єктів, залучених у робочий потік, відобразити їх на діаґрамі. Показати їхні змінні значення і стану, наскільки це необхідно для передавання призначення потоків об'єктів.

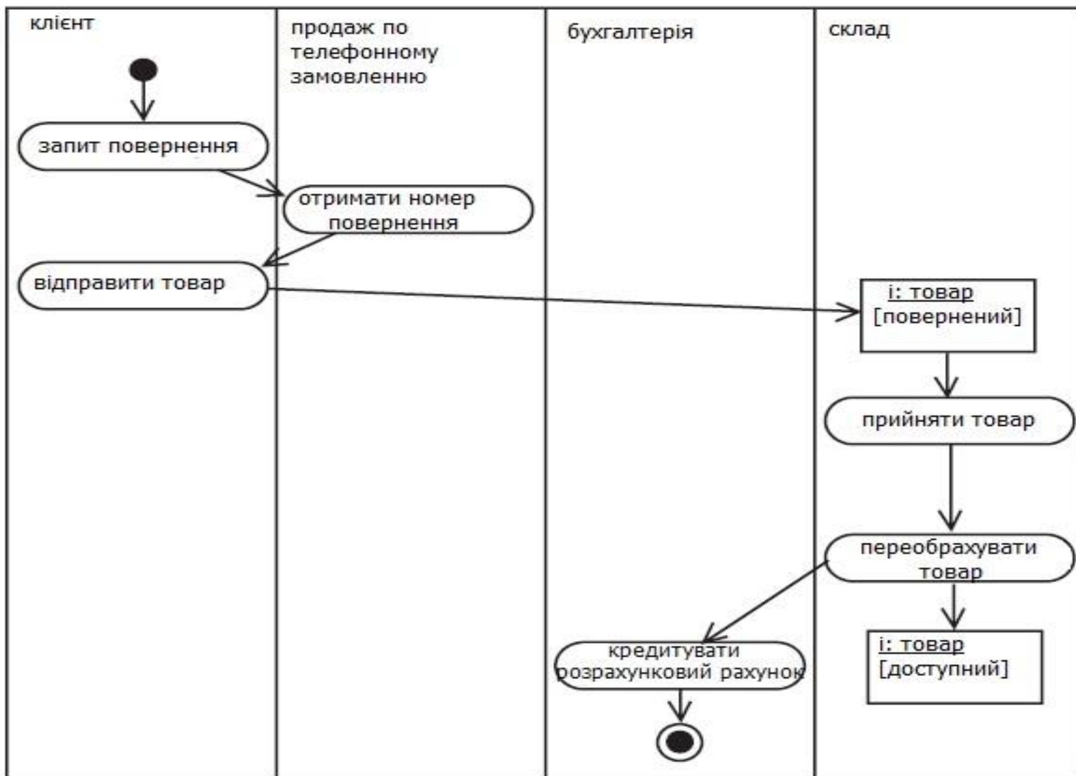


Рис. 3.2.24. Моделювання потоку робіт

На рис. 3.2.24 показана діаґрама діяльності для системи роздрібної торгівлі, яка специфікує потік робіт, що виникає при поверненні замовником продукту висланою поштою. Робота починається з дії Request Return (Повернення замовлення), яке виконує об'єкт Customer (Замовник), потім потік проходить через відділ Telesales (Поштові продажі) – Get return number (Одержати номер повернення), назад до замовника – Ship item (Відправити товар), потім потрапляє на Warehouse (Склад) – Receive item (Прийняти позицію) і Restock item (Поповнити запас) – і, нарешті, завершується у відділі Accounting (Бухгалтерія) – Credit account (Кредитувати рахунок). Як показано на діаґрамі, один важливий об'єкт – екземпляр Item (Позиція товару) – також подорожує в потоці, змінюючи свій стан від returned (повернутий) до available (доступний). Тут немає розгалужень, поділів і з'єднань, що є в складніших потоках робіт.

Моделювання операцій. Діаґрами діяльності можуть бути пов’язані з будь-яким моделюючим елементом для ВСКД його поведінки. Найчастіше елементом, для якого розробляється діаґрама діяльності, буде операція. Використана діаґрама виступає в якості блок-схеми дій, що виконуються даною операцією. Головна перевага діаґрами діяльності в тому, що всі елементи, зазначені на ній, семантично зв’язуються в одну модель. [2,13] Наприклад, будь-яка інша операція або сиґнал, на який посилається дія, може піддатися перевірці на відповідність класу цільового об’єкта. Щоб змоделювати операцію, необхідно: зібрати разом усі абстракції, що беруть участь у ній (*параметри і тип значення, що повертається, якщо таке є, атрибути класу, що включає й деякі сусідні класи*), ідентифікувати предумови початкового стану операції й післяумови її кінцевого стану, а також будь-які інваріанти класу, які включає, що повинні зберігатися протягом її виконання; з початку операції специфікувати всі діяльності і дії, що розвертаються в часі, і зобразити їх на діаґрамі разом зі станами діяльності й станами дій. За необхідності використовувати розгалуження для описування умовних шляхів та ітерацій. У випадку, якщо операцією володіє активний клас, використовувати поділ і з’єднання для показу паралельних потоків керування.

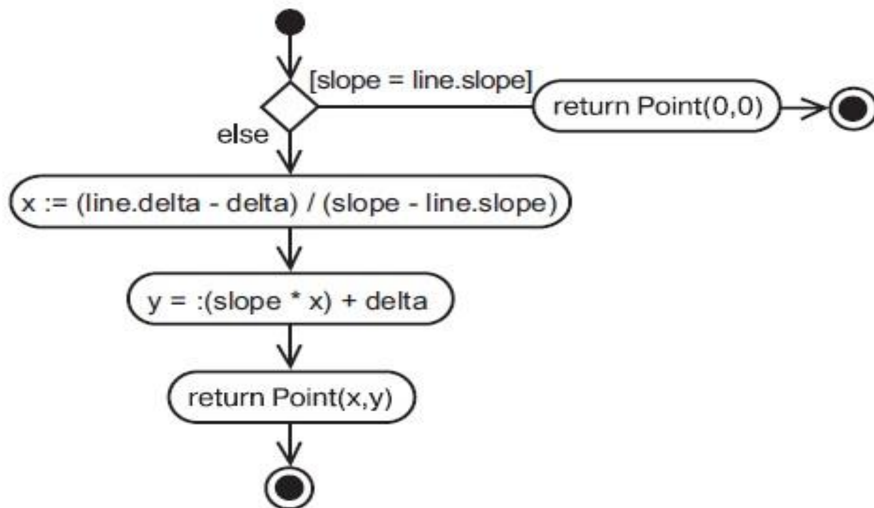


Рис. 3.2.25. Моделювання операції

На рис. 3.2.25 наведена діаґрама діяльності для класу Line (Лінія), що специфікує алґоритм операції intersection (перетинання), сиґнатура якої включає один параметр (line класу Line) і одне значення, що повертається (класу Point – Точка). Клас Line має два атрибути, що нас цікавлять: slope (нахил) і 1delta (зсув лінії відносно початку координат). Алґоритм операції простий. По-перше, є захисна умова, яка перевіряє, чи не рівний нахил slope даної лінії нахилу лінії-параметра line. Якщо це так, лінії не перетинаються й повертається точка Point(0,0). А якщо ні, то операція спочатку обчислює значення точки перетину x, потім – значення y. Як x, так і y – об’єкти, локальні стосовно операції. І, нарешті, повертається точка Point (x, y).

Використовуючи діаграму, що на рис. 3.2.25, можна згенерувати наступний код операції intersection на C++:

```
Point Line::intersection (Line : line){
  if (slope == line.slope) return Point(0,0);
```

```
  int x = (line.delta — delta) / (slope — line.slope);
  int y = (slope * x) + delta;
  return Point(x, y);
}
```

Зворотне проектування можливе для діаґрам діяльності для коду, що містить тіло операції. Згадувана діаграма могла бути згенерована на основі реалізації класу Line. Цікавіша в порівнянні зі зворотним проектуванням моделі з коду може виявитися анімація моделі за працюючою системою. Маючи в розпорядженні все ту ж діаґраму (рис. 3.2.25), інструмент міг би анімувати стан дій на ній в міру їх виконання в працюючій системі.

Добре структурована діаґрама діяльності повинна фокусуватися на вираженні лише одного аспекту динаміки системи; містити тільки елементи та рівень деталізації, важливі для розуміння даного аспекту. При створенні діаґрами діяльності необхідно починати з моделювання головного потоку, застосовувати розгалуження, паралельність і потік об'єктів – у другу чергу, на окремих діаграмах.

4. Основи моделювання подій

4.1. Моделювання подій сиґналів

Основні питання:

- *Події сиґналу, виклику, часу і зміни*
- *Моделювання серії сиґналів*
- *Моделювання виняткiв*
- *Опрацювання подій в активних і пасивних об'єктах*

Реальний світ насичений подіями, які дуже часто настають раптово та одночасно. В UML будь-яке явище, яке може мати місце в дійсності, моделюється як подія.

Усі системи, пов'язані з реальністю, тією чи іншою мірою є динамічними, причому динаміку зумовлюють саме події, що відбуваються всередині системи або за її межами. Роботу банкомата ініціює користувач, який натискає кнопку для здійснення банківської транзакції. Автономний робот починає діяти, коли зустрічається з деяким предметом. Мережевий маршрутизатор реагує на виявлення переповнення буферів повідомлень. На хімічному заводі сиґналом до дії стає закінчення періоду, необхідного для завершення реакції.

Подія (event) – це опис істотного факту, що відбувся в певному часі й просторі. Отримання сиґналу, закінчення проміжку часу, зміна стану – це приклади асинхронних подій, які можуть відбутися в будь-який момент. [2] У контексті кінцевих автоматів події використовуються для моделювання певного впливу, який може викликати перехід з одного стану в інший. До подій належать сиґнали, виклики, закінчення певного проміжку часу або зміна стану. Події можуть бути синхронними й асинхронними; їхнє моделювання – одна зі складових *моделювання процесів і потоків*. У сенсі автоматів подія означає вплив, який може викликати перехід з одного стану в інший.

Настання події – зміна станів. Графічна UML-нотація подій дозволяє реалізовувати оголошення подій у вигляді сиґналів (сиґнал OffHook (Трубка повішена)) і показати, як настання події призводить до переходу між станами: сиґнал OffHook викликає перехід телефону зі стану Active (Активний) у стан Idle (Очікування) й виконання дії drop connection (розірвати зв'язок) (рис. 4.1.1).

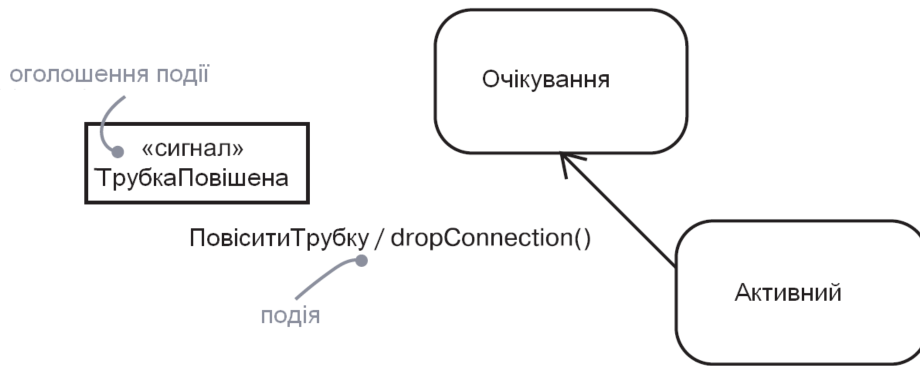


Рис. 4.1.1. Події

Сиґнал (signal) є *різновид події*, при використанні якого *повідомлення передається асинхронно* від одного екземпляра до іншого. Події можуть бути внутрішніми або зовнішніми. Зовнішні події передаються між системою і актантами (*натискання кнопки або переривання від сенсора запобігання зіткнень*), а внутрішні – між об'єктами, що існують у самій системі (*вимкнення, що генерується при переповненні*).

Види подій. В UML можна моделювати всі чотири види подій: **сиґнали**, **виклики**, **закінчення проміжку часу** і **зміна стану**.

Повідомлення є іменований об'єкт, який асинхронно посилається одним об'єктом і приймається іншим. *Сиґнал є класифікатором для повідомлень* і сам є типом повідомлення. У сиґналів є багато загального з класами. Можна говорити про екземпляри сиґналів. Сиґнали можуть брати участь у зв'язках узагальнення, що дозволяє моделювати **ієрархії подій**, де одні є загальними (*сиґнал* NetworkFailure (Збій мережі)), а інші – їх спеціалізованими версіями (WarehouSeserverFailure (Відмова складського сервера) є спеціалізаціяація події Networkfailure). Як і класи, сиґнали можуть мати атрибути й операції.

Моделювання сиґналів класами. Сиґнал може відправлятися в результаті виконання якоїсь дії в процесі **переходу** (*зміни стану*) в автоматі. Сиґнал можна моделювати як повідомлення, передане між двома ролями при деякій їхній взаємодії. При виконанні методу також можуть передаватися сиґнали. При моделюванні поведінки класів та інтерфейсів важливою частиною специфікації поведінки є **вказівки** сиґналів, які операції можуть їх посилати. [1] Сиґнали в UML моделюються *класами зі стереотипами* (рис. 4.1.2). Для моделювання вказівки, що деяка операція (moveTo()) посилає сиґнал, використовується залежність зі стереотипом «send» (*вказівка*).

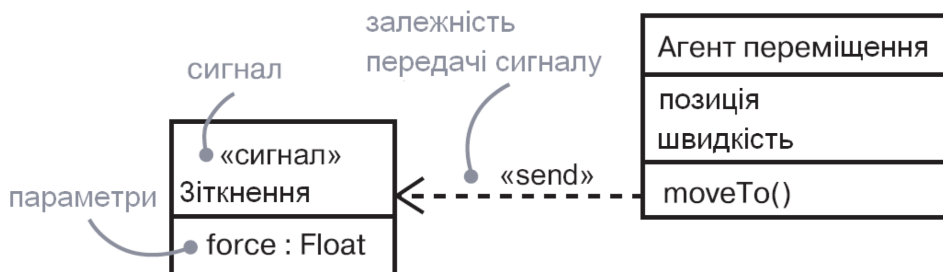


Рис. 4.1.2. Моделювання сиґналів класами

Подія сиґналу є його екземпляр (*як сам факт передавання сиґналу*).

Подією виклику (call event) є отримання **запиту** деяким об'єктом на виконання операції над ним. Виклик є *синхронною подією*, що використовується для запуску якоїсь операції. [1,2] Подія виклику може призвести до *переходу між станами* в автоматі або *виклику методу* на цільовому об'єкті. Конкретний варіант задається у визначенні операції класу.

Синхронність події виклику означає, що коли один об'єкт ініціює виконання операції на іншому об'єкті, у якого є свій автомат, керування передається від відправника отримувачеві, спрацьовує відповідний перехід, операція завершується, отримувач переходить у новий стан і повертає керування відправникові. [13] У випадках, коли відправник не має потреби чекати відповіді, виклик може бути визначений як асинхронний.

В моделі подія виклику не відрізняється від події сиґналу (рис. 4.1.3). В обох випадках подія разом зі своїми параметрами має вигляд *тригера для переходу стану*.

Рис. 4.1.3. Події виклику

Подія часу є закінченням якогось проміжку часу. В UML така подія моделюється за допомогою ключового слова after (після), за яким іде вираз, що визначає цей проміжок (рис. 4.1.4). Вираз може бути простим: after 2 seconds (*через 2 секунди*) або складним: after 1 ms since exiting Idle (*через 1 мс після виходу зі стану очікування*). Якщо явно не зазначене інше, відлік часу починається з моменту входу в поточний стан. Для позначення кінця періоду використовується слово at. Запис at (1 Jan 2012, 1200 UT) вказує на те, що подія часу закінчиться в перший день нового 2012-го року.

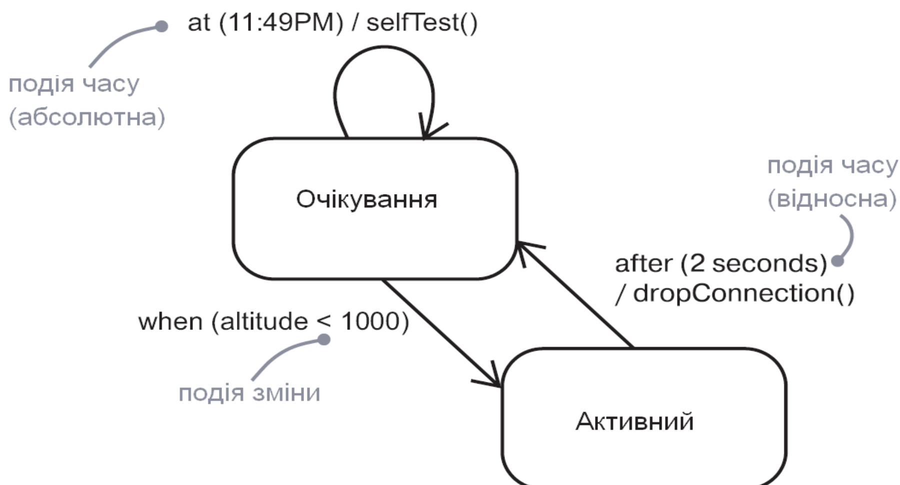


Рис. 4.1.4. Події часу і зміни

Подія зміни описує зміну стану або виконання деякої умови. В UML подія зміни моделюється за допомогою ключового слова when (*коли*), за яким іде булевий вираз (рис. 4.1.4). [2] Такий вираз може використовуватися для позначення абсолютного моменту часу (when time=11:59, «коли час=11:59») або перевірки умови (when altitude-1000, «коли висота<1000»).

Подія зміни відбувається один раз при зміні значення умови з неістинного на істинне (*а не навпаки*). Поки умова залишається істинною, подія не повторюється.

У подіях сиґналу і виклику беруть участь щонайменше два об'єкти: той, який посилає сиґнал або ініціює операцію, і той, якому подія адресована. Оскільки сиґнали по своїй природі асинхронні, а асинхронні виклики самі по собі є сиґналами, семантика подій перегукується із семантикою *активних* і *пасивних* об'єктів.

Екземпляр будь-якого класу може послати сиґнал об’єкту-отримувачу або викликати його операцію. Відправивши сиґнал отримувачеві, він продовжує свій потік керування, не чекаючи від нього відповіді. Після того, як дійова особа, що взаємодіє з банкоматом, пошле сиґнал pushbutton (*натиснути кнопку*), він може виконувати інші дії незалежно від того, що робить система, якій був посланий сиґнал. І навпаки, якщо об'єкт викликає операцію, він повинен дочекатися відповіді від отримувача. У трейдерській системі екземпляр класу Trader (Трейдер) може викликати операцію confirmtransaction (підтвердити транзакцію) у деякому екземплярі класу Trade (Угода), тим самим побічно змінивши стан останнього. Якщо це синхронний виклик, то об'єкт Trader буде чекати, поки операція закінчиться.

Стан «рандеву». Будь-який екземпляр будь-якого класу може бути отримувачем *події виклику* або *сиґналу*. Якщо це синхронна подія виклику, то відправник і отримувач перебувають у стані «*рандеву*» протягом усього виконання операції. Це означає, що **потік управління відправника блокується**, поки операція не завершиться. Якщо це сиґнал, то відправник і отримувач не входять у стан «*рандеву*»: відправник посилає сиґнал, але не чекає відповіді від отримувача. У кожному разі подія може бути загублена (*якщо явно не зазначено, що потрібна відповідь*), може викликати перехід стану в автоматі отримувача, якщо він існує, або ініціювати звичайний виклик методу.

Моделювання події виклику. В UML події виклику, які отримує об’єкт, моделюються як *операції класу* цього об'єкта. Іменовані сиґнали, отримувані об’єктом, моделюються шляхом перерахування в додатковому розділі класу (рис. 4.1.5).

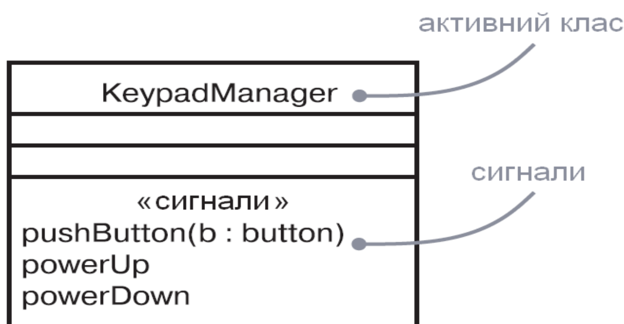


Рис. 4.1.5. Сиґнали й активні класи

У більшості систем, керованих подіями, події сиґналів утворюють ієрархію. Автономний робот може розрізняти зовнішні сиґнали, такі, як Collision (Зіткнення), і внутрішні, наприклад, Hardware Fault (Апаратна відмова). Однак багато зовнішніх і внутрішніх сиґналів можуть перетинатися. І навіть всередині цих двох широких областей класифікації можна виявити часткові різновиди. Hardware Fault можна розділити на Batteryfault (Відмова батарей) і Movement Fault (Відмова рухового механізму). Допускається й подальша спеціалізація. Різновидом сиґналу Movement Fault є Motorstall (Зупинка електромотора).

Поліморфні події. Моделюючи подібним чином ієрархії сиґналів, можна специфікувати поліморфні події. Нехай автомат, у якому деякий перехід спрацьовує

тільки при отриманні сиґналу Motorstall. Оскільки цей сиґнал є в ієрархії листовим, то перехід ініціюється тільки ним, так що поліморфізм відсутній. Для автомата, у якому перехід спрацьовує при отриманні сиґналу Hardwarefault, перехід буде поліморфний: його викликає як сам сиґнал Hardwarefault, так і будь-яка його спеціалізація (різновид), включаючи Batteryfault, Movement Fault і Motorstall.

Для моделювання множини сиґналів необхідно: розглянути всі різновиди сиґналів, на які може відповідати дана множина активних об'єктів; виявити схожі види сиґналів і вмістити їх в ієрархію типу «узагальнення/спеціалізація», використовуючи спадкування. На верхніх рівнях ієрархії будуть розташовуватися загальні сиґнали, а на нижніх – спеціалізовані; визначити можливості поліморфізму в автоматах цих активних об’єктів. При виявленні поліморфізму скорегувати ієрархію, додавши, якщо буде потреба, проміжні абстрактні сиґнали.

На рис. 4.1.6 наведена множина сиґналів, які б могли оброблятися автономним роботом. Кореневий сиґнал Robotsignal (Сиґнал робота) є абстрактним, тобто безпосереднє створення його екземплярів неможливе. У цього сиґнала є дві конкретні спеціалізації (Collision і Hardwarefault), одна з яких (Hardwarefault) спеціалізується й далі. В сиґнала Collision є один параметр.

Важлива частина ВСКД поведінки класу або інтерфейсу – виявлення аварійних ситуацій, які можуть породжувати його операції. Для певного класу чи інтерфейсу операції можуть бути зрозумілі з опису, але може бути незрозуміло, які аварійні ситуації вони можуть викликати, якщо це явно не зазначене в моделі.

В UML аварійні ситуації є різновидом подій і моделюються як сиґнали. Події-помилки можна приєднати до специфікації операцій. Моделювання винятків у деякому змісті протилежне моделюванню сімейств сиґналів. Основна мета останнього – специфікувати, які сиґнали можуть отримувати активні об’єкти; мета моделювання аварійних ситуацій полягає в тому, щоб показати, які аварійні ситуації може породжувати об’єкт.

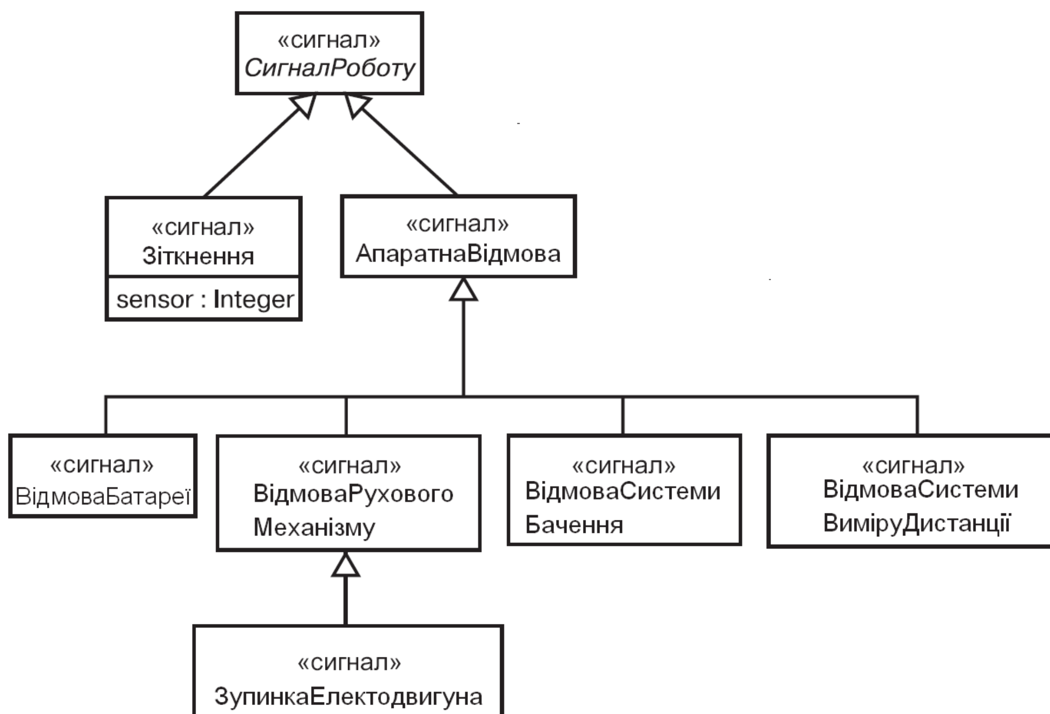


Рис. 4.1.6. Моделювання сімї сиґналів

Щоб виконати це завдання, потрібно для кожного класу й інтерфейсу та для кожної певної в них операції розглянути нормальні й аварійні ситуації та змоделювати їх у вигляді сиґналів, переданих між об'єктами. Організувати ієрархію сиґналів: на верхніх рівнях розмістити загальні сиґнали, на нижніх – спеціалізовані, і за необхідності ввести проміжні виключення. Вказати для кожної операції, які аварійні ситуації (сиґнали) вона може породжувати. Це можна зробити явно на діаграмі (провівши залежності зі стереотипом send від операції до її сиґналів) або ж використовувати діаграми послідовності, що зображують різні сценарії.

На рис. 4.1.7 представлена модель ієрархії аварійних ситуацій, які можуть породжуватися контейнерними класами зі стандартної бібліотеки, наприклад, класом-шаблоном Set (Установка). У корені цієї ієрархії перебуває абстрактний сиґнал Set Error (Встановити помилку), а нижче розташовані спеціалізовані види помилок: Duplicate (Дублювання), Overflow (Переповнення) і Underflow (Втрата значущості). Очевидно, операція add (додати) може породити сиґнали Duplicate і Overflow, а операція remove (вилучити) – тільки сиґнал Underflow. Замість цього можна було б забрати залежності з переднього плану, перелічивши їх у специфікації кожної операції. Знаючи, які сиґнали може породити операція, можна успішно створювати програми, що використовують клас Set.

При моделюванні подій слід будувати ієрархію сиґналів так, щоб можна було скористатися загальними для них властивостями. Не забувайте асоціювати підходящий автомат з кожним елементом, який може отримувати події. Слід обов'язково моделювати не тільки ті елементи, які можуть отримувати події, але й ті, які можуть їх посилати. Зображуючи подію чи ім'я, у загальному випадку слід моделювати ієрархії подій явно, а їх використання специфікувати на задньому плані тих класів або операцій, які посилають або приймають подію.

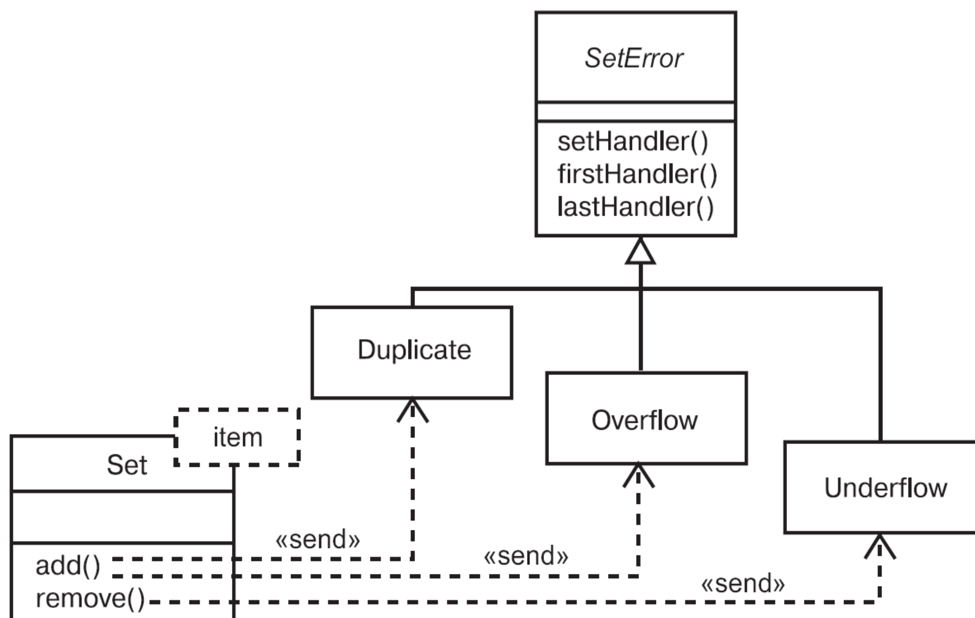


Рис. 4.1.7. Моделювання умов помилки

4.2. Кінцеві автомати. Моделювання поведінки працюючих спільно об’єктів

Основні питання:

- *Стани, переходи і діяльності*
- *Моделювання ЖЦ об’єкта*
- *Створення добре структурованих алґоритмів*

Використовуючи взаємодію, можна моделювати поведінку набору об’єктів, що спільно працюють, з допомогою автоматів. Здебільшого цей процес включає специфікацію ЖЦ екземплярів класу, ВВ або системи в цілому. Ці екземпляри можуть реагувати на такі події, як сиґнали, операції або закінчення якогось періоду часу. Коли відбувається подія, залежно від поточного стану об’єкта спостерігається деякий ефект.

Розглянемо ЖЦ домашнього термостата в погожий осінній день. Вранці ніщо не тривожить його спокій, температура в будинку стабільна. Ближче до світанку спостерігається інша картина. Сонце встає, і температура повітря на вулиці злегка підвищується. Починають прокидатися члени родини; хтось встав з ліжка і підкрутив регулятор. Обидві ці події важливі для системи терморегуляції будинку. Термостат починає поводитися як усі пристрої його класу: віддає команду або обігрівачу (щоб підігріти повітря), або кондиціонеру (щоб його остудити). Після того як мешканці пішли з дому, рух припиняється і температура знову стає постійною. Але тут може втрутитися автоматика, яка наказує термостату знизити температуру, щоб не витрачати даремно електроенергію і газ. Термостат знову береться за роботу. Ближче до кінця дня автоматична програма знову «оживає», велівши підняти температуру: мешканці в повному зборі, до цього моменту будинок повинен стати затишним. Ввечері для підтримання теплового режиму в будинку термостат може вмикати обігрівач чи кондиціонер…

Багато програмних систем поводяться подібно термостату. Серцевий стимулятор працює цілодобово, адаптуючись до змін кров’яного тиску чи виконуваної людиною роботи. Мережевий маршрутизатор працює безупинно, непомітно перенаправляючи потоки бітів, змінюючи свою поведінку у відповідь на команди адміністратора мережі. Стільниковий телефон також працює за різними запитами…

Динамічні аспекти системи в UML моделюються **кінцевими автоматами** (state machines). Автомат – це поведінка, яка специфікує послідовність станів об'єкта, через які він проходить протягом свого ЖЦ у відповідь на події, а також реакції на ці події. [2]

У той час, як взаємодія моделює співтовариство об'єктів, що спільно працюють для виконання деяких дій, автомат моделює ЖЦ окремого об’єкта – екземпляр класу, ВВ або навіть усієї системи. За час ЖЦ в об’єкті може відбуватися безліч різноманітних подій, таких, як передавання і приймання сиґналів, виклики операцій, створення та знищення об’єкта, закінчення періоду часу, відведеного на якусь дію, або зміна якихось умов. У відповідь на ці події об’єкт виконує певну дію, що являє собою обчислення, а потім змінює свій стан на інше. Тому поведінка такого об’єкта залежить від минулого, принаймні в тому ступені, у якому воно впливає на його поточний стан. Об’єкт може прийняти подію, відповісти на нього дією, потім змінити свій стан. Коли ж він приймає іншу подію, його реакція може бути іншою – залежно від поточного стану, який є результатом попередньої події.

Події, стани, переходи й ефекти в ЖЦ об’єктів. Автомати використовуються для моделювання поведінки будь-якого елемента, найчастіше класу, ВВ або всієї системи. Динаміку виконання деякого процесу можна візуалізувати двома способами: визначаючи потік керування від однієї діяльності до іншої (*діаграми діяльності*) або виділяючи потенційні стани об’єкта і переходи між ними (*діаграми станів*). Автомати можуть бути добре візуалізовані на діаграмах станів. Можна зосередитися на поведінці

об'єкта, що залежить від послідовності подій, що особливо зручно для моделювання реактивних систем. Добре структуровані автомати подібні добре структурованим алґоритмам: вони ефективні, прості, адаптовані й зрозумілі.

Графічна нотація *станів*, *переходів*, *подій* і *ефектів* в UML дозволяє показати поведінку об'єкта таким чином, щоб виділити важливі елементи його ЖЦ (рис. 4.2.1).

Кожен об'єкт має свій ЖЦ: спочатку створюється, потім припиняє своє існування. Між цими двома подіями об'єкт може впливати на інші об'єкти, посилаючи їм повідомлення, а також попадати під їхній вплив, приймаючи повідомлення від них. У багатьох випадках такі повідомлення являють собою прості синхронні виклики операцій. Наприклад, екземпляр класу Customer (Покупець) може викликати операцію get Account Balance (Запитати баланс) на екземплярі класу Bank account (Банківський рахунок). Подібним об'єктам не потрібний автомат для описування їх поведінки, оскільки їх поточна поведінка не залежить від минулої.

У системах інших типів можна зустріти об'єкти, які повинні відповідати на **сиґнали**, що являють собою *асинхронні повідомлення*, передані між екземплярами. Мобільний телефон повинен відповідати на випадкові телефонні виклики з інших телефонів, на події натискання клавіш користувачем, сиґнали з мережі. Крім того, можна зустріти об'єкти, поведінка яких залежить від минулого. Наприклад, поведінка системи керованої ракети «повітря-повітря» залежить від її поточного стану, такого, як Not Flying (Не летить) – не надто гарна ідея запускати ракету з літака, що знаходиться на землі, – або Searching (Пошук): не варто пускати ракету, поки не обрана ціль.

Поведінка об'єкта, який повинен відповідати на асинхронні повідомлення або його поточну поведінку залежить від минулого, найкраще специфікувати за допомогою автоматів. Це стосується екземплярів класів, які можуть приймати сиґнали, включаючи багато активних об'єктів. Фактично об'єкт, який приймає сиґнал, але не має для нього переходу у своєму поточному стані й не відкладає реакцію на сиґнал у цьому стані, буде просто його ігнорувати. Інакше кажучи, відсутність переходу для сиґналу не є помилкою; це означає, що в даній точці сиґнал не представляє інтересу. Ви також будете використовувати автомати для моделювання поведінки систем у цілому, особливо реактивних, які повинні відповідати на сиґнали діючих осіб за межами системи.

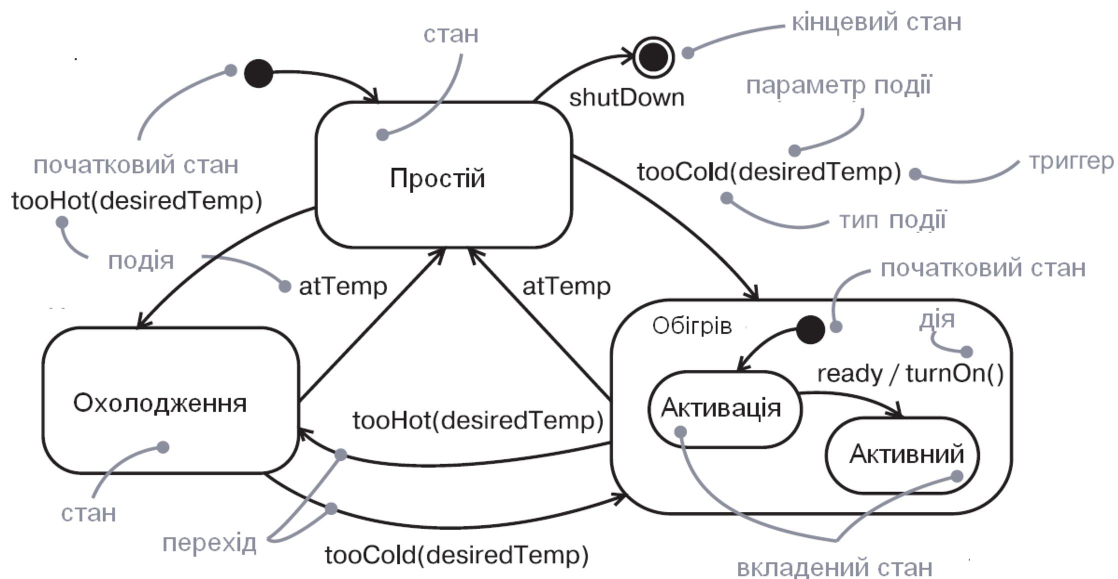


Рис. 4.2.1. Автомати

Стан (state) – це ситуація в ЖЦ об'єкта, у якій він задовольняє задані умови, здійснює якусь діяльність або очікує певної події. Об'єкт перебуває в тому або іншому стані обмежений період часу. [13] *Обігрівач у будинку може бути в кожному із чотирьох станів: простий (очікування команди на ввімкнення обігріву), активація (газ поданий, але очікується досягнення певної температури), функціонування (газ і казан ввімкнені), вимикання (подача газу припинилася, але казан увімкнений, залишкове тепло скидається із системи).*

Коли автомат об'єкта перебуває в певному стані, то кажуть, що в цьому стані перебуває сам об'єкт. Наприклад, екземпляр Heater (Обігрівач) може здійснювати Idle (Простій) або Shuttingdown (Вимикання).

Стан характеризується кількома частинами:

- **ім'я –** текстовий рядок, що відрізняє даний стан від інших. Стан може бути анонімним, тобто без імені;
- **вхідний/вихідний ефекти –** дії, виконувані відповідно при вході і виході зі стану;
- **внутрішні переходи –** переходи, які опрацьовуються без зміни стану;
- **підстани –** вкладені стани, у тому числі неортогональні (послідовно активні) або ортогональні (паралельно активні) підстани;
- **відкладені події –** список подій, які не опрацьовуються в даному стані, але відкладаються і містяться в черзі для опрацювання об'єктом в іншому стані.

Стан зображується у вигляді прямокутника із заокругленими кутами (рис. 4.2.2).

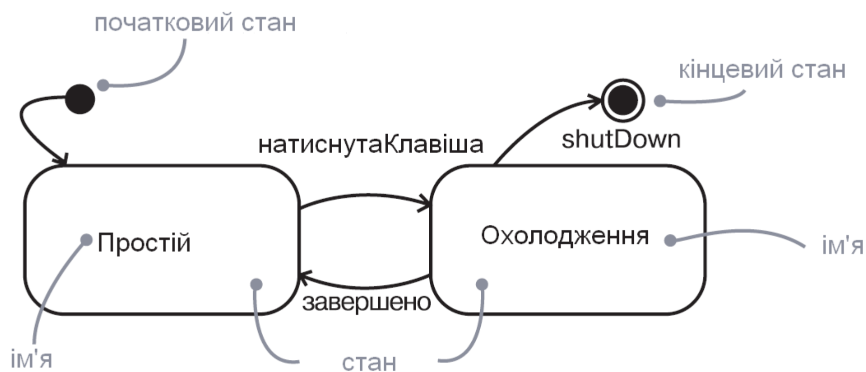


Рис. 4.2.2. Стани

Є два особливі стани, які можуть бути визначені для автомата об'єкта. Перший з них – початковий стан, який означає початкову точку, за замовчуванням встановлену для автомата або підстану. Він зображується зафарбованим чорним кругом. Другий стан – кінцевий; він означає, що робота автомата завершена. Кінцевий стан представлений у формі зафарбованого чорного круга, вписаного в коло.

Перехід (transition) – це зв'язок між двома станами, який означає, що об'єкт у першому стані повинен виконати певні дії й перейти в другий стан, коли відбудеться певна подія та будуть задоволені задані умови. [2] Доти, поки перехід не відбувся, про об'єкт кажуть, що він перебуває у **вихідному стані;** після переходу він перебуває в **цільовому стані.** *Наприклад, Heater (Обігрівач) може перейти зі стану Idle (Простій) у стан Active (Активація), коли відбудеться подія toocold (надто холодно) з параметром desiredtemp (потрібна температура).*

Перехід складається з п’яти частин:

□ **вихідний стан –** стан, на який впливає перехід. Якщо об’єкт перебуває у вихідному стані, то вихідний перехід може відбутися, коли об’єкт прийме подію, що ініціює перехід, і буде виконана захисна умова (якщо вона є);

□ **ініціююча подія** – подія, яка, будучи розпізнана об'єктом у вихідному стані, ініціює виконання переходу, забезпечуючи істинність його захисної умови;

□ **захисна умова** – булевий вираз, який обчислюється при ініціюванні переходу після отримання події. Якщо вираз дійсний, то виконання переходу дозволяється. А якщо ні, то перехід не виконується, і якщо немає інших переходів, які можуть бути ініційовані тією ж подією, то подія втрачається;

□ **ефект** – поведінка, що виконується (подібна до дії), яка може бути притаманною об’єкту, що володіє даним автоматом, і опосередкованою – з іншими об'єктами, які видимі даному;

□ **цільовий стан** – стан, у якому виявляється об’єкт після завершення переходу.

Як показано на рис. 4.2.3, перехід зображується у вигляді суцільної лінії зі стрілкою, спрямованої від вихідного стану до цільового. **Перехід у себе** (self-transition) – це перехід, вихідний і цільовий стани якого збігаються.

Подія (event) в контексті автоматів - це вплив, який може викликати перехід з одного стану в інший. Під подіями можуть вважатися: вступ сиґналу, виклик, закінчення певного періоду часу або зміна стану (рис. 4.2.3). [2,13] Сиґнал або виклик можуть мати параметри, які доступні переходу, у тому числі вираження захисної умови та дії. Допускається існування завершального переходу (тобто такого, який не має ініціюючої події). Завершальний перехід ініціюється неявно, коли вихідний стан завершує свою діяльність, якщо така спостерігається.

Захисна умова представлена у вигляді булевого виразу, укладеного у квадратні дужки і вміщено після тригера події (рис. 4.2.3). Таким чином, допускається наявність множинм переходів з одного вихідного стану, з тим самим тригером події, доти, поки умови не перекриваються. Захисна умова обчислюється лише один разу для кожного переходу в момент виникнення події, але вона може обчислюватися і повторно, якщо перехід ініціюється ще раз. Всередині булевого виразу можна задати умови перебування об’єкта в тому або іншому стан, наприклад, вислів «Обігрівач у стані простою» набуває значення «істина», якщо об’єкт Heater (Обігрівач) перебуває в стані Idle (Простій). Якщо умова виявляється не дотриманою при його перевірці, то подія не повторюється пізніше, коли умова стане дійсною. Щоб змоделювати таку поведінку, слід використовувати подію зміни.

Ефект – це поведінка, яка має місце, коли ініціюється перехід. Під ефектами може йтися про вбудовані обчислення, виклики операцій (для об’єкта, що володіє автоматом, а також для інших видимих об’єктів), створення й знищення іншого об’єкта або передавання сиґналу деякому об’єкту. Щоб відзначити факт відправлення сиґналу, можна постачати ім’я сиґналу префіксом – ключовим словом send.

Переходи допускаються тільки за умови, що автомат неактивний (перебуває в стані спокою), тобто не виконується ефект (дія) від попереднього переходу. Спрацьовування ефекту переходу в будь-яких асоційованих вхідних і вихідних ефектів повинно завершитися перш, ніж будь-яким додатковим подіям буде дозволено викликати додаткові переходи. У цьому полягає відмінність від виконання діяльності, яка може перериватися деякими подіями.

Розширені UML – засоби моделювання поведінки. Моделювання різної поведінки в UML у більшості випадків вимагає використання тільки базових засобів

станів і переходів UML. Застосовуючи їх, можна специфікувати простий автомат, що описуватиме поведінкові моделі лише дугами (*переходами*) і вершинами (*станами*).

Автомати UML включають множину засобів, що допомагають управляти *складними поведінковими моделями*. Ці засоби часто дозволяють зменшити кількість необхідних станів і переходів, а також змоделювати багато загальних і іноді досить складних ідіом, які довелося б відображати у вигляді простих автоматів. Деякі із цих розширених засобів включають вхідні й вихідні ефекти, внутрішні переходи, діяльності й відкладені події. Зображуються вони у вигляді рядків тексту всередині піктограми стану (рис. 4.2.4).

Іноді модельована ситуація вимагає виконання деякої настановної дії при вході в стан – незалежно від того, який перехід привів вас сюди. Аналогічно, коли об'єкт залишає стан, потрібно виконати деяке очищення незалежно від того, який перехід з нього веде. У системі керування ракетою можна побажати явно оголосити ефект ontrack (веде переслідування), коли система входить у стан Tracking (Переслідування), і offtrack (припинити переслідування) – коли залишає його. Використовуючи простий автомат, можна домогтися подібного ефекту, вміщуючи ці дії відповідно на кожному вхідному і вихідному переході. Однак тут велика ймовірність помилитися: необхідно постійно пам’ятати, що дії потрібно створювати при додаванні кожного нового переходу. Більше того, модифікація дії означає, що будуть порушені всі сусідні переходи. UML передбачає скорочення для цієї ідіоми (рис. 4.2.4). У піктограму стану можна включити вхідний ефект (позначений ключовим словом entry) і вихідний ефект (позначений ключовим словом exit), – кожен із вказівкою на відповідну дію. Щоразу при вході в стан викликається вхідна дія, а при виході з нього – вихідна.

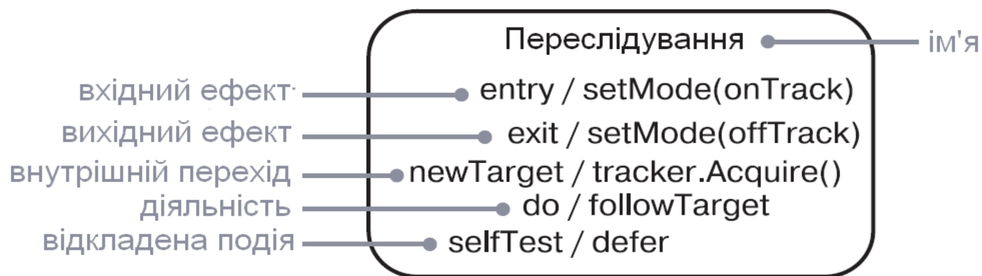


Рис. 4.2.4. Розширені стани і переходи

Вхідний і вихідний ефекти можуть не мати аргументів або захисних умов, але перший з них на верхньому рівні автомата класу може мати параметри аргументів, які автомат приймає при створенні об’єкта.

Всередині стану часто виникає необхідність обробляти певні події, не залишаючи його. У такому випадку можна говорити про внутрішні переходи, які мають тонку відмінність від переходів у себе. Коли справа стосується переходу в себе (рис. 4.2.3), подія ініціює перехід, об’єкт залишає стан, виконується деяка дія (якщо вона є) і потім об’єкт знову входить у той же стан. Оскільки даний перехід виходить і входить у стан, він ініціює дію виходу, потім дію самого переходу і, нарешті, дію входу.

Якщо потрібно опрацювати подію, не виконуючи при цьому вхідних і вихідних дій, UML передбачає скорочення для цієї ідіоми у вигляді внутрішнього переходу.

Внутрішній перехід (internal transition) – це такий перехід, який реагує на подію відповідним ефектом, не змінюючи стану. На рис. 4.2.4 подія newtarget (нова ціль)

позначає внутрішній перехід; якщо ця подія відбувається, коли об'єкт перебуває в стані Tracking (Переслідування), то при цьому виконується дія tracker.Acquire() (переслідувач.Одержати()), але стан залишається колишнім, і ніякого вхідного або вихідного ефекту не спостерігається. Внутрішній перехід символізується включенням рядка переходу (разом з іменем події, необов'язковою захисною умовою й ефектом) у символ стану замість стрілки переходу. Відзначимо, що слова entry (вхід), exit (вихід) і do (діяти) є ключовими і не можуть бути використані в якості імен подій. Щораз, коли об'єкт перебуває в деякому стані й відбувається подія, якою позначений внутрішній перехід, виконується відповідний ефект, але об'єкт при цьому не залишає стану і не повертається в нього повторно. Таким чином, подія опрацьовується без виклику дій виходу та входу.

Коли об'єкт перебуває в деякому стані, він зазвичай простоює, очікуючи настання події. Однак іноді стає необхідно змоделювати якусь діяльність, яка виконується в цьому стані (do-activity). Іншими словами, перебуваючи в стані, об'єкт може що-небудь робити доти, поки це не буде перерваний подією. Перебуваючи в стані переслідування – Tracking, об'єкт може іти за ціллю (followtarget). Вв UML застосовується спеціальний перехід do, що описує роботу, яка повинна виконуватися усередині стану після того, як буде виконана вхідна дія (рис. 4.2.4). Можна також специфікувати поведінку як послідовність дій. Якщо поява події викличе перехід, який призведе до виходу зі стану, будь-яка виконувана в стані діяльність негайно припиниться.

Стан Tracking (Переслідування). Як показано на рис. 4.2.3, припустимо, що є тільки один перехід до цього стану, викликаний подією contact (контакт). Поки об'єкт перебуває в стані Tracking, будь-які події, відмінні від contact, а також від тих, які опрацьовуються підстанами, будуть втрачені. Це означає, що подія може відбутися, але буде проігнорована й у результаті не викличе ніяких дій.

У будь-якій ситуації, пов'язаній з моделюванням, важливо *розпізнавати одні події та ігнорувати інші*. Перші включаються в модель як події, що ініціюють переходи; решта залишається без уваги. Однак у певних ситуаціях необхідно приймати деякі події і при цьому відкладати реакцію на них на потім. Наприклад, поки об'єкт перебуває в стані Tracking, доведеться відкладати реакцію на такі сиґнали, як selftest (самоперевірка). В UML подібну поведінку можна специфікувати, використовуючи відкладені події.

Відкладена подія (deferred event) – це така, опрацювання якої відкладається доти, поки об'єкт не перейде в інший стан. Якщо подія в цьому новому стані вже не є відкладеною, вона опрацьовується і може викликати перехід як ніби вона відбулася щойно. Якщо автомат проходить через ряд станів, у яких дана подія є відкладеною, він зберігається доти, поки не настане стан, у якому він перестане вважатися таким. За цей період можуть виникати інші події, що не є відкладеними. Відкладену подію можна відзначити, додавши до її імені ключове слово defer (рис. 4.2.4). У цьому прикладі події selftest можуть мати місце в стані Tracking, але втримуються доти, поки об'єкт не прийде в стан Engaging (Включення); тоді вони опрацьовуються так, начебто відбулися щойно.

До автомата можна звернутися з іншого автомата. Такі автомати називають **вкладеними** (submachines). Їх зручно використовувати, пропрацьовуючи структуру великих моделей станів. Подробиці наведено в книзі «UML».

Розглянуті властивості станів і переходів вирішують цілий ряд типових проблем моделювання автоматів. Однак в автоматів, розглянутих в UML, є властивість, яка дозволяє ще більше спростити моделювання складної поведінки, – **підстан** (substate),

тобто такий стан, який входить до складу іншого. Heater (Обігрівач) може перебувати в стані Heating (Обігрів) і в той же час – у вкладеному стані Activating (Активація). У цьому випадку правильно буде сказати, що об'єкт перебуває одночасно в станах Heating і Activating.

Простий стан – такий, який не має внутрішньої структури. Стан, що має вкладення, тобто підстани, називають **складеним** (композитним). Підстани можуть бути паралельними (*ортогональними*) або послідовними (*неортогональними*). В UML складений стан зображується так само, як і простий, але з додатковим розділом, у якому показано вкладений автомат. Глибина вкладення станів необмежена.

Моделювання поведінки банкомата. Система обслуговування може перебувати в одному із трьох основних станів: Idle (Простій – очікування взаємодії з користувачем), Active (Активний – виконує транзакцію, запитану користувачем) або Maintenance (Обслуговування – можливо, поповнюється запас готівки). У стані Active поведінка банкомата описується простою схемою: перевірити рахунок користувача, вибрати тип транзакції, виконати транзакцію, надрукувати чек. Після цього банкомат знову повертається в стан Idle. Перераховані етапи поведінки можна подати як стани Validating (Перевірка), Selecting (Вибір), Processing (Обробка), Printing (Друк). Може бути необхідно налаштувати систему таким чином, щоб вона давала користувачеві право вибрати й здійснити кілька транзакцій після того, як виконана перевірка рахунку, але до роздрукування загального чека.

Проблема в тому, що на будь-якому етапі в користувача повинна бути можливість скасувати транзакцію й повернути банкомат у стан Idle. Такого ефекту можна досягти, використовуючи прості автомати, але це вже небажана процедура. Оскільки користувач може скасувати транзакцію в будь-який момент часу, довелося б включати відповідний перехід з будь-якого стану в послідовності Active. Це спричиняє помилки, тому що передбачити всі необхідні переходи непросто, а наявність безлічі подій, що переривають основний потік керування, призведе до появи великої кількості переходів з різних вихідних станів, які будуть закінчуватися в одному й тому ж цільовому. При цьому в кожного з них будуть ті самі ініціюючі події, захисні умови і дії.

За допомогою вкладених підстанів можна спростити моделювання цього завдання (рис. 4.2.5). Тут у стану Active є внутрішній автомат, у який входять підстани Validating, Selecting, Processing, Printing. Банкомат переходить зі стану Idle у стан Active, коли користувач вставляє в карткоприймач кредитну карту. При вході в стан Active виконується дія readcard (читати карту). Почавши з вихідного стану внутрішнього автомата, керування переходить до стану Validating, потім до Selecting і, нарешті, до Processing. Після виходу з останнього керування може повернутися в стан Selecting (якщо користувач обрав іншу транзакцію) або перейти в стан Printing. Звідси відбувається безумовний завершальний перехід у стан Idle. Відзначимо, що в стані Active є дія виходу, яка забезпечує автоматичне повернення кредитної карти.

Також слід звернути увагу на перехід зі стану Active у стан Idle, ініціюючий подію cancel (скасування). У будь-якому підстані стану Active користувач може скасувати транзакцію і повернути банкомат у стан простою (але тільки після добування кредитної карти з картоприймача – це дія, що супроводжує вихід зі стану Active, відбувається незалежно від того, що послужило причиною виходу). Без підстанів потрібен був би перехід, ініціюючий подією cancel, для кожного стану підструктури.

Неортогональні (nonorthogonal) **стани.** Такі підстани, як Validating і Processing, називають *неортогональними*, або *непересічними*. За наявності ряду неортогональних підстанів у контексті складеного стану, об'єкт може перебувати

одночасно в цьому складеному стані і тільки в одному з його підстанів (*або в кінцевому стані*). Таким чином, неортогональні підстани розділяють простір складеного стану на непересічні стани.

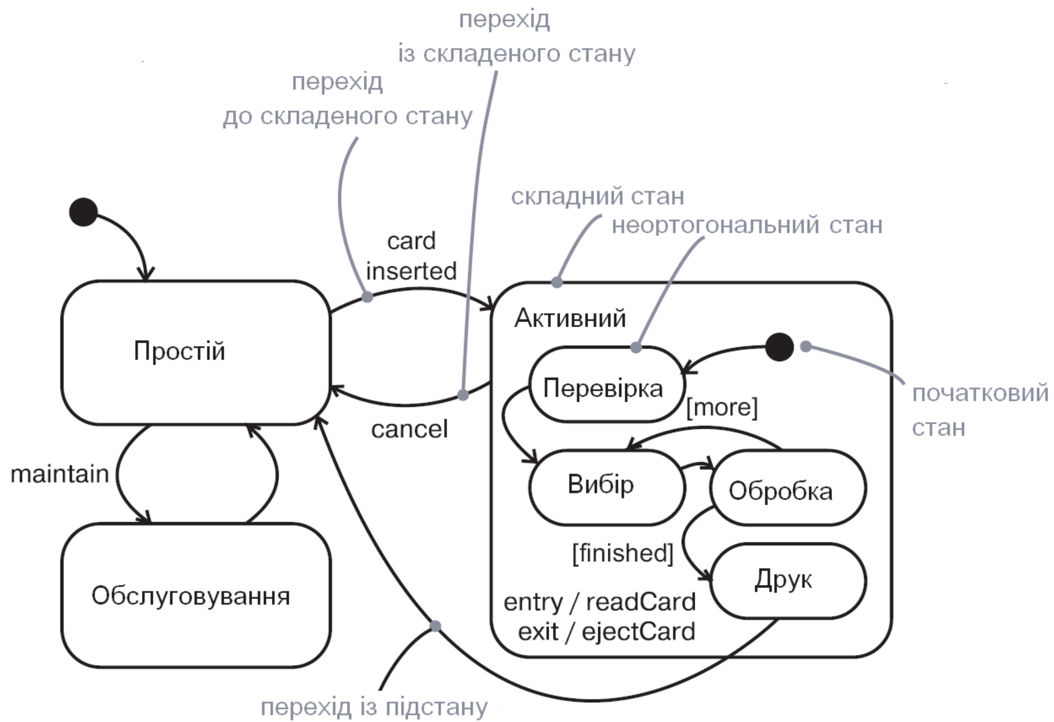


Рис. 4.2.5. Послідовні підстани

Перехід від вихідного стану, зовнішнього відносно вміщувального складеного стану, може призвести або до останнього, або до одного із його підстанів. Якщо метою переходу є складений стан, то вкладений автомат повинен мати початковий стан, якому передається керування після входу в складений стан і після виконання його вхідної дії, якщо такий присутній. Якщо ціль переходу – вкладений підстан, то керування передається йому після виконання вхідної дії (якщо вона є) складеного стану і (якщо є) вхідної дії цього цільового підстану.

Перехід, що веде зі складеного стану, може мати в якості вихідного або його самого, або один з його підстанів. У кожному разі керування спочатку залишає вкладений стан (при цьому виконується його вхідна дія, якщо така є), а потім складений стан (при цьому виконується його вихідна дія, якщо вона є). Перехід, початком якого є складений стан, по суті, перериває діяльність вкладеного автомата. Завершальний перехід складеного стану виконується, коли керування досягає його кінцевого підстану.

Автомат описує динамічу поведінку об’єкта, стан якого залежить від його минулого. Автомат, по суті, специфікує можливий порядок станів, які може приймати об'єкт за час свого ЖЦ. [1,2,8]

Якщо не зазначено інше, то коли перехід призводить до складеного стану, діяльність вкладеного автомата починається з його початкового стану (якщо тільки безпосередньою метою переходу не є конкретний підстан). Однак іноді виникає необхідність моделювати об’єкт, що «пам’ятає» підстан, який був активним перед тим, як він покинув складений стан. При моделюванні поведінки агента, який виконує автоматичне резервне копіювання на всіх комп’ютерах мережі, бажано пам’ятати, на чому він зупинився, коли, наприклад, був перерваний по запиту оператора.

Теоретично тут допускається моделювання за допомогою простого автомата, але це буде некрасиво. Потрібно буде, щоб кожен послідовний підстан посилав значення деякої змінної, локальної стосовно складеного стану. Початковий стан у цьому складеному стані повинен мати перехід, обладнаний захисною умовою, що перевіряє змінну у кожному зі своїх підстанів. Таким чином, при виході зі складеного стану можна буде запам'ятати останній підстан, для того, щоб при наступному вході в складений стан відразу здійснити перехід у нього. Це незручно, адже доведеться пам'ятати про те, що потрібно «обійти» усі підстани і встановити для кожного відповідну вихідну дію. У результаті з'явиться безліч переходів з того самого початкового стану, які ведуть до різних цільових підстанів з дуже схожими (але все-таки різними) захисними умовами.

Історичний стан (history state). В UML передбачений простіший спосіб моделювання подібних ідіом – за допомогою історичних станів, що дозволяють складеному стану, який включає неортогональні підстани, «пам'ятати» підстан, який був активним на момент останнього переходу зі складеного стану зовні. [1] Як показано на рис. 4.2.6, історичний стан зображується у вигляді маленького кружечка з буквою «H» (History).

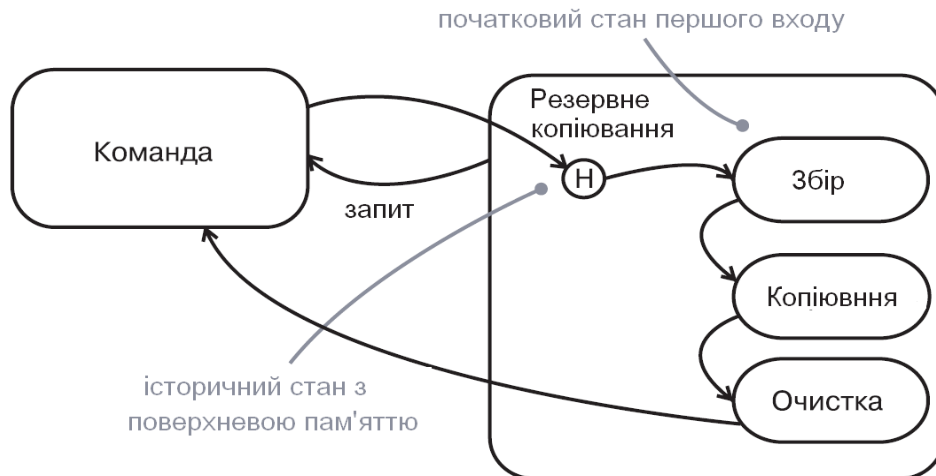


Рис. 4.2.6. Історичні стани

Якщо потрібно, щоб перехід активував останній підстан, то він показується як ведучий зі складеного стану зовні, безпосередньо до історичного. Коли вперше входиться в складений стан, він ще не має історії. Це означає, що в наявності є єдиний перехід від історичного стану до послідовного підстану, – у даному прикладі це Collecting (Збір).

Ціль цього переходу специфікується початковий стан вкладеного автомата при першому вході. Припустимо, що під час стану BackingUp (Резервне копіювання) і Copying (Копіювання) посилається подія query (запит). Керування залишає Copying і Backingup (*викликаючи за необхідності їх вихідні дії*) і повертається до стану Command (Команда). Коли завершиться діяльність Command, перехід, що завершується, повернеться до історичного стану складеного Backingup. Цього разу, оскільки існує історія вкладеного автомата, керування передається стану Copying, минаючи стан Collecting, тому що саме Copying було останнім активним підстаном перед останнім виходом з Backingup. У кожному разі, якщо вкладений автомат досяг

завершального стану, то він втрачає всю збережену в ньому історію і поводиться так, начебто ніякого входу в нього ще не було.

Неортогональні підстани – це загальний випадок використання вкладених автоматів, з яким доводиться зустрічатися на практиці. Однак у деяких ситуаціях моделювання може знадобитися специфікація **ортогональних областей**. [1,5,13] Вони дозволяють описати кілька автоматів, що працюють паралельно в контексті об'єкта, який його включає.

На рис. 4.2.7 показано розширення стану Maintenance (Обслуговування) з рис. 4.2.5. Воно декомпозується на дві ортогональні області – Testing (Тестування) і Commanding (Обробка команд), показані у вигляді вкладень в Maintenance, але розділені пунктирною лінією. Кожна із цих ортогональних областей, у свою чергу, розбита на підстани. Коли керування передається зі стану Idle (Простій) в Maintenance, то загальний потік розділяється на два паралельні – об'єкт, що включає, буде одночасно перебувати в областях Testing і Maintenance. Більше того, об'єкт, що перебуває в області Commanding, також перебуває у стані Waiting (Очікування) або Command (Команда).

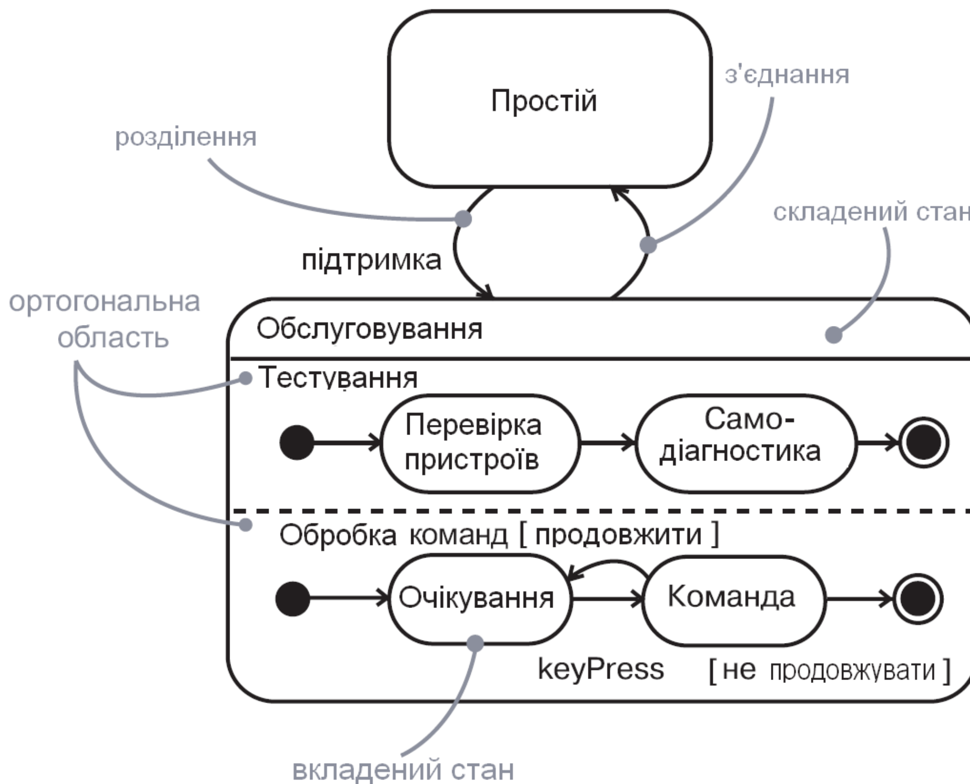


Рис. 4.2.7. Паралельні підстани

Проходження по кожній із цих двох ортогональних областей здійснюється паралельно. У підсумку кожен вкладений автомат досягає свого кінцевого стану. Якщо одна ортогональна область переходить у кінцевий стан раніше іншої, керування в ній очікує, поки інша область не досягне свого кінцевого стану. Коли ж обидві області виявляються у своїх кінцевих станах, керування з них зливається назад у загальний потік. Коли є перехід у складений стан, розбитий на ортогональні області, керування завжди розділяється на стільки ж паралельних потоків, скільки існує таких областей. Аналогічно, за наявності переходу зі складеного стану, розбитого на ортогональні

області, паралельні потоки керування зливаються воєдино. Це вірно в усіх випадках. Якщо всі ортогональні області досягають свого кінцевого стану або ж є явний перехід із вміщувального складеного стану, керування зливається в один потік.

Вхід у складений стан з ортогональними областями призводить до початкового стану кожної із цих областей. Але також можливо здійснити перехід із зовнішнього стану безпосередньо в один або кілька ортогональних станів. Такий процес називають **поділом** (fork), оскільки керування передається від одного стану декількам ортогональним відразу. Графічно це виражається товстою чорною лінією з однією вхідною стрілкою й кількома вихідними, кожна з яких указує на ортогональний стан. Якщо одна або кілька ортогональних областей не мають цільових станів, то неявно вибираються початкові стани цих областей. Перехід до єдиного ортогонального стану усередині складеного – також неявний поділ; початкові стани всіх інших ортогональних областей є неявними учасниками поділу.

Аналогічним чином перехід від будь-якого стану всередині складеного з ортогональними областями ініціює вихід із усіх ортогональних областей. Такий перехід часто є помилковою умовою, що викликає переривання всіх паралельних обчислень.

З'єднання (join) – це перехід з кількома вхідними стрілками і однією вихідною. Кожна вхідна стрілка повинна виходити зі станів у різних ортогональних областях одного і того ж складеного стану. З'єднання може мати ініціюючу подію. Перехід зі з'єднанням ефективний тільки тоді, коли активні всі вихідні стани; статус інших ортогональних областей складеного стану неважливий. [12,16] Якщо подія відбувається, то керування залишає всі ортогональні області складеного стану, а не лише ті, звідки ведуть стрілки.

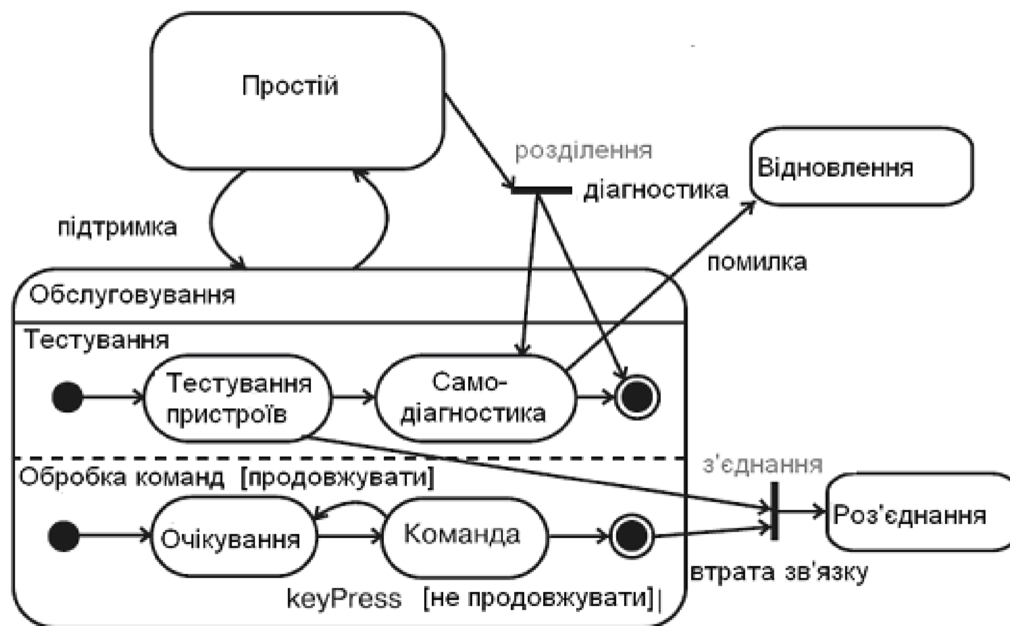


Рис. 4.2.8. Переходи з розділеним з'єднанням

Рис. 4.2.8 – варіант попереднього прикладу з явними переходами поділу і з'єднання. Перехід maintain (підтримка) до складеного стану Maintenance (Обслуговування) – це, як і раніше, неявний поділ, що призводить до початкових станів за замовчуванням двох ортогональних областей. Тут є присутній перехід з явним поділом від Idle (Простій) до вкладених станів Self diagnose (Самодіагностика) і

кінцевого стану області Commanding (Обробка команд). Кінцевий стан – це явний стан, який може бути метою переходу. Якщо виникає подія помилки під час Self diagnose, то відбувається неявне злиття з переходом до Repair (Відновлення). При цьому й Self diagnose, і будь-який активний стан області Commanding завершується. Є також явний перехід зі злиттям до стану Offline (Роз'єднання). Він здійснюється тільки при виникненні події disconnect (втрата зв'язку), коли активні і стан Testing devices (Тестування пристроїв), і кінцевий стан області Commanding (Обробка команд). Якщо обидва стани неактивні, дана подія не має ніякого ефекту.

Інший спосіб моделювання паралелізму – застосування **активних об'єктів.** Тобто замість розбивки одного автомата об'єкта на кілька паралельних областей можна визначити два активні об'єкти, кожен з яких реалізує поведінку однієї з паралельних гілок. [12,14,16,17] Якщо на поведінку одного із цих паралельних потоків впливає стан іншого, це можна змоделювати за допомогою ортогональних областей. Якщо ж поведінка одного потоку залежить від повідомлень, посланих іншим або іншому, тоді доцільно скористатися активними об'єктами. Якщо між паралельними потоками відбувається лише мінімальне спілкування або взагалі не спостерігається ніякого, то в більшості випадків застосування активних об'єктів зробить ваш дизайн наочнішим.

Найбільш загальне призначення автоматів – моделювання ЖЦ об'єкта, особливо якщо мова йде про екземпляри класів, ВВ і про систему в цілому. У той час, як взаємодія моделює поведінку набору працюючих разом об'єктів, автомат моделює поведінку одного об'єкта протягом його ЖЦ – наприклад, як це буває з користувацькими інтерфейсами, контролерами і різними технічними пристроями.

Коли моделюється ЖЦ об'єкта, доводиться явно специфікувати три моменти: події, на які об'єкт може реагувати, безпосередню реакцію на ці події і вплив більш ранньої поведінки на поточну. Крім того, моделювання ЖЦ об'єкта містить у собі прийняття рішень про порядок, у якому об'єкт може осмислено реагувати на події, починаючи з моменту його створення і закінчуючи знищенням.

Щоб змоделювати ЖЦ об'єкта, необхідно визначити контекст автомата (*чи це клас або система в цілому; якщо мова йде про контекст класу або ВВ, знайти сусідні класи, включаючи всіх батьків даного класу і усіх доступних йому по асоціаціях або залежностях*). Ці сусіди – потенційні цілі дій, а також кандидати на включення в захисні умови; якщо ж мова йде про контекст системи в цілому, необхідно зосередити увагу на якомусь одному аспекті її поведінки. Теоретично кожен об'єкт системи може брати участь у моделюванні її ЖЦ, і, за винятком роботи з найпростішими системами, такі повні моделі будуть недоступні для розуміння. Встановити початковий і кінцевий стани об'єкта (*можливо встановити перед- і післяумови початкового і кінцевого станів відповідно*). Прийняти рішення щодо подій, на які повинен реагувати об'єкт *(їх можна виявити в інтерфейсах об'єкта, якщо вони специфіковані, якщо ні, розглянути, які об'єкти можуть взаємодіяти з даним у наявному контексті і які події вони можуть передавати і приймати*). Від початкового стану до кінцевого виділити ті стани вищого рівня, у яких може перебувати об'єкт, з'єднати їх переходами, що ініціюються відповідними подіями. Продовжувати роботу, додаючи дії до цих переходів, ідентифікувати вхідні і вихідні дії (*особливо якщо виявиться, що відповідна ідіома використовується в автоматі*). За необхідності розширити виділені стани підстанами. Перевірити, чи всі згадані події в автоматі відповідають подіям, очікуваним в інтерфейсі об'єкта. Аналогічно перевірити, чи всі події, очікувані інтерфейсом об'єкта, опрацьовуються в автоматі. Розглянути місця, де слід явно ігнорувати події. Переконатися, що всі дії, згадані в автоматі, підтримуються зв'язками, методами й

операціями об'єкта, що їх включає. Провести трасування автомата, щоб перевірити його на відповідність очікуваним послідовностям подій і реакцій на них (*особливо уважно слід шукати недосяжні стани і ті, у яких автомат може зациклитися*). Після реорганізації автомата знову перевірити його на предмет того, чи не змінилася семантика об'єкта.

На рис. 4.2.9 представлений автомат контролера домашньої системи сиґналізації, яка відповідає за відстеження показників різноманітних сенсорів, розташованих по периметру будинку.

У ЖЦ цього контролера є чотири основні стани: Intializing (Ініціалізація – *запуск роботи*), Idle (Простій – *контролер готовий і очікує сиґналів тривоги або команд користувача*), Command (Команда – опрацювання команд користувача) і Active (Активний – опрацювання сиґналу тривоги). Вперше створюваний об'єкт контролера спочатку потрапляє в стан Intializing, а потім переходить у стан Idle. Подробиці цих двох станів не показані, за винятком переходу в себе після закінчення певного періоду часу (10 с) у стані Idle. Тимчасові подібні події часто зустрічаються у вбудованих системах, де є таймер, що викликає періодичну перевірку працездатності системи.

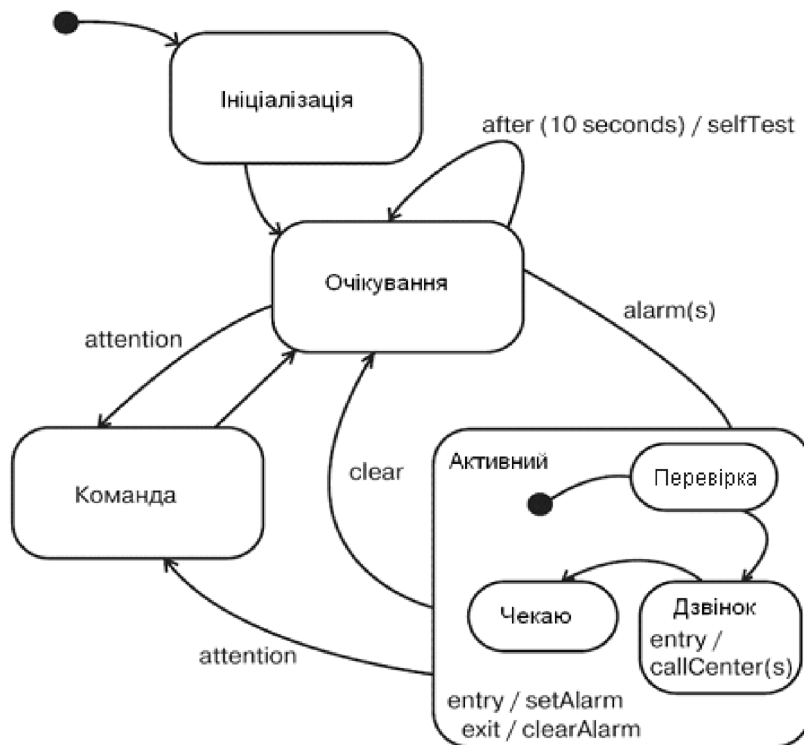


Рис. 4.2.9. Моделювання життєвого циклу об'єкта

Керування передається від стану Idle у стан Active після отримання події alarm (*тривога*). Останнє супроводжується параметром s, що ідентифікує сенсор, який був задіяний. При вході в стан Active у якості вхідного виконується дія set Alarm (підняти Тривогу) і керування передається спочатку стану Checking (Перевірка), потім стану Calling (Виклик), що забезпечує виклик охоронної компанії для реєстрації сиґналу тривоги, і, нарешті, стану Waiting (Очікування). Стани Active і Waiting завершуються тільки при «очищенні» сиґналу тривоги (clearalarm) або ж з ініціативи користувача, при виклику події attention (увага), що передує команді. Кінцевого стану тут немає, що є типовим для вбудованих систем, які повинні працювати безупинно.

Добре структурований автомат повинен бути простим і не включати зайвих станів чи переходів; мати доступ до всіх видимих об'єктів; реалізовувати свою поведінку з оптимальним балансом витрат часу і ресурсів; бути зрозумілим, іменуючи свої стани і переходи в термінології словника системи; не допускати занадто глибоких рівнів вкладеності (*не більше 2-х*); помірковано використовувати ортогональні області (краще активні класи). При зображенні автомата уникати пересічних переходів.

4.3. Процеси і потоки керування

Основні питання:

- *Активні об'єкти, процеси і потоки*
- *Моделювання множини потоків керування*
- *Моделювання міжпроцесорної комунікації*
- *Побудова абстракцій захищених потоків*

При моделюванні реальних систем, в яких одочасно може відбуватися велика кількість різних подій, слід враховувати їх вигдяд з точки зору процесів, де основна увага приділяється процесам і потокам, що лежать в основі механізмів паралелізму і синхронізації. Це потребує складніших механізмів обміну інформацією і синхронізацією, ніж для послідовних систем.

Для собаки в будці, розпорядок дня простий і послідовний. Їмо, спимо, ганяємося за кішкою. Знову їмо, мріємо, як будемо ганятися за кішкою. Забратися в будку, щоб поспати або укритися від дощу, не складає проблеми, оскільки, крім собаки, ні в кого не виникне потреби скористатися входом. Ніякої конкуренції за ресурси.

Сімейні турботи не настільки прості. Кожен член родини живе своїм власним життям, але в той же час взаємодіє з іншими мешканцями будинку (разом обідають, дивляться телевізор, прибирають). Члени родини спільно користуються загальними ресурсами. У дітей загальна спальня, на всю родину один телефон, два ноутбуки тощо. Родичі розподіляють між собою обов'язки: батько ходить у пральню і бакалійну лавку, мати оплачує рахунки і готує їжу, діти допомагають прибирати в будинку. Боротьба за використання ресурсів і координація домашніх обов'язків стають предметом суперечок. Наявність однієї ванної кімнати в ситуації, коли всі одночасно збираються до школи, – це проблема, а обід не буде готовий, якщо тато не сходив до магазину.

По-справжньому складне життя хмарочоса і його орендарів. Сотні й тисячі людей приходять в одну і ту ж будівлю, і в кожного свої плани. Усі проходять через визначену кількість входів, турникетів, загороджувальних бар'єрів, супроводжуваних непробивними вахтерами. Усі користуються загальними ліфтам. Відвідувачів обслуговують ті самі системи опалення, електро-, водопостачання та каналізації. На всіх одне паркування. Якщо люди прагнуть працювати злагоджено, то повинні спілкуватися і синхронізувати свої дії.

В UML кожен незалежний потік керування моделюється у вигляді активного об'єкта.

Активний клас (active class) – це клас, екземплярами якого є активні об'єкти. Активні класи – це різновид класів, тому їх UML нотація включає всі відповідні розділи – для імені класу, атрибутів і операцій та зображується у вигляді прямокутника з потовщеними бічними границями (рис. 4.3.1). [2,13] Активні класи

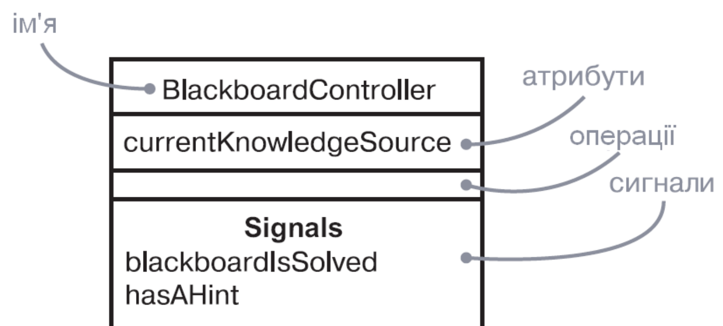


Рис. 4.3.1. Активний клас

часто отримують сиґнали, які зазвичай перелічуються в додатковому розділі нотації.

Активний об'єкт (active object) – це об'єкт, який володіє процесом або потоком і може ініціювати керуючий вплив. Активний об'єкт є екземпляром активного класу. [2]

Процес (process) – це ресурсомісткий потік керування, який може виконуватися паралельно з іншими процесами.

Потік (thread) – це полегшений потік керування, який може виконуватися паралельно з іншими потоками в рамках одного процесу. Процеси й потоки зображуються у вигляді активних класів зі стереотипами, а на діаграмах взаємодії часто виступають у ролі послідовностей.

Подібно до інших об'єктів, активні об'єкти можуть спілкуватися один з одним, посилаючи повідомлення, передавання яких повинно бути доповнене семантикою паралелізму, щоб полегшити взаємодію незалежних потоків керування. Багато мов програмування безпосередньо підтримують концепцію активного об'єкта. У мовах Java, Smalltalk паралелізм вбудований. C++ підтримує паралелізм за рахунок різних бібліотек, в основі яких лежать механізми паралелізму, забезпечувані операційною системою.

Потік керування. У строго послідовній системі в кожен момент часу виконується одна дія з опрацювання подій. Послідовність виконання цих дій називають потоком керування. [13] В послідовній програмі в довільний момент часу завжди має місце тільки один потік керування. При її запуску керування переходить до позначки входу в програму, після чого виконується одна операція за іншою. Навіть якщо дійові особи функціонують паралельно, послідовна програма буде опрацьовувати по одній події, відкидаючи тим самим одночасні з нею події. Якщо трасувати виконання послідовної програми, то позначка виконання буде переміщатися з одного рядка до іншого – послідовно. Зустрічаються, звичайно, розгалуження, цикли, переходи, а за наявності рекурсії або ітерації – рух потоку навколо однієї позначки. Але, незважаючи на все це, у послідовній системі є тільки один потік виконання.

У паралельній же системі є *кілька потоків керування*. Що означає, що у той самий момент часу має місце різна діяльність. Кожен з одночасно виконуваних потоків керування починається із позначки входу в деякий процес або потік. Якщо зробити моментальний знімок паралельної системи під час її роботи, то можна побачити кілька позначок керування (принаймні, логічних).

Для моделювання процесу або потоку в UML використовується активний клас, у контексті якого здійснюється незалежний потік керування, що може працювати паралельно з іншими. Активні класи, що володіють досить специфічною властивістю. Активний клас являє собою незалежний потік керування, тоді як звичайний клас не пов'язаний з таким. На відміну від активних, звичайні класи неформально називають *пасивними*, тому що вони не здатні ініціювати незалежний керуючий вплив.

Активні класи застосовуються для моделювання сімейств процесів або потоків. [1] На практиці це означає, що *активний об'єкт є екземпляр активного класифікатора* і матеріалізує процес або потік. При моделюванні паралельних систем за допомогою активних об'єктів присвоюється ім'я кожному незалежному потоку керування. У результаті створення активного об'єкта запускається асоційований з ним потік керування; після знищення об'єкта цей потік завершується. Активні класи мають ті ж властивості, як будь-які інші класи: можуть мати екземпляри, атрибути й операції, а також брати участь у зв'язках залежності, узагальнення й асоціації (включаючи агрегації). Активні класи мають право користуватися надаваними UML механізмами розширення, у тому числі стереотипами, позначеними значеннями й обмеженнями.

Зустрічаються активні класи – реалізації інтерфейсів. Активні класи можуть реалізовуватися за допомогою кооперацій, а їх поведінка може специфікуватися за допомогою автоматів. Допускається участь активних класів у коопераціях. На діаграмах активні об'єкти зустрічаються там же, де й пасивні. Можна моделювати кооперації активних і пасивних об'єктів за допомогою діаграм взаємодії (послідовності й комунікації). Активний об'єкт може виступати в ролі цільового об'єкта події в автоматі. В автоматах як активні, так і пасивні об'єкти можуть посилати й отримувати події сиґналів і викликів.

Способи комунікації між об'єктами в потоках керування. Об'єкти, що кооперуються між собою, взаємодіють шляхом обміну повідомленнями. У системі, де є одночасно активні й пасивні об'єкти, слід розглядати чотири можливі комбінації обміну повідомленнями.

По-перше, повідомлення може бути передано одним пасивним об'єктом іншому такому ж. Якщо припустити, що в будь-який момент часу існує лише один потік керування, що проходить через обидва об'єкти, така взаємодія – не що інше, як простий виклик операції.

По-друге, повідомлення може бути передане від одного активного об'єкта іншому активному. Тут слід сказати про **міжпроцесну комунікацію,** яка може здійснюватися двома способами. У першому варіанті деякий активний об'єкт спроможний синхронно викликати операцію іншого. Такий спосіб має семантику рандеву: об'єкт, що вимагає виконання операції й чекає, поки сторона, що приймає, отримає виклик, виконає деяку операцію і поверне деякий об'єкт (якщо є що повертати); потім обидва об'єкти продовжують працювати незалежно один від одного. Протягом усього часу виконання виклику обидва потоки керування будуть блоковані. У другому варіанті один активний об'єкт може асинхронно послати сиґнал іншому або викликати його операцію. Семантика такого способу нагадує поштову скриньку (mailbox): сторона, яка викликає, посилає сиґнал або викликає операцію, після чого продовжує роботу. Тим часом сторона, що отримує, приймає сиґнал або виклик, як тільки буде до цього готова. Поки вона опрацьовує запит, усі нові події або виклики ставляться в чергу. Відреагувавши на запит, об'єкт, що приймає, продовжує свою роботу. Семантика поштової скриньки проявляється в тому, що обидва об'єкти не синхронізовані – просто один залишає повідомлення для іншого.

В UML синхронне повідомлення зображується зафарбованою стрілкою, а асинхронне – звичайною (рис. 4.3.2).

По-третє, повідомлення може бути передане від активного об'єкта пасивному. Труднощі виникають у випадку, коли відразу кілька активних об'єктів передають свій потік керування тому самому пасивному. У такій ситуації слід дуже акуратно моделювати синхронізацію потоків.

По-четверте, пасивний об'єкт може передавати повідомлення активному. На перший погляд, це може здатися некоректним, але якщо згадати, що кожен потік керування належить деякому активному об'єкту, то стає зрозуміло, що передавання пасивним об'єктом повідомлення активному має ту ж семантику, що й обмін повідомленнями між двома активними об'єктами.

Коли потік проходить через деяку операцію, то вважається, що ця операція є позначкою виконання. Якщо операція визначена в деякому класі, то можна сказати, що позначкою виконання є конкретний екземпляр цього класу. В одній операції (в одному об'єкті) можуть одночасно перебувати кілька потоків керування, а буває й так, що різні потоки перебувають у різних операціях, але все-таки в одному об'єкті.

Проблема виникає тоді, коли в одному об'єкті перебувають відразу кілька потоків керування. Якщо не виявити обережність, то більш ніж один потік може модифікувати той самий атрибут, що призведе до некоректної зміни стану об’єкта або втрати інформації. Це класична проблема взаємного виключення. Помилки при опрацюванні такої ситуації можуть стати причиною різних видів конкуренції між потоками і їх взаємної інтерференції.

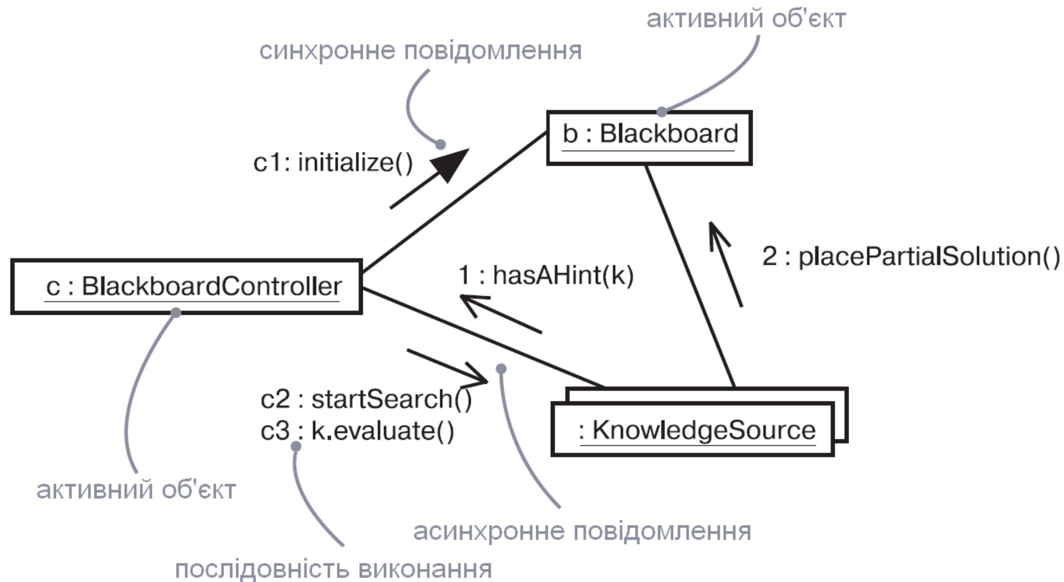


Рис. 4.3.2. Комунікація

Ключ до розв’язку проблеми – серіалізація доступу до критичного об’єкта. У даного підходу є три різновиди, суть кожного з яких полягає в приєднанні до операцій, визначених у класі, деяких синхронізуючих властивостей. UML дозволяє моделювати всі три можливості:

Sequential (послідовна) – викликаючі сторони повинні координувати свої дії ще до входу в об’єкт, що викликається, так, щоб у будь-який момент часу усередині об'єкта перебував рівно один потік керування. За наявності кількох потоків керування не можуть гарантуватися семантика і цілісність об’єкта.

Guarded (захищена) – семантика і цілісність об’єкта гарантуються при наявності кількох потоків керування шляхом упорядкування викликів усіх захищених операцій об’єкта. По суті, у кожен момент часу може виконуватися одна операція над об'єктом, що зводить такий підхід до послідовного. При цьому існує небезпека взаємного блокування.

Concurrent (паралельна) – семантика й цілісність об’єкта за наявності кількох потоків керування гарантується за рахунок відділення операцій зміни даних від операцій читання. Це досягається завдяки ретельному дотриманню правил проектування.

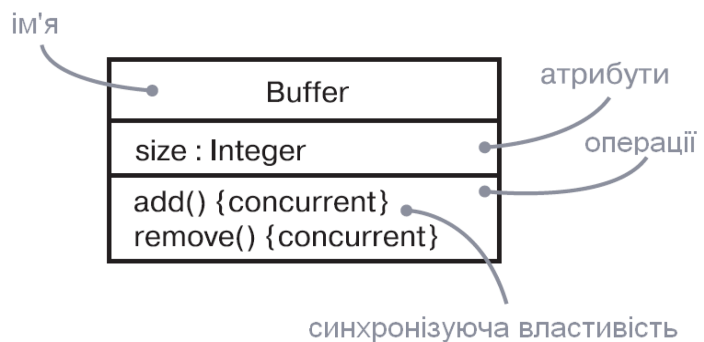


Рис. 4.3.3. Синхронізація

Деякі мови програмування підтримують перераховані конструкції безпосередньо. У мові Java є властивість synchronized, еквівалентна властивості

concurrent в UML. У будь-якій мові, що підтримує паралельність, усі три підходи можна реалізувати за допомогою **семафорів** (semaphores).

На рис. 4.3.3 показано, як ці властивості приєднуються до операції, – шляхом застосування нотації, прийнятої в UML для обмежень. Зверніть увагу, що одночасність повинна бути оголошена окремо як для кожної операції, так і для цілого об'єкта. Оголошення одночасності для операції означає безпроблемне одноразове виконання її численних викликів. Оголошення одночасності для об'єкта дозволяє викликам різних операцій виконуватися одночасно й без помилок. Це суворіша умова.

Побудова системи з кількома потоками керування – непросте завдання. Треба не тільки з'ясувати, як розподілити роботу між паралельними активними об'єктами, але й продумати механізми комунікації та синхронізації між активними й пасивними об'єктами в їх системі, що гарантують правильність поведінки в присутності кількох потоків керування. Тому корисно візуалізувати способи взаємодії цих потоків. В UML це можна зробити за допомогою діаграм взаємодії (для описування динамічної семантики), у якій беруть участь активні класи й об'єкти.

Для моделювання кількох потоків керування потрібно: ідентифікувати можливості розпаралелення дій і матеріалізувати кожен потік керування у вигляді активного класу. Згрупувати загальні множини активних об'єктів в активний клас. Важливо уникати ускладнення системи внаслідок надто великої кількості паралельних потоків керування; розглянути баланс розподілу обов'язків між цими активними класами, а потім досліджувати, з якими іншими активними й пасивними класами статично кооперується кожен з них. Переконатися, що кожен активний клас має внутрішню структуру з високим ступенем зчеплення й слабо пов'язаний із сусідніми класами, і що для кожного класу правильно вибрано набір атрибутів, операцій і сиґналів. Відобразити статичні рішення у вигляді діаграм класів, явно виділивши кожний активний клас. Розглянути, як кожна група класів динамічно кооперується з іншими. Відобразити свої розв'язки на діаграмах взаємодії. Явно показати активні об'єкти як початкові позначки відповідних потоків керування. Ідентифікувати кожну зв'язану послідовність, привласнюючи їй ім'я активного об'єкта. Звернути особливу увагу на комунікації між активними об'єктами, у міру необхідності користуючись як синхронними, так і асинхронними повідомленнями. Забезпечити синхронізацію активних об'єктів і тих пасивних об'єктів, з якими вони кооперуються. Застосовувати найоптимальнішу семантику – послідовну, захищену або паралельну. [12]

На рис. 4.3.4 показана частина вигляду трейдерської системи з точки зору процесів. Ви бачите три об'єкти, які паралельно постачають інформацію в систему Stockticker (Біржовий тікер), IndexWatcher (Переглядач індексу) і Cnnnewsfeed (Стрічка новин-CNN), названі відповідно s, I та c. Два з них, s і I, обмінюються кожний зі своїм екземпляром класу Analyst (Аналітик) - a1 і a2. У рамках цього невеликого фрагмента моделі клас Analyst може бути спроектований у спрощеному вигляді – з розрахунку на те, що в кожен момент часу в кожному з його екземплярів може бути активний тільки один потік керування. Однак обидва екземпляри класу Analyst одночасно спілкуються з об'єктом Alert Manager (Диспетчер оповіщення), якому ми дали ім'я m. Отже, m необхідно спроектувати так, щоб він зберігав свою семантику в присутності кількох потоків керування. Об'єкти m і з одночасно спілкуються з t – об'єктом класу TradingManager (Менаджер продажів). Кожному з'єднанню привласнений порядковий номер, зумовлений тим, який потік керування ним володіє.

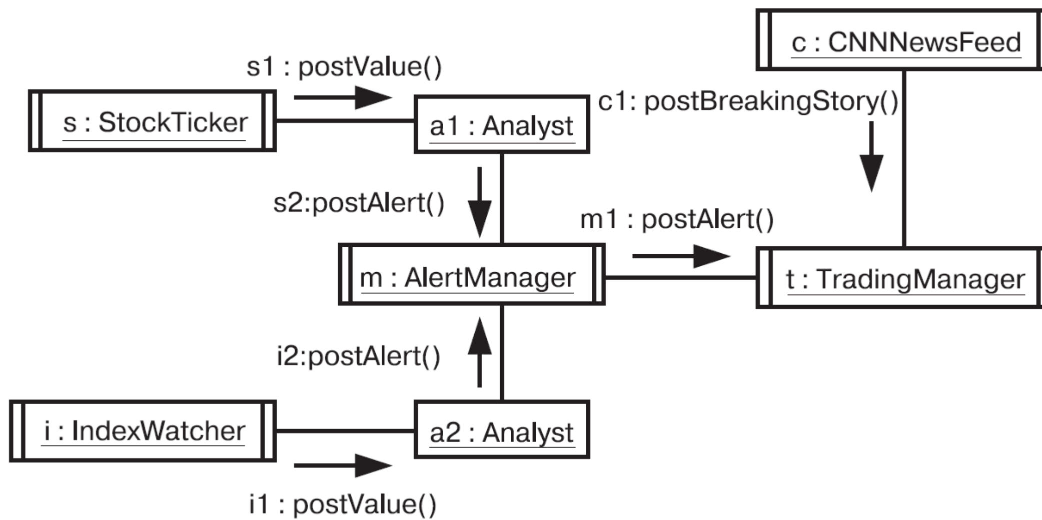


Рис. 4.3.4. Моделювання потоків управління

До подібних діаграм часто приєднують автомати з ортогональними станами, щоб у деталях передати поведінку активних об'єктів.

При під'єднанні в систему кількох потоків керування необхідно також розглянути механізми, за допомогою яких об'єкти з різних потоків керування взаємодіють один з одним. Об'єкти, що перебувають у різних потоках (існуючих у рамках того самого процесу), можуть спілкуватися за допомогою подій, сиґналів або викликів, причому останні можуть мати синхронну й асинхронну семантику. Для обміну інформацією через межі процесів, у кожного з яких свій власний адресний простір, застосовуються інші механізми.

Складність проблеми міжпроцесорної комунікації збільшується ще й тим, що в розподілених системах процеси можуть виконуватися на різних вузлах. Існує два класичні підходи до міжпроцесорної комунікації: передавання повідомлень і виклики вилучених процедур. В UML ці механізми моделюються як асинхронні й синхронні події відповідно. Але оскільки це вже не прості виклики усередині одного процесу, то проект необхідно збагатити додатковою інформацією.

Для моделювання міжпроцесорної комунікації необхідно: змоделювати кілька потоків керування; змоделювати обмін повідомленнями за допомогою асинхронних комунікацій, а виклики вилучених процедур – за допомогою синхронних; неформально описати використовуваний механізм за допомогою приміток або більш строго, за допомогою кооперацій. [13]

На рис. 4.3.5 показана розподілена система бронювання квитків, у якій процеси виконуються в чотирьох вузлах. Кожен об'єкт маркований стереотипом process, а також позначеним значенням location, яке визначає його фізичне положення. Комунікації між об'єктами ReservationAgent (Агент бронювання), Ticketingmanager (Диспетчер видачі білетів) і Hotelagent (Готельний агент) асинхронні. У примітці написано, що комунікації побудовані на основі служби повідомлень, реалізованої на Javabeans. Комунікація між об'єктами Tripplanner (Планувальник маршруту) і Reservationsystem (Система резервування) синхронна. Семантика їх взаємодії показана в кооперації, названої CORBA ORB. Tripplanner виступає в ролі клієнта, а ReservationAgent – сервера. Розкривши цю кооперацію, ви побачите деталі спільної роботи сервера і клієнта.

Добре структуровані активний клас і активний об’єкт являють собою незалежний потік керування, який максимально використовує можливості дійсного паралелізму в системі. Вони не настільки малі, щоб вимагати наявності інших активних елементів, надлишок яких може створити переускладнену і нестабільну архітектуру процесів; акуратно керують комунікацією між активними елементами, вибираючи найоптимальніший в даному випадку механізм – синхронний або асинхронний; виходять із трактування кожного об'єкта як критичної області, використовуючи відповідні синхронізуючі властивості для збереження його семантики в присутності кількох потоків керування.

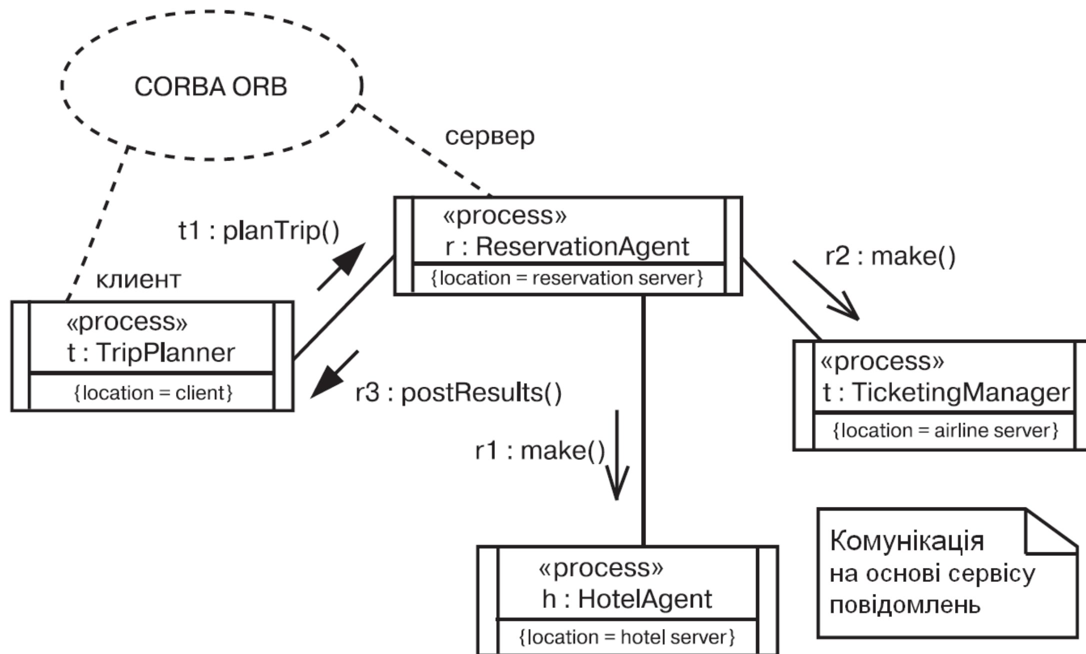


Рис. 4.3.5. Моделювання міжпроцесорної комунікації

При моделюванні в UML активних класів і об'єктів слід показувати тільки ті атрибути, операції й сиґнали, які важливі для розуміння абстракції у відповідному контексті, явно показувати всі синхронізуючі властивості операції. Слід уникати як переускладнення моделей (*за наявності надто великого числа паралельних потоків керування система почне пробуксовувати*), так і надмірного спрощення (*недостатній ступінь паралелізму не дозволить оптимізувати пропускну здатність системи*).

4.4. Моделювання систем реального часу.

Основні питання:

- *Час, тривалість і місце розташування*
- *Моделювання тимчасових обмежень*
- *Моделювання розподілу об'єктів*
- *Моделювання мігруючих об'єктів*
- *Системи реального часу і розподілені системи*

Події в реальному світі (*що не вибачає помилок*) відбуваються в непередбачені моменти часу, проте вимагають адекватної і своєчасної реакції. Системні ресурси можуть бути розподілені по усьому світу, а деякі здатні переміщатися в просторі, – у зв’язку із цим виникають проблеми затримок, синхронізації, безпеки і якості послуг.

Моделювання часу і простору – важливий елемент будь-якої розподіленої системи або системи реального часу (СРЧ). Якісна модель СРЧ повинна висвітлити всі необхідні й достатні просторово-часові властивості системи. Моделювання більшості програмних систем грунтується на ідеальності середовища: припускають, що доставка повідомлення здійснюється миттєво, мережеві збої неможливі, робочі станції не виходять із ладу, а завантаження мережі завжди рівномірне. На жаль, реальність аж ніяк не укладається в цю схему: повідомлення доставляються з деяким затриманням (*іноді й зовсім не доставляються*), мережа «падає», робочі станції «зависають» і завантаження мережі далеке від збалансованого. Тому при моделюванні таких систем необхідно брати до уваги як просторові, так і тимчасові характеристики.

Система реального часу (real time system) повинна виконувати свої функції в строго певний абсолютний або відносний момент часу і витрачати на це передбачуваний і обмежений час. Серед подібних систем бувають такі, для яких необхідний час реакції обчислюється нано- або мілісекундами. Але зустрічаються системи «майже реального часу», для яких можливий час реакції – секунди і навіть більше.

Розподілена система (distributed system) – система, компоненти якої можуть бути фізично розміщені на різних вузлах, що можуть представляти різні процесори, змонтовані в одному корпусі, або різні комп'ютери, що перебувають у різних точках земної кулі. [2,11]

Для моделювання та ВСКД СРЧ і розподілених систем (РС) UML пропонує різні засоби, включаючи оцінки часу, тимчасові вирази, обмеження й позначені значення, графічний вигляд яких показано на рис. 4.4.1.

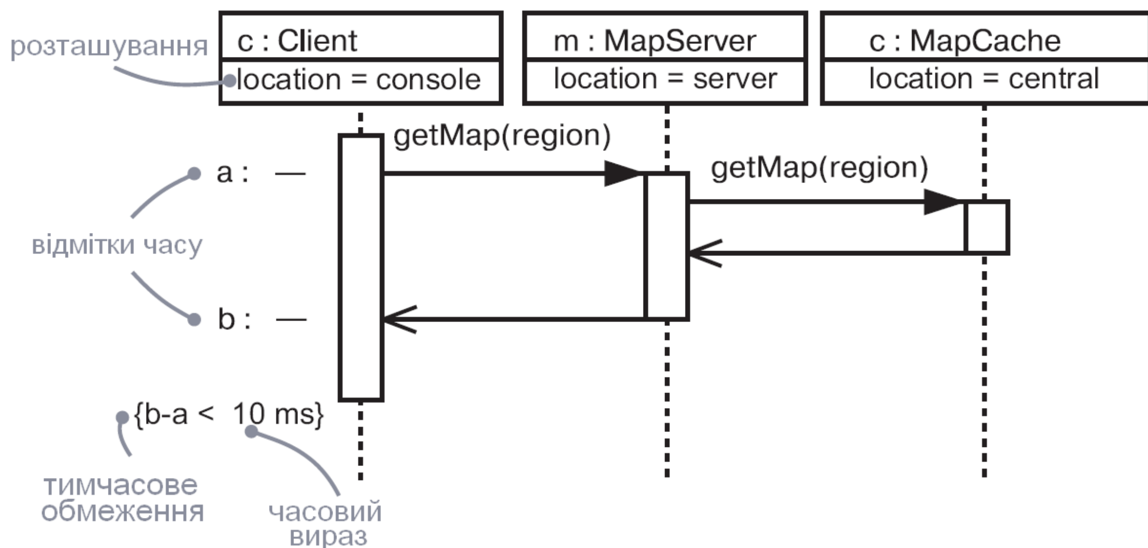


Рис. 4.4.1. Часові обмеження і місце розташуваня

Оцінка часу (timing mark) служить для позначення моменту часу, в який відбулася подія. Вона зображується у вигляді зарубки (*короткої горизонтальної лінії*) на границі діаграми послідовності. [10]

Тимчасовий вираз (time expression) – значенням його є абсолютний або відносний час. Тимчасовий вираз може бути також сформований з використанням імені повідомлення і вказівки стадії його опрацювання, наприклад request.sendtime або request.receivetime. [2,13]

Тимчасове обмеження (timing constraint) – це семантичне твердження про відносний або абсолютний час. Графічно тимчасове обмеження зображується як будь-яке інше – рядком у дужках і пов'язаною з деяким елементом залежністю. [1,2]

Місце розташування (location) – розміщення компонента у вузлі. Є атрибутом об'єкта. [1]

Для СРЧ важлива своєчасність реакції. Події в них можуть відбуватися регулярно або спонтанно, але в кожному разі час реакції на подію повинен бути передбачуваним – або в абсолютному виразі, або відносно моменту виникнення події.

Передавання повідомлень – це один з динамічних аспектів будь-якої системи, тому при моделюванні її тимчасових особливостей за допомогою UML можна кожному повідомленню, що бере участь у взаємодії, дати ім'я, яке буде використовуватися як оцінка часу. При зображенні повідомлень використовується ім'я події, наприклад сиґналу або виклику. При цьому не можна вводити ім'я події в запис виразу, оскільки та сама подія може викликати різні повідомлення. Якщо з'явилась така неоднозначність, слід присвоїти явне ім'я повідомленню в оцінці часу, щоб надалі його можна було використовувати в тимчасовому виразі. Якщо задане ім'я повідомлення, допускається застосування кожної із трьох його функцій – sendtime, receivetime, transmissiontime (*ці функції не описані в базовому* UML). СРЧ можуть включати й інші функції. Для синхронних викликів можна також вказати посилання на час **кругового** (round-trip) повідомлення – executiontime (*ще одне нововведення*). Ці функції зручні для побудови складних тимчасових виразів (у тому числі, що включають вагу і зсув), що є константами або змінними (*якщо їх можна обчислити в момент виконання*). Тимчасовий вираз можна вмістити всередину тимчасового обмеження, щоб описати поведінку системи (рис. 4.4.2) у часі. Як і будь-які інші обмеження, вони зображуються поруч із відповідним повідомленням або явно приєднуються за допомогою зв'язків залежності.

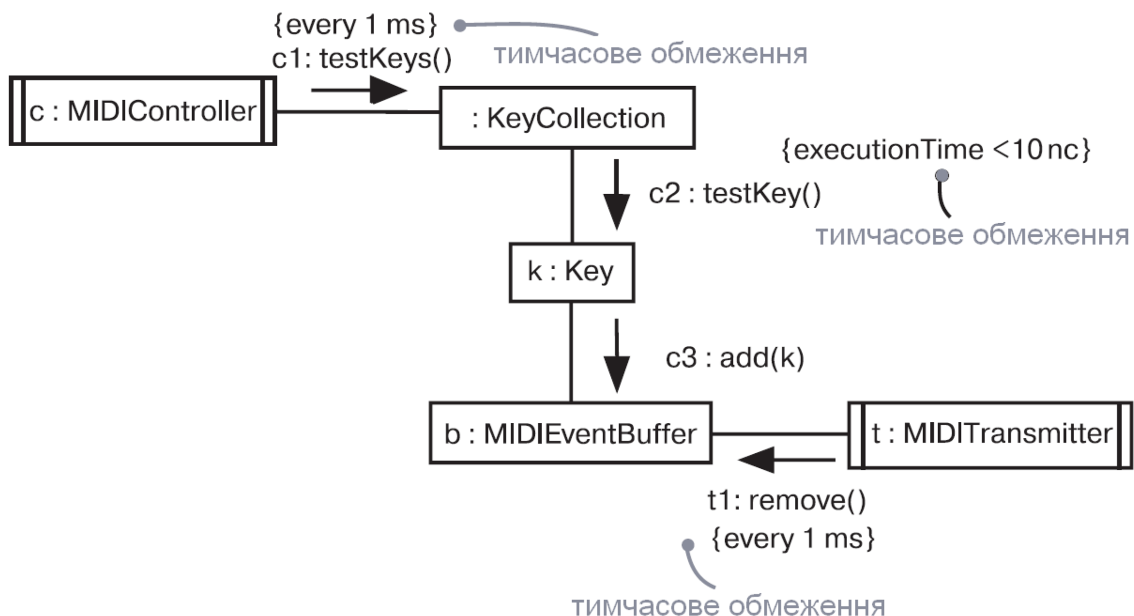


Рис. 4.4.2. Час

РС за своєю природою складаються з компонентів, фізично розподілених по різних вузлах. Дуже часто **місце розташування** (location) компонентів фіксується в момент

установки системи. Але зустрічаються й такі системи, у яких компоненти мігрують із одного вузла на інший.

В UML вигляд системи з точки зору розміщення моделюється за допомогою діаграм розміщення, що описують топологію процесорів і пристроїв, на яких функціонує система. В цих вузлах розміщаються артефакти, виконавчі модулі, бібліотеки і таблиці. Кожен екземпляр вузла володіє власними екземплярами тих або інших артефактів, а кожен екземпляр артефакту належить рівно одному екземпляру вузла (хоча різні екземпляри артефакту одного виду можуть перебувати на різних вузлах).

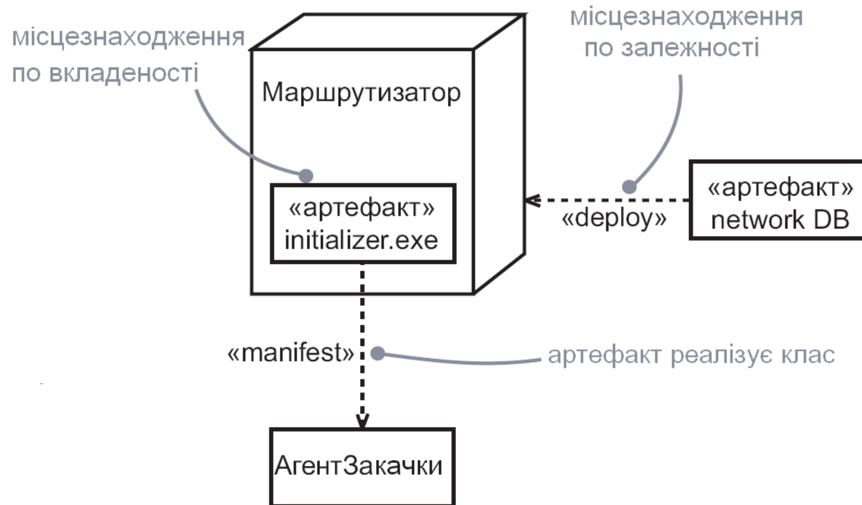


Рис. 4.4.3. Місцезнаходження

Компоненти і класи можуть бути матеріалізовані у вигляді артефактів. На рис. 4.4.3 клас Loadagent (Агент Ззавантаження) матеріалізований у вигляді артефакту initializer.exe, який розміщується на вузлі типу Router (Маршрутизатор). Як бачимо з рис. 4.4.3, моделювати положення артефакту в UML можна двома способами. По-перше, як у випадку з Router, можна фізично вмістити елемент (його текстове або графічний опис) у додатковий розділ вміщаючого вузла. По-друге, можна використовувати залежність із ключовим словом deploy від артефакту до вузла, який його містить.

Абсолютний час події і відносний час між подіями є основними тимчасовими аспектами СРЧ, при моделюванні яких знаходять застосування тимчасові обмеження.

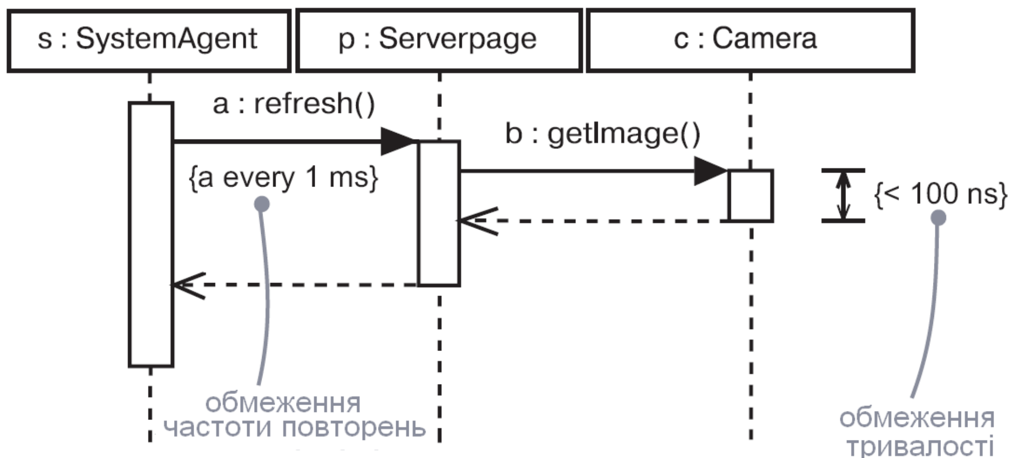


Рис. 4.4.4. Моделювання тимчасових обмежень

При моделюванні тимчасових обмежень потрібно: для кожної події у взаємодії розглянути, чи повинна вона починатися в певний абсолютний момент часу та промоделювати цю властивість за допомогою тимчасового обмеження на повідомлення; для кожної послідовності, що представляє інтерес, повідомлень у взаємодії розглянути, чи обмежений час її виконання, промоделювати її за допомогою тимчасового обмеження на послідовність. [13]

Наприклад, ліве обмеження на рис. 4.4.4 специфікує початковий час повторюваної події виклику refresh. Тимчасове обмеження, що перебуває праворуч, специфікує максимальну тривалість виклику getImage. Як правило, рекомендується вибирати для повідомлень короткі імена, щоб не плутати їх з іменами операцій.

При моделюванні топології розподіленої системи слід розглянути фізичне розташування як вузлів, так і артефактів. Якщо в центрі уваги перебуває керування конфігурацією розгорнутої системи, то моделювання розподілу вузлів особливо важливе для ВСКД розміщення таких фізичних сутностей, як модулі, бібліотеки і таблиці. Якщо більше цікавить функціональність, масштабованість і пропускна здатність системи, то важливіше всього моделювати розподіл об’єктів.[1,10]

Прийняти рішення щодо розподілу об’єктів в системі складно, і не тільки тому, що воно тісно пов’язане з питаннями паралелізму. При моделюванні розподілу об'єктів слід: для кожного класу об’єктів розглянути місцезнаходження його посилань, виявити всіх його сусідів і їх місце розташування. Сильно зв’язане розташування означає, що логічно сусідні об’єкти перебувають поруч, а слабко зв’язане – що вони фізично вилучені один від одного (*при обміні інформацією між ними будуть тимчасові затримки*). Бажано розміщувати об'єкти поруч із дійовими особами, які ними маніпулюють; розглянути зразки взаємодії між взаємозалежними наборами об’єктів, розташувати поруч набори тісно взаємодіючих об’єктів, щоб знизити витрати на комунікацію та розділити набори слабо взаємодіючих об’єктів; розглянути розподіл обов’язків у системі, перерозподілити об’єкти так, щоб збалансувати завантаження кожного вузла (*враховувати безпеку, мінливість і якість послуг*); співвіднести об’єкти з артефактами, щоб тісно зв’язані об’єкти виявилися в тому самому артефакті; співвіднести артефакти з вузлами, щоб обчислювальні потреби кожного вузла виявилися в межах його можливостей, при необхідності додати додаткові вузли. Збалансувати продуктивність і витрати на комунікацію, розміщуючи тісно зв'язані артефакти на одному вузлі. [11,13]

На рис. 4.4.5 представлена діаграма об’єктів, яка моделює розподіл об’єктів у системі роздрібної торгівлі. Цінність цієї діаграми в тому, що вона дозволяє візуалізувати фізичне розміщення ключових об'єктів. Очевидно, два об’єкти Order (Замовлення) і Sales (Продажі) перебувають у вузлі Workstation (Робоча станція), два інших (Observeragent – Агент спостереження й Product – Продукт) у вузлі Server і один (Producttable – Таблиця продуктів) у вузлі DataWarehouse (Сховище даних).

Добре структурована модель СРЧ описує тільки ті просторово-часові властивості, які необхідні й достатні для розуміння бажаної поведінки системи; централізує використання цих властивостей так, щоб їх було легко знайти й модифікувати.

Зображуючи в UML просторову або часову властивість, потрібно: давати позначкам часу осмислені імена (*відповідні до повідомлень*); проводити явну відмінність між тимчасовими виразами, значеннями яких є абсолютний і відносний час; показувати просторові властивості тільки тоді, коли важливо візуалізувати місцезнаходження елементів у розгорнутій системі; для складніших випадків використовувати профіль

UML «**UML Profile for Schedulability, Performance, and Time»** (*дана специфікація OMG призначена для високопродуктивних СРЧ*).

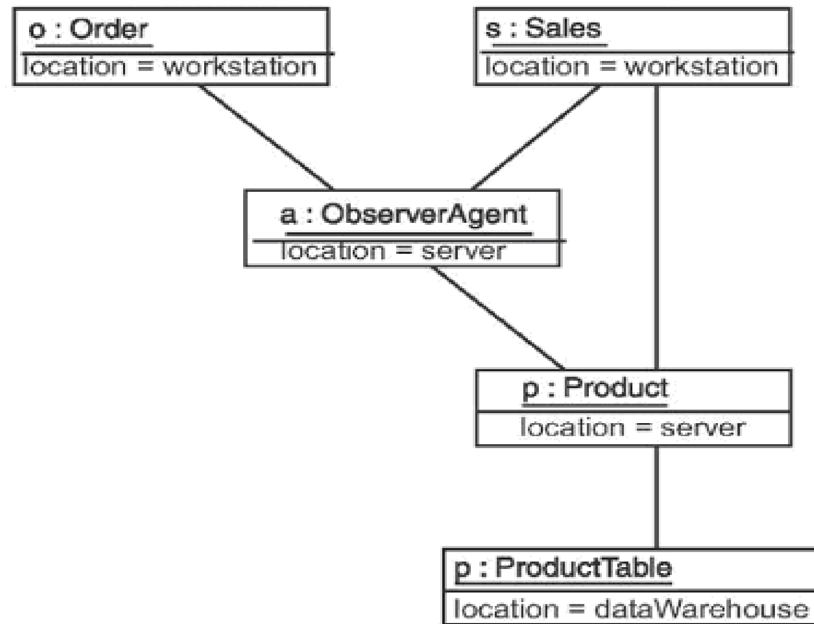


Рис. 4.4.5. Моделювання розподілених об'єктів

4.5. Діаграми станів

Основні питання:

- *Моделювання реактивних об'єктів*
- *Пряме і зворотне проектування*

Діаграми станів як і діаграми діяльності використовуються для моделювання ЖЦ об'єкта. Діаграма станів показує кінцевий автомат. Однак у той час, як діаграма діяльності демонструє потік керування від однієї діяльності до іншої через множину об'єктів, діаграма станів відображає потік керування від стану до стану всередині окремого об'єкта. Діаграми стану застосовуються для моделювання динамічних аспектів поведінки систем.

Інвестора, який фінансує будівництво хмарочосу, не будуть цікавити подробиці процесу будівництва. Вибір матеріалу, планування закупівель, безліч нарад для обговорення деталей – усе це важливо для будівельників, але мало цікавить того, хто фінансує проект. Інвестор зацікавлений у вигідному поверненні своїх інвестицій і в тому, щоб захистити їх від ризику. Дуже довірливий капіталовкладник дає забудовникові купу грошей, потім надовго зникає і повертається тільки тоді, коли забудовник готовий вручити йому ключі від будинку. Такий інвестор насправді зацікавлений лише в кінцевому результаті будівництва.

Прагматичніший інвестор у цілому довіряє забудовникові, однак перш ніж розлучитися з грішми, воліє небагато постежити за роботою. З цією метою обережний інвестор установлює чіткі етапи виконання проекту, завершення кожного з яких буде означати виконання певного обсягу робіт і послужить сиґналом для фінансування наступного етапу. Наприклад, спочатку невелика частина грошей іде на оплату роботи архітектора. Після того, як архітектурний проект затверджений, трохи більша частина грошей відпускається на здійснення інженерних (розрахункових) робіт. Після завершення даного етапу, якщо

зацікавлені сторони задоволені результатом, ще більша частина грошей виділяється для того, щоб забудовник міг почати копати котлован.

Увесь процес будівництва – від риття котловану до отримання сертифіката володіння будинком розділений на етапи, кожен з яких фіксує деякий проміжний стан проекту: архітектурна частина, інженерна частина, земельні роботи, підведення інфраструктури, зведення стін і т.д. Для інвестора бачити стан будівництва *важливіше, ніж стежити за* потоком діяльності*; останнє має більше значення для будівельника і може здійснюватися за допомогою графіків Перта, призначених для моделювання потоку робіт проекту.*

Аналогічно при моделюванні програмних систем виявиться, що найбільш природний спосіб ВСКД поведінки деякого виду об'єктів – **зосередитися на потоці керування**, що веде від *стану* до *стану*, а не від однієї діяльності до іншої. Друге зазвичай робиться за допомогою блок-схем (*в* UML *– за допомогою діаграм діяльності*). Розглянемо модель вбудованої домашньої системи сиґналізації. Система працює безупинно, реагуючи на такі зовнішні події, як наприклад, розбивання вікна. До всього характеризується описом стабільних станів (наприклад, Idle – Простий, Armed – Готовність, Active – Активність, Checking – Перевірка і т.п.), подій, що ініціюють перехід з одного стану в інший, а також дій, що виконуються при кожному такому переході. [1,2,10]

В UML керована подіями поведінка об'єкта моделюється за допомогою **діаграм станів**. Діаграма станів – це просто уявлення автомата, яке підкреслює потік керування від одного стану до іншого (рис. 4.5.1).

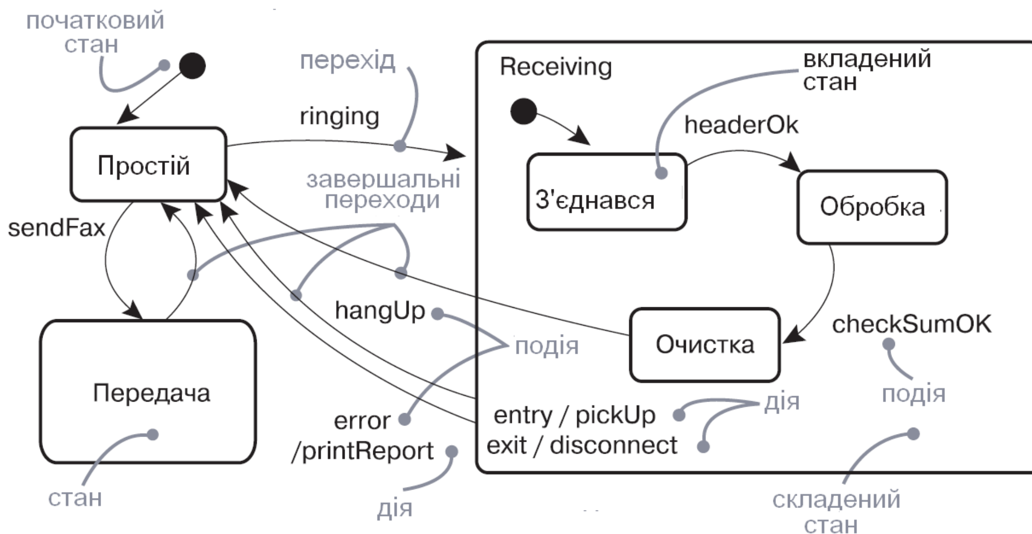


Рис. 4.5.1. Діаграма станів

Діаграма станів (state diagram) показує автомат, зосереджуючи увагу на потоці керування від одного стану до іншого. Зображується у вигляді графа з вершинами й дугами (ребрами). [2б13]

Автомат (state machine) – опис послідовності станів, через які проходить об'єкт протягом життєвого циклу, реагуючи на події, а також опис реакції на ці події. [2]

Стан (state) – ситуація в життєвому циклі об'єкта, протягом якої він задовольняє деяку умову, виконує деяку діяльність або очікує деякої події. [1,2]

Подія (event) – специфікація істотного факту, який відбувається в часі й просторі. У контексті автомата подія – це вплив, який викликає перехід між станами. [2]

Перехід (transition) – зв’язок між двома станами, який показує, що об’єкт, який перебуває в першому стані, повинен виконати деякі дії і перейти в другий, як тільки відбудеться певна подія і будуть виконані певні умови. [13]

Діяльність (activity) специфікує роботу, що відбувається усередині автомата. [2]

Дія (action) – примітивне виконуване обчислення, яке призводить до зміни стану моделі або повернення значення. [1,2]

Діаграма станів має властивості, загальні для всіх діаграм – ім’я і графічний зміст, що є проекцією моделі. Від інших діаграм вона відрізняється саме своїм змістом. Діаграми стану містять прості й складені стани, а також переходи, дії та події, включаючи примітки й обмеження.

Діаграми станів застосовуються для моделювання динамічних аспектів поведінки системи (зумовлена порядком виникнення подій поведінка об’єктів будь-якого виду в будь-якому зображенні системної архітектури, включаючи класи (у т.ч. активні), інтерфейси, компоненти і вузли). Діаграми станів можна використовувати для моделювання окремого динамічного аспекту системи, до того ж у контексті майже будь-якого елемента моделі (*підсистеми, класу*), можна приєднувати до ВВ (*для моделювання сценаріїв*).

При моделюванні динамічних аспектів системи, класу або ВВ діаграми станів зазвичай застосовуються для моделювання реактивних об’єктів.

Реактивний (reactive), або **керований подіями**, об’єкт – такий, поведінка якого найкраще характеризується його реакцією на події, прийняті їм ззовні його контексту. Зазвичай реактивний об’єкт простоює, чекаючи події. Коли він отримує цю подію, його реакція звичайно залежить від попередніх подій. Відреагувавши певним чином, об’єкт знову повертається в стан простою, очікуючи наступної події. Розглядаючи подібний об’єкт, ви зосереджуєте увагу на його стабільних станах, подіях, що ініціюють переходи від одного стану до іншого, а також на діях, які виконуються при кожній зміні стану.

Найчастіше діаграми станів використовуються при моделюванні поведінки реактивних об'єктів (екземплярів класів, ВВ і системи в цілому). У той час, як взаємодії моделюють поведінку набору об’єктів, що працюють разом, діаграма станів моделює поведінку окремого об’єкту під час його ЖЦ. Діаграма станів моделює потік керування від однієї події до іншої (*діаграма діяльності – потік керування від однієї діяльності до іншої*).

При моделюванні поведінки реактивного об’єкта, по суті, специфікуються три речі: стабільні стани, у яких об’єкт може перебувати, події, що викликають переходи від стану до стану, і дії що, виконуються при кожній зміні станів. Крім того, моделювання поведінки реактивного об'єкта включає моделювання його ЖЦ з виділенням стабільних станів, у яких він може перебувати. **Стабільний стан** представляє умову, при якій об’єкт може існувати протягом деякого певного періоду часу. Коли відбувається подія, об’єкт може переходити з одного стану в інший. Ці події також можуть викликати переходи в себе і внутрішні переходи, коли вихідний і цільовий стану об’єкта збігаються. Реагуючи на події і зміни стану, об'єкт може виконувати дії.

Щоб змоделювати реактивний об’єкт, необхідно: вибрати контекст автомата – клас, ВВ чи систему в цілому; встановити початковий і кінцевий стани об'єкта (можливо, щоб управляти іншими частинами моделі, знадобиться також установити перед- і післяумови початкового і кінцевого станів відповідно); прийняти рішення щодо стабільних станів об’єкта, розглянувши умови, в яких об’єкт може існувати протягом певного періоду часу. Почати зі станів вищого рівня і тільки після цього перейти до розгляду можливих підстанів; прийняти рішення відносно осмисленого часткового впорядкування станів

протягом часу життя об'єкта; прийняти рішення щодо подій, які можуть викликати переходи з одного стану в інший. Змоделювати ці події як тригери переходів між станами; Приєднати дії до переходів (як у машині Мілі) і/або до станів (як у машині Мура). Розглянути можливості спрощення автомата за рахунок застосування підстанів, розгалужень, поділів, з'єднань та історичних станів. Переконатися, що всі стани досяжні при деякому стіканні подій. Перевірити, чи немає яких-небудь «мертвих» станів, з яких об'єкт не зможе вийти ні при якій комбінації подій. Провести трасування автомата (вручну або із застосуванням інструментальних засобів), щоб підтвердити відповідність очікуваним послідовностям подій і реакціям на них. [5,13]

Як приклад, на рис. 4.5.2 показана діаграма станів, призначена для розбору деякої простої контекстно-вільної мови – схожої до того, що можна зустріти в системах, які опрацьовують вихідні й вхідні потоки повідомлень XML.

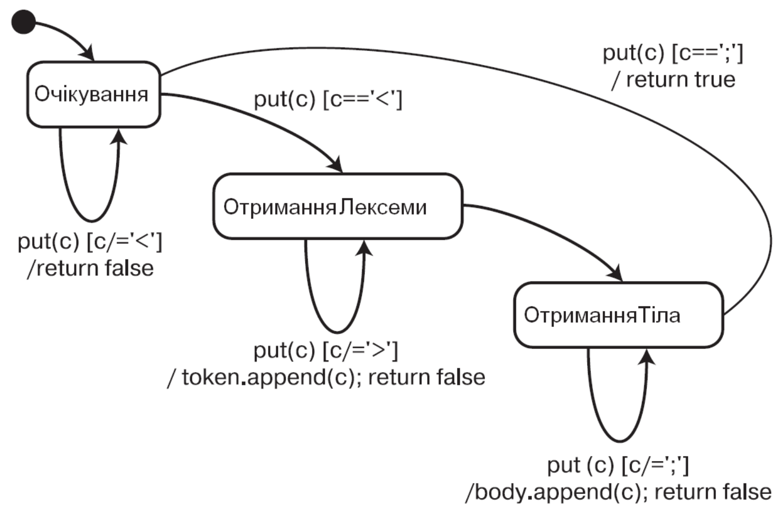


Рис. 4.5.2. Моделювання реактивних об'єктів

У цьому випадку автомат призначений для розбору потоку символів, відповідних до синтаксису

message : '<' string '>' string ';'

Перший рядок представляє тег, другий – тіло повідомлення. З отриманого потоку символів приймаються повідомлення тільки із правильним синтаксисом.

Як показано на рисунку, в автомата є тільки три стабільні стани: Waiting (Очікування), Gettingtoken (Отримання слова) і Gettingbody (Отримання тіла). Стан спроектований у вигляді машини Мілі – з діями, приєднаними до переходів. Фактично є тільки одна подія, яка може цікавити нас при розгляді цього автомата – виклик put з параметром c (символом). У стані Waiting автомат відкидає будь-який символ, який не означає початок слова (оголошеного в захисній умові). Як тільки зустрічається початок слова, стан об'єкта змінюється на Gettingtoken. Перебуваючи в цьому стані, автомат зберігає кожен символ, який не є закінченням слова. Коли приходить символ закінчення слова, стан об'єкта змінюється на Gettingbody. Перебуваючи в цьому стані, автомат зберігає кожен символ, який не означає закінчення тіла повідомлення (що оголошене в захисній умові). Як тільки надходить закінчення повідомлення, стан об'єкта змінюється на Waiting і повертається значення, яке вказує на те, що повідомлення було розібрано й автомат готовий отримувати наступне.

Відзначимо, що останній стан специфікує автомат, який працює безупинно, – кінцевого стану не існує.

Пряме проектування (створення коду з моделі) можливе для діаграми станів, особливо якщо контекстом такої є клас. Наприклад, використовуючи діаграму станів, показану на рис. 4.5.2, інструмент прямого проектування може згенерувати наступний Java-код для класу Messageparser (Аналізатор повідомлень):

```
class MessageParser
 {
   public
   boolean  put(char c)
      {
       switch (state)
        {
          case Waiting: if (c == ‘<’) { state =Gettingtoken;
                                         token = new Stringbuffer();
                                         body = new Stringbutfer();}
                         break;
        case Gettingtoken : if (c == ‘>’) state = Gettingbody;
                           else token.append(c);  break;
        case Gettingbody :  if (c == ';') state = Waiting;
                            else  body, append(c);
                             return true;
        }  // end switch
        return false;
        Stringbuffer gettoken() {
        return token;
       }  // end put()

   Stringbuffer getbody()
    { return body; }
 private
   final static Int Waiting = 0;
   final static Int Gettingtoken = 1;
   final static Int Gettingbody = 2;
   Int state = Waiting;
   StringBuffer token, body;
} // end class MessageParser
```

Інструмент прямого проектування повинен генерувати необхідні закриті атрибути і кінцеві статичні константи.

Зворотне проектування теоретично можливо, але практично не дуже корисно. Вибір того, що становить осмислений стан об'єкта, залежить від погляду проектувальника. Інструменти зворотного проектування не мають здатності до абстрагування, а тому не можуть генерувати осмислені діаграми станів. Цікавіше, чому зворотне проектування моделі з коду, може виявити анімацію моделі на основі роботи реальної системи. Наприклад, якщо розглянути діаграму, що на рис. 4.5.2, інструментальний засіб міг би анімувати стани автомата на ній у міру того, як їх ухвалювала б працююча система. Аналогічним чином можна анімувати виконання переходів, показуючи момент отримання події і результат виконання дії. Під керуванням відладника ви могли б контролювати швидкість виконання, установлювати позначки переривання для припинення роботи в цікаві моменти часу, щоб перевірити значення атрибутів окремих об’єктів. [2,5,11]

Створюючи діаграму станів в UML, слід пам'ятати, що вона являє собою всього лише проекцію однієї моделі динамічних аспектів системи. Окремо взята діаграма станів може охопити семантику одного реактивного об'єкта, але семантику непростої системи не в змозі передати.

5. Основи моделювання архітектури ПЗ

5.1. Поняття про архітектуру та раціональний уніфікований процес

ВСКД програмних систем передбачають розгляд ПЗ з різних точок зору. Різні зацікавлені особи – кінцеві користувачі, аналітики, розробники, системні інтегратори, тестувальники і менеджери проекту мають різні погляди про проект. Кожен з них бачить систему по-різному в різні моменти його ЖЦ.

Сліпі і слон. В село, де проживали шестеро сліпих, привели слона. Сліпі, які не мали ніякого поняття, що таке слон, вирішили: *«Оскільки ми не можемо його побачити, то підемо і хоч доторкнемося до нього».* Вони підійшли до слона і кожен помацав його. *«Він схожий на дерево зі зморщеною корою. Я ледве зумів його обняти»,* – сказав перший сліпий, охопивши його ногу. *«О, ні! Він схожий на канат»,* – сказав другий, взявши слона за хвіст. *«Що ти кажеш? Він схожий на товсту змію»,* – сказав третій сліпий, вхопивши слона за хобот. *«Ти не правий! Він схожий на великий волохатий лист лопуха»,* – сказав четвертий сліпий, помацавши слона за вухо. *«Та ні! Він схожий на велику бочку»,* – сказав п'ятий сліпий, торкнувши його за живіт. *«Ви всі помиляєтесь! Слон твердий і гладкий, як камінь, загострений на кінці. Ним можна вбити людину»,* – сказав шостий сліпий, взявши його за бивень. Вони довго сперечалися і кожен наполягав на своєму…

Кожен говорив про слона по-різному і був правий по-своєму, хоча ніхто з них так і не зміг представити цілого слона.

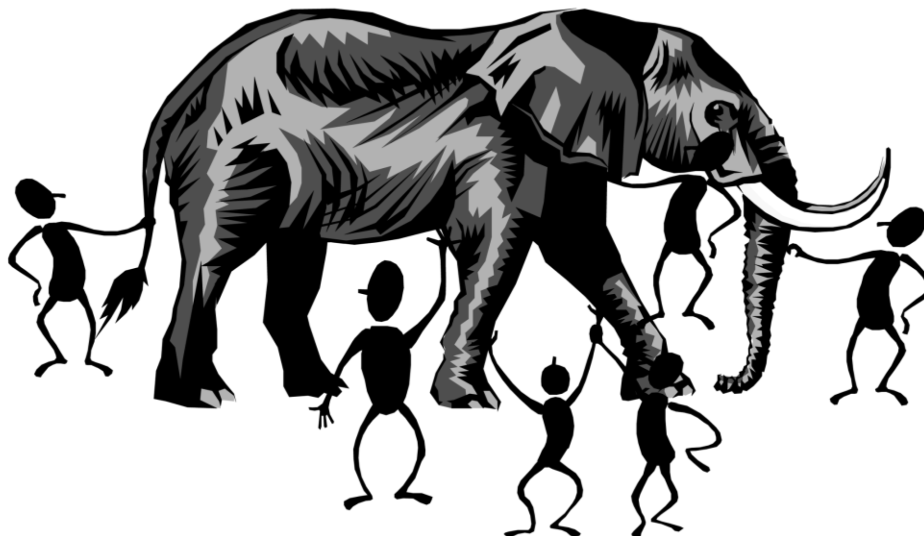


Рис. 5.1.1. Слон і сліпі

Системна архітектура є найважливішим артефактом, який може бути використаний для керування всіма цими різноманітними точками зору і тим самим

управляє ітеративним і покроковим процесами розроблення системи протягом усього її ЖЦ. **Архітектура** – це набір суттєвих рішень відносно:

- організації програмної системи;
- вибору структурних елементів, що складають систему, їх інтерфейсів;
- поведінки цих елементів, специфікованої в їхніх коопераціях;
- об’єднання цих структурних елементів та елементів поведінки у більші підсистеми;
- архітектурного стилю, що визначає організацію системи: статичні й динамічні елементи, їх інтерфейси, кооперацію і композицію.

Архітектура ПЗ стосується не тільки його структури і поведінки, але й вимог та особливостей використання, функціональності, продуктивності, гнучкості, можливості повторного застосування коду, зрозумілості, економічних і технологічних обмежень та компромісів, а також естетичних питань.

Архітектурні вигляди. Архітектура програмної системи може бути описана за допомогою **п’яти взаємозалежних архітектурних виглядів** (рис. 5.1.2). Кожен вигляд – проекція організації і структури системи, зосереджена на певному її аспекті.

Рис. 5.1.2. Моделювання виглядів системної архітектури

Вигляд з точки зору **варіантів використання** (Use case view) системи охоплює ВВ, що описують поведінку системи з погляду кінцевих користувачів, аналітиків і тестувальників. Цей вигляд специфікує не дійсну організацію програмної системи, а ті рушійні сили, що формують системну архітектуру. В UML статичні аспекти цього вигляду передаються діаграмами ВВ, а динамічні його аспекти – діаграмами взаємодій, станів і діяльності. [2,10,13]

Вигляд з точки зору **проектування** (Design view) охоплює класи, інтерфейси і кооперації, що формують словник проблеми і її рішення. Цей вигляд, в основному, підтримує функціональні вимоги до системи, тобто сервіс, який вона повинна надавати кінцевим користувачам. Статичні аспекти цього вигляду в UML зосереджені в діаграмах класів і об'єктів, а динамічні передаються діаграмами взаємодій, станів і діяльності. Також використовуються **діаграми внутрішньої структури класів**. [2,13]

Вигляд з точки зору **процесів** (Process view) системи показує потік керування, що проходить через різні її частини, включаючи можливі механізми паралелізму і синхронізації. Цей вигляд стосується продуктивності, масштабованості й пропускної

здатності системи. Статичні і динамічні аспекти цього вигляду в UML представлені в деяких видах діаграм, що використовуються у вигляді проектування, але сформовані на активних класах, які керують системою, і переданих між ними повідомленнях. [2,10,12]

Вигляд з точки зору **реалізації** (Implementation view) системи охоплює артефакти, що використовуються для складання і фізичної реалізації системи. Цей вигляд, у першу чергу, відноситься до керування конфігурацією версій системи, що складається з окремих файлів і компонентів, зібраних різними способами для формування працюючої системи. Він також пов'язаний з відображенням з логічних класів і компонентів у фізичні артефакти. В UML статичні аспекти цього вигляду відображені в діаграмах артефактів, динамічні аспекти – у діаграмах взаємодії, станів і діяльності. [2,10,13]

Вигляд з точки зору **розгортання** (Deployment view) системи охоплює вузли, що утворюють топологію устаткування, на якому працює система. Цей вигляд, в основному, пов’язаний з розподілом, доставкою та встановленням частин, що складають фізичну систему. Його статичні аспекти в UML описуються діаграмами розміщення, а динамічні – діаграмами взаємодій, станів і діяльності. [2,5,13]

Кожен із цих п’яти виглядів може бути достатнім для різних зацікавлених осіб, що мають відношення до розроблення і експлуатації системи, дозволяючи їм зосередитися тільки на тих аспектах архітектури, які безпосередньо їх стосуються. Однак усі ці вигляди також взаємодіють один з одним. Вузли вигляду розгортання містять компоненти з вигляду реалізації, які, у свою чергу, є фізичною реалізацію класів, інтерфейсів, кооперацій і активних класів з виглядів проектування і процесів. UML дозволяє виразити кожен з цих п'яти виглядів і їх взаємодію.

Життєвий цикл розроблення ПЗ

UML не прив’язана безпосередньо до певного процесу розроблення ПЗ. Щоб якомога більше скористатися перевагами процесу розроблення ПЗ, варто розглянути його як процес:

- керований варіантами використання;
- що ґрунтується на архітектурі (архітектурно-центрований);
- ітеративниий та інкрементний (покроковий).

Процес, керований ВВ (use case driven), означає, що ВВ використовуються в якості *первинних робочих продуктів*, на підставі яких визначається бажана поведінка системи, перевіряється і підтверджується правильність обраної системної архітектури, проводиться тестування і проводиться спілкування із зацікавленими особами, що мають відношення до проекту.

Процес, що ґрунтується на архітектурі (architecture-centric), означає, що архітектура системи є первинним робочим продуктом для процесу концептуалізації, конструювання, керування й розвитку системи під час її розроблення. [2]

Ітеративний процес (iterative) – містить керування потоком виконавчих версій системи. [4,10]

Інкрементний або покроковий процес (incremental) ґрунтується на безперервній інтеґрацій системної архітектури з метою випуску версій, кожна наступна з яких удосконалена в порівнянні з попередньою. [2,4,10]

Процесом з керованими ризиками (risk-driven). Процес, що є ітеративним і покроковим, називають процесом з керованим ризиками (risk-driven), оскільки при випуску кожної нової версії (**релізу**) серйозна увага приділяється виявленню чинників, що складають найбільший ризик для проекту в цілому, та зведення їх до мінімуму.

Такий керований ВВ процес, сконцентрований на архітектурі, ітеративний і покроковий може бути розбитий на фази. [2,4,8,12]

Фаза (phase) – це відрізок часу між двома важливими **контрольними точками** (milestones) процесу, у яких досягаються добре визначені певні цілі, завершене створення робочих продуктів і ухвалюється рішення про перехід до наступної фази.

Існують чотири фази ЖЦ розроблення ПЗ (рис. 5.1.3): аналіз і планування вимог, проектування, побудова і впровадження. На діаграмі потік робіт зображений у контексті цих фаз, у результаті чого демонструється різний рівень концентрації на виконанні цього потоку робіт для різних його складових у різних фазах.

Аналіз і планування вимог (Requirements) – перша фаза процесу, впродовж якої початкова ідея набуває достатнього обґрунтування (принаймні, внутрішнього) для забезпечення переходу до фази розробки. На цій фазі добре розроблена ідея перетворюється в концепцію готового продукту, і створюється бізнес-план розролення цього продукту. На цій фазі повинні бути отримані відповіді на питання:

- *Що повинна робити система, в першу чергу, для її основних користувачів?*
- *Який вигляд повинна мати архітектура системи?*
- *Який план і в що обійдеться розроблення продукту?*

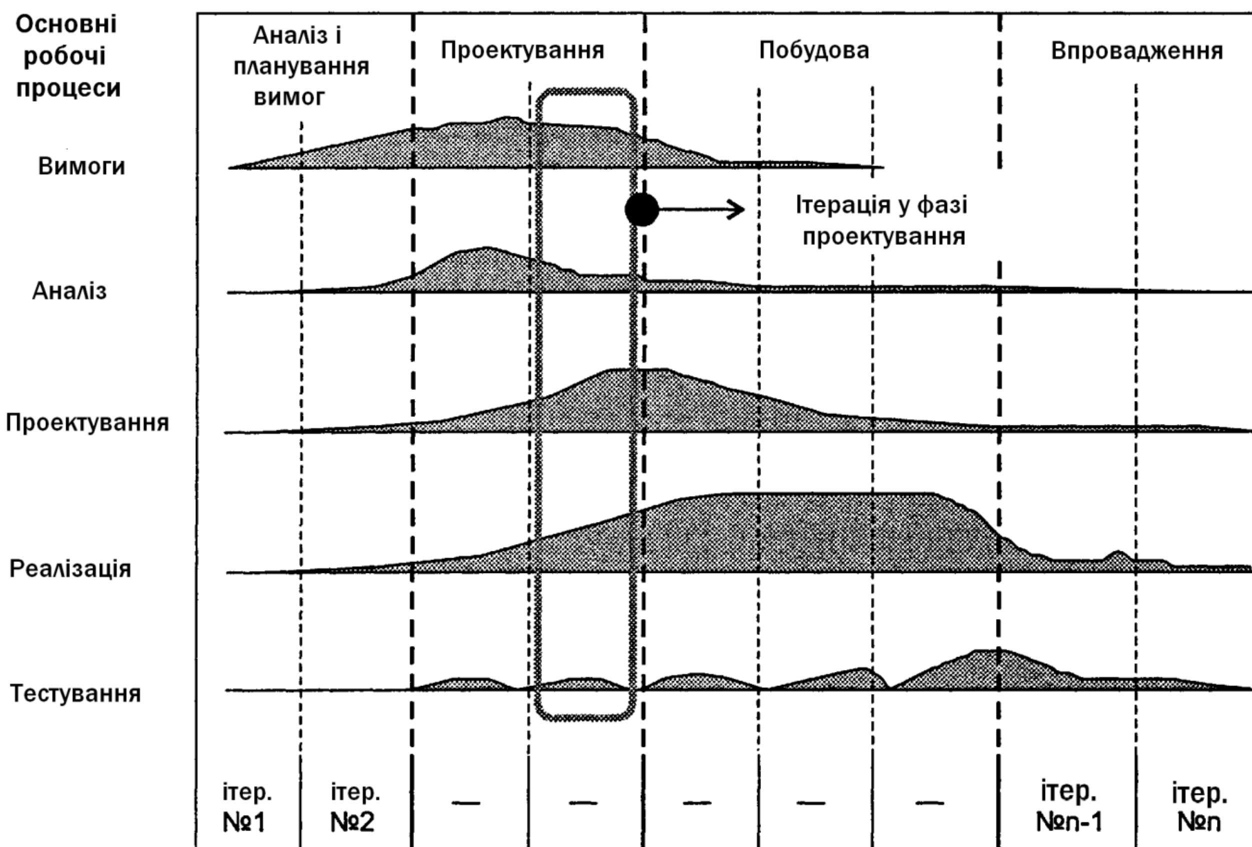


Рис. 5.1.3. Життєвий цикл розроблення програмного забезпечення

У **фазі проектування** (Design) детально описуються більшість ВВ і розробляється архітектура системи. Між архітектурою системи і системою є прямий і нерозривний зв'язок. Архітектурний прототип набуває форми, у якій він може бути представлений користувачам. На цьому етапі вимоги до системи, особливо критерії оцінювання,

постійно переглядаються відповідно до мінливих потреб. Також виділяються необхідні ресурси для зменшення ризиків.

У фазі побудови (Construction) відбувається створення продукту – до архітектури (*скелету*) додаються закінчені програми (*м'язи*). На цій фазі базовий рівень архітектури розростається до повної розвиненої системи. Концепції розвиваються до продукту, готового до передавання користувачам. У фазі побудови продукт до рівня, що містить в собі всі ВВ, які керівництво і замовник домовилися включити в поточний випуск. Архітектура системи стабільна, однак, оскільки розробники можуть знайти найкращі способи структурування системи, від них можуть виходити пропозиції про внесення в архітектуру системи незначних змін.

Упровадження (Transition) – четверта фаза процесу, у ході якої ПЗ передається користувачам. Але процес розроблення не часто завершується на цьому, тому що навіть на даній фазі система безупинно удосконалюється: усуваються помилки і додаються нові функціональні можливості, що не ввійшли в більш ранні версії.

Один елемент, який відрізняє процес розроблення в цілому і є присутнім у всіх чотирьох фазах, – це ітерація – певна чітка послідовність дій із зрозуміло сформульованим планом і критеріями оцінювання, що призводить до появи нової версії системи, яка може бути виконана, протестована й оцінена. Система, що реалізовується, не обов'язково повинна бути реалізована в повному обсязі. Оскільки ітерація породжує продукт, що виконується, досягнутий прогрес і рівень ризику можуть бути оцінені заново після кожної такої ітерації. Це означає, що життєвий цикл розроблення програмного забезпечення можна характеризувати як безперервний потік версій, які виконуються, реалізують архітектуру системи із проміжною корекцією для зниження потенційного ризику. Це підкреслює значення архітектури як найважливішого елемента програмної системи й фокусує UML на моделюванні різних її виглядів. Такий підхід до розроблення ПЗ є найефективнішим сьогодні, його називають **раціональним уніфікованим підходом** (Unified Software Development Process).

5.2. Моделювання архітектурних зразків

Основні питання:

- *Зразки і каркаси*
- *Моделювання зразків проектування*
- *Моделювання архітектурних зразків*
- *Забезпечення доступності зразків*

Механізми і каркаси як основа архітектури системи. Усі добре структуровані системи побудовані на зразках проектування (patterns), що пропонують типові вирішення проблеми в даному контексті. **Механізм** – це зразок проектування, що застосовується до співтовариства класів; **каркас** (framework) – **архітектурний зразок**, що пропонує розширюваний шаблон для додатків у деякій предметній області. Зразки використовуються для специфікації механізмів і каркасів, що утворюють архітектуру системи. Доступність зразка полягає в ідентифікації всіх «штепсельних вилок», «розеток», «кнопок», «циферблатів», за допомогою яких користувач налаштовує зразок для застосування його в певному контексті.

Існують тисячі способів зібрати будинок з купи дощок. Кожен архітектор, ґрунтуючись на власному досвіді, вибере той архітектурний стиль, який найкраще задовольняє вимоги замовника, а потім злегка підкоректовує з урахуванням побажань клієнта і обмежень, що накладаються обраним місцем, а також з місцевими традиціями, будівельними нормами і правилами. При проектуванні будинку не можна зняти з рахунку і деякі загальні проблеми, що

встановлюють певні межі архітекторським фантазіям. Є лише обмежене число перевірених способів конструювання кроків, на які опирається дах, і обмежене число способів проектування несучої стіни з дверними і віконними прорізами. Кожному будівельникові для вирішення цих типових проблем доводиться вибирати ті чи інші механізми, пристосовуючи їх до загального архітектурного стилю і місцевих норм.

Побудова програмних систем підкоряється тим же принципам. Щораз, відриваючи око від конкретних рядків коду, виявляться типові механізми, які визначають *спосіб організації класів* і інших абстракцій. Наприклад, у керованої подіями системі застосування зразка проектування, відомого під іменем «ланцюг обов'язків», – це типовий спосіб організації обробника подій. Якщо подивитись трохи вище рівня цих механізмів, то можна побачити типові каркаси, що формують архітектуру всієї системи. Наприклад, в інформаційних системах застосування тришарової архітектури розповсюджений спосіб досягнення чіткого поділу обов'язків між інтерфейсом користувача, зберіганням інформації, бізнес-об'єктами та правилами.

В UML часто доводиться моделювати зразки проектування (механізми), які можна представити у вигляді кооперацій. Аналогічним чином архітектурні зразки моделюються як каркаси, що представляються у вигляді пакетів зі стереотипами.

Для зразків обох типів в UML передбачений особливий графічний вигляд (рис. 5.2.1).

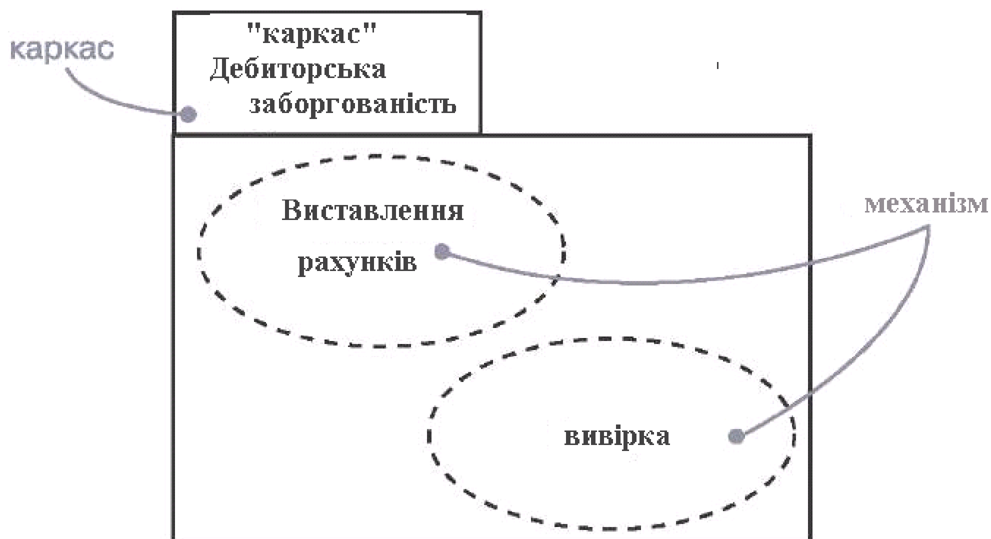


Рис. 5.2.1. Механізми і каркаси

Зразок (pattern) – це загальний розв'язок типової проблеми в даному контексті. **Механізм** – це зразок проектування, що застосовується до співтовариства класів. **Каркас** (framework) – архітектурний зразок, що пропонує розширюваний шаблон для додатків у деякій предметній області. [1,2,5]

Розроблення архітектури нової системи чи розвиток існуючої, як правило, не починається з нуля, а навпаки, колишній досвід і погодження наштовхують на застосування типових прийомів вирішення типових проблем. Наприклад, при побудові системи, що активно взаємодіє з користувачем, можна задіяти випробуваний зразок «модель-огляд-контролер» (model-view-controller), який дозволяє чітко відокремити об'єкти (модель) від їхнього зовнішнього вигляду та від агента й забезпечує їхню синхронізацію (контролер). А при створенні систем дешифрування перевіреним

способом організації системи є застосування архітектури «класної дошки» (blackboard), добре пристосованої для розв’язання складних завдань методом проб і помилок. Це – приклади зразка як типового розв'язання типових завдань у даному контексті.

Добре структурована система містить у собі множину зразків на різних рівнях абстракції. Зразки проектування описують структуру і поведінку співтовариства класів, а архітектурні зразки – структуру й поведінку системи в цілому. Зразки входять до UML і є важливою складовою словника розробника. Явно виділяючи їх у системі, остання робиться зрозумілішою і простою в супроводі. При модифікації невідомого вихідного тексту потрібно витратити багато часу, щоб здогадатися, як зв’язані між собою його частини. Якщо для цього тексту відомо, що виділені класи взаємодіють на основі механізму «публікація-підпис» (publish-and-subscribe), то отримується чіткіша візія як усе працює. Та ж ідея застосовна до системи в цілому. Одна лише фраза «*система організована як набір конвеєрів і фільтрів*» дуже багато чого говорить про системну архітектуру. Зрозуміти це, дивлячись лише на код класів, є куди складніше.

Зразки допомагають ВСКД артефакти програмної системи, зокрема реалізовувати пряме проектування системи, вибираючи відповідний набір зразків і застосовуючи їх до абстракцій, специфічних для даної предметної області, або зворотне проектування, виявляючи зразки, що містяться в ній (складніша задача). При моделюванні системи важлтвим завданням є якісно описати характерні для неї зразки з метою подальшої допомоги команді, яка в майбутньому буде повторно використовувати або модифікувати створюване ПЗ. З практичної точки зору найважливішими є зразки проектування і каркаси.

Механізми (*як зразки проектування співтовариства класів*). Існує низка типових проблем проектування, зокрема для програмістів, що працюють на одній із мов ООП (Java): як змінити клас, налаштований реагувати на деяку множину подій, щоб він також реагував на інші події, не зачіпаючи вихідного коду класу. Типове вирішення проблеми – застосування зразка адаптера (adaptor pattern), структурного зразка проектування, який конвертує один інтерфейс в інший. Цей зразок (*як багато подібних*) є настільки загальним, що доцільно дати йому певну назву, а потім використовувати його в різних моделях при виникненні подібної проблеми.

Шаблони й параметризовані кооперації. При моделюванні механізми проявляють себе двояко. По-перше, механізм просто іменує набір абстракцій, що працюють разом для реалізації певної типової поведінки системи (див. рис. 5.2.1). Такі механізми моделюються як прості кооперації, оскільки вони є всього лише **іменами** для співтовариства класів. Розкривши таку кооперацію, можна побачити її структурні аспекти (зображувані на діаграмі класів), а також поведінкові аспекти (зображувані на діаграмах взаємодії). Кооперації подібного типу охоплюють різні рівні абстракції системи, де певний конкретний клас може брати участь у кількох коопераціях.

По-друге, як показано на рис. 5.2.2, механізм іменує **шаблон** для набору абстракцій, що працюють спільно для забезпечення деякої типової поведінки. Такі механізми моделюються у вигляді **параметризованих кооперацій**, що зображуються в UML подібно до шаблонних класів. Розкривши таку кооперацію, можна побачити її структурні й поведінкові аспекти. Якщо згорнути її, то можна побачити, як зразок застосовується до системи, зв’язуючи шаблонні частини кооперації з існуючими абстракціями.

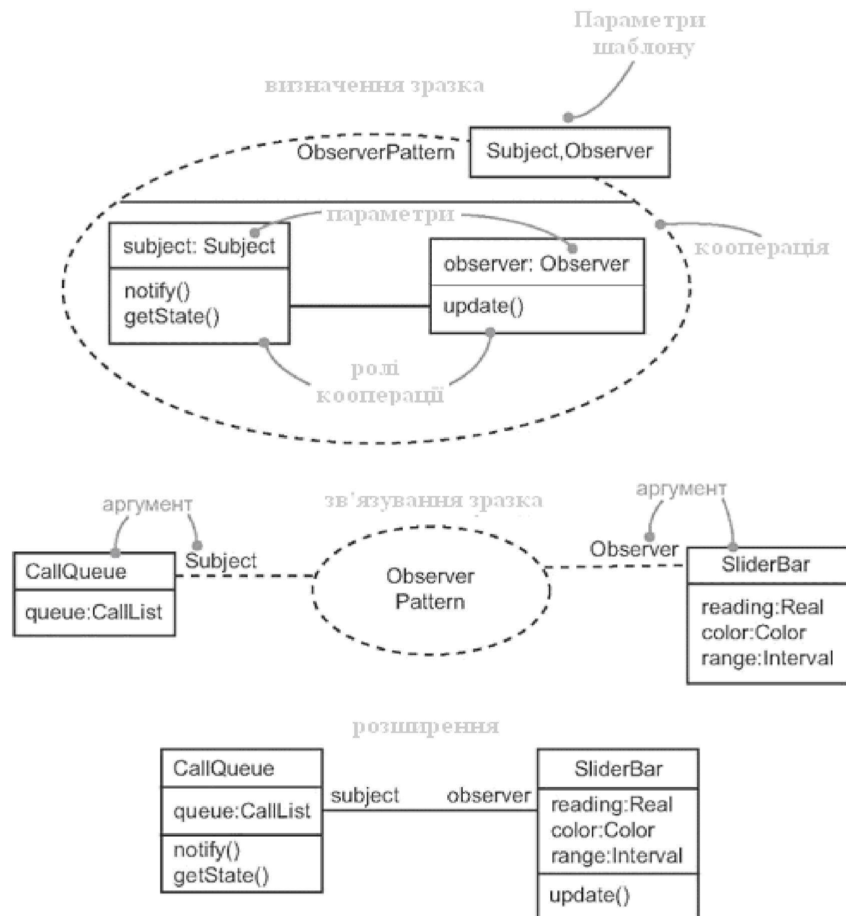


Рис. 5.2.2. Механізми

При моделюванні механізму у вигляді параметризованої кооперації описуються всі ці «*штепсельні вилки*», «*розетки*», «*кнопки*» і «*циферблати*», з допомогою яких користувач налаштовує (адаптовує) зразок для застосування його в певному контексті, змінюючи значення його параметрів. Подібні кооперації можуть з'являтися в різних частинах системи і зв'язуватися з різними абстракціями. На рис. 5.2.2 класи зразка Subject (Суб'єкт) і Observer (Спостерігач) пов'язані з конкретними класами Callqueue (Черга задач) і Slidebar (Повзунок) відповідно.**Каркас** (framework) – архітектурний зразок, що пропонує розширюваний шаблон для додатків у деякій предметній області. Наприклад, у системах реального часу часто можна зустріти архітектурний зразок «циклічний виконавець» (cyclic executive), який розділяє час на кадри й підкадри, де опрацювання відбувається в строгих тимчасових рамках. Вибір цього зразка замість керованої подіями архітектури впливає на всю систему. Даний зразок (як і його альтернатива) настільки загальний, що є сенс назвати його каркасом.

Каркас можна вважати різновидом мікроархітектури (*це більше, ніж механізм*). Він містить в собі множину механізмів, що спільно працюють над вирішенням типової проблеми для типової предметної області. Специфікуючи каркас, тим самим описується **«кістяк» архітектури** разом з усіма її органами керування, що застосовуються користувачем для адаптації до потрібного контексту.

В UML каркас моделюється у вигляді **пакета зі стереотипом**, всередині якого є механізми для кожного з виглядів системної архітектури. Він містить не тільки параметризовані кооперації, але також ВВ (що пояснюють, як треба працювати з каркасом), прості кооперації (набір абстракцій, на базі яких можна будувати систему, наприклад, шляхом породження класів-нащадків).

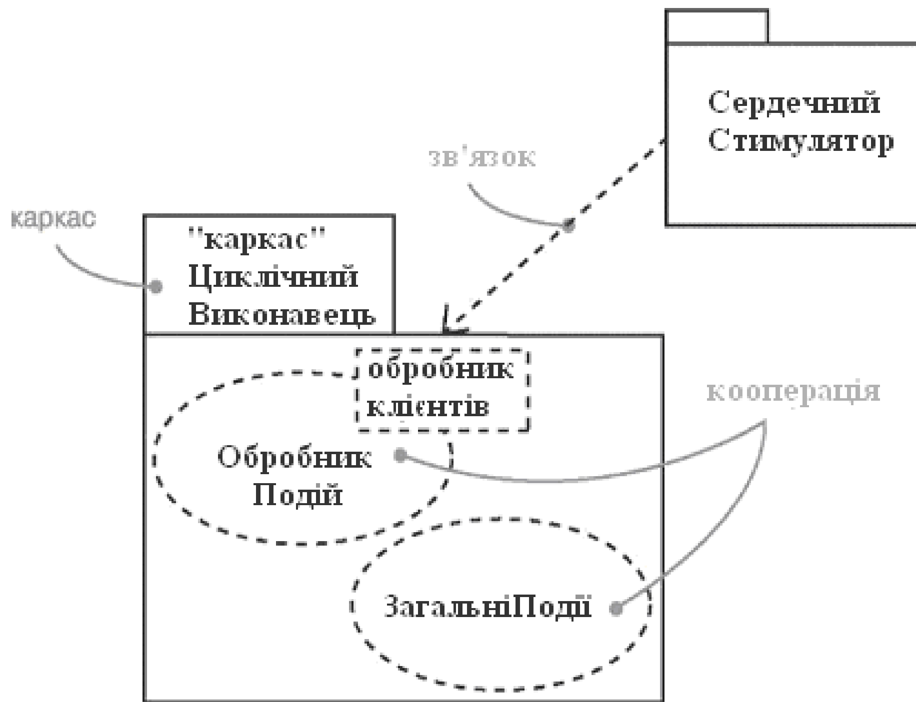


Рис. 5.2.3. Каркаси

На рис. 5.2.3 зображено такий каркас, названий Cyclicexecutive (Циклічний виконавець). Крім іншого, цей каркас включає кооперацію CommonEvents (Загальні події), що охоплює множину класів подій, і механізм EventHandler (Обробник подій), призначений для циклічного опрацювання подій. Побудований на базі цього каркаса клієнт Pacemaker (Сердечний стимулятор) може користуватися абстракціями з кооперації CommonEvents шляхом породження похідних класів, а також застосовувати механізм EventHandler.

Зразок проектування (*вигляд зовні*) зображується як параметризована кооперація. Будучи кооперацією, зразок є набором абстракцій, структура і поведінка яких повинні забезпечити в ході спільної роботи виконання певної корисної функції. Параметри кооперації іменують ті елементи, які користувач зразка повинен з чимось пов’язати. Отже, зразок проектування перетворюється в шаблон, який використовується в конкретному контексті шляхом підстановки елементів, відповідних до параметрів шаблону.

При вигляді зсередини зразок проектування представляється простою кооперацією і відображається зі своїми структурною і поведінковою складовими. Кооперація моделюється за допомогою діаграм класів (для структурної складової) і діаграм взаємодій (для поведінкової складової). Параметри кооперації іменують певні структурні елементи, які при звязуванні з певним контекстом конкретизуються абстракціями цього контексту.

Щоб змоделювати зразок проектування, необхідно: ідентифікувати типовий розв'язок типової проблеми і матеріалізувати його у вигляді механізму; змоделювати механізм як кооперацію, описавши її структурний і поведінковий аспекти; ідентифікувати ті елементи зразка проектування, які повинні бути пов'язані з елементами в конкретному контексті, і відобразити їх у вигляді параметрів кооперації.

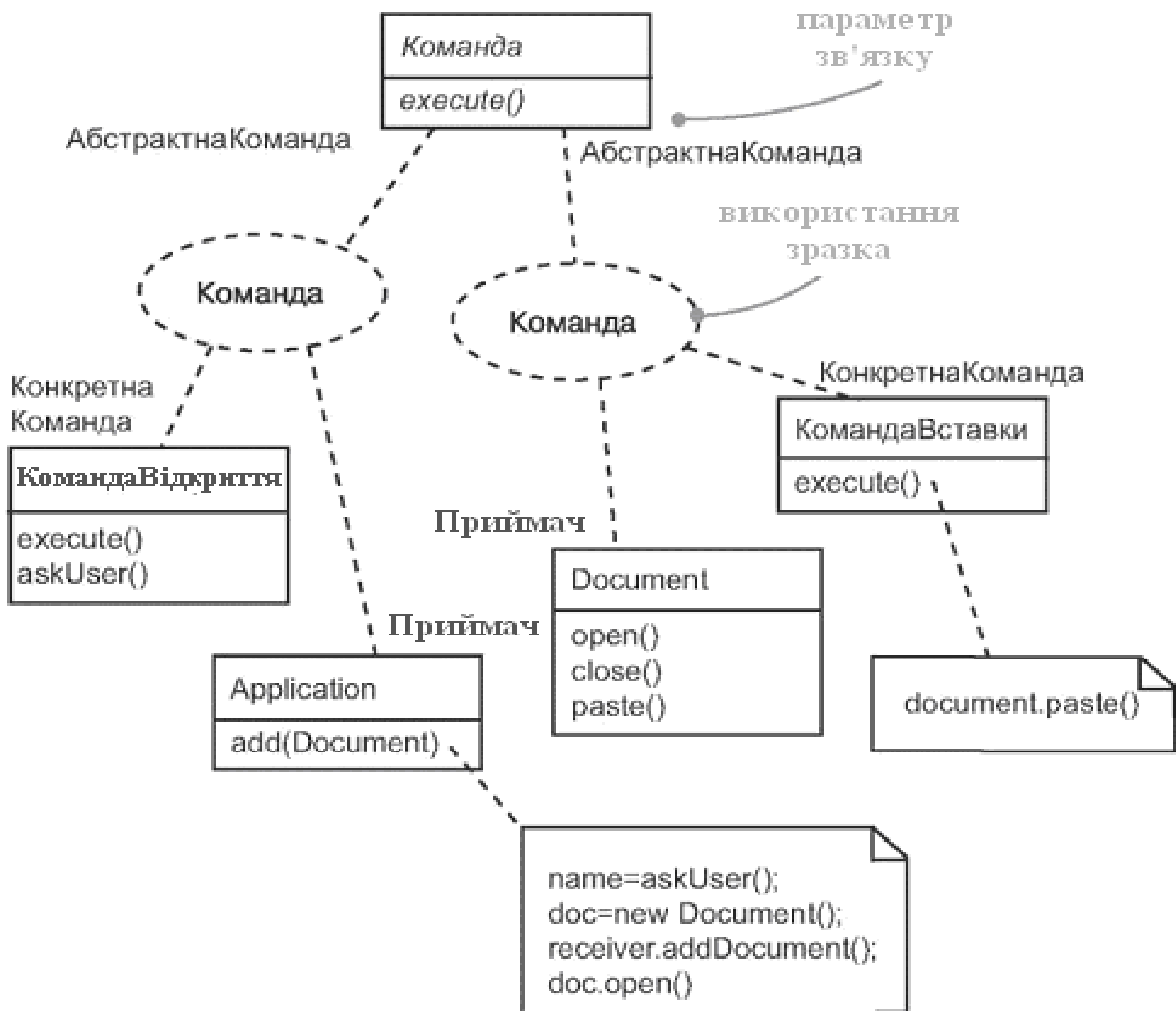


Рис. 5.2.4. Моделювання зразка проектування

На рис. 5.2.4 показане використання зразка проектування Command [*Gamma et. al. Design Patterns. - Reading, Massachusetts: Addison-Wesley, 1995*]. Як встановлено в описі даного зразка, він «інкапсулює запит у вигляді об'єкта, дозволяючи тим самим параметризувати клієнтів, що подають різні запити, ставити ці запити в чергу і протоколювати їх, а також підтримувати операції, що допускають скасування». Як бачимо з моделі, цей зразок проектування має три параметри, які, будучи застосовані до нього, повинні бути пов'язані з елементами в даному контексті. Модель демонструє два з'єднання, у яких класи Pastecommand (Команда вставки) і Opencommand (Команда відкриття) прив'язуються до параметрів зразка.

Тут параметрами є Abstractcommand (Абстрактна команда), яка повинна бути пов'язана з таким же абстрактним суперкласом у кожному випадку, Concretecommand (Конкретна команда), яка зв'язується з різними конкретними класами в різних випадках зв'язування, і Receiver (Приймач), що зв'язується з класом, над яким команда здійснює дію. Клас Command (Команда) може бути створений зразком, але оформлення його у вигляді параметра дозволяє створювати множину ієрархій команд. Класи Pastecommand і Opencommand є підкласами класу Command. Система може повторно (багаторазово) використовувати цей зразок

проектування із різноманітними зв'язками, що робить процес розроблення на основі зразків досить ефективним.

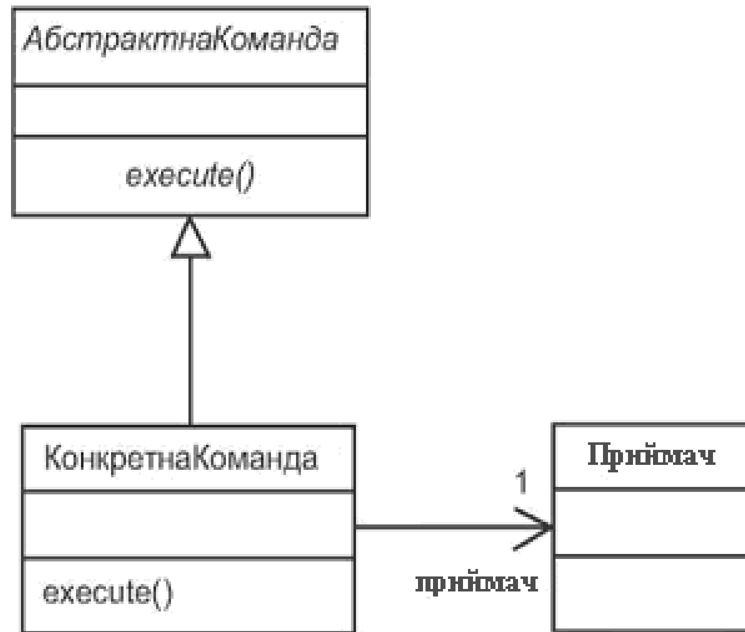


Рис. 5.2.5. Моделювання структурного аспекту зразка проектування

Щоб завершити модель зразка проектування, потрібно специфікувати його структурну і поведінкову складові, які складають внутрішній зміст кооперації. На рис. 5.2.5 зображена діаграма класів, що представляє структуру цього зразка. Тут показано, як використовуються класи, названі параметрами зразка. Рис. 5.2.6 демонструє діаграму послідовності, що відображає поведінку того ж зразка.

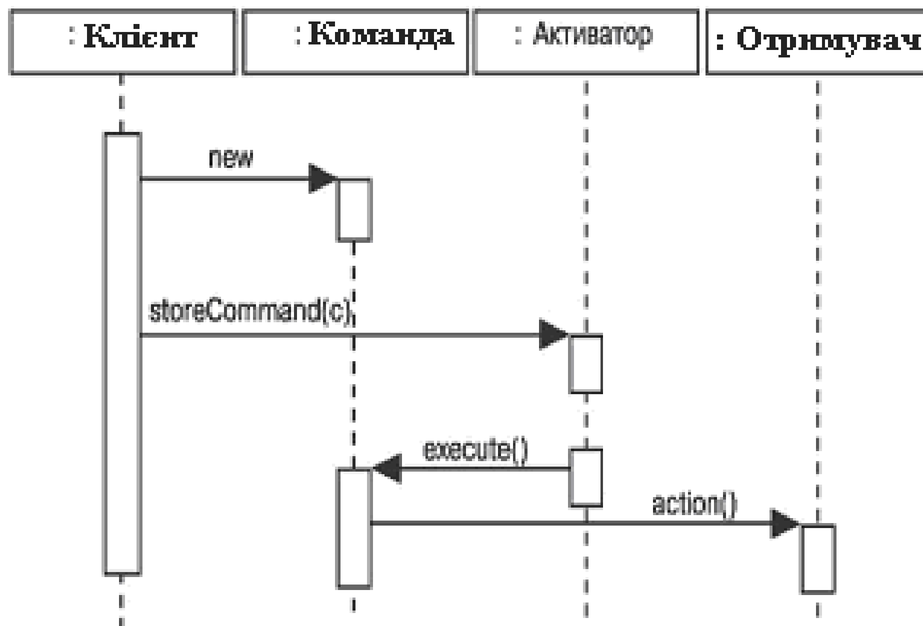


Рис. 5.2.6. Моделювання поведінкового аспекту зразка проектування

Зразки також підходять для моделювання типових архітектурних рішень. При моделюванні каркаса розробляється інфраструктура всієї архітектури, яка потім планується повторно використовуватися і адаптуватися до якогось контексту.

Каркас зображується у вигляді пакета зі стереотипом, що представляє ряд елементів, включаючи класи, інтерфейси, ВВ, компоненти, вузли, кооперації та навіть інші каркаси. Фактично в каркас вміщуються усі абстракції, що працюють спільно, формують розширюваний шаблон для додатків у певній області. Деякі із цих елементів будуть відкриті й стануть ресурсами, доступними для використання клієнтами. Це ті частини каркаса, які можна під'єднати до абстракцій свого контексту. Деякі з таких відкритих елементів стануть зразками проектування і становитимуть ресурси, з якими зв'язуються клієнти. Саме ці частини каркаса потрібно наповнювати, зв'язуючи їх зі зразком проектування. Окремі елементи каркаса можуть бути інкапсульованими (закритими або захищеними) і невидимими зовні. [13]

Зразок, по суті, є описом архітектури, хоча й неповним і, можливо, параметризованим. Все, що застосовується до моделювання добре структурованої архітектури, повною мірою є застосовне до добре структурованих каркасів, що не проектуються у відриві від системи. В основі каркасів лежать існуючі архітектури, що довели свою працездатність. Потім каркаси розбудовуються, щоб знайти елементи керування і стикування, необхідні й достатні для забезпечення можливості їх адаптації до нових областей.

Рис. 5.2.7. Моделювання архітектурного зразка

Щоб змоделювати архітектурний зразок, необхідно: закласти в основу каркаса перевірену існуючу архітектуру; змоделювати каркас як пакет зі стереотипом, що містить усі елементи (особливо зразки проектування), необхідні й достатні для описування різних виглядів каркаса; розкрити елементи, що включаються, інтерфейси і параметри, необхідні для адаптації каркаса у формі зразків проектування і кооперацій.

Це робиться з метою, щоб користувач зразка розумів, які класи повинні бути розширені, які операції реалізовані і які сигнали опрацьовані.

На рис. 5.2.7 показана специфікація архітектурного зразка Blackboard (Класна дошка) [Buschmann et al. Pattern-Oriented Software Architecture. – New York, NY: Wiley, 1996]. Як випливає з його опису, цей зразок застосуємо до завдань перетворення даних у високорівневі структури, що не мають простого детермінованого розв'язку. В основі архітектури лежить зразок Blackboard, що визначає порядок спільної роботи класів Knowledgeset (Джерело знань), Blackboard і Controller (Контролер). Цей каркас включає також зразок проектування Reasoning engine (Процесор логічного введення), що визначає загальний механізм роботи класу Knowleclgesource. І, нарешті, якбачимо з рис. 5.2.7, каркас розкриває один ВВ - Apply new knowledge sources (Застосувати нові джерела знань), що пояснює клієнтові, як можна цей каркас адаптувати.

Зразки (механізми і каркаси) використовуються на багатьох рівнях абстракції – від окремих класів до системи в цілому. Добре структурований зразок: вирішує типову проблему типовим чином, включає структурну і поведінкову складові; розкриває елементи керування і стикування, за допомогою яких його можна налаштувати на різні контексти; охоплює різні індивідуальні абстракції в системі. Зображуючи зразок у UML, необхідно розкривати ті його елементи, які слід адаптувати для застосування в конкретному контексті, приводити ВВ зразка та способи його адаптації.

5.3. Моделювання кооперації

Основні питання:
- *Кооперації, реалізації й взаємодії*
- *Моделювання реалізації варіанта використання*
- *Моделювання реалізації операції*
- *Моделювання механізму*
- *Матеріалізація взаємодій*

У контексті системної архітектури кооперація дозволяє виділити концептуальну частину, яка підкреслює і статичні, й динамічні аспекти. *Можна уявити собі один з найкращих будинків світу (Нотр Дам де Парі, резиденцію Буковинських митрополитів чи палац Люксембург). Краса цих будівель не піддається опису. Їхня архітектура наповнена гармонією, тонкими лініями, і вишуканістю, хоча не всі закладені в них ідеї лежать на поверхні. Якщо подивитися уважніше, можна помітити подробиці, прекрасні самі по собі, які в комплексі створюють враження гармонії і функціональності значно більші, ніж в кожній окремій деталі. Можна згадати найбезглуздіший і найогидніший будинок («хрущовку» чи забігайлівку по сусідству з вокзалом). Тут наявна відсутність жодного архітектурного стилю: у загальний модерністський дизайн ніяк не вписується тупий дах, безликі декорації та некомбіновані кольори, які просто знущаються над поглядом стороннього. Це є нікчемні «шедеври» посередніх напівремісників, що несуть у собі тільки ідею функціональності, без жодних претензій на стиль, хіба що тільки упадковий («угар непу» чи «епоха Керенского», як сказав класик).*

Що відрізняє ці два види архітектури? По-перше, у справжніх витворах є добірність дизайну, який відсутній у забігайлівках. У конструкції Нотр Дам присутні складні, симетричні й збалансовані геометричні елементи. Взагалі, в архітектурі не так вже є багато стилів, які вдало поєднуються між собою, і тільки справжньому архітектору в душі під силу таке поєднання. По-друге, будинки спроектовані зі смаком, мають загальну структуру, якій підвладні всі окремі елементи конструкції (окремі стіни є опорою купола собору, а частина з них поряд з іншими елементами використовується для відведення води тощо…).

Повертаючись до ПЗ, у контексті з проведеною будівельною аналогію, можна зробити висновок, що якісна програмна система не тільки виконує покладені на неї функції, але й демонструє гармонію, тобто **врівноваженість проекту**, завдяки чому легко піддається модифікації. Ця гармонійність і збалансованість найчастіше пояснюється тим, що добре структуровані об'єктно-орієнтовані системи містять множину повторюваних структурних **елементів-зразків** (patterns). В якісній об'єктно-орієнтованій системі можна виявити елементи, що взаємодіють між собою для реалізації деякої спільної поведінки. Остання є значно більшою, ніж поведінка суми всіх складових. Багато елементів добре структурованих систем у різних комбінаціях беруть участь у функціонуванні різних механізмів. В UML механізми моделюються за допомогою кооперацій. [1,5,13]

Кооперація (collaboration) – це співтовариство класів, інтерфейсів та інших елементів, які працюють спільно для забезпечення певної поведінки, більш значущої, ніж поведінка суми всіх тих же складових. [2] Кооперація іменує сукупність взаємодіючих будівельних блоків системи, включаючи як структурні, так і поведінкові елементи. Наприклад, можна розглянути розподілену систему керування інформацією, база даних якої розміщається на кількох вузлах. З точки зору користувача відновлення інформації виглядає як атомарна операція. Якщо ж глянути на неї зсередини, усе виявиться не так просто, оскільки у відновленні даних бере участь кілька машин. Для створення ілюзії простоти необхідно ввести механізм транзакцій, за допомогою якого клієнт може привласнити ім'я якоїсь операції, яка представляється єдиною і неподільною, незважаючи на те, що торкається кількох баз даних. У роботі такого механізму могли б брати участь кілька скооперованих класів, що спільно забезпечують транзакцію. Багато із цих класів будуть залучені і в інші механізми – наприклад, у механізм зберігання інформації. Такий набір класів (структурна складова), узятий разом із взаємодіями між ними (поведінкова складова), утворює механізм, який в UML представляється кооперацією.

Кооперація не тільки іменує системні механізми, але й служить у якості реалізації ВВ і операцій. Нотація кооперації (*еліпс з пунктирною межею*) в UML (рис 5.3.1) дозволяє візуалізувати структурні й поведінкові будівельні блоки системи, особливо в ситуаціях, коли вони перетинаються із класами, інтерфейсами та іншими елементами.

Кооперація також показує, як деякий елемент, наприклад, класифікатор (включаючи клас, інтерфейс, компонент, вузол або ВВ) або операція, реалізується набором класифікаторів і асоціацій, кожна з яких певним чином відіграє деяку роль. Кооперація допомагає специфікувати реалізацію ВВ і операцій, а також моделювати архітектурно значущі механізми системи.

Кожна кооперація повинна мати **ім'я**, що відрізняє її від інших кооперацій. Ім'я – це текстовий рядок; узяте окремо, воно називається **простим. Кваліфіковане ім'я** кооперації постачене префіксом – іменем пакета, у якому вона перебуває. Як правило, при зображенні кооперації вказується тільки її ім'я (рис. 5.3.1).

Рис. 5.3.1. Кооперації

Кооперації мають дві складові: структурну, яка описує класи, інтерфейси й інші спільно працюючі елементи, і поведінкову, яка описує динаміку взаємодії цих елементів.

Структурна складова кооперації – це внутрішня (складова) структура, яка може включати будь-яку комбінацію класифікаторів, таких, як класи, інтерфейси, компоненти, вузли. Всередині кооперації ці класифікатори можуть бути організовані з використанням усіх звичайних зв'язків UML, включаючи асоціації, узагальнення й залежності. Фактично структурні аспекти кооперацій можуть використовувати повний діапазон засобів структурного моделювання UML.

Відмінність кооперації і класів. На відміну від структурованих класів, *кооперація не володіє своїми структурними елементами*. Замість цього вона просто **посилається на класи**, інтерфейси, компоненти, вузли та інші структурні елементи, оголошені в іншому місці, або використовує їх. Тому *кооперація іменує концептуальний, а не фізичний фрагмент системної архітектури*. Кооперація може поширюватися на багато рівнів системи. Більше того, один елемент системи може брати участь у кількох коопераціях (а деякі елементи не будуть частиною ні однієї з них).

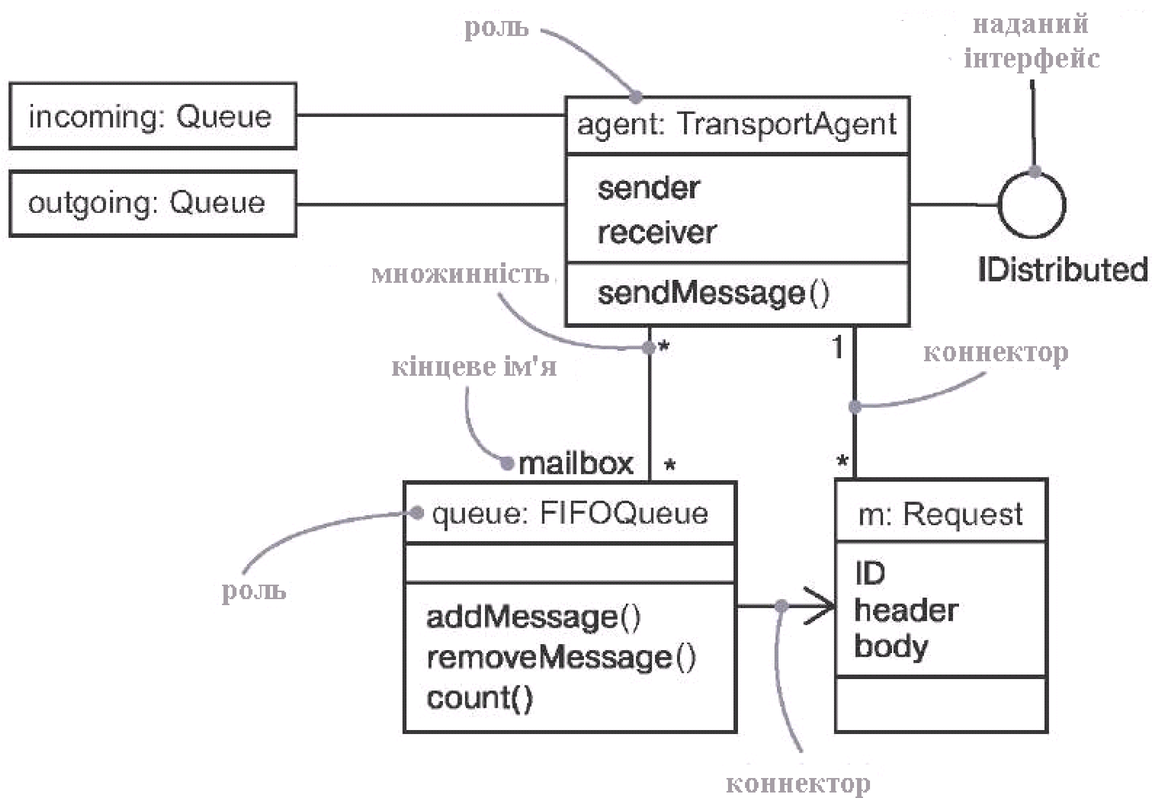


Рис. 5.3.2. Структурні аспекти моделювання

Наприклад, система роздрібної торгівлі через Internet описана більше десяти ВВ. Серед них є такі, як Purchase Items (Купівля товару), Return Items (Повернення товару), Query Order (Перегляд замовлення) і т.п.. Кожен з них може бути реалізований окремою кооперацією. До того ж, ці кооперації будуть розділяти деякі загальні структурні елементи, такі, як класи Customer (Клієнт) і Order (Замовлення), але організовані по-різному. На більш глибших рівнях системи також виявляться кооперації, що представляють архітектурно значущі механізми. Може бути присутнім кооперація Internode messaging (Міжвузлові повідомлення), яка описує деталі захищеного передавання повідомлень між вузлами.

Якщо є кооперація, що іменує концептуальний фрагмент системи, то можна заглянути і в її середину, щоб розглянути приховані усередині структурні деталі. На рис

5.3.2 показано, який набір класів, зображений на діаграмі класів, виявиться при розкритті кооперації Internode messaging.

Якщо структурна складова кооперації зображується як *діаграма складової структури*, то поведінкова в більшості випадків подана як *діаграма взаємодії*. Ця діаграма описує взаємодію, що відповідає поведінці, суть якої – обмін повідомленнями між об'єктами в деякому контексті для досягнення певної мети. Контекст взаємодії встановлює сама кооперація, що визначає класи, інтерфейси, компоненти, вузли та інші структурні елементи, екземпляри яких можуть брати участь у взаємодії.

Поведінкову складову кооперації можна описати однією або кількома діаграмами взаємодії. Якщо необхідно *підкреслити часовий порядок повідомлень*, потрібно використовувати *діаграму послідовності*; якщо ж основний акцент потрібно зробити на *структурних зв'язках* між об'єктами, що виникають у ході спільної діяльності, слід застосовувати *діаграму комунікації*. В обох випадках підходить будь-який вид діаграм, оскільки в більшості випадків вони семантично еквівалентні.

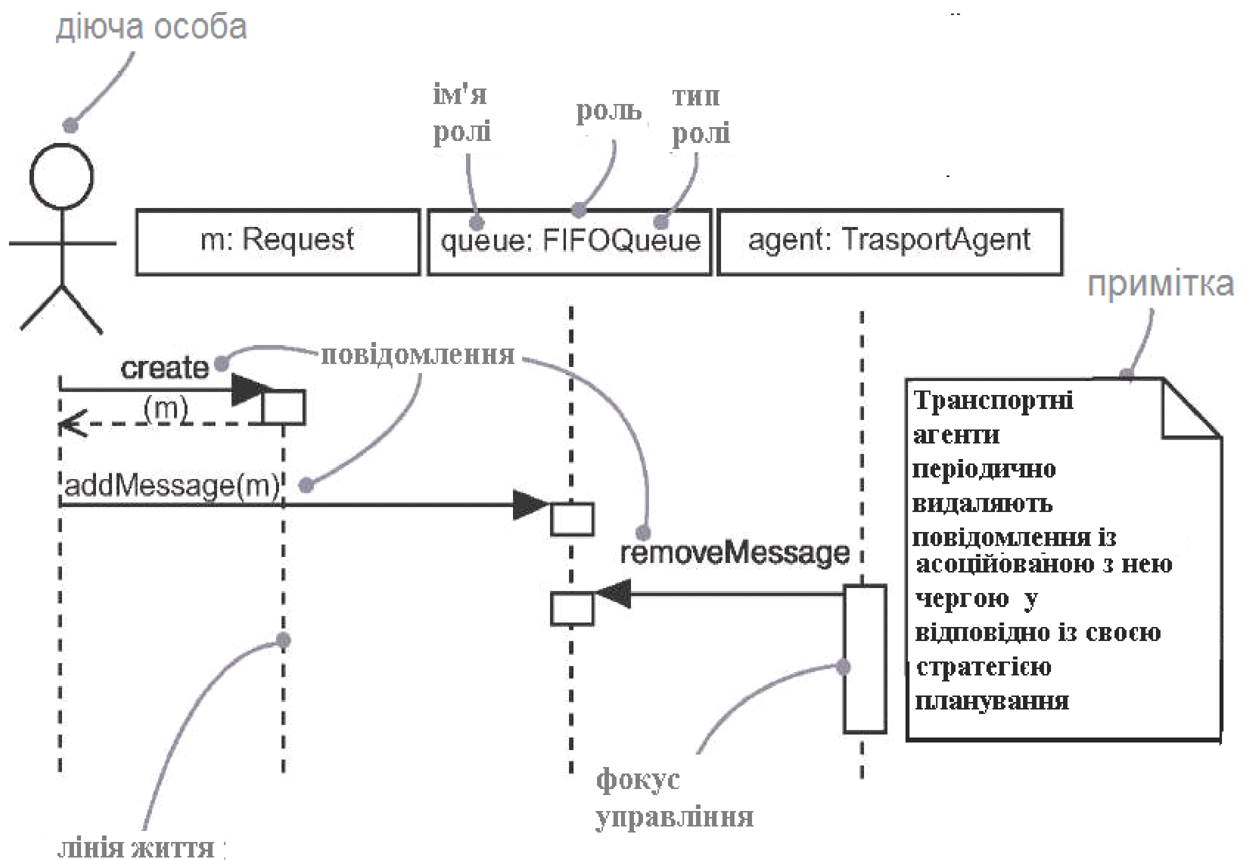


Рис. 5.3.3. Аспекти поведінки кооперацій

Це означає, що, моделюючи взаємодії співтовариства класів усередині деякої кооперації, кооперацію Internode messaging можна розкрити і ознайомитися з подробицями поведінки з допомогою діаграми взаємодії (рис. 5.3.3).

У коопераціях системи перебуває серце її архітектури, тому механізми, що лежать в основі системи, представляють істотні проектні рішення. Усі добре структуровані об'єктно-орієнтовані системи складаються з регулярної (*побудованої відповідно до чітких правил*) множини кооперацій відносно невеликого розміру, тому дуже важливо

навчитися їх організовувати. Існує два види зв'язків, що мають відношення до кооперацій, які варто брати до уваги.

По-перше, це **зв'язок між кооперацією і тим, що вона реалізує**. Кооперація може реалізовувати або класифікатор, або операцію. Це означає, що кооперація описує структурну або поведінкову реалізацію відповідного класифікатора чи операції. Наприклад, ВВ, що іменує набір сценаріїв, які виконує система, може бути реалізований у вигляді кооперації. Цей ВВ разом із його діючими особами та навколишніми ВВ представляє контекст кооперації. Аналогічно кооперацією може бути реалізована й операція, що *іменує реалізацію деякої системної послуги*. У такому випадку контекст формує дана операція разом зі своїми параметрами і можливим значенням, що повертається. Такий зв'язок між ВВ або операцією і реалізованою кооперацією моделюється за допомогою зв'язку реалізації. [2,13]

По-друге, існують **зв'язки між самими коопераціями**. Одні з них можуть уточнювати описи інших; це може бути змодельоване у вигляді **зв'язку уточнення**. Подібні зв'язки між коопераціями відображають, як правило, зв'язки уточнення між ВВ, які вони представляють. [13]

Обидва види зв'язків проілюстровано на рис 5.3.4.

Рис. 5.3.4. Організація кооперацій

Об'єкти представляють окремі сутності в статичному положенні або в динаміці. Разом з тим часто прагнуть показати деякі загальні частини в певному контексті. Частину в межах контексту називають **роллю** (role). [2,10] Можливо, найважливіше призначення ролей полягає в моделюванні динамічних взаємодій. У таких випадках не доводиться моделювати конкретні екземпляри, що існують у реальному світі. Замість цього моделюються ролі в деякому зразку, що повторно використовується, у межах якого вони, по суті, *заміщають об'єкти*, які з'являються в індивідуальних екземплярах

зразка. Наприклад, якщо потрібно змоделювати варіанти реакції вікон в операційній системі на рухи і клацання миші, потрібно намалювати діаграму взаємодій, що містить ролі, типи яких включають вікна, події й обробники.

Щоб змоделювати ролі, необхідно: визначити контекст, у якому взаємодіють об'єкти; ідентифікувати необхідні й достатні ролі, за допомогою яких можна ВСКД модельований контекст; зобразити їх як ролі в структурованому контексті, можливості привласнити кожній ролі ім’я. Якщо для якої-небудь із них не можна підібрати осмислене ім’я, зобразити її як безіменну. Розкрити властивості кожної ролі, які необхідні й достатні для моделювання контексту, та представити ролі і їх зв’язки на діаграмі взаємодії або діаграмі класів. [13]

На рис. 5.3.5 зображена діаграма взаємодії, що описує частковий сценарій здійснення телефонного виклику в контексті комутатора. Тут у наявності чотири ролі: а (Callingagent – Викликаючий абонент), з (Connection – Під’єднання), а також t1 і t2 (екземпляри Terminal – Термінал). Усі вони за своєю суттю є концептуальними «заступниками» конкретних об’єктів реального світу.

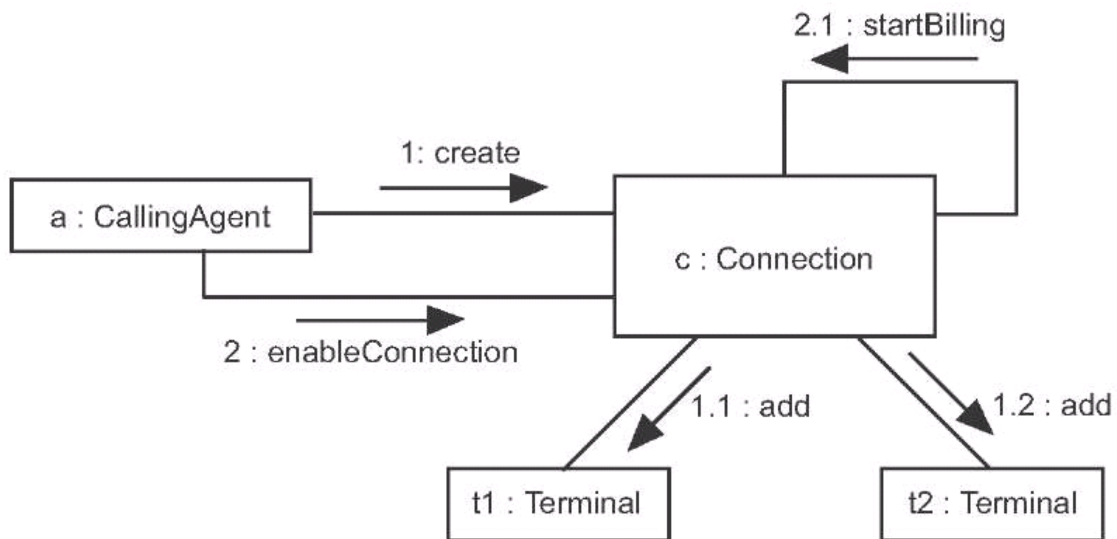


Рис. 5.3.5. Моделювання ролей

Одне із призначень кооперацій полягає в моделюванні реалізації ВВ. Як правило, аналіз системи диктується тими ВВ, які ідентифікуються. Переходячи ж до етапу реалізації, потрібно матеріалізувати їх у вигляді конкретних структур і поведінок. У загальному випадку кожен ВВ повинен бути реалізований однією або кількома коопераціями. Якщо розглядати систему в цілому, то класифікатори, що беруть участь у кооперації, яка пов'язана з деяким ВВ, будуть брати участь і в інших коопераціях. Таким чином, структурний зміст кооперацій має тенденцію перекриватися.

Щоб змоделювати реалізацію ВВ, необхідно: ідентифікувати структурні елементи, необхідні й достатні для вираження семантики ВВ; організувати ці структурні елементи в діаграми класів; розглянути окремі сценарії, що представляють даний варіант використання (кожен сценарій описує конкретний шлях виконання ВВ); відобразити динаміку цих сценаріїв на діаграмах взаємодії, використовувати діаграми послідовності, якщо потрібно підкреслити часовий порядок повідомлень, або діаграми комунікації, якщо важливішими є структурні зв'язки між об’єктами, що беруть участь у

кооперації; організувати ці структурні й поведінкові елементи як кооперацію, яку можна з'єднати з варіантом використання за допомогою зв'язку реалізації.

На рис. 5.3.6 зображені ВВ, що відносяться до системи контролю кредитних карток, включаючи основні: Place order (Розмістити замовлення) і Generate bill (Видати рахунок), а також підлеглі: Detect card fraud (Виявити шахрайство) і Validate transaction (Перевірити транзакцію). Хоча в більшості випадків не виникає необхідності в явному моделюванні цього зв'язку – таке завдання можна покласти на інструментальні засоби, – на даному рисунку показана явна модель реалізації Place order за допомогою кооперації Order management (Керування замовленнями). У свою чергу, ця кооперація може бути розкладена на структурний і поведінковий аспекти, що в підсумку дає діаграми класів і діаграми взаємодії. Таким чином, через зв'язок реалізації ВВ зв'язується з його сценаріями.

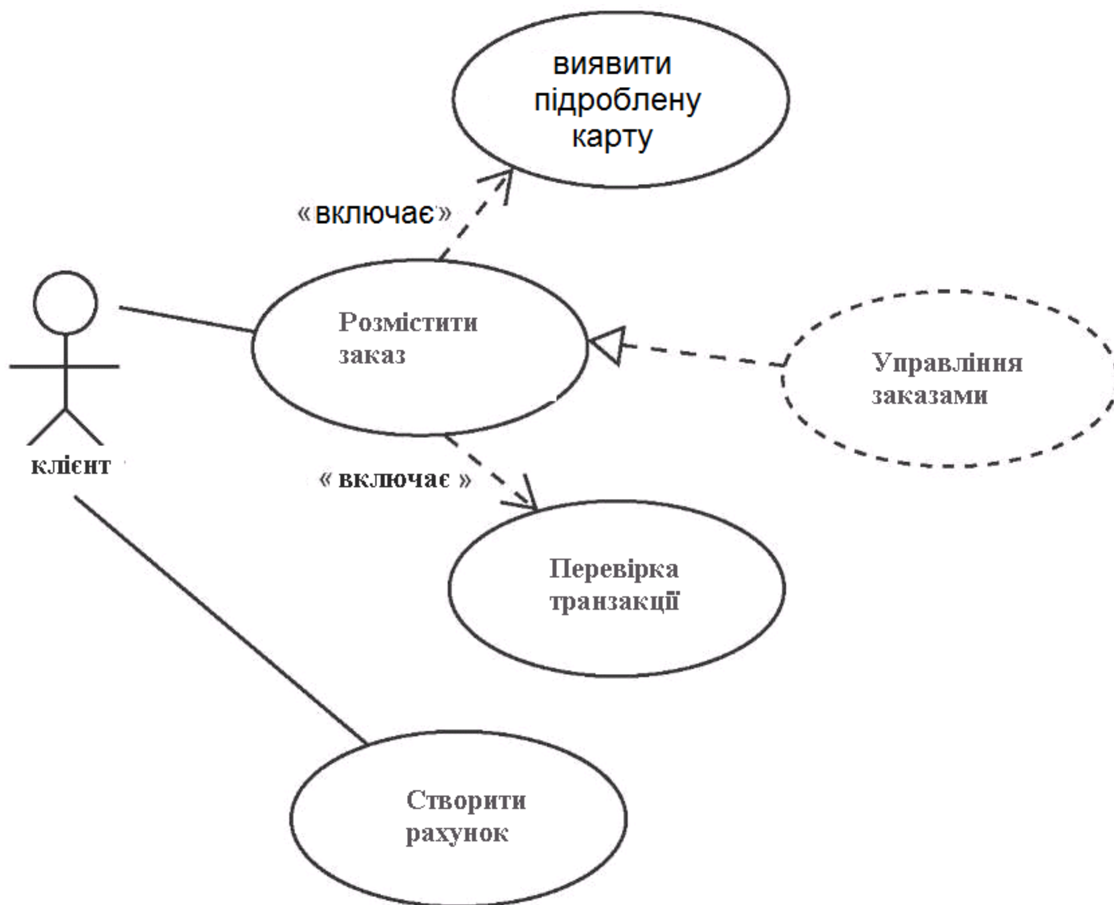


Рис. 5.3.6. Моделювання реалізації варіанту використання

У більшості випадків немає необхідності явно моделювати зв'язок між ВВ та реалізуючою його кооперацією. Можна залишити це з'єднання на задньому плані моделі, а потім дозволити інструментальним засобам використовувати його, щоб спростити навігацію між ВВ і його реалізацією.

Ще одне призначення кооперацій – моделювання реалізації операцій. Часто її можна специфікувати безпосередньо в коді, однак для тих операцій, які вимагають спільної роботи кількох об'єктів, перед написанням коду краще змоделювати реалізацію за допомогою кооперації. [13]

Контекст реалізації операції становлять параметри, що повертають значення й об'єкти, локальні відносно неї. Таким чином, ці елементи видимі для структурного аспекту кооперації, яка реалізує дану операцію (подібно тому якактанти видимі для структурного аспекту кооперації, що реалізує ВВ). Зв'язки між цими частинами можна змоделювати за допомогою діаграм складеної структури, що описують структурну частину кооперації.

Щоб змоделювати реалізацію операції, необхідно: ідентифікувати параметри, що повертають значення й інші об'єкти, видимі операції (*вони виступлять як ролі кооперації*); якщо операція досить проста, представити її реалізацію безпосередньо в коді, який можна вмістити на задній план моделі або явно візуалізувати у примітці; якщо операція складна алгоритмічно, змоделювати її реалізацію за допомогою діаграми діяльності; якщо операція складна або вимагає тривалого й ретельного проектування, представити її реалізацію у вигляді кооперації. Надалі можна розгорнути структурну й поведінкову складові кооперації, використовуючи відповідно діаграми класів і діаграми взаємодії. [4]

На рис 5.3.7 показано активний клас Renderfrarne (Побудова фрейма) і розкрито три його операції. Функція progress (Обробка) досить проста й може бути реалізована відразу в коді, наведеному в примітці. А ось операція render (візуалізувати) набагато складніша, тому її реалізація покладена на кооперацію Ray trace (Трасування променів). Кооперацію можна вивчити зсередини і побачити її структурні й поведінкові аспекти.

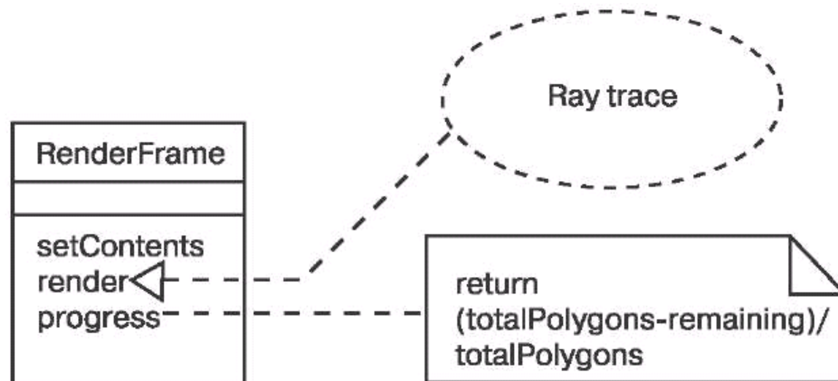


Рис. 5.3.7. Моделювання реалізації операції

У добре структурованій об'єктно-орієнтованій системі завжди присутній цілий спектр зразків. На одному кінці цього спектра можна виявити ідіоми, що представляють стійкі конструкції мови реалізації, а на іншому – архітектурні зразки і каркаси, що утворюють систему в цілому і задають певний стиль. У середині ж спектра розташовуються механізми, що описують розповсюджені зразки проектування, за допомогою яких системи взаємодіють один з одним. Механізми в UML представляють за допомогою кооперацій.

Механізми – це автономні кооперації; їхнім контекстом є не якийсь один варіант використання або одна операція, а система в цілому. Будь-який елемент, видимий у деякій частині системи, – кандидат на участь у механізмі. [2,4]

Механізми представляють архітектурно значущі проектні розв'язки. Як правило, механізми пропонує системний архітектор, і з кожною новою версією вони змінюються. Застосування механізмів робить систему простішою (*оскільки в механізмі реалізовані*

типові взаємодії), зрозумілою (*тому що до розуміння системи можна підійти з боку її механізмів)* і гнучкою (*налаштовуючи кожен механізм, налаштовується система в цілому*).

Щоб змоделювати механізм, необхідно: ідентифікувати основні механізми, що утворюють архітектуру системи (їхній вибір диктується загальним архітектурним стилем, який ви вирішили покласти в основу своєї реалізації, поряд зі стилем, найбільш підходящим до предметної області); представити кожен механізм у вигляді кооперації; розкрити структурну й поведінкову складові кожної кооперації, в міру можливості намагаючись знаходити елементи, що спільно використовуються; затвердити механізми на ранній стадії ЖЦ розроблення (*вони мають стратегічне значення*), але згодом у кожній новій версії розбудовувати їх у міру прояснення деталей реалізації.

Кожна кооперація має представляти реалізацію ВВ або служити автономним механізмом рівня всієї системи. Добре структурована кооперація включає як структурний, так і поведінковий аспекти; є чіткою абстракцією деякої ідентифікованої взаємодії в системі; не часто буває повністю незалежною, але частіше перекривається зі структурними елементами інших кооперацій; є зрозумілою і простою. Зображувати кооперацію явно в UML слід тоді, коли це необхідно для розуміння її зв'язків з іншими коопераціями, класифікаторами, операціями або системою в цілому. В інших випадках їх слід залишати на задньому плані моделі; організовувати кооперації відповідно до представлених їх класифікаторами або операціями або вміщувати в пакети, асоційовані з системою в цілому.

5.4. Моделювання пакетів як способів організації елементів моделі

Основні питання:

- *Пакети, видимість, імпорт і експорт*
- *Моделювання ґруп елементів*
- *Моделювання архітектурних виглядів*
- *Масштабування великих систем*

ВСКД великих систем припускають роботу з великою множиною класів, інтерфейсів, вузлів, компонентів, діаграм та інших елементів. Масштабуючи такі системи, стикаються з необхідністю організовувати ці сутності у великі блоки. В UML для організації елементів моделі в ґрупи застосовуються пакети.

Пакет – це спосіб організації елементів моделі в блоки, якими можна розпоряджатися як єдиним цілим. [2] Можна управляти видимістю елементів пакета так, що деякі будуть видні користувачеві, а інші – приховані. Крім того, за допомогою пакетів зображуються різні вигляди архітектури системи. Добре структурований пакет поєднує семантично близькі елементи, що мають тенденцію змінюватися спільно. Такі пакети характеризуються слабкою зв'язаністю і високою погодженістю, причому доступ до вмісту пакета строго контролюється. [5,6,13]

Конструкція собачої будки не є чимось складним: чотири стіни, лаз для собаки в одній з них і дах. Для виготовлення будки знадобиться всього кілька дощок. Житлові будинки влаштовані складніше. Стіни, стеля, підлога з'єднуються один з одним, утворюючи більшу сутність, названу кімнатою. Кімнати організовані в ще більші блоки – житлова зона, зона дозвілля тощо. Такі блоки є абстракцію, що поєднує під загальною назвою ряд кімнат, функціонально зв'язаних одна з одною для передбачуваного використання домашнього простору. Хмарочоси влаштовані ще складніше. Там крім найпростіших конструкцій (стін, підлог, стель) є ще більші утворення – місця, відкриті для загального доступу, здані в оренду квартири і офіси. Ці блоки ґрупуються в ще більші, такі, як орендована площа,

інфраструктура обслуговування будинку тощо. Вони не мають нічого спільного із самим будинком, а є лише артефактами, які застосовували при плануванні хмарочоса.

Складна система складається з кількох подібних шарів. По-справжньому зрозуміти її можна, об'єднавши абстракції, які до неї входять, у великі ґрупи. Більшість блоків середнього розміру (кімнати) є абстракціями, схожими на класи, в них буває багато екземплярів. Великі блоки часто є абстракціями («частина будинку для продажу»), в яких не буває екземплярів. Вони не реалізуються на фізичному рівні; єдина їх ціль – полегшити розуміння системи. *Такі блоки не будуть представлені в розгорнутій системі – вони існують лише на рівні моделі.*

Організуючі сутності. В UML організуючі модельні блоки називають пакетами. **Пакет** (package) є універсальним механізмом організації елементів у ґрупи, що спрощують розуміння моделі. [2,13] Крім того, пакети дозволяють контролювати доступ до свого змісту, що полегшує роботу зі з’єднаннями в архітектурі системи.

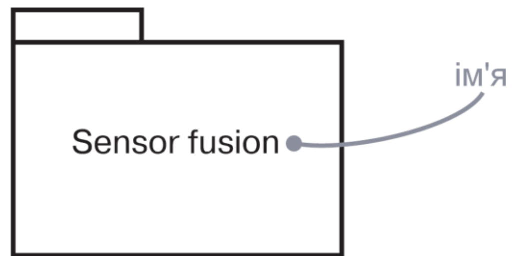


Рис. 5.4.1. Пакети

Графічно пакети в UML представлені нотацією у вигляді папки із закладкою, що дозволяє візуалізувати ґрупи елементів, з якими можна працювати як з єдиним цілим, контролюючи при цьому видимість і можливість доступу до окремих елементів (рис. 5.4.1). Ім’я пакета зазначене на папці, якщо вміст останньої не показано, а якщо ні, – то на закладці.

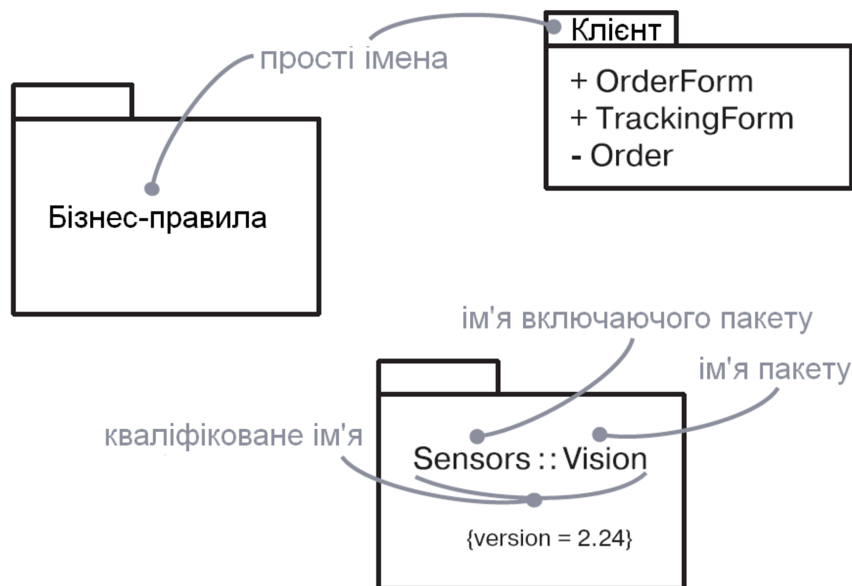


Рис. 5.4.2. Прості й кваліфіковані імена пакетів

Ім’я пакета. У кожного пакета повинно бути ім’я, що відрізняє його від інших пакетів. Просте ім’я включає текстовий рядок. Кваліфіковане ім’я випереджається іменем пакета, що включає даний, якщо таке вкладення має місце. Подвійна двокрапка (::) використовується в якості роздільника імен пакетів. Звичайно, зображуючи пакет, вказують тільки його ім’я (рис. 5.4.2), але, як і у випадку з класами, можна доповнювати

пакети присвоєними значеннями або додатковими розділами, щоб прояснити деталі. Ім'я пакета повинно бути унікальним всередині його пакета.

Елементи, що належать пакету. Пакет може володіти іншими елементами, у тому числі класами, інтерфейсами, компонентами, вузлами, коопераціями, ВВ, діаґрамами та іншими пакетами. **Володіння** (ownership) – це зв'язок композиції, який означає, що елемент оголошений всередині пакета. Якщо пакет віддаляється, то знищується і належний йому елемент. Кожен елемент може належати тільки одному пакету.

Простір імен пакета. Пакет визначає свій простір імен, тобто елементи одного виду повинні мати імена, унікальні в контексті пакета, що їх включає. В одному пакеті не може бути двох класів Queve (Черга), але може бути один клас Queve в пакеті P1, а інший – у пакеті P2. P1::Queve й P2::Queve мають різні кваліфіковані імена й тому є різними класами.

Елементам різного виду можна присвоювати однакові імена в межах пакета. Припустима наявність класу Timer (Таймер) і компонента Timer в одному й тому пакеті. Однак, щоб не було плутанини, краще призначати всім елементам пакета унікальні імена.

Один пакет може містити інші, отже, допускається ієрархічна декомпозиція моделі. Може існувати клас Camera (Камера), що належить пакету Vision (Оптичний пристрій), який, у свою чергу, утримується в пакеті Sensors (Сенсори). Повне ім'я цього класу – Sensors::Vision::Camera. Краще уникати надто глибокої вкладеності пакетів: два-три рівні – межа керованості. За необхідності для організації пакетів варто використовувати імпорт, а не вкладення.

Описана семантика володіння робить пакети важливим механізмом масштабування системи. Без них довелося б створювати більші плоскі моделі, усі елементи якої повинні мати унікальні імена. Такі конструкції були б зовсім некеровані, особливо якщо вхідні в модель класи та інші елементи створені різними колективами. Пакети дозволяють контролювати елементи, що утворюють систему, у процесі її еволюції.

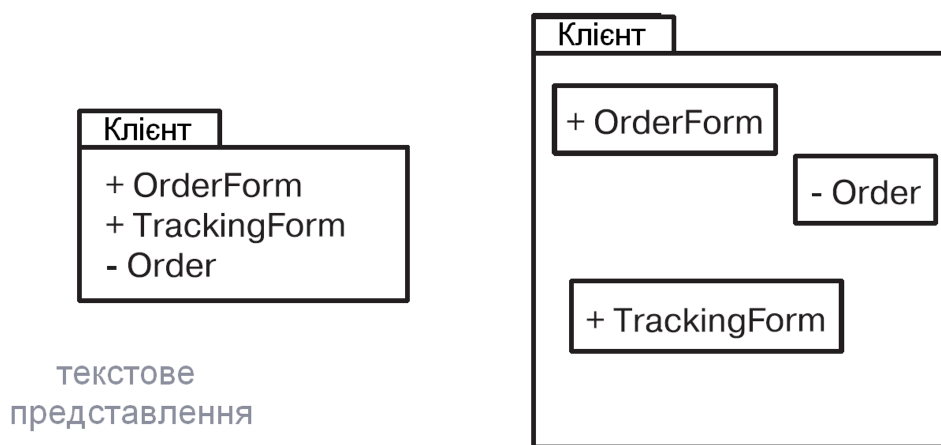


Рис. 5.4.3. Елементи, що належать пакету

Зміст пакета можна подати графічно або в текстовому вигляді (рис. 5.4.3). Якщо вказувати належні пакету елементи, то ім'я пакета пишеться усередині закладки. Втім,

вміст пакета не показують таким чином, а застосовують наявні інструментальні засоби, що дозволяють розкрити пакеті переглянути його вміст.

Видимість елементів пакета (відкриті, захищені і закриті). Видимість належних пакету елементів можна контролювати так само, як видимість атрибутів і операцій класу. За замовчуванням такі елементи є *відкритими*, тобто видимі для всіх елементів, що втримуються в будь-якому пакеті, що імпортує даний. *Захищені* елементи видимі тільки для нащадків, а *закриті* взагалі невидимі поза своїм пакетом. На рис. 5.4.3 Order Form (Бланк замовлення) – це відкрита частина пакета Client (Клієнт), а Order (Замовлення) – закрита. Будь-який пакет, що імпортує дані, може «бачити» об'єкт Order Form, але «не бачить» Order. При цьому повне кваліфіковане ім'я для Orderform буде Client::OrderForm.

Інтерфейс пакета. Видимість елемента пакета позначається відповідним символом перед його іменем. Для відкритих елементів використовується знак + (*плюс*), як у випадку з Orderform (рис.5.4.3). Всі *відкриті частини пакета* в сукупності *становлять його* інтерфейс. Так само як і відносно класів, для імен захищених елементів використовують символ # (*дієз*), а для закритих додають символ – (*мінус*). *Захищені елементи* будуть *видимі тільки для пакетів, що успадковують даний*, а *закриті* взагалі *невидимі поза пакетом*, у якому оголошені.

Пакетна видимість показує, що клас видимий тільки для інших класів, оголошених у тому ж самому пакеті, але схований для класів, оголошених в інших пакетах. Пакетна видимість зображується за допомогою символу ~ (*тильда*) перед іменем класу.

Імпорт і експорт. Нехай в моделі є два класи одного рівня, розташовані поруч: A і B. Клас A «бачить» B, і навпаки. Тобто кожен з них може залежати від іншого. Обоє вони утворюють тривіальну систему і для них не треба створювати ніяких пакетів. Припустимо тепер, що є кілька сотень рівноправних класів. Розмір мережі зв'язків, яку можна «*виткати*» між ними, не піддається уяві. Більше того, настільки величезну ґрупу неорганізованих класів просто неможливо сприйняти в її цілісності. Це реальна проблема великих систем: простий необмежений доступ не дозволяє здійснити масштабування. У таких випадках для організації абстракцій і застосовуються пакети.

Зв'язок імпорту як залежність. Припустимо, що клас A розташований в одному пакеті, а клас B – в іншому, причому обидва пакети рівноправні. Нехай також A і B оголошені відкритими частинами у своїх пакетах. Ця ситуація докорінно відрізняється від двох попередніх. Хоча обидва класи оголошені відкритими, вільний доступ одного з них до іншого неможливий, тому що *границі пакетів непрозорі*. Однак якщо пакет, що містить клас A, *імпортує пакет,* що містить клас B, то A зможе «бачити» B, хоча B, як і раніше, не буде «бачити» A. *Імпорт дає елементам одного пакета однобічний доступ до елементів іншого*. В UML зв'язок імпорту моделюється як залежність, доповнена стереотипом import. Упаковуючи абстракції в семантично осмислені блоки і контролюючи доступ до них за допомогою імпорту, можна керувати складністю систем, які нараховують безліч абстракцій.

Відкриті елементи пакета називають **експортованими**. На рис.5.4.4 пакет GUI експортує два класи – Window (Вікно) і Form (Форма). Клас EventHandler (Обробник подій) є захищеною частиною пакета. Досить часто експортованими елементами пакета є інтерфейси.

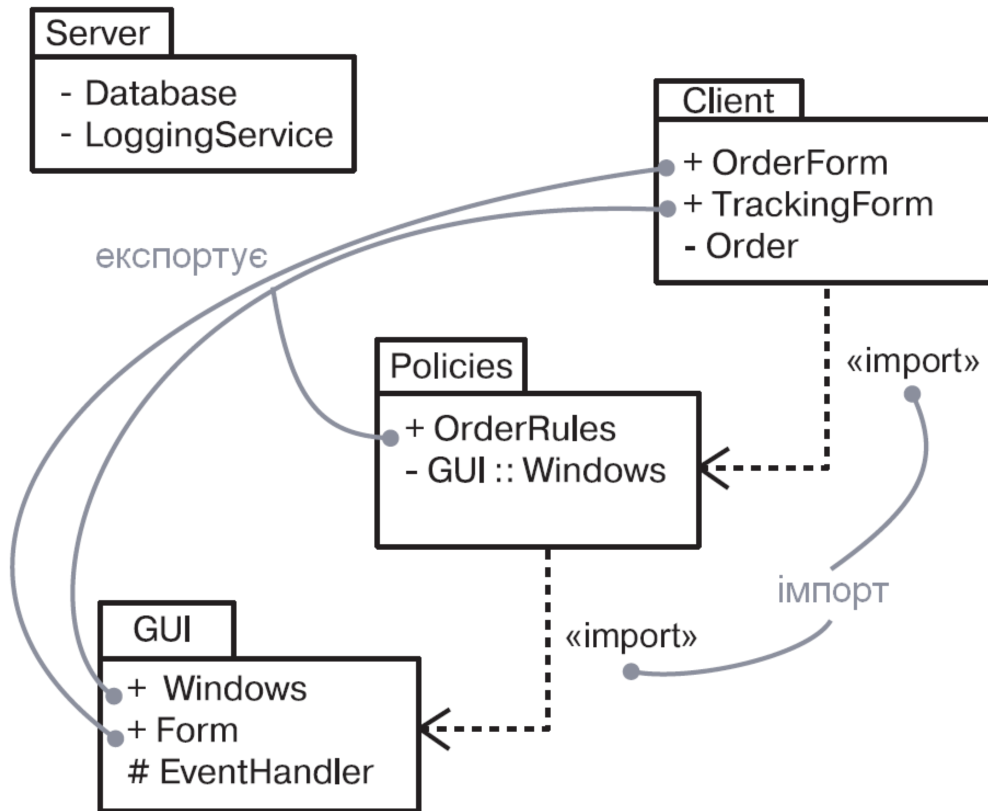


Рис. 5.4.4. Імпорт і експорт

Експортовані елементи будуть видимі для вмісту тих пакетів, для яких даний пакет є видимий. На рис. рис. 5.4.4 пакет Policies (Політики) явно імпортує пакет GUI. Таким чином, класи GUI::Window і GUI::Form будуть видимі для вмісту пакета Policies як класи із простими іменами Window і Form. Але клас GUI:: Event Handler буде схований, тому що є захищеним. З іншого боку, пакет Server не імпортує GUI, тому елементи Server мають доступ до відкритих елементів GUI, але для цього їм знадобляться кваліфіковані імена, наприклад GUI::Window. Аналогічно в GUI немає доступу до вмісту пакета Server навіть із використанням кваліфікованих імен, оскільки він закритий.

Залежності імпорту і доступу є транзитивними. У даному прикладі пакет Client імпортує Policies, а Policies – GUI, так що можна сказати, що Client транзитивно імпортує GUI. Якщо Policies має доступ у GUI, не імпортуючи його, то Client не додає елементи GUI до свого простору імен, але може звертатися до них, використовуючи кваліфіковані імена (наприклад, GUI::Window).

Моделювання ґруп елементів. Найчастіше пакети застосовують для організації елементів моделювання в іменовані ґрупи, з якими потім можна буде працювати як з єдиним цілим. Створюючи простий додаток, можна взагалі обійтися без пакетів, оскільки всі ваші абстракції прекрасно розмістяться в єдиному пакеті. У складніших системах скоріше виявиться, що багато класів, компоненти, вузли, інтерфейси й навіть діаграми природно розділяються на ґрупи, які моделюються у вигляді пакетів.

Відмінність між класами і пакетами. Між класами і пакетами є одна значна відмінність: класи є абстракції сутностей предметної області, а пакети – це механізми організації таких сутностей у моделі. **Пакети не видні в працюючій системі**, вони служать винятково механізмом організації розроблення. [1,2,10]

Найчастіше за допомогою пакетів елементи однакових базових типів організовують у ґрупи. Наприклад, класи і їх зв'язки у вигляді системи з точки зору проектування можна розбити на кілька пакетів і контролювати доступ до них за допомогою *залежностей імпорту*. Компоненти вигляду реалізації можна організувати в такий же спосіб.

Пакети також застосовуються для ґрупування елементів різних типів. Наприклад, якщо система створюється кількома колективами розробників, розташованими в різних місцях, то пакети можна використовувати для керування конфіґурацією, розміщаючи в них усі класи й діаграми, так щоб члени різних колективів могли незалежно діставати їх зі сховища й вміщувати назад. На практиці пакети часто застосовують для ґрупування елементів моделі й асоційованих з ними діаграм. Щоб змоделювати ґрупи елементів, необхідно: переглянути елементи моделі в деякому вигляді архітектури системи з метою пошуку ґруп елементів, близьких семантично або концептуально; вмістити кожну таку ґрупу в пакет; для кожного пакета визначити, які елементи повинні бути доступні ззовні, позначити їх як відкриті, а інші – як захищені або закриті (у сумнівних випадках приховати елемент); явно з'єднати пакети залежностями імпорту з тими пакетами, від яких вони залежать; за наявності сімейств пакетів з'єднати спеціалізовані пакети з більш загальними зв'язками узагальнення. [13]

На рис. 5.4.5 показано пакети, які організують у класичну трихрівневу архітектуру класи, що є частинами вигляду інформаційної системи з погляду проектування. Елементи пакета User Services (Користувацькі сервіси) надають візуальний інтерфейс для введення й виведення інформації. Елементи пакета Data Services (Сервіси даних) забезпечують доступ до даних і їх відновлення. Пакет Business Services (Бізнес сервіси) є сполучною ланкою між елементами перших двох пакетів; він охоплює всі класи й інші елементи, відповідальні за виконання бізнес-завдання, у тому числі бізнес-правила, які диктують стратеґію маніпулювання даними.

Усі абстракції простої системи можна вмістити в один пакет. Однак, організовуючи класи й інші елементи вигляду системи з точки зору проектування в три пакети, можна не тільки зробити модель зрозумілішою, а й контролювати доступ до її елементів, приховуючи одні та експортуючи інші. [1,13]

Моделювання архітектурних виглядів. Використання пакетів для ґрупування родинних елементів досить важливе – без нього не можна розробити складну модель. Даний підхід

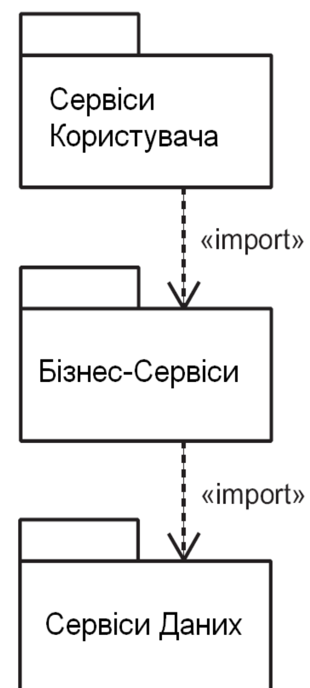


Рис. 5.4.5. Моделювання ґруп елементів

застосуємо до організації таких елементів, як класи, інтерфейси, компоненти, вузли і діаграми. Але при розгляді архітектурних зображень програмних систем виникає потреба у ще більших блоках. Архітектурні зображення теж можна моделювати за допомогою пакетів. [1]

Виглядом (view) називають проекцію організації і структури системи, в якій увага акцентується на одному з конкретних її аспектів. З цього визначення випливають два наслідки. По-перше, систему можна розкласти на майже ортогональні пакети, кожен з яких має справу з *набором архітектурно важливих рішень* (*можна створити вигляд з точки зору проектування, взаємодії, реалізації, розміщення і варіантів використання*). По-друге, цим пакетам будуть належати всі абстракції, що відносяться до даного вигляду. Тому всі компоненти моделі належать пакету, який моделює зображення реалізації. У той же час пакети можуть містити посилання на елементи інших пакетів. [1,2,5]

Для моделювання архітектурних виглядів необхідно: ідентифікувати набір архітектурних виглядів, важливих у контексті проблеми (на практиці цей набір включає вигляди проектування, взаємодії, реалізації, розміщення і ВВ); розмістити ті елементи й діаграми, необхідні та достатні для ВСКД семантики кожного вигляду у відповідному пакеті. За необхідності виконати подальше групування цих елементів у пакети.

Між елементами різних виглядів виникають залежності. Таким чином, кожен вигляд на верхньому рівні системи повинен бути відкритим для всіх інших виглядів цього рівня. На рис. 5.4.5 показана канонічна декомпозиція верхнього рівня, типова для більшості складних систем.

Пакети потрібні тільки для організації елементів моделі. Якщо є абстракції, безпосередньо матеріалізовані як обє’кти в системі, не слід користуватися пакетами, а застосовувати класи чи компоненти.

Добре структурований пакет: є внутрішньо погоджений і окреслює чітку границю навколо ґрупи родинних елементів; *слабко зв’язаний* і експортує в інші пакети тільки ті елементи, які вони дійсно повинні «бачити», а імпортує лише ті, які принципово важливі для роботи його власних елементів; має невелику глибину вкладеності (*людина не здатна сприймати надто глибоко вкладені структури*); володіючи збалансованим набором елементів, пакет у системі не повинен бути ні надто великим (*при потребі розщеплювати на дрібніші*), ні надто малим (*поєднувати елементи, якими можна маніпулювати як одним цілим*).

Зображуючи пакет UML, слід: застосовувати просту форму піктограми пакета,

Рис. 5.4.6. Моделювання архітектурних зображень

якщо не потрібно явно розкрити його вміст; розкриваючи вміст пакета, показувати тільки принципово необхідні елементи для розуміння його призначення в даному контексті; моделюючи за допомогою пакета сутності керування конфігурацією, слід розкривати значення позначок, пов’язаних з номерами версій.

Перелік використаних джерел

1. Буч, Г. Введение в UML от создателей языка; пер. с англ. [Текст] / Г. Буч, Дж. Рамбо, И. Якобсон – ДМК пресс, 2011.
2. Буч, Г. Язык UML. Руководство пользователя; пер. с англ. [Текст] / Г. Буч, Дж. Рамбо, И. Якобсон – ДМК пресс, 2007.
3. Буч, Г. Объектно-ориентированный анализ и проектирование с примерами приложений на С++; пер. с англ. [Текст] / Г. Буч – М.-СПб.: БИНОМ – Невский диалект, 2001. – 560 с.
4. Рамбо Дж. UML 2.0. Объектно-ориентированное моделирование и разработка. – 2-е издание; пер. с англ. [Текст] / Дж. Рамбо, М. Блаха – Питер, 2007.
5. Якобсон, А., Уніфікований процес розробки програмного забезпечення. [Текст] / А. Якобсон, Г. Буч, Дж. Рамбо – СПб.: Питер, 2002. – 496 с.
6. Фаулер, М. UML основы. Второе издание. Краткое руководство по унифицированному языку моделирования. [Текст] / М. Фаулер, К. Скотт – СПб.: Символ-плюс, 2002.– 192 с.
7. Лафоре, Р. Объектно-ориентированное программирование в С++.– 4-е издание. [Текст] / Р. Лафоре – СПб.: Питер, 2005.– 928 с.
8. Макконелл, С. Совершенный код. Мастер-класс; пер. з англ. [Текст] / С. Макконелл – М.: Русская Редакция; СПб.: Питер, 2007. – 896с.
9. Фаулер М. Предметно-ориентированные языки программирования; пер. с англ. [Текст] / М. Фаулер, Р. Парсонс – М.: Вильямс, 2011.
10. Ларман К. Применение UML 2.0 и шаблонов проектирования; пер. с англ. [Текст] / К Ларман – М.: Вильямс, 2009.
11. Арлоу Дж. UML 2 и Унифицированный процесс. Практический объектно-ориентированный анализ и проектирование; пер. с англ. [Текст] / Дж. Арлоу, А. Нейштадт – Символ-Плюс, 2007
12. Хасан Г. UML. Проектирование систем реального времени, распределенных и параллельных приложений, 2-е издание; пер. с англ. [Текст] / Г. Хасан – ДМК Пресс, 2011.
13. Иванов Д. Моделирование на UML. Учебно-методическое пособие; - 2010.
14. http://habrahabr.ru/
15. http://www.omg.org/
16. http://www.uml.org/
17. http://uml3.ru/

Редактор *Є.І. Гриценко*
Коректор *М.Д. Радик*
Комп'ютерне макетування *В.П .Наворинський*

Формат 60×90 Папір ксероксний.
Обл. вид. арк. 12.8
Наклад 35 прим. Зам. № 2279

Видавництво Тернопільського національного
технічного університету імені Івана Пулюя

вул. Руська, 56, м. Тернопіль, 46001
E-mail: vydavnytstvo@tu.edu.te.ua

© Тернопільський національний технічний університет імені Івана Пулюя
Навчально-методична література