ʃ ʻɖ ʻɦlsjtɕɦlsəts tsɦəʻlsʃ ʻ ɖ0ƶʕʃ υ ʕtɕ0ʾɖʃ

ʊjtɕɖtsʧʻɖɻɦɻʕʃʔ ɖ0ʯʻtsɖ0ɖɻɖʃʔ jʕtsɖtsʤʻɥɖʃʔ ƶɖ ʻəjtɕɦʃlsjls

ʌ 0ʕƶɖɻlsjls ʕtsʤʧ' ʁ lsjtɕɖʃ ʻɖ ʷtstɕʤ0ʯʻʔɖʃ lsjɔɖtsɖtsɢʔ

ʃ tsʜjɖ ʁ əƟɖɖʷ ʙʻʟɖjɦ-ʧtɕtsʯjɦʻə

lsƟ Ɵtɕ⊐ʻlsjʕlslƶtɕƟ ʧtɕtsɢtɕƟɖʤɖtsɢts ʟƟʙjʟʧjʯjɖɖʷ

ʃ ʧtstɕɖʃʔ ʕtsɖɦʧjʕls ɖjʕʯʻʔ

ʊjtɕɖtsʧʻɖɻ

2015

ЗМІСТ

ПЕРЕДМОВА

Метою навчального посібника є вивчення основ і освоєння термінологічного апарату об'єктно-орієнтованого моделювання, оволодіння навичками моделювання мовою *UML* (Unified Modeling Language) у середовищі об'єктно-орієнтованого засобу моделювання Rational Rose 2003.

На відміну від традиційного підходу, коли головну увагу приділяють інформації, з якою працює система, за об'єктно-орієнтованого підходу увагу приділяють як інформації, так і поведінці системи, що дає змогу створювати гнучкі системи, які допускають зміну поведінки і/або інформації, що міститься у них.

Порівняно зі структурним підходом, де головну увагу приділяють функціональній декомпозиції, в об'єктно-орієнтованому підході предметну область розбивають на деяку множину незалежних сутностей – об'єктів. Об'єктна декомпозиція, відображена у специфікаціях і кодах застосувань, є головною особливістю об'єктно-орієнтованого підходу.

Стандартною нотацією для моделювання великих *інформаційних систем* (ІС) на базі об'єктно-орієнтованої методології слугує уніфікована мова моделювання UML, що підтримується багатьма CASE-засобами. Однією з найрозповсюдженіших вважають Rational Rose.

Важливим аспектом успішного створення складної ІС є використання методології розробки проекту, у рамках якої вводяться етапи роботи, ставляться задачі аналітикам, проектувальникам, програмістам, тестувальникам, системним інтеграторам тощо у рамках методології *Rational Unified Process* (RUP).

Методологія RUP базується на побудові системи канонічних діаграм – одиничних описів фрагментів системи. Діаграми ілюструють різні аспекти системи. У кожної діаграми є своя мета і своя категорія користувачів.

З метою опису виконання послідовності операцій у середовищі об'єктно-орієнтованого засобу моделювання Rational Rose 2003 використовуватимемо систему позначень, запозичену у [7]:

Позначення	Виконання операції
➢Назва команди	*Вибір* у поточному меню команди з певною назвою. Встановити курсор миші на команду і натиснути на *ліву клавішу миші* (ЛКМ)
❑Назва кнопки	*Натискання* кнопки з зазначеною назвою в активному діалоговому вікні. Встановити курсор миші на кнопці та натиснути на ЛКМ)
⇩Назва списку	*Розкриття* списку. Встановити курсор миші на кнопку розкриття списку, натиснути на ЛКМ
⇗Елемент списку	*Вибір* елемента списку. Встановити курсор миші на зазначений елемент, натиснути на ЛКМ
🗀Назва закладки	*Вибір* закладки з зазначеною назвою. Встановити курсор миші на ярлику закладки та натиснути на ЛКМ
Назва поля:=*значення*	*Уведення значення* з клавіатури у текстове поле введення, список або лічильник. Значення лічильника можна змінювати і за допомогою кнопок-регуляторів
⊙Назва перемикача (радіокнопки)	*Вибір* перемикача з зазначеною назвою в активному діалоговому вікні. Встановити курсор миші на перемикач і натиснути на ЛКМ
☑Назва індикатора (прапорця, опції)	*Позначення* індикатора з зазначеною назвою в активному діалоговому вікні. Встановити курсор миші на індикаторі та натиснути на ЛКМ

1. ОСНОВИ ОБ’ЄКТНО-ОРІЄНТОВАНОГО МОДЕЛЮВАННЯ

План викладу матеріалу:

1. Класифікація програмних систем.
2. Життєвий цикл програмних систем.
3. Вступ у процес моделювання.
4. Класи й об’єкти.
5. Методологія об’єктно-орієнтованого моделювання.

Ключові терміни розділу

- *Програмні системи*
- *Типи програмних систем*
- *Процес розробки програми*
- *Життєвий цикл програми*
- *Технічне завдання*
- *Словник предметної області*
- *Артефакти*
- *Модель життєвого циклу*
- *Модель. Види моделей*
- *Задачі, які розв’язує модель*
- *Стилі програмування*
- *Методи моделювання*
- *Алгоритмічна модель*
- *Об’єктно-орієнтована модель (ООМ)*
- *Головні властивості ООМ*
- *Класи й об’єкти*
- *Головні властивості класів*
- *Атрибути та методи класів*
- *Об’єктно-орієнтований аналіз*
- *Об’єктно-орієнтоване проектування*
- *Нефункціональні вимоги*
- *Функціональні вимоги до програм*

1.1. Класифікація програмних систем

Програмна система (або *програмний продукт*) – організована сукупність програм і/або програмних модулів постійного застосування для розв’язування задач у різноманітних сферах людської діяльності.

Програмний модуль (або просто *модуль*) – частина програми, оформлена у вигляді, який допускає її незалежну компіляцію і використання. Програма – організована сукупність модулів, серед яких один з модулів є *головним*. Головний модуль організовує роботу інших модулів програми і забезпечує інтерфейс користувача.

Програмна система може складатися з однієї чи декількох програм (комплексу програм), які певним чином організовані та взаємодіють між собою. Програми комплексу можуть використо-

вувати певний набір модулів (бібліотеку модулів). Модулі та їхні набори у різних мовах програмування організовані по-різному.

Термін "програмна система" узагальнений, отож вводять інші подібні терміни, які враховують область застосування чи масштаб використання програмної системи.

Програмна система за *областю застосування* може бути:

- *прикладною програмою* (синоніми: застосування, додаток, аплікація – від *application*), призначеною для розв'язування певного класу однотипних задач і отримання конкретних результатів (наприклад, нарахування заробітної плати);
- *пакетом* (чи *системою*) *програм*, призначених для підтримання певної професійної діяльності людини (наприклад, математичні пакети, графічні пакети, видавничі системи, системи мультимедіа тощо);
- *системною програмою* (синонім: програмне забезпечення), призначеною для забезпечення роботи прикладних програм (наприклад, операційна система);
- *системою керування базами даних* (СКБД) – комплексом програм і мовних засобів, призначених для створення, ведення і використання *баз даних* (БД);
- *застосуванням реального часу* – застосуванням, в якому врахування реального астрономічного часу є критично важливим для виконання головних функцій (наприклад, диспетчерські системи на транспорті);
- *вертикальним застосуванням* – застосуванням, зробленим за конкретним індивідуальним замовленням;
- *горизонтальним застосуванням* – застосуванням[1], розрахованим на масовий попит, яке можна придбати у торговельній мережі (наприклад, комп'ютерні ігри);
- *офісним застосування* – застосуванням (вертикальним чи горизонтальним), призначеним для введення/виведення, зберігання та опрацювання документів у рамках організації чи підприємства, яке не зв'язане критично з часом;

[1] Операційні системи, пакети, настільні СКБД, деякі офісні застосування (наприклад, MS Office) можна зачислити до горизонтальних застосувань

- *застосування баз даних* – офісне застосування, головна функціональність якого полягає у забезпеченні зручного доступу до бази даних.

Зауваження. Вживання однини, наприклад, у термінах *прикладна програма*, *застосування* тощо є історичною традицією. Насправді більшість сучасних програмних систем є комплексами програм.

За *масштабом використання* програмної системи вирізняють:

- *настільні застосування* (для роботи одного користувача);
- *групові застосування* (для роботи однієї категорії користувачів локальної мережі);
- *корпоративні програмні системи* (для роботи різних категорій користувачів локальної чи глобальної мережі).

1.2. Життєвий цикл програмних систем

Програмні системи за час існування зазнають найрізноманітніших змін своєї форми, які залежать від стану процесу розробки та експлуатації застосування. Послідовність цих змін позначають терміном *життєвий цикл*. З цим терміном тісно переплітається поняття *процесу розробки* застосування.

У буденному використанні “*розробка*” означає створення “*чогось*”, розробку завершено. Після розробки розпочинається використання “чогось” (*експлуатація*). Для програмних застосувань чітко відокремити фази розробки та експлуатації не вдається. Отож тут доцільніше послуговуватись терміном “*розробка*, *що продовжується*”, який поєднує в собі як термін “розробка” у звичному розумінні, так і модифікацію програми у процесі експлуатації.

Отже, терміни “*життєвий цикл*” і “*процес розробки*” можна вважати двома різними сторонами одного поняття. При вживанні терміна “життєвий цикл” мають на увазі погляд з точки зору програми, а при вживанні терміна “процес розробки” – погляд з точки зору програміста.

Відповідно до стандарту ISO/IEC 12207 життєвий цикл програми налічує такі етапи (рис. 1.1):

- аналіз предметної області і формулювання системних вимог (постановка задачі);

- проектування структури програми;
- реалізація програми в кодах (власне програмування);
- впровадження програми і тестування;
- супровід програми під час експлуатації;
- відмовлення від використання програми.

На *етапі аналізу предметної області і формулювання вимог* здійснюється визначення функцій, що повинна виконувати програма, а також концептуалізація (визначення об'єктів) предметної області. Цю роботу виконують аналітики спільно з фахівцями предметної області. Результатом є деяка *концептуальна схема*, що містить опис базових компонентів (об'єктів) і тих функцій, які вони виконуватимуть.

Здебільшого, на цьому етапі формується *словник* (*глосарій*) предметної області, який містить текстовий опис термінів, сутностей, користувачів тощо, а також формується *технічне завдання*, в якому подано функціональні та нефункціональні вимоги до системи.

Етап *проектування* структури програми полягає в розробці детальної схеми, на якій зазначено класи, їхні властивості і методи, а також різні взаємозв'язки між ними. Зазвичай, на цьому етапі у роботі можуть брати участь аналітики, архітектори системи, а також окремі кваліфіковані програмісти.

Результатом цього етапу повинна стати деталізована схема програми, за якою описано всі класи і взаємозв'язки між ними в процесі функціонування програми.

Етап *програмування* є найтрадиційнішим для програмістів. Поява інструментаріїв швидкої розробки додатків (Rapid Application Development, RAD) дає змогу істотно скоротити час і витрати на виконання цього етапу. Результатом такого етапу є програмний додаток, що має необхідну функціональність і здатний вирішувати потрібні задачі у конкретній предметній області.

Етапи *впровадження* і *супроводу* програми зв'язані з необхідністю настроювання і конфігурації середовища програми, а також з усуненням помилок, які виникли під час її використання. Іноді окремим етапом вважають *тестування* програми, під яким розуміють перевірку працездатності програми на деякій сукупності вихід-

них даних чи за деяких спеціальних режимів експлуатації. Результатом цих етапів є підвищення надійності додатка, що виключає виникнення критичних ситуацій.

Життєвий цикл програми налічує певні етапи, які циклічно повторюються. На кожному етапі виникають або модифікуються певні матеріали (*артефакти*), які використовують на наступному етапі як вхідні дані. На рис. 1.1 назви артефактів підкреслено.

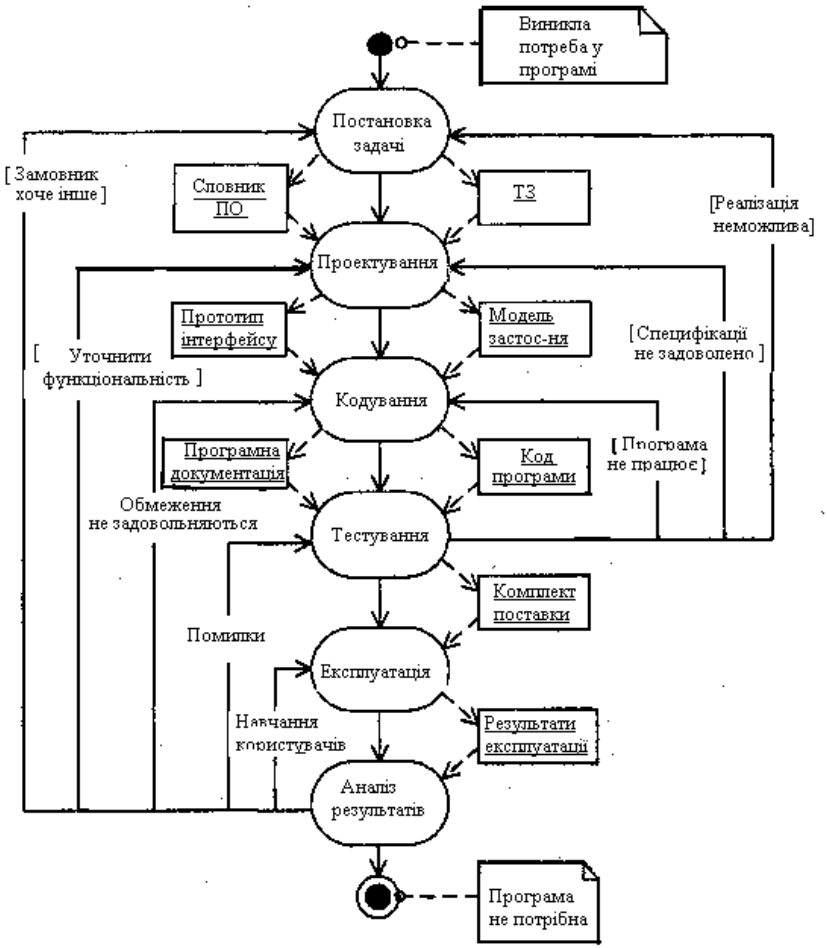


Рис. 1.1. Життєвий цикл програми

Розглядаючи різні етапи життєвого циклу програми, необхідно зазначити: якщо виникнення RAD-інструментаріїв дає змогу істотно скоротити терміни етапу програмування, то відсутність відповідних засобів для перших двох етапів тривалий час стримувала процес розробки додатків. Розвиток методології *об'єктно-орієнтованого аналізу і проектування* (ООА і П) програмних систем було спрямовано на автоматизацію другого, а потім і першого етапів життєвого циклу програми.

Методологія ООА і П тісно переплітається з концепцією автоматизованої розробки програмних систем (Computer Aided Software Engineering, CASE). До перших CASE-засобів ставилися з певною недовірою. Згодом з'явилися як захоплені відгуки про їхнє застосування, так і критичні оцінки їхніх можливостей. Причин для таких суперечливих думок було декілька. Перша з них полягає у тому, що ранні CASE-засоби були простою надбудовою над деякою системою керування базами даних. Хоча візуалізація процесу розробки концептуальної схеми бази даних має велике значення, проте вона не вирішує проблем розробки додатків інших типів.

Друга причина має складнішу природу, оскільки зв'язана з графічною нотацією, реалізованою у CASE-засобі. Якщо мови програмування мають строгий синтаксис, то спроби запропонувати придатний синтаксис для візуального зображення концептуальних схем БД сприйняли не надто прихильно. Отож розробку і стандартизацію *уніфікованої мови моделювання* (Unified Modeling Language, UML), орієнтованої на об'єктно-орієнтований аналіз і проектування програмних систем, співтовариство корпоративних програмістів сприйняло з великим оптимізмом.

У кожної організації, яка спеціалізується на розробці програмних систем, зазвичай існують власні моделі життєвого циклу та процесу розробки програми. Однак внаслідок широкого використання Case-засобів останніми роками набувають поширення *універсальні* та певним чином *стандартизовані* моделі.

Найпоширенішими сьогодні є моделі MS Solution Framework (MSF) корпорації Microsoft та Unified Software Development Process (USDP) фірми Rational. Модель життєвого циклу рис. 1.1 слугує певним узагальненням і спрощенням цих двох моделей.

1.3. Вступ у процес моделювання

Компанії, які займаються випуском програмних систем, досягають успіху у випадку, коли їхня продукція має високу якість і максимально враховує запити користувачів.

Для швидкої та ефективної розробки програмного продукту необхідно залучити кваліфіковану робочу силу, вибрати правильні інструменти і визначити правильний напрям роботи. Процес розробки проекту необхідно добре продумати, щоб швидко адаптувати його до можливих змін вимог користувачів, потреб бізнесу чи технології. Цього можна досягнути, якщо у процесі розробки проекту використовувати *моделі*.

Безперечно, розробка сучасних програмних продуктів неможлива без попереднього моделювання. Накопичений досвід засвідчує: чим більшим і складнішим є проект, тим важливішим стає *моделювання* майбутньої системи. Не варто сподіватися на успішність проекту, якщо не приділено достатньої уваги попередньому моделюванню системи.

Моделювання використовують не тільки під час створення великих систем. Адже чим більша і складніша система, тим більшого значення набуває моделювання при її розробці. Справа у тому, що моделювати складну систему необхідно у будь-якому випадку, оскільки інакше ми не зможемо її представити як єдине ціле.

Моделі наочно демонструють бажану структуру та поведінку системи, відображають її архітектуру та допомагають уточнити деталі проекту з замовником для мінімізації майбутніх ризиків.

Модель – це спрощене представлення реальності, своєрідне “креслення” системи. Кожну систему можна описати по-різному, використовуючи різні моделі, кожна з яких є *семантично замкнутою абстракцією* системи.

Моделі програмних систем відображають певні аспекти системи. *Структурна модель*, наприклад, відображає статичну організацію системи, а *модель поведінки* підкреслює динамічні процеси, притаманні системі.

Зазвичай, моделювання будь-якої системи супроводжується створенням *множини моделей* для відображення різних аспектів

системи. Окрім цього, моделі можуть мати різні *рівні абстракції*, які відображають одні й ті ж аспекти системи з різним ступенем деталізації.

Моделювання дає змогу розв'язати такі задачі:

- *візуалізація системи* – візуальне відображення програмної системи у її поточному чи бажаному стані;
- *специфікація системи* – визначення структури і/або поведінки системи;
- *конструювання системи* – отримання шаблону, який допоможе сконструювати систему;
- *документування системи* – фіксація прийнятих рішень на основі отриманих моделей.

При розробці програмних систем існує декілька методів моделювання, які відрізняються своєю орієнтацією на *стиль програмування.* Зазвичай, вирізняють п'ять стилів:

- *процедурно-орієнтований* – спрямований на представлення програми як множини процедур, які за чергою викликаються;
- *об'єктно-орієнтований* – спрямований на представлення програми як набору взаємодіючих об'єктів;
- *логіко-орієнтований* – спрямований на виконання цілей, які передано у термінах обчислення предикатів;
- *орієнтований на правила* – виконання правил "якщо-то";
- *орієнтований на обмеження*.

Як стверджують автори [1], неможливо визнати один стиль програмування найкращим у *всіх* областях практичного застосування, однак об'єктно-орієнтований стиль найприйнятніший для найширшого кола задач.

Алгоритмічний метод моделювання передбачає процедурно-орієнтований стиль програмування, у рамках якого програміст створює процедури, що викликають одна одну для виконання поставлених задач і обробки даних.

Головним будівельним блоком є процедура чи функція, а увага приділяється насамперед питанням передачі керування і декомпозиції великих алгоритмів на менші. Нічого поганого у цьому

немає, якщо не зважати на те, що системи важко адаптуються при зміні вимог чи збільшенні розміру додатка.

За використання об’єктно-орієнтованого стилю програміст створює програмні об’єкти і наділяє їх визначеною поведінкою, реакцією на зміни зовнішніх умов. Такі об’єкти взаємодіють між собою, виконують визначені задачі, приймають, обробляють і передають дані.

На об’єктно-орієнтованому програмуванні базується *об’єктно-орієнтований* метод моделювання. Якщо об’єктно-орієнтоване *програмування* спрямоване на правильне й ефективне використання об’єктів у рамках конкретних мов, то об’єктно-орієнтоване *моделювання* спрямоване на правильне й ефективне *структурування складних систем*.

Об’єктно-орієнтований метод моделювання передбачає такий *аналіз* предметної області, за якого уже на початкових етапах розробки програмної системи можна було б вирізняти набори взаємодіючих об’єктів.

Об’єктно-орієнтована модель має чотири головні властивості:

- *абстрагування* – виокремлення істотних характеристик об’єкта, що вирізняють його з-поміж інших видів об’єктів;
- *інкапсуляція* – приховування внутрішньої реалізації об’єкта за наданим цим об’єктом інтерфейсом;
- *модульність* – здатність системи розкладатися на внутрішньо сильно чи слабко зв’язані між собою модулі;
- *ієрархія* – упорядкування абстракцій і розташування їх за рівнями.

Ці властивості є головними, і за відсутності будь-якого з них модель не буде об’єктно-орієнтованою.

Існує також три додаткових властивості, корисні в об’єктній моделі, без яких, однак, можна обійтися:

- *типізація* – створення об’єктів на основі шаблонів визначеного типу;
- *паралелізм* – здатність системи обробляти декілька повідомлень чи задач паралельно;
- *збережуваність* – здатність системи зберігати не тільки дані, але й об’єкти у проміжку між окремими запусками системи.

Об'єктно-орієнтоване моделювання програмних систем довело свою корисність при побудові систем будь-якого розміру і складності у різних областях людської діяльності. Окрім того, сучасні мови програмування, інструментальні засоби та операційні системи, здебільшого, є тією чи іншою мірою об'єктно-орієнтованими, а це дає вагомі підстави трактувати світ у термінах об'єктів.

1.4. Класи та об'єкти

Для створення об'єктно-орієнтованої програми необхідно створити деякий набір об'єктів з визначеною поведінкою та схемою їхнього взаємозв'язку. У свою чергу, для створення об'єктів необхідно попередньо створити їхній опис, який у термінах C++ називають класом.

Клас – це абстракція сукупності об'єктів, які мають спільний набір властивостей і володіють однаковою поведінкою. Об'єктом називають екземпляр відповідного класу. Об'єкти, які не мають ідентичних властивостей чи не володіють однаковою поведінкою, за визначенням не можуть належати одному класу.

Клас – це шаблон, на основі якого створено об'єкти. Не можна плутати клас і об'єкт. Клас – це лише матриця, на основі якої створюють об'єкти.

Клас визначеного типу може бути тільки один, а об'єктів у програмі може бути скільки завгодно (точніше, наскільки вистачить ресурсів системи). Перевірити правильність опису створеного класу можна тільки після створення об'єктів на його основі. Коли об'єкти починають працювати і взаємодіяти, тільки тоді можна оцінити точність поводження об'єкта, описаного класом.

Якщо взяти приклад телефонного апарата, то його електронна схема – те саме, що клас, а сам апарат – те саме, що об'єкт. На основі однієї електронної схеми можна зробити скільки завгодно апаратів, і усі вони працюватимуть і виглядатимуть однаково за умови, що на заводі не допущено браку. У цьому відмінність програмування від реального виробництва. Під час створення програми за точністю виготовлення об'єкта стежить мовний компілятор, а програмістові необхідно зосередитись на правильному описі класу.

Найважливішими властивостями класів вважають інкапсуляцію, успадкування і поліморфізм.

Інкапсуляція класу аналогічна властивості об’єктно-орієнтованої моделі і має на увазі приховання непотрібних деталей реалізації класу. У самому класі можуть зберігатися дані і методи їхньої обробки, доступні тільки за допомогою наданого класом інтерфейсу і захищені від небажаного впливу ззовні. Для використання класу немає необхідності знати його внутрішню будову (адже для того, щоб скористатися телефоном, немає необхідності знати його електронну схему). Необхідно лише знати, як набрати номер чи відповісти на дзвінок, тобто знати опис зовнішнього інтерфейсу.

Успадкування – одна із найважливіших властивостей класу. Ця властивість дає змогу створювати на основі одного чи декількох *батьківських* класів *дочірні* класи (підкласи) із властивостями батьківських і власними додатковими можливостями.

Можливість успадкування властивостей дає змогу програмістові скоротити обсяг ручного кодування і змінити поведінку всіх дочірніх об’єктів унаслідок зміни поведінки батьківського класу.

Якщо в реальному виробництві у схемі уже випущених телефонних апаратів буде знайдено помилку, то всю партію доведеться викинути на смітник. У програмуванні – навпаки: виправляючи знайдену помилку в батьківському класі, ми після запуску програми змушуємо правильно працювати усі дочірні класи.

Поліморфізм – можливість об’єктів, створених на основі класів, змінювати свою реакцію на ті ж самі впливи за різних зовнішніх умов.

Класи мають атрибути і методи. У програмі *атрибути* (або *властивості*) – це змінні, описані в тілі класу, що можуть бути як сховані від зовнішнього впливу (зміна властивостей виробляється за допомогою доступних ззовні методів), так і доступні для зміни. *Методи* – це функції, визначені в тілі класу, що можуть бути доступні чи сховані від зовнішніх програм.

1.5. Методологія об’єктно-орієнтованого моделювання

Метод об’єктно-орієнтованого моделювання передбачає послідовне виконання двох етапів: об’єктно-орієнтованого *аналізу* та об’єктно-орієнтованого *проектування*. Тому термін “об’єктно-орієнтоване моделювання” еквівалентний терміну “об’єктно-орієнтований аналіз і проектування” (ООА і П) [5].

Аналіз – широке поняття. Його зміст детальніше відображають терміни *аналіз системних вимог* (тобто дослідження вимог до майбутньої програмної системи) та *об'єктний аналіз* (тобто дослідження об'єктів предметної області).

При цьому під *предметною областю* розуміють ту частину реального світу, що має істотне значення чи безпосереднє відношення до процесу функціонування програми. Іншими словами: предметна область містить у собі тільки ті об'єкти і взаємозв'язки між ними, які необхідні для опису вимог і умов розв'язання деякої задачі.

У загальному випадку виділення базових об'єктів (чи компонентів) предметної області є нетривіальною задачею. Складність виявляється у неформальному характері процедур чи правил, які можна застосовувати з цією метою. Окрім того, таку роботу необхідно виконувати спільно з фахівцями чи експертами, що добре знають предметну область.

У процесі *проектування* головну увагу звертають на концептуальні рішення, які забезпечують виконання системних вимог, а не на питання реалізації. У процесі об'єктно-орієнтованого проектування визначають програмні об'єкти та способи їхньої взаємодії і/або схеми баз даних.

Під час *аналізу системних вимог* необхідно з'ясувати потреби *замовника*, аналізуючи отриману інформацію від керівництва компанії та майбутніх користувачів системи. Під час аналізу необхідно визначити:

- *функціональні вимоги* до системи (або *бізнес-процеси*), тобто встановити *варіанти використання* програмної системи для реалізацій конкретних функцій чи дій у даній предметній області ("визначити те, що система *має робити*");
- *потоки даних* для кожного бізнес-процесу;
- *границі* системи;
- *користувачів* системи та процеси їхньої взаємодії з системою.

У результаті аналізу, зазвичай, оформляють *словник* (або *глосарій*) предметної області (містить текстовий опис термінів, сутностей, користувачів тощо) і *технічне завдання*, у якому сформульовано функціональні та нефункціональні вимоги до системи. До *не-*

функціональних вимог зачислено питання надійності, зручності використання, продуктивності, можливості супроводу програм, питання безпеки, проектні та апаратні обмеження тощо.

Технічне завдання є гарантією єдиного трактування вимог замовниками і проектувальниками. Воно дає змогу також вирішувати спірні питання з приводу функцій системи, що виникають у процесі її створення.

В UML певним синонімом терміна "*технічне завдання*" є *специфікація вимог* до системи (Software Requirement Specification, SRS), в яких визначаються межі системи, користувачів і функціональні вимоги. SRS є текстовою основою *формалізації* етапу постановки задачі за допомогою діаграм прецедентів.

Приклад 1.1. Власник невеликої авіакомпанії, що забезпечує перельоти на місцевих авіалініях, хоче надати клієнтам можливості перегляду інформації про польоти і бронювання місць за допомогою системи реєстрації на web-вузлі компанії. Система налічує такі вимоги:

- на web-вузлі можуть обслуговуватися як корпоративні клієнти (компанії), так і фізичні особи;
- відвідувачі web-вузла матимуть змогу одержувати інформацію про розклад авіарейсів;
- відвідувачі можуть бронювати місця в літаку, зазначаючи номер рейсу і необхідну кількість місць;
- для бронювання місць відвідувачі web-вузла заповнюють реєстраційну форму, у якій повинні бути зазначені дані:
  ```
  прізвище, ім'я, поштову адресу, назву компанії
  (для корпоративних клієнтів),номер телефону,
  номер факсу, адресу електронної пошти
  ```
- відвідувачі, не зареєстровані на web-вузлі, можуть переглядати інформацію про авіарейси, однак не можуть бронювати місця;
- після бронювання місць система надсилатиме клієнтові підтвердження електронною поштою;
- корпоративні клієнти, що часто користуються послугами компанії, мають визначену знижку; деяку знижку в придбанні квитків отримують й інші постійні клієнти;

- замовлення може бути відмінене не пізніше, ніж за тиждень до вильоту, у цьому випадкові клієнтові повертають 80% вартості квитка;
- співробітники компанії, що займаються продажем авіаквитків, можуть переглядати та поновлювати інформацію про рейси.

Цей простий приклад демонструє, як декількома короткими твердженнями можна визначити усі функціональні можливості системи. Тут описано вимоги до системи з погляду її користувачів.

? Запитання для самоперевірки

1. Дайте визначення програмної системи.
2. Як можна класифікувати програмні системи?
3. Перелічіть типи програмних систем за областю застосування.
4. Перелічіть типи програмних систем за масштабом використання.
5. Що розуміють під процесом розробки програми?
6. Що розуміють під життєвим циклом програми?
7. Опишіть стандартну модель життєвого циклу програми.
8. Що таке артефакт?
9. Що містить технічне завдання?
10. Що містить словник предметної області?
11. Чи можна за допомогою однієї моделі описати складну систему?
12. Які задачі дає змогу розв'язати модель програмної системи?
13. Охарактеризуйте алгоритмічний метод моделювання.
14. Охарактеризуйте об'єктно-орієнтований метод моделювання.
15. Охарактеризуйте головні властивості об'єктно-орієнтованого методу моделювання.
16. Дайте визначення класу.
17. Дайте визначення об'єкта.
18. Дайте визначення атрибута класу.
19. Дайте визначення методу класу.
20. Що розуміють під об'єктно-орієнтованим аналізом?
21. Що розуміють під об'єктно-орієнтованим проектуванням?
22. Охарактеризуйте функціональні вимоги до програм.
23. Охарактеризуйте нефункціональні вимоги до програм.

2. ОСНОВИ УНІФІКОВАНОЇ МОВИ МОДЕЛЮВАННЯ (UML)

План викладу матеріалу:

1. Загальна характеристика UML.
2. Архітектурний базис UML.
3. Відношення.
4. Діаграми UML.
5. Правила і загальні механізми мови UML.
6. Представлення моделі.

Ключові терміни розділу

- *Визначення UML*
- *Структурні сутності*
- *Анотаційна сутність*
- *Основні типи діаграм*
- *Позначені значення*
- *Основні властивості UML*
- *Сутність групування* (*пакет*)
- *Основні типи відношень*
- *Стереотипи*
- *Обмеження*

2.1. Загальна характеристика UML

Уніфікована мова моделювання (Unified Modeling Language, UML) – це графічна мова для *специфікації*, *візуалізації*, *конструювання* і *документування* програмних систем. За допомогою UML можна розробити детальний план такої системи, який відображатиме і *концептуальні* елементи системи (системні функції та бізнес-процеси), і особливості її *реалізації* (класи, схеми баз даних, програмні компоненти багаторазового використання тощо).

Авторами мови є Грейді Буч (Grady Booch), Джеймс Рамбо (James Rumbaugh) і Айвар Якобсон (Ivar Jacobson). У січні 1997 року внаслідок об'єднання розробок цих авторів випущено версію UML 1.0, а в листопаді 1997 року – версію UML 1.1. Наступні версії: UML 1.3 – квітень 1999 року; UML 1.4 – жовтень 2001 року.

Зауважимо, що базові ідеї та конструкції мови практично не змінювалися з версії UML 1.1. У наступних версіях уточнювали визначення, добавляли розділи, регламентували зв'язки UML з іншими технологіями тощо. Паралельно розвивалися інструментальні засоби, які підтримували UML (Rational Rose Enterprise, Objecteering, Magic Draw UML, MS Visio, Visual UML та ін.).

Сьогодні UML є загальновизнаним стандартом, який використовує більшість розробників системного та прикладного програмного забезпечення. Знань UML вимагають не лише від системних аналітиків та проектувальників, але й від звичайних програмістів і тестувальників програмного забезпечення. Постійно збільшується ринок UML-орієнтованих інструментальних засобів, призначених для автоматизації процесу розробки програм.

Безперечно, UML відіграватиме важливу роль у галузі розробки програмного забезпечення і в майбутньому. Розвиток UML буде спрямований на спрощення розв'язку однієї з найскладніших задач в галузі інформаційних технологій – задачі проектування програмного забезпечення.

UML призначено для моделювання програмних систем. Самі автори UML визначають її як графічну мову моделювання загального призначення, яку використовують для *специфікації*, *візуалізації*, *конструювання* і *документування* усіх артефактів[1], які створюються під час розробки програмних систем.

- *Специфікація* – це декларативний опис того, як усе побудоване або працює. UML надає достатньо формальні та універсальні засоби для специфікації усіх можливих артефактів, що дає змогу знизити ризик неоднозначного сприйняття специфікації.
- *Візуалізація* – представлення інформації у графічній формі, придатній для сприйняття людиною. Часто моделювання є єдиним засобом, який дає змогу уявити систему загалом як одне ціле. Проблема полягає в обмеженому сприйнятті людиною складних сутностей. Моделювання передбачає розділення складної системи на дещо простіші складові та окремо розглядає кожну з цих складових. Також моделювання дає змогу створювати високорівневі моделі всієї системи, відкидаючи деталі, несуттєві для цього рівня абстракції.
- *Конструювання* – отримання набору програмних модулів, які утворюють застосування або його компонент. Розроблені мо-

[1] Будь-який створений і відчужений матеріал проекту (модель, програмний код, документація тощо).

делі системи утворюють деякий базовий каркас, на основі якого можна будувати систему. Сучасні CASE-засоби дають змогу деякою мірою автоматизувати конструювання програмного коду на підставі розроблених моделей.

- *Документування проектних рішень.* Для підтримки та розвитку програмних продуктів потрібна вичерпна та якісна документація. Моделювання дає змогу одержати документи, які визначають високорівневу організацію системи. Такі документи необхідні для початкового ознайомлення з системою.

Серед головних властивостей UML можна виокремити такі:

- *UML – це мова моделювання*, яку використовують в контексті деякого процесу розробки програмних засобів.
- *UML – це функціонально завершена мова*, яка забезпечує повний цикл моделювання програмного забезпечення, починаючи від формування концепції майбутньої системи і завершуючи питаннями програмної реалізації системи.
- *UML – це об'єктно-орієнтована мова*, яку найефективніше можна застосовувати саме в контексті об'єктно-орієнтованих методів аналізу та проектування.
- *UML – це формальна мова*, яка дає змогу будувати завершені моделі, яким властива однозначна інтерпретація. Наслідком формалізації мови є можливість її інтерпретації не лише людьми, але й машинами.
- *UML – це мова візуалізації*, орієнтована на представлення моделей програмних систем переважно в графічній формі. Водночас UML припускає долучення до моделі текстової інформації з метою додаткової конкретизації деталей.
- *UML – це стандартна мова*, яка гарантує вільне розуміння та поширення UML-моделей між різними розробниками.
- *UML – це універсальна мова*, яка з однаковим успіхом придатна для моделювання як великих, так і малих програмних систем.
- *UML – це незалежна мова*, яка не накладає обмежень на мови програмування для реалізації моделей і може бути сумісною з будь-якою об'єктно-орієнтованою мовою.

Використання UML найефективніше в інформаційних системах масштабу підприємства, банківських і фінансових установах, телекомунікаціях, на транспорті, у торгівлі, науці тощо.

UML можна використовувати, наприклад, для моделювання документообігу в юридичних фірмах чи керівних структурах, для опису структури та функціонування системи обслуговування пацієнтів у лікарнях, для проектування апаратних засобів тощо.

Зазначимо ще одну область, у якій активно застосовують графічну нотацію – це виконання робіт з приведення системи менеджменту якості у відповідність зі стандартом ISO 9001:2000 у рамках сертифікації підприємств і компаній. У цій області мову UML застосовують для візуального моделювання та документування бізнес-процесів. Розроблені діаграми і пояснення до них надсилають міжнародним сертифікаційним органам для отримання відповідного сертифікату.

Концептуальна модель (або метамодель) UML складається з трьох частин: *архітектурного базису*, *правил мови* та *загальних механізмів мови*.

2.2. Архітектурний базис UML

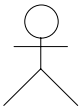
Архітектурний базис UML визначає базові поняття, якими оперує мова: *сутності*, *відношення* та *діаграми*.

Сутності – це певні абстракції, які є базовими елементами моделей. В UML є чотири типи сутностей: *структурні* (актори, класи, інтерфейси, компоненти, вузли), *поведінки* (прецеденти, діяльності, стани і повідомлення), *групування* та *анотаційні*.

Структурні сутності – це статичні поняття, які відповідають концептуальним, логічним чи фізичним елементам системи. Структурні сутності, зазвичай, позначають *іменниками*. Розрізняють п'ять *головних* структурних сутностей: *актори*, *класи*, *інтерфейси*, *компоненти*, *вузли*. Кожна з сутностей може мати свої підвиди[1].

[1] Поділ на головні сутності та їхні підвиди строго не фіксується, отож у літературі трапляється й дещо інша класифікація сутностей.

Актор (Actor) – це суб'єкт, який перебуває поза системою, що моделюється, і безпосередньо з нею взаємодіє. Графічно акторів зображають значком "худа людина", під яким вказують ім'я актора (рис. 2.1).



Адміністратор

Рис. 2.1. Зображення актора

Клас (Class) – це сукупність *однотипних сутностей* предметної області (*об'єктів*) зі спільними атрибутами, операціями, відношеннями та семантикою.

В UML класи зображають прямокутником, розділеним на три секції, в яких записують назву класу, атрибути та операції, відповідно (рис. 2.2). Назву абстрактного класу позначають *курсивом*. Атрибути та операції мають чітко визначені формати запису, які відображають їхні найважливіші характеристики (назви, типи тощо). За необхідності секції атрибутів і/або операцій опускають.

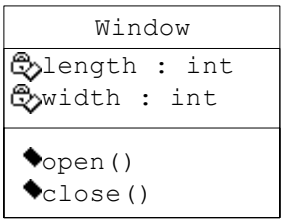


Рис. 2.2. Зображення класу

Об'єкти (Objects) – це *екземпляри класів* з конкретними значеннями атрибутів. Об'єкт має зображення, подібне до зображення класу, проте назву об'єкта підкреслюють і записують у вигляді: <назва об'єкта>:<назва класу>. Якщо ідентифікація об'єкта неважлива, то вказують лише назву класу, до якого належить об'єкт: :<назва класу>. При зображенні об'єктів секції атрибутів та операцій, здебільшого, опускають.

Інтерфейс (Interface) – це сукупність операцій, що формують деякий сервіс, який надає клас чи компонент. Інтерфейс лише декларує операції, а реалізація операцій покладається на клас або компонент, який підтримує цей інтерфейс. Інтерфейси зображають колом, під яким вказують назву інтерфейсу (рис. 2.3).

Visible

Рис. 2.3. Зображення інтерфейсу

Компонент (Component) – це фізично заміщувана частина системи, яка відповідає певному набору інтерфейсів і/або забезпечує реалізацію іншого набору інтерфейсів. Компоненти фізично існують під час виконання програми. Графічне зображення компонента – прямокутник з двома виступами з лівого боку і назвою усередині (рис. 2.4).

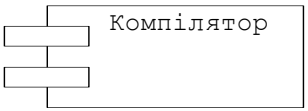


Рис. 2.4. Зображення компонента

В означенні компонента цілковито викладено його семантику:

- компонент має *фізичну* природу – він існує в реальному світі бітів, а не у світі концепцій;
- компонент *заміщуваний* – замість одного компонента можна підставити інший, якщо він відповідає тому ж самому набору інтерфейсів;
- компонент – це *частина* системи.

У багатьох аспектах компоненти подібні до класів: мають назви, можуть реалізувати інтерфейси, вступати у певні відношення, бути вкладеними, вступати у взаємодії. Однак між ними є суттєві розбіжності:

- класи – логічна абстракція, а компоненти – фізичні сутності (можуть розміщуватися у *вузлах*);

- компоненти слугують фізичною упаковкою логічних сутностей (класів, кооперацій тощо), отож є на іншому рівні абстракції порівняно з класами;
- класи володіють атрибутами та операціями, а компоненти – лише операціями, доступними через їхні інтерфейси.

Можна виокремити три групи компонентів:

1. *Компоненти розгортання* (Deployment components): динамічно під'єднувані бібліотеки (DLL) і програми виконання (EXE).
2. *Компоненти – робочі продукти* (Work product components): файли з вихідними текстами програм чи даними; бази даних або таблиці баз даних; документи; виконавчі модулі із закритими механізмами комунікації тощо.
3. *Компоненти виконання* (Execution components) – наслідок роботи системи (прикладом є об'єкт COM+, екземпляр якого створюється з DLL).

Вузол (Node) – це фізичний елемент системи, який існує під час виконання програми і представляє обчислювальний ресурс. Вузли зображають кубом, в якому вказується назва вузла (рис. 2.5). Вузол володіє певним обсягом пам'яті і, можливо, процесором.

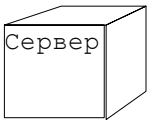


Рис. 2.5. Зображення вузла

Вузли надають засоби фізичного розгортання компонентів. Найпоширеніший приклад використання вузлів – це моделювання процесорів і пристроїв, які утворюють топологію автономної, вбудованої, клієнт-серверної чи розподіленої комп'ютерної системи.

Сутності *поведінки* (прецеденти, діяльності, стани і повідомлення) розглядатимемо під час вивчення відповідних *діаграм*.

Сутності *групування – пакети* (packages) можуть містити структурні сутності, сутності поведінки та інші сутності групування. На відміну від компонентів, які реально існують під час роботи програми, пакети мають чисто концептуальний характер (існують тільки під час процесу розробки).

Пакет зображають прямокутником із “закладкою” – меншим прямокутником, приєднаним до верхнього лівого кута (рис. 2.6).

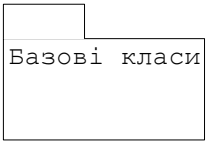


Рис. 2.6. Зображення пакета

Анотаційна сутність – це коментар для пояснення чи зауваження до будь-якого елемента моделі. Є тільки один тип анотаційної сутності – *примітка* (note). Графічно *примітку* зображають прямокутником із загнутим правим верхнім кутом (рис. 2.7).

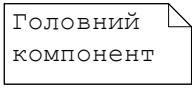


Рис. 2.7. Зображення примітки

2.3. Відношення

Відношення – це відображення семантичного зв’язку між сутностями. У мові UML визначено чотири основні типи відношення: *залежності*, *асоціації*, *узагальнення* та *реалізації*.

Залежність (dependency) – це відношення *використання*, за якого зміна однієї сутності (*незалежної*) може вплинути на іншу сутність, яка її використовує, причому зворотне використання, зазвичай, неприпустиме. Для зображення залежності використовують пунктирну лінію зі стрілкою (рис. 2.8), спрямованою у бік *незалежної* сутності.

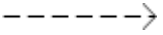


Рис. 2.8. Зображення залежності

Найчастіше залежності використовують під час моделювання класів, щоб відобразити у сигнатурі операції той факт, що один клас використовує інший клас (незалежну сутність) аргументом.

Асоціація (association) – це структурне відношення, що описує множину зв'язків (з'єднань) між об'єктами. Різновид асоціації – *агрегування* (aggregation) – це структурне відношення між цілим і його частинами. Графічно асоціацію зображають у вигляді лінії (іноді завершується стрілкою), поруч з якою можуть бути додаткові позначення (кратність, назви ролей тощо). На рис. 2.9 зображено приклад відношення цього вигляду.

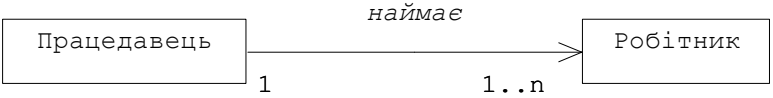


Рис. 2.9. Приклад асоціації

Узагальнення (generalization) – це відношення типу "спеціалізація/узагальнення", за якого об'єкт спеціалізованого елемента (*нащадок*) може бути підставлений замість об'єкта узагальненого елемента (*батька*, *предка*), проте не навпаки. За принципом об'єктно-орієнтованого програмування, нащадок (child) успадковує структуру і поведінку свого предка (parent). Графічно відношення узагальнення зображають лінією з незафарбованою стрілкою, яка вказує на предка (рис. 2.10).

Рис. 2.10. Приклад узагальнення

Здебільшого, у нащадка, окрім унаслідуваних, є і власні атрибути та операції. Операція нащадка з тією ж самою сигнатурою, що й у предка, заміняє відповідну операцію предка (властивість *поліморфізму*).

Найчастіше узагальнення використовують при моделюванні класів. Клас може мати одного (Single inheritance) або декілька предків (Multiple inheritance), чи не мати їх зовсім. Клас, у якого немає предків, а є нащадки, називають *базовим* (або *кореневим*). Клас, у якого немає нащадків, називають *листковим*.

Узагальнення використовують також з метою відображення наслідування між класами та інтерфейсами або з метою відображення наслідування між пакетами тощо.

Реалізація (realization) – це відношення між класифікаторами, за якого один класифікатор визначає зобов'язання, а інший гарантує їхнє виконання. Відношення реалізації трапляються у двох випадках:

- між інтерфейсами і класами/компонентами, що їх реалізують;
- між прецедентами і кооперaціями, що їх реалізують.

Відношення реалізації зображають у вигляді пунктирної лінії з незафарбованою стрілкою, як щось середнє між відношенням узагальнення і залежності (рис. 2.11):

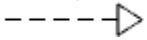


Рис. 2.11. Зображення реалізації

Ми розглянули чотири типи відношень, які є базовими у моделях UML. Існують також їхні варіанти: *уточнення* (refinement), *трасування* (trace), *долучення* і *розширення* для залежностей тощо.

2.4. Діаграми UML

Діаграма UML – це графічне зображення елементів системи у формі зв'язаного графа з *вершинами* (сутностями) і *ребрами* (відношеннями). Діаграми можуть містити будь-яку комбінацію сутностей, однак у практиці моделювання застосовують порівняно невелику кількість типових комбінацій, а саме:

- *Діаграми класів* (class diagram) – зображають класи, інтерфейси, об'єкти і кооперації, а також відношення між ними. Ці діаграми відображають *статичні* аспекти системи.
- *Діаграми об'єктів* (object diagram) – зображають об'єкти і відношення між ними. Це *статичні* знімки екземплярів сутностей, зображених на діаграмах класів.
- *Діаграми прецедентів* (use case diagram) – зображають прецеденти й акторів, а також відношення між ними. Ці діаграми відображають *статичні* аспекти системи.

- *Діаграми взаємодії* – зображають об'єкти та повідомлення, якими об'єкти можуть обмінюватися. Зазвичай, розглядають два часткові випадки таких діаграм: *діаграми послідовностей* (sequence diagram), що відображають *часову* упорядкованість повідомлень, і *діаграми кооперації* (collaboration diagram), на яких зображають структурну організацію об'єктів, що обмінюються повідомленнями. Ці типи діаграм є ізоморфними. Діаграми взаємодії відображають *динамічні* аспекти системи.
- *Діаграми станів* (statechart diagram) – зображають автомат, що налічує стани, переходи, події і види дій. Ці діаграми відображають *динамічні* аспекти системи.
- *Діаграми діяльностей* (activity diagram) – це окремий випадок діаграми станів (на діаграмі зображають переходи потоку керування від одного виду діяльності до іншого усередині системи).
- *Діаграми компонентів* (component diagram) – зображають сукупність компонентів та існуючі між ними залежності. Ці діаграми відображають *статичні* аспекти системи.
- *Діаграми розміщення* (deployment diagram) – зображають конфігурацію вузлів системи і розміщених у них компонентів. Ці діаграми відображають *статичні* аспекти системи. Вони зв'язані з діаграмами компонентів, оскільки у вузлі розміщують один чи декілька компонентів.

Програмна система (або деяка інша система) – це сутність аналізу та проектування, яку розглядають з різних позицій за допомогою моделей, які відображають діаграмами. Діаграма – графічна проекція елементів системи. Один елемент можна зображати на різних діаграмах. Кожна діаграма відображатиме одне з можливих представлень елементів системи.

2.5. Правила і загальні механізми мови UML

Елементи мови UML комбінуються один з одним за певними семантичними *правилами*, які дають змогу коректно та однозначно визначати:

- *назви*, які можна надавати сутностям, відношенням і діаграмам;
- *область дії* – контекст, в якому назва має певне значення;

- *видимість*– контекст, в якому назва може використовуватися іншими елементами;
- *цілісність* – узгоджена поведінка та взаємодія елементів;
- *виконання* – правильне розуміння поведінки системи в динаміці.

Моделювання спрощується і здійснюється ефективніше, якщо дотримуватися деяких угод. Роботу з UML істотно полегшує послідовне використання загальних механізмів: *специфікації* (specifications), *доповнення* (adornments), *поділу* (common divisions) і *розширення* (extensibility mechanisms).

Щодо кожного з елементів графічної моделі UML складається *специфікація*, яка містить текстове представлення елемента.

Наприклад, піктограмі класу відповідає специфікація, що цілковито описує його атрибути, операції (з вичерпними сигнатурами) і поведінку, а також може містити й інші деталі: видимість атрибутів і операцій, коментар про те, що клас є абстрактним тощо. Візуально ж піктограма класу відображає його найважливіші аспекти: назву, атрибути й операції.

Отже, графічну нотацію UML використовують для візуалізації системи, а за допомогою специфікацій описують її деталі. Специфікації UML визначають *семантичний план*, що містить у собі складові частини усіх моделей системи і забезпечує їхнє узгодження між собою.

Практично кожен з елементів моделі має унікальне графічне зображення, на якому відображено найважливіші аспекти цього елемента. Однак в моделі можуть використовуватися додаткові елементи зображення цього елемента, які *доповнюють* характеристику цього елемента іншими аспектами. Як головні, так і доповнюючі аспекти елемента моделі описано у специфікації UML.

Доповнення (adornments) – це текстові або графічні об’єкти, долучені до базової нотації елемента, які застосовують для візуалізації деталей його специфікації.

Наприклад, базова нотація асоціації – це лінія, однак її можна доповнювати стрілками, кратностями і назвами ролей.

При роботі з класами, компонентами чи вузлами уточнюючу інформацію можна розмістити у *додатковому* розділі, який розта-

шовують нижче від головних розділів. З метою однозначного розуміння додатковий розділ рекомендують називати явно.

Під час моделювання об'єктно-орієнтованих систем існує *поділ* на *класи*/*об'єкти* та *інтерфейс*/*реалізацію*.

Клас – це абстракція, а *об'єкт* – конкретне втілення цієї абстракції (або екземпляр класу). Наприклад, є прецеденти й екземпляри прецедентів, компоненти й екземпляри компонентів, вузли й екземпляри вузлів тощо. У графічному зображенні об'єкта прийнято його назву підкреслювати.

Інтерфейс декларує зобов'язання, а *реалізація* представляє конкретне втілення цих зобов'язань і точно відповідає оголошеній семантиці інтерфейсу. Майже всі конструкції UML характеризуються дихотомією *інтерфейс/реалізація*. Наприклад, прецеденти реалізуються кооперацією, а операції – методами.

UML – це стандартна мова розробки моделей програмних систем, однак жодна замкнута мова не в змозі охопити нюанси всіх можливих моделей у різних предметних областях. Отож UML є *відкритою* мовою (допускає контрольовані *розширення*). Механізми розширення UML налічують:

- *стереотипи* (stereotype) – розширюють словник UML шляхом створення нових сутностей мови на основі існуючих сутностей;
- *позначені значення* (tagged value) – розширюють властивості основних елементів UML, даючи змогу долучати нову інформацію до специфікації елемента;
- *обмеження* (constraints) – розширюють семантику елементів, даючи змогу створювати нові і скасовувати існуючі правила.

В UML чотири головні типи сутностей дають змогу моделювати величезну кількість систем. Однак іноді доцільно вводити нові сутності, специфічні для предметної області, що моделюється.

Стереотип – це деякий метаклас, який дає змогу вводити нові сутності. У найпростішому випадку стереотип зображають як назву в типографічних лапках (наприклад, «`global`»). Для наочності стереотипу можна призначити піктограму (розмістити її праворуч від назви) чи застосувати новий графічний символ. Усі три підходи проілюстровано на рис. 2.12.

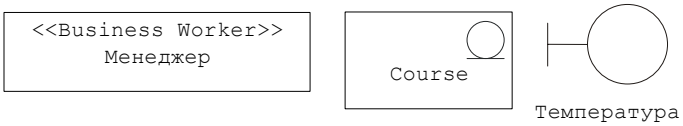


Рис. 2.12. Способи зображення стереотипів

У кожної сутності є фіксований набір властивостей: класи мають назви, атрибути й операції; асоціації –- назви й кінцеві точки (кожна зі своїми властивостями) і т. д. Позначені значення дають змогу додавати нові властивості.

Позначене значення (або *позначка*) – це *метадане* (його значення застосовують щодо сутності), яке зображають так:

```
{ [ НазваПозначки = ] Значення }
```

Фігурні дужки – обов'язкові елементи позначки. Можна вказувати тільки значення, якщо воно допускає однозначну інтерпретацію. Позначки можна визначати для існуючих сутностей UML або застосовувати щодо окремих стереотипів. Позначки розташовують під назвою сутності чи стереотипу. У фігурних дужках може бути декілька позначених значень, які розділяються комами. На рис. 2.13 наведено приклад використання позначених значень у варіанті використання.

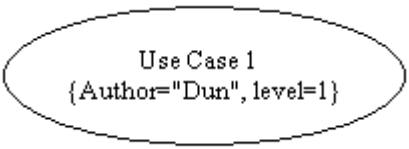


Рис. 2.13. Приклад використання позначених значень

Обмеження – це логічне твердження щодо значень властивостей елементів моделі. Задаючи обмеження, ми тим самим розширюємо семантику елементів.

Обмеження зображають рядком у фігурних дужках, який розташовують після назви елемента чи в окремій примітці, приєднаній до відповідного елемента. Залежно від інструментальних засобів рядок може бути неформальним текстом, логічним виразом мови програмування чи виразом на іншій формальній мові. Обмеження

можна приєднувати відразу до декількох елементів. На рис. 2.14 наведено приклад використання обмеження.

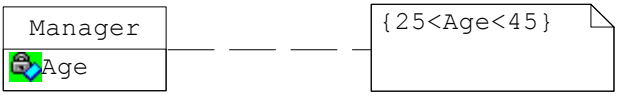


Рис. 2.14. Приклад використання обмеження

Примітка. Використання для позначок та обмежень однієї і тієї ж синтаксичної конструкції (фігурних дужок) не є випадковим. Позначене значення можна вважати обмеженням, яке може набувати тільки одного конкретного значення.

Визначення стереотипу здійснюють так. Беруть за основу деякий існуючий елемент моделі і до нього додають нові позначені значення, нові обмеження і доповнення. Після того, як стереотип визначено, його можна використовувати як елемент моделі.

В UML визначено кілька десятків *стандартних стереотипів*, які доволі часто застосовують під час моделювання. Наприклад, *актор* не є самостійною сутністю метамоделі – це стереотип класу. Ми не описуватимемо усі стандартні стереотипи – чимало з них вам ніколи не знадобиться. Деякі стандартні стереотипи розглянемо при подальшому викладі матеріалу.

2.6. Представлення моделі

Для моделювання програмних систем необхідно розглядати їх з різних позицій. Усі, хто має відношення до проекту, – кінцеві користувачі, аналітики, прикладні програмісти, системні адміністратори, тестувальники, менеджери проектів тощо – переслідують власні інтереси, і кожен дивиться на створювану систему порізному в різні моменти її життя.

Абстрактний граф моделі, який складається з множини різнотипних сутностей і відношень, не підлягає конструюванню загалом. З нього виокремлюють тільки сутності та відношення, актуальні для певної точки зору (*представлення*).

Набір представлень моделі є ще менш формальним і догматичним, ніж набір канонічних типів діаграм. Найпопулярнішим вважають набір представлень, описаних авторами UML:

- *Представлення прецедентів* (Use case view) – це опис поводження системи з позиції зовнішніх користувачів, аналітиків і тестувальників.
- *Логічне представлення* (Logical view) – це опис системи з позиції проектування. Він охоплює класи, інтерфейси та кооперації, що формують словник предметної області та її розв'язок.
- *Представлення процесів* (Process view) – це опис взаємодії процесів під час роботи системи. Він віддзеркалює такі аспекти, як паралелізм, синхронізація, продуктивність, масштабованість і пропускна здатність.
- *Представлення компонентів* (Component view) – це опис конфігурації системи з позиції реалізації, який охоплює компоненти і файли, що використовуються для збирання і випуску готового програмного продукту.
- *Представлення розміщення* (Deployment view) віддзеркалює топологію зв'язків апаратних засобів і розміщення на них компонентів.

? Запитання для самоперевірки

1. Дайте визначення UML.
2. Перелічіть головні властивості UML.
3. Перелічіть структурні сутності UML.
4. Дайте визначення актора.
5. Дайте визначення класу й об'єкта.
6. Дайте визначення вузла й компонента.
7. Перелічіть основні типи відношень.
8. Дайте визначення стереотипу.
9. Дайте визначення позначеного значення.
10. Яким шляхом вводять обмеження у мові UML.
11. Для чого використовують представлення прецедентів?
12. Для чого використовують логічне представлення?
13. Для чого використовують представлення процесів?
14. Для чого використовують представлення компонентів?
15. Для чого використовують представлення розміщення?

3. ОСНОВИ МОДЕЛЮВАННЯ ПОВЕДІНКИ СИСТЕМИ

План викладу матеріалу:

1. Прецеденти використання системи.
2. Діаграма прецедентів (Use Case Diagrams).
3. Організація прецедентів.
4. Створення прецедентів Case-засобом Rational Rose.
5. Специфікації прецедентів.
6. Діаграми видів діяльності.
7. Попередній архітектурний аналіз системи.

Ключові терміни розділу

- *Актори*
- *Організація прецедентів*
- *Дії та види діяльності*
- *Смуги синхронізації*
- *Прецеденти* (*Use case*)
- *Специфікації прецедентів*
- *Переходи та рішення*
- *Зони відповідальності*

3.1. Прецеденти використання системи

Будь-яка програмна система працює у деякому контексті, що визначає зовнішнє оточення системи. Таке оточення формують *користувачі* (або *актори*) системи, якими можуть слугувати як люди, так і системи. Кожен з акторів взаємодіє з системою за своєю власною схемою та очікує від системи певної поведінки й реакції. Схему взаємодії актора з системою називають *прецедентом* (Use case). Синонімами терміна *прецедент* є терміни: *варіант використання*, *сценарій поведінки*.

Загалом *прецедент* специфікує деякий аспект поведінки системи або її частини, не втручаючись в їхню реалізацію, та визначає множину послідовностей дій, спрямованих на досягнення актором очікуваного результату.

Таке визначення прецеденту містить кілька важливих моментів, які вимагають дещо докладнішого пояснення:

- прецеденти можуть існувати на будь-якому *рівні*, починаючи від всієї системи і завершуючи окремими класами;
- прецедент описує деякий *аспект поведінки* відповідного компонента системи;
- прецеденти не дають жодного уявлення про *реалізацію* системи;

- прецедент визначає *множину послідовностей дій*, оскільки аспекти поведінки системи, зазвичай, не є лінійними, а містять різні можливі варіанти розвитку подій;

Зазначимо, що прецеденти описують лише ті аспекти поведінки системи, які є *суттєвими* для акторів системи. Реально, система може виконувати додаткові дії, які також спрямовані на досягнення закладеної в систему функціональності, проте безпосередньо не відображають потреб акторів системи. Такі дії не вважають прецедентами.

Прецеденти застосування системи або її частини зображають в UML за допомогою діаграм *прецедентів* (Use Case Diagrams).

3.2. Діаграма прецедентів (Use Case Diagrams)

Діаграма прецедентів – це узагальнене представлення функціонального призначення системи, яке має відповідати на головне питання моделювання: що робить система у зовнішньому світі?

На діаграмі прецедентів застосовують два типи базових сутностей: варіанти використання (*прецеденти*) і діючі особи (*актори*), між якими встановлюють такі типи відношень:

– асоціація між актором та прецедентом;
– узагальнення між акторами;
– узагальнення між прецедентами;
– залежності чи асоціації (різних типів) між прецедентами.

Під час задання прецедентів акцентують увагу на поводженні системи стосовно заданого актора. Прецеденти допомагають визначити можливості та межі системи. Нотація для прецеденту досить скромна – це текст (*наказова дієслівна форма*), який, залежно від Case-засобу, розташований в овалі або під овалом (рис. 3.1).

Рис. 3.1. Приклад найпростішої діаграми прецедентів

Прецеденти визначають за специфікацією вимог до системи. Актором може бути будь-який взаємодіючий із системою зовнішній суб'єкт: фізична особа (наприклад, торговий агент), зовнішня програмна система (скажімо, програма підготовки і роздруку рахунків) чи пристрій (наприклад, датчик температури). Усі процеси взаємодії між діючими суб'єктами і системою розглядають як прецеденти.

Діаграми прецедентів відображають *статичні* аспекти системи з позиції користувачів. Ця позиція охоплює, здебільшого, поведінку системи, тобто видимі ззовні сервіси, які надає система.

На початкових стадіях моделювання статичних аспектів системи діаграми прецедентів застосовують двома способами:

- *для моделювання контексту системи*: умовно відмежовують систему і виявляють діючих осіб, що перебувають за цією межею і взаємодіють із системою (діаграми прецедентів потрібні на цьому етапі для ідентифікації акторів і семантики їхніх ролей);
- *для моделювання вимог до системи*: визначають те, що система повинна робити з позиції зовнішнього користувача, незалежно від того, як вона повинна це робити (діаграми прецедентів потрібні на цьому етапі для ідентифікації бажаної поведінки системи, яка слугує тут "чорною скринькою").

Будь-яка система містить усередині деякі сутності, водночас інші сутності залишаються за її межами. Наприклад, у системі перевірки кредитних карток (рис. 3.2) наявні *рахунки*, *транзакції* і *механізми перевірки* рахунків. Водночас власники кредитних карток і торгові підприємства перебувають поза системою. Сутності усередині системи відповідають за реалізацію поведінки, якої очікують сутності, що перебувають зовні. Сутності, що перебувають поза системою і взаємодіють з нею, складають її *контекст*.

UML дає змогу моделювати контекст за допомогою діаграм прецедентів, у яких увагу акцентують на зовнішніх акторах. Важливо правильно визначити акторів, оскільки це даватиме змогу описати класи сутностей, що взаємодіють із системою. Ще важливіше визначити, хто не є актором, адже при цьому обмежується

оточення системи (у ньому залишаються тільки ті елементи, що беруть участь у її роботі).

Наприклад, на рис. 3.2 подано контекст системи, що працює з кредитними картками, де головну увагу приділяють зовнішнім акторам (діаграму сформовано Case-засобом Objecteering).

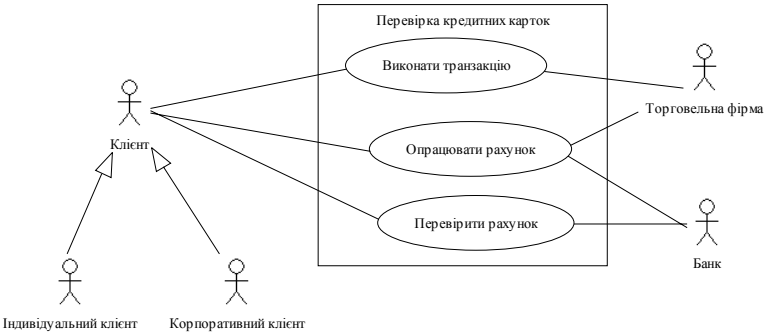


Рис. 3.2. Моделювання контексту системи

Вимоги до системи можна виразити по-різному, від простого тексту до виразів на формальній мові. Певну частину вимог можна сформулювати за допомогою діаграми прецедентів (рис. 3.3).

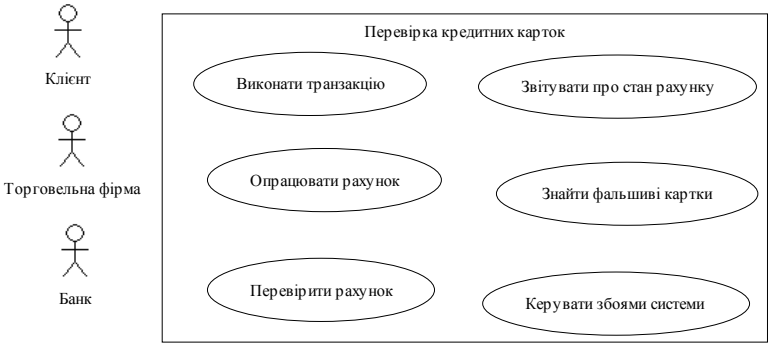


Рис. 3.3. Моделювання вимог до системи

Рис. 3.3 розширює попередню діаграму прецедентів, вводячи додаткові прецеденти, які описують важливі аспекти поведінки системи.

На наступних стадіях моделювання статичних аспектів системи застосовують *комбіновані* діаграми прецедентів, які об'єднують у собі можливості діаграм прецедентів контексту та вимог до системи.

3.3. Організація прецедентів

Прецеденти можна організувати, визначаючи між ними відношення *узагальнення*, *долучення* і *розширення*. Ці відношення застосовують, щоб виділити деяку *загальну* поведінку чи, навпаки, її варіації.

Відношення *узагальнення* означає, що прецедент-нащадок успадковує поведінку і семантику свого предка, може заміщати його чи доповнювати його поведінку та, крім цього, може фігурувати усюди, де тільки з'явиться його предок (як батько, так і нащадок можуть мати конкретні екземпляри).

Наприклад, у системі перевірки карток можлива наявність прецеденту *Перевірити клієнта*, що відповідає за встановлення особистості клієнта (рис. 3.4). Він може мати двох спеціалізованих нащадків (*Перевірити пароль* і *Перевірити сітківку*), які поводяться так само, як прецедент-предок *Перевірити клієнта*, і можуть використовуватися скрізь, де використовується їхній предок, але при цьому кожний з них додає свою власну поведінку (перший перевіряє текстовий пароль, а другий - малюнок сітківки ока).

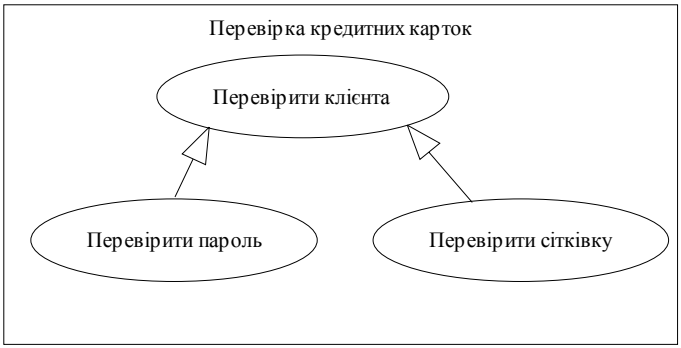


Рис. 3.4. Приклад відношення узагальнення між прецедентами

Відношення *долучення* між прецедентами означає, що в деякій точці базового прецеденту використовують поведінку іншого прецеденту. Прецедент, що долучається, може існувати автономно, а базовий – не може. Можна вважати, що базовий прецедент запозичає поведінку автономного прецеденту, що долучається. Відношення долучення зображають у вигляді залежності чи асоціації зі стереотипом *include*.

На рисунках 3.4 і 3.5 зображено одне і те ж саме відношення долучення, створене за допомогою різних інструментів, між базовим прецедентом *Бронювати місця* та автономним прецедентом *Переглянути інформацію про рейси*.

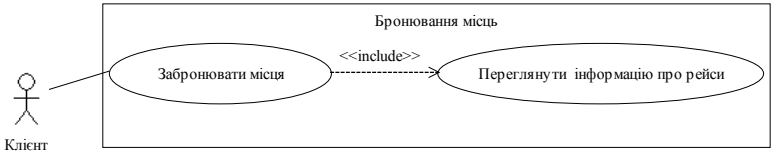


Рис. 3.4. Приклад відношення долучення (Case-засіб Objecteering)

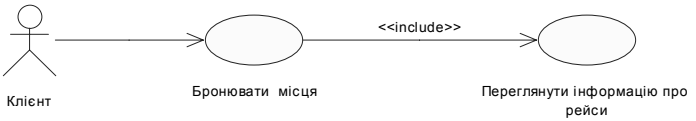


Рис. 3.5. Приклад відношення долучення (Case-засіб Rational Rose)

Відношення *розширення* між прецедентами означає, що базовий прецедент неявно містить поведінку інших прецедентів. Відношення розширення зображають зі стереотипом *extend*.

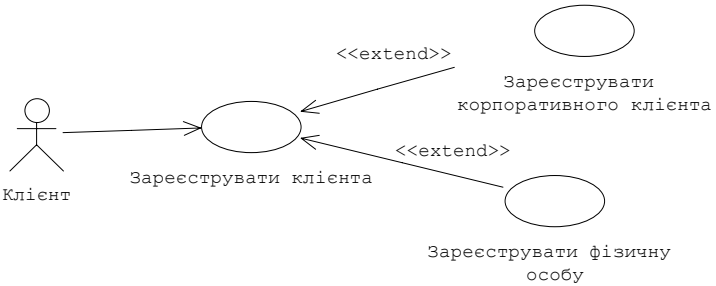


Рис. 3.6. Приклад відношення розширення (Case-засіб Rational Rose)

На рис. 3.6 зображено два відношення, які розширюють базовий прецедент *Зареєструвати клієнта* прецедентами *Зареєструвати корпоративного клієнта* і *Зареєструвати фізичну особу*. На основі базового прецеденту можна розробити головну процедуру реєстрації клієнта, а потім розробити спеціальні процедури, які доповнюватимуть головну процедуру.

Відношення розширення застосовують для моделювання таких частин прецеденту, які користувач сприймає як необов'язкову поведінку системи. Відношення розширення використовують також для моделювання окремих субпотоків, які виконуються лише за визначених обставин. Зрештою, їх застосовують для моделювання декількох потоків, які можуть зливатися в деякій точці.

3.4. Створення прецедентів Case-засобом Rational Rose

При запуску Rational Rose (коротко Rose) у вікні Create New Model оберіть варіант Rational Unified Process. П'ять го

ловних елементів інтерфейсу Rose – це браузер, вікно документації, панелі інструментів, вікно діаграми і журналу.

Браузер (browser) – це ієрархічна структура, що дає змогу здійснювати навігацію по моделі. Усе, що додається до неї (актори, прецеденти, класи, компоненти тощо) буде зображено у вікні браузера. За допомогою браузера можна:

- додавати до моделі елементи;
- переглядати існуючі елементи моделі;
- переглядати існуючі зв'язки між елементами моделі;
- переміщати і/або перейменовувати елементи моделі;
- додавати елементи моделі до діаграми;
- зв'язувати елемент з файлом чи адресою Інтернет;
- групувати елементи в пакети;
- працювати з деталізованою специфікацією елемента;
- відкривати діаграму.

Браузер організований у вигляді дерева, яке підтримує чотири представлення: представлення прецедентів (Use Case View), компонентів (Component View), розміщення (Deployment View) і логічне представлення (Logical View).

За допомогою *вікна документації* (documentation window) можна детально описувати (коментувати) елементи моделі Rose. Усе, що буде написане у цьому вікні, з'явиться потім як коментар у згенерованому коді. Ці коментарі виводитимуться також у звітах середовища Rose.

Панелі інструментів Rose (toolbars) забезпечують швидкий доступ до найрозповсюдженіших команд. Існує два типи панелей інструментів: стандартна панель і панель діаграми. Стандартну панель бачимо завжди; її кнопки відповідають командам, що можуть використовуватися для роботи з будь-якою діаграмою. Панель діаграми є унікальною для кожного типу діаграм UML.

Усі панелі інструментів можуть бути змінені і перебудовані користувачем (➢Tools ➢ Options… 🗀Toolbars).

Вікно діаграми (diagram window) використовується для перегляду і редагування однієї чи декількох діаграм UML.

Журнал (log) застосовується для перегляду помилок і звітів про результати виконання різних команд.

Щоб розмістити нового актора у браузері, виконуємо таке:

⇩ Use Case View ⇗ Business Use Case Model ❑ ПКМ
➢ New ➢ Actor // у браузері з'явиться актор NewClass
⇗ NewClass:= *Назва_актора* // введіть назву нового актора
⇗ *Назва_актора* ❑ ПКМ ➢ Open Specification
⇩ Stereotype ⇗ Business Actor ❑ ОК

Щоб розмістити новий прецедент у браузері, виконуємо таке:

⇩ Use Case View ⇗ Business Use Case Model ❑ ПКМ
➢ New ➢ Use Case // у браузері з'явиться прецедент NewUseClass
⇗ NewUseClass:= *Назва_прецеденту* // введіть назву
⇗ *Назва_прецеденту* ❑ ПКМ ➢ Open Specification
⇩ Stereotype ⇗ Business Use Case ❑ ОК

Під час викладу матеріалу опиратимемося на учбовий приклад реєстрації на курси. Попередньо виокремимо таких акторів:

- *Студент* – записується на курси і переглядає табель успішності.

- *Професор* – обирає курси викладання і ставить оцінки за курси.
- *Система розрахунку* – одержує інформацію щодо оплати за курси.
- *Каталог курсів* – база даних, що містить інформацію про курси.

Послідовно розміщуємо у браузері акторів: *Студент*, *Професор*, *Система розрахунку* і *Каталог курсів*.

Аналізуючи предметну область, можна попередньо виокремити такі прецеденти (*business use case*): *зареєструватися на курси*; *переглянути табель успішності*; *обрати курси викладання*; *поставити оцінки*. Послідовно розміщуємо у браузері ці прецеденти.

Для створення нової діаграми прецедентів виконуємо таке:

⇩ Use Case View ⇗ Business Use Case Model ❑ ПКМ

➢ New ➢ Use Case Diagram // отримуємо діаграму NewDiagram

⇗ NewDiagram:= *Початкова схема* // назва діаграми

⇗ *Початкова схема* ❑ ЛКМ (*двічі*) // відкриття діаграми

// Для розміщення актора чи прецеденту на діаграмі перетягніть

// його мишею з браузера на діаграму

❑ Unidirectional Association // проведення асоціацій.

Діаграму прецедентів “Початкова схема” зображено на рис. 3.7.

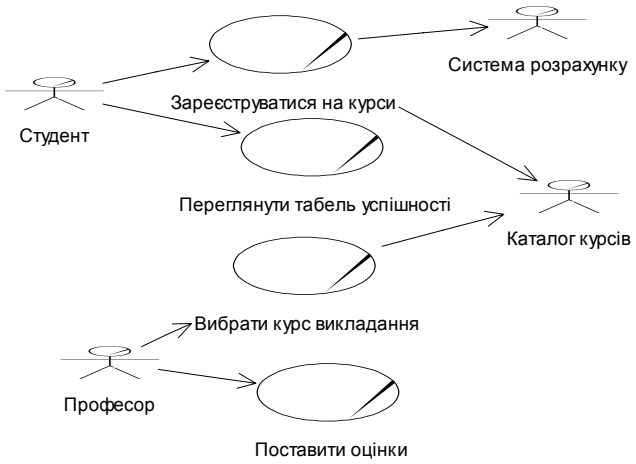


Рис. 3.7. Діаграма прецедентів *Початкова схема*

3.5. Специфікації прецедентів

Специфікації прецедентів, зазвичай, записують окремо від графічних позначень і доповнюють діаграми прецедентів. UML не визначає точний формат специфікацій прецедентів. На практиці специфікації часто представляють собою стислий та інформативний опис поведінки системи, що відповідає прецеденту.

Специфікації прецедентів можуть налічувати *передумови* та *постумови*. Передумови визначають стан, в якому перебуватиме система до виконання прецеденту. Постумови специфікують стан, в який перейде система після виконання прецеденту. Перед- та постумови відображають лише ті аспекти стану, які є важливими для певного прецеденту.

Описуючи специфікації прецедентів, використовуватимемо шаблон, запозичений з регламенту Rational Unified Process:

Заголовок прецеденту
1. Короткий опис
2. Потоки подій
    2.1. Головний потік подій
    2.2. Підлеглі потоки подій
    2.3. Альтернативні потоки подій
3. Спеціальні вимоги
4. Передумови
5. Постумови
6. Додаткові зауваження

У деяких специфікаціях прецедентів окремі пункти можуть не вказуватися. Створимо за допомогою MS Word два документи з описами специфікацій прецедентів "*Зареєструватися на курси*" та "*Обрати курси викладання*".

Прецедент "*Зареєструватися на курси*"

1. Короткий опис. Прецедент дає змогу студентові зареєструватися на доступні курси у поточному семестрі. Студент може змінити свій вибір (обновити чи видалити курси), якщо зміна виконується у встановлений час на початку семестру. Система *каталогу курсів* надає список усіх доступних курсів у поточному семестрі.

2. Потоки подій.

2.1. Головний потік подій.

Цей прецедент розпочне виконуватися, коли студент захоче зареєструватися на конкретні курси чи змінити свій графік курсів:

1) система запитує необхідну дію (створити графік, обновити графік, видалити графік);
2) коли студент зазначає дію, виконується один з підлеглих потоків (створити, обновити, видалити чи прийняти графік).

Якщо реєстрацію на поточний семестр уже завершено, то виконуватиметься потік 2.3.5.

2.2. Підлеглі потоки подій.

2.2.1. Створити графік:

1) система каталогу курсів виконує пошук доступних курсів і виводить їхній список; якщо система каталогу курсів недоступна, то виконується потік 2.3.4;
2) студент обирає зі списку 4 базові і 2 альтернативні курси;
3) після вибору система створює графік студента;
4) виконується підлеглий потік “Прийняти графік”.

2.2.2. Обновити графік:

1) система виводить поточний графік студента; якщо система не може знайти графіка студента, то виконується потік 2.3.3;
2) система каталогу курсів виконує пошук доступних курсів і виводить їхній список; якщо система каталогу курсів недоступна, то виконується потік 2.3.4;
3) студент може обновити свій вибір курсів, видаляючи чи додаючи запропоновані курси;
4) після вибору система обновляє графік;
5) виконується підлеглий потік “Прийняти графік”.

2.2.3. Видалити графік:

1) система виводить поточний графік студента; якщо система не може знайти графіка студента, то виконується потік 2.3.3;
2) система запитує у студента підтвердження видалення графіка.
3) студент підтверджує видалення; якщо студент скасує видалення, то виконується потік 2.3.6:
4) система видаляє графік; якщо графік містить курси, на які записався студент, то студент вилучається зі списків цих курсів.

2.2.4. *Прийняти графік.* Для кожного обраного, проте ще не зафіксованого курсу в графіку система перевіряє виконання таких вимог: проходження студентом попередніх курсів, факт відкриття запропонованого курсу і відсутність конфліктів графіка. Якщо усі вимоги виконано, то система додає студента у список курсу. Інакше виконується потік 2.3.2. Курс фіксується у графіку студента.

2.3. *Альтернативні потоки.*

2.3.1. *Зберегти графік.* У будь-який момент студент може замість прийняття графіка зберегти його. Тоді потік "Прийняти графік" заміняється на такий:

1) "незафіксовані" конкретні курси позначаються у графіку як "обрані";

2) графік зберігається в системі.

2.3.2. *Не виконані попередні вимоги, курс заповнений чи простежуються конфлікти графіка.* Видається повідомлення про помилку. Студент може або вибрати інший пропонований курс і продовжити виконання прецеденту, або зберегти графік, або скасувати операцію (після чого головний потік почнеться з початку).

2.3.3. *Графік не знайдений.* Видається повідомлення про помилку. Після того, як студент підтвердить це повідомлення, головний потік почнеться з початку.

2.3.4. *Система каталогу курсів недоступна.* Видається повідомлення про помилку. Після того, як студент підтвердить це повідомлення, прецедент завершиться.

2.3.5. *Реєстрація на курси завершена.* Видається відповідне повідомлення і прецедент завершується.

2.3.6. *Видалення скасоване.* Головний потік почнеться з початку.

3. Спеціальні вимоги. Відсутні.

4. Передумова. Перед виконанням прецеденту студент має увійти в систему.

5. Постумова. Якщо прецедент завершиться успішно, графік студента буде створений, обновлений чи вилучений. Інакше стан системи не зміниться.

Прецедент "*Обрати курси викладання*"

1. Короткий опис. Прецедент надає професору можливість вибору до чотирьох курсів, які викладатимуть у визначеному семестрі.

2. Потоки подій.

2.1. Головний потік подій.

Цей прецедент виконується, коли професор вибирає курси для викладання:

1) професор задає поточний або майбутній семестр; за неправильного задання семестру виконується потік 2.3.1;
2) система запитує необхідну дію (додавання курсу, видалення, перегляд курсів, роздрук курсів і вихід);
3) коли професор зазначає дію, виконується відповідний підлеглий потік.

2.2. Підлеглі потоки подій.

2.2.1. Додати курс:

1) система відображає вікно з полями назви і номера курсу;
2) професор вводить назву і номер курсу; у випадку задання невірної інформації виконується потік 2.3.2;
3) система відтворює текст із пропозицією на ведення курсу і професор підтверджує свій вибір; якщо найменування курсу не відображено, то виконується потік 2.3.3;
4) система зв'язує поточний суб'єкт *Професор* із зазначеним курсом; якщо зв'язок не можна створити, виконується потік 2.3.4.

2.2.2. Видалити курс:

1) система відображає список з назвами і номерами раніше запропонованих курсів;
2) професор вводить назву чи номер курсу; у випадку задання невірної інформації виконується потік 2.3.2;
3) система видаляє зв'язок між курсом і суб'єктом *Професор*; якщо зв'язок видалити не можна, виконується потік 2.3.5.

2.2.3. Перегляд курсів:

1) система відображає інформацію про всі пропозиції на ведення курсів, поданих поточним суб'єктом *Професор*; якщо дані одержати не можна, виконується потік 2.3.6.
2) після завершення перегляду прецедент активізується з початку.

2.2.4. *Роздрукувати курси*:

1) система посилає на принтер розклад занять суб'єкта *Професор*; якщо не можна друкувати, виконується потік 2.3.7;

2) після завершення перегляду прецедент активізується з початку.

2.2.5. *Вихід.* Виконання функцій, передбачених прецедентом, завершується.

2.3. *Альтернативні потоки.*

2.3.1. *Неправильний семестр.* Видається відповідне повідомлення про невірне введення даних про семестр. Система пропонує повторити операцію чи завершити прецедент.

2.3.2. *Невірна комбінація найменування і номера курсу.* Видається відповідне повідомлення про невірне введення даних про найменування і номер курсу і пропонує повторити операцію чи прецедент.

2.3.3. *Помилка відтворення тексту з пропозицією на ведення курсу.* Видається повідомлення про те, що в даний момент запитаний курс недоступний. Прецедент активізується з початку.

2.3.4. *Помилка створення зв'язку.* Інформація збережена, система створить зв'язок пізніше; прецедент активізується з початку.

2.3.5. *Помилка видалення зв'язку.* Інформацію збережено, система видалить зв'язок пізніше; прецедент активізується спочатку.

2.3.6. Помилка витягу даних про пропозиції на ведення курсів. Видається повідомлення про те, що в даний момент інформація недоступна; прецедент активізується з початку.

2.3.7. *Помилка роздруку розкладу занять.* Видається повідомлення про те, що в даний момент функція недоступна; прецедент активізується з початку.

3. Спеціальні вимоги. Відсутні.

4. Передумова. Перед виконанням прецеденту професор має увійти в систему.

5. Постумова. Якщо прецедент завершиться успішно, розклад буде створено, обновлено чи вилучено. Інакше стан системи не зміниться.

Для прикріплення до прецеденту файла з його специфікацією виконуємо таке:

⇩ Use Case View ⇗ Business Use Case Model
⇗ *Назва_прецеденту* ❑ ПКМ ➢ Open Specification
🗀 File ⇗ ❑ ПКМ // на білому фоні
➢ Insert File // через браузер файлів знаходимо необхідний файл
❑ Open.

Проаналізувавши детальніше головні прецеденти "*Зареєструватися на курси*" та "*Обрати курси викладання*" можна зробити такі висновки:

- У системі необхідно зберігати дані про студентів і професорів.
- За роботу системи повинен відповідати працівник деканату (*реєстратор*), який формуватиме навчальний план і каталог курсів, вестиме усі дані про курси, професорів і студентів.
- Вхід у систему користувачеві необхідно здійснювати за допомогою пароля.
- Оскільки вхід у систему цілком однаковий для реєстратора, студента і професора, їхню поведінку можна узагальнити і ввести нового актора "*Користувач*" (супертип).
- У модель необхідно ввести прецедент "*Увійти в систему*", який зв'язується з актором "*Користувач*".
- У модель необхідно ввести прецедент "*Підтримувати дані про професорів*", який зв'язується з актором "*Реєстратор*".
- У модель необхідно ввести прецедент "*Підтримувати дані про студентів*", який зв'язується з актором "*Реєстратор*".
- У модель необхідно ввести прецедент "*Закрити реєстрацію*", який зв'язується з актором "*Реєстратор*".

Нову діаграму прецедентів *Уточнена схема* (уточнення діаграми *Початкова схема*) зображено на рис. 3.8. Наведемо специфікацію прецедентів "*Увійти в систему*" і "*Закрити реєстрацію*".

Прецедент "*Увійти в систему*"

1. Короткий опис. Описує вхід користувача у систему реєстрації.

2. Потоки подій.

2.1. Головний потік подій:

1) система запитує ім'я користувача і пароль;

2) користувач вводить ім’я і пароль;

3) система підтверджує ім’я та пароль і відкриває доступ; якщо допущена помилка, виконується потік 2.3.1.

2.3. Альтернативні потоки.

2.3.1. Неправильне ім’я чи пароль. Система виводить повідомлення про помилку. Користувач може повернутися до початку головного потоку чи відмовитися від входу в систему, у цьому випадку прецедент завершується.

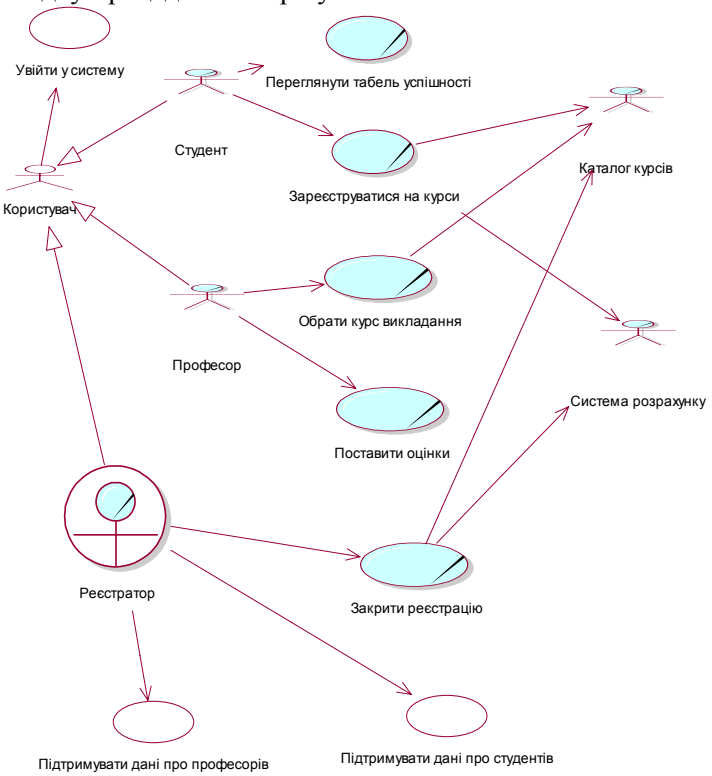


Рис. 3.8. Діаграма прецедентів *Уточнена схема*

Прецедент “*Закрити реєстрацію*”

1. Короткий опис. Прецедент дає змогу реєстраторові закривати процес реєстрації. Пропоновані курси, на які не записалося достат-

ньої кількості студентів (менше трьох), скасовуються. У систему розрахунку передається інформація про кожного студента з кожного доступного курсу, щоб студенти внесли оплату за курси.

2. Потоки подій.

2.1. Головний потік подій. Такий прецедент розпочне виконуватися, коли реєстратор запитує припинення реєстрації:

1) система перевіряє стан реєстрації; якщо реєстрація ще виконується, видається повідомлення і прецедент завершується;
2) для кожного доступного курсу система перевіряє, чи читає його будь-який з професорів і чи записалося на нього не менше трьох студентів; якщо ці умови виконуються, система фіксує доступний курс у кожному графіку, що включає даний курс (у протилежному випадку виконується потік 2.3.1);
3) для кожного студентського графіка перевіряється наявність у ньому максимальної кількості головних курсів; якщо їх недостатньо, система намагається доповнити альтернативними курсами зі списку даного графіка (обирається перший доступний альтернативний курс); якщо таких курсів немає, то жодних доповнень не відбувається;
4) система закриває усі доступні курси;
5) система розраховує плату за навчання для кожного студента в поточному семестрі і скеровує інформацію в систему розрахунку, яка розсилає студентам рахунки для оплати з копією їхніх остаточних графіків; якщо неможливо встановити зв'язок з системою розрахунку, виконується потік 2.3.2.

2.3. Альтернативні потоки.

2.3.1. Курс ніхто не читає і/або записалося мало студентів. Курс скасовується. Система вилучає такий курс із кожного графіка.

2.3.2. Система розрахунку недоступна. Через деякий установлений час система знову намагатиметься зв'язатися з нею; спроби будуть повторюватися доти, доки зв'язок не встановиться.

3. Спеціальні вимоги. Відсутні.

4. Передумова. Перед виконанням прецеденту реєстратор має увійти в систему.

5. Постумова. Якщо прецедент завершиться успішно, реєстрація закривається. В іншому випадку стан системи не зміниться.

3.6. Діаграми діяльності

При моделюванні поведінки системи виникає необхідність деталізувати особливості алгоритмічної та логічної реалізації операцій. Традиційно з цією метою використовують блок-схеми або структурні схеми алгоритмів. Кожна така схема акцентує увагу на *послідовності виконання певних дій* (або елементарних операцій), які в сукупності спричинюють до отримання бажаного результату.

Для моделювання процесу виконання операцій у мові UML використовують *діаграму діяльності*, яка зображається графом, вершинами якого є стани (дій і/або видів діяльності), а дугами – переходи від одного стану (дій/виду діяльності) до іншого стану (дій/виду діяльності).

Головним напрямом використання діаграм діяльності є візуалізація особливостей реалізації операцій класів, коли необхідно зобразити алгоритми їхнього виконання. При цьому кожен стан дії може бути виконанням операції деякого класу, а вид діяльності – послідовністю таких дій.

На етапі моделювання прецедентів *діаграми діяльності* зображають потоки функцій керування, зазначають, які гілки процесу керування можуть виконуватися паралельно, і визначають альтернативні шляхи досягнення мети. Потоки керування можуть містити декілька прецедентів або протікати у рамках одного прецеденту.

У контексті мови UML *вид діяльності* (activity) є деякою *сукупністю* окремих елементарних обчислень. Щодо цього окремі елементарні обчислення можуть спричинювати до деякого результату або дії (action). На діаграмі діяльності відображається логіка або послідовність переходу від однієї діяльності до іншої, у цьому випадку увага фіксується на результаті діяльності, який може спричинити зміну стану системи або повернення деякого значення.
Примітка. Хоча діаграму діяльності призначено для моделювання поведінки систем, час у явному вигляді відсутній на цій діаграмі.

Елементи діаграми діяльності: стани *видів діяльності* (activity), стани *дій* (action state, чи state), *переходи* (transition), точки прийняття рішення або *галуження* (decision), вертикальна/горизонтальна смуги *синхронізації* (synchronization) і *доріжки* (swimlanes).

Після того, як діаграму діяльності буде активовано, панель інструментів набуде такого вигляду (рис. 3.9).

Рис. 3.9. Панель інструментів діаграми діяльності

Наведемо детальні характеристики *виду діяльності*:

- послідовність дій, яка виконується об'єктом регулярно;
- неатомарний (тобто його можна перервати);
- можна розбити на декілька дрібніших видів діяльності;
- записують на природній мові.

Дія є атомарною, тобто її перервати не можна. Стан дії не може мати внутрішніх переходів, оскільки він є елементарним. Звичне використання станів дії полягає в моделюванні одного кроку виконання алгоритму (процедури) або потоку керування. Водночас вид діяльності моделює блок, процедуру чи потік керування як єдине ціле.

Дія може бути записана на природній мові, на деякому псевдокоді або мові програмування. Жодних додаткових або неявних обмежень під час запису дій не накладається. Рекомендується для назви простої дії використовувати дієслово зі словами пояснень. Якщо ж дію можна представити в деякому формальному вигляді, то доцільно записати її на тій мові програмування, на якій передбачено реалізацію конкретного проекту.

У мові UML вид діяльності та стан дії позначають однаково: прямокутником з вигнутими назовні бічними сторонами (для виду діяльності ця випуклість є більшою порівняно з дією). Різниця між видом діяльності та станом дії є очевидною за контекстом.

Кожна діаграма видів діяльності повинна мати єдиний початковий (start state) і єдиний кінцевий стан (end state). Саму діаграму діяльності прийнято розташовувати так, щоб види діяльності/дії слідували зверху вниз. У цьому випадку початковий стан зображатиметься у верхній частині діаграми, а кінцевий – в її нижній частині.

Під час побудови діаграми діяльності використовують *переходи*, які спрацьовують відразу після завершення діяльності або

виконання відповідної дії. Цей перехід переводить процес діяльності в наступний стан відразу, як тільки закінчиться дія у попередньому стані. На діаграмі перехід зображають лінією зі стрілкою.

Якщо зі стану дії виходить єдиний перехід, то його можна не позначати. Якщо ж таких переходів декілька, то спрацювати може тільки один з них. У цьому випадку для кожного з таких переходів необхідно записати *сторожову умову* в квадратних дужках.

Подібний випадок трапляється тоді, коли послідовно виконувана діяльність повинна розділитися на альтернативні гілки залежно від значення деякого проміжного результату. Таку ситуацію називають *галуженням*.

Графічно галуження на діаграмі діяльності позначається невеликим ромбом, усередині якого немає жодного тексту. В цей ромб може входити тільки одна стрілка від того стану дії, після виконання якого потік керування повинен бути продовжений однією з гілок, що взаємно виключаються. Прийнято вхідну стрілку приєднувати до верхньої або лівої вершини символу галуження. Стрілок, що виходять, може бути дві або більше, але для кожної з них явно зазначається відповідна сторожова умова у формі логічного виразу. Приклад галуження наведено на рис. 3.10.

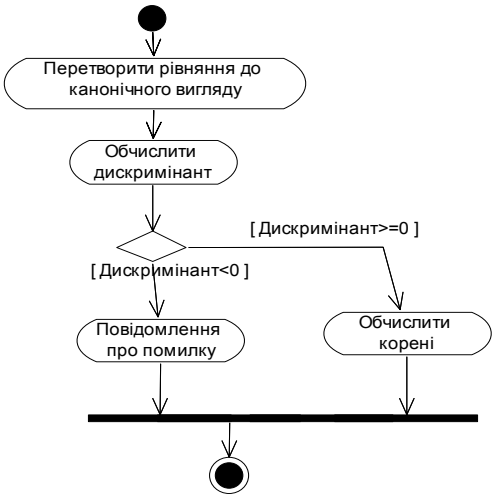


Рис. 3.10. Діаграма видів діяльності розв’язування квадратного рівняння

Станами діаграми діяльності на рис. 3.10 є тільки види діяльності. На рис. 3.11 діаграма містить стани дій і видів діяльності.

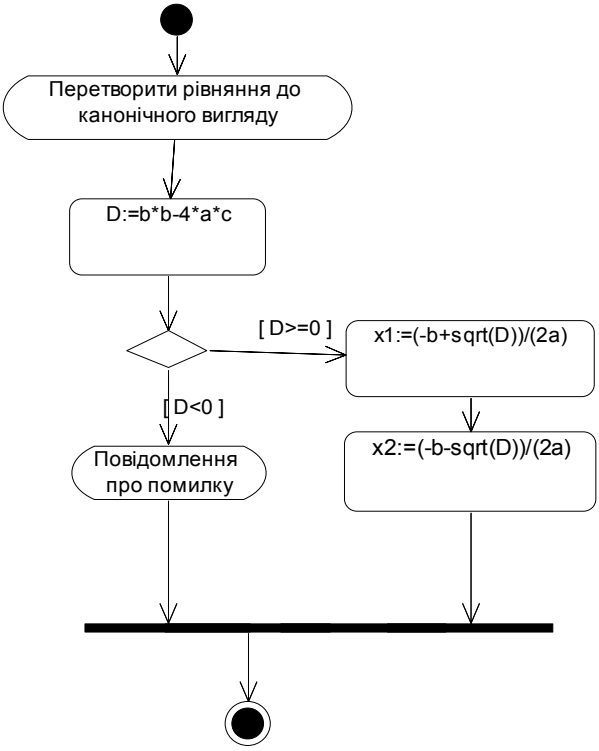


Рис. 3.11. Детальна діаграма видів діяльності розв'язування квадратного рівняння

Для відображення на діаграмах діяльності моментів розділення і злиття паралельних обчислень чи потоків керування використовують горизонтальну/вертикальну товсту лінію (смугу *синхронізації*). На рис. 3.10 і 3.11 горизонтальну смугу синхронізації використовують для злиття паралельних обчислень.

Діаграми діяльності можна використовувати не тільки для специфікації алгоритмів обчислень чи потоків керування у програмних системах. Не менш важливою є область їхнього застосування, що зв'язана з моделюванням бізнес-процесів.

Дійсно, діяльність будь-якої компанії (фірми) є не що інше, як сукупність окремих дій, спрямованих на досягнення необхідного результату. Проте стосовно бізнес-процесів бажано виконання кожної дії асоціювати з конкретним підрозділом компанії чи виконавцем. У цьому випадку підрозділ чи окремий виконавець несе відповідальність за реалізацію окремих дій, а сам бізнес-процес зображається як перехід дій з одного підрозділу/виконавця до іншого.

Для моделювання цих особливостей у мові UML використовують спеціальну конструкцію – *доріжку* (swimlanes). Щодо цього всі дії на діаграмі діяльності діляться на окремі групи, які відокремлено одну від одної вертикальними лініями. Дві сусідні лінії утворюють доріжку, а група станів між цими лініями виконується окремим підрозділом (відділом, групою, філіалом) компанії.

Назви підрозділів явно зазначаються у верхній частині доріжки. Перетинати лінію доріжки можуть тільки переходи, які в цьому випадку позначають вихід або вхід потоку керування у відповідний підрозділ компанії. Порядок проходження доріжок не несе жодної семантичної інформації і визначається міркуваннями зручності.

Створимо діаграму видів діяльності "*Створення каталогу курсів*". Спочатку перейменуємо назву "*Business Use Case Model*" на осмислену назву "*Курси*":

⇩ Use Case View ⇗ Business Use Case Model ❑ ПКМ

➢ Rename // вводимо назву *Курси*

Створюємо діаграму видів діяльності:

⇩ Use Case View ⇗ *Курси* ❑ ПКМ

➢ New ➢ Activity Diagram // отримуємо діаграму NewDiagram
// і перейменовуємо її на *Створення каталогу курсів*

⇗ *Створення каталогу курсів* ❑ ЛКМ (двічі)

// Відкриваємо діаграму *Створення каталогу курсів*

Створюємо дві доріжки з назвами "*Реєстратор*" і "*Професор*":

❑ Swimlane ❑ ЛКМ // на діаграмі і змінюємо назву

Використовуючи техніку переміщення, розташуємо у межах доріжок необхідні елементи діаграми (рис. 3.12). Для приведення лінії до ортогонального вигляду треба: ⇗ *лінія* ➢ Format ➢ Line Style ⇗ Rectilinear.

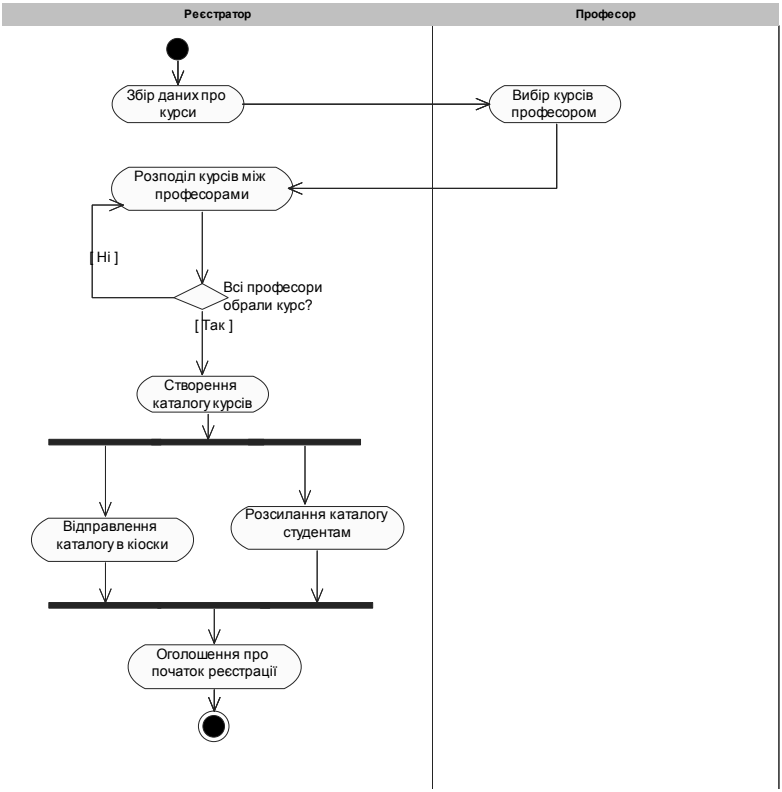


Рис. 3.12. Діаграма видів діяльності *Створення каталогу курсів*

3.7. Попередній архітектурний аналіз системи

Уже на початкових етапах створення програмної системи необхідно зробити попередній архітектурний аналіз цієї системи. Архітектурний аналіз виконується архітектором системи і містить:

- затвердження загальних стандартів (угод) моделювання і документування системи;
- попереднє виявлення архітектурних механізмів;
- формування набору головних абстракцій предметної області (класів аналізу);
- формування початкового зображення архітектурних рівнів.

Угоди моделювання визначають:

- елементи і діаграми, які використовуються у моделі;
- правила їхнього застосування;
- угоди з найменування елементів моделі;
- організацію моделі (пакети).

Приклад набору угод моделювання:

- Назви прецедентів мають бути короткими дієслівними фразами.
- Назви класів мають бути іменниками, що певним чином відповідають поняттям предметної області.
- Назви класів повинні розпочинатися з великої літери.
- Назви атрибутів і операцій мають починатися з малої літери.
- Складені назви мають бути суцільними, без підкреслень, кожне окреме слово має розпочинатися з великої літери.
- Усі класи й діаграми, що описують попередній системний проект, мають міститися у пакеті з назвою *Analysis Model*.
- Діаграми класів, що реалізують прецедент, і діаграми взаємодії (відображають взаємодію об'єктів у процесі реалізації сценаріїв прецедентів) мають міститися у кооперації з назвою даного прецеденту і стереотипом "*use-case realization*".
- Кожну кооперацію оформляють окремим пакетом.
- Усі пакети кооперації мають міститися у пакеті з назвою *Use Case Realizations*.
- Зв'язок між прецедентом і його реалізацією зображається на спеціальній діаграмі прецедентів *Traceabilities* (рис. 3.13).

Створення пакетів і діаграми Traceabilities:

1. ⇩ Logical View // розкриваємо вузол дерева
2. ⇗ Design Model {❑ ПКМ ➢ New ➢ Package // при потребі}
// Створіть або відкрийте пакет Use-Case Realizations, а потім усе-
// редині нього створіть пакети: Use-Case Realization-Login
// Use-Case Realization-Close Registration
// Use-Case Realization-Register for Courses
3. У кожному з пакетів, які належать до пакета Use-Case Realization, створіть кооперації Close Registration, Login і Register for

Courses (кожна кооперація – прецедент зі стереотипом "*use-case realization*", який задається у його специфікації).

4. Створіть у пакеті Use-Case Realizations нову діаграму прецедентів з назвою Traceabilities відповідно до рис. 3.13.

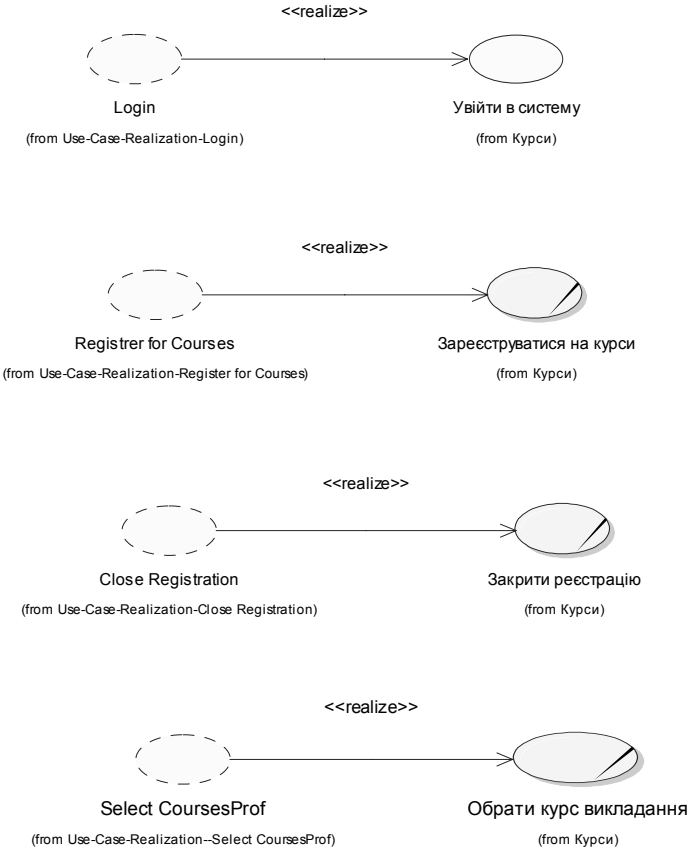


Рис. 3.13. Зв'язок між прецедентами та їхніми реалізаціями

Ідентифікація головних абстракцій полягає у попередньому визначенні набору класів системи (класів аналізу) на основі опису предметної області і специфікації вимог до системи. Наприклад, для системи реєстрації на курси можна попередньо ідентифікувати п'ять класів аналізу: *Student* (Студент); *Professor* (Професор); *Sche-*

dule (Навчальний графік студента); *Course* (Курс); *CourseOffering* (Доступний курс).

Архітектурні рівні моделі зберігаються у пакеті *Design Model* у вигляді пакетів зі стереотипом “layer”. Кількість і структура рівнів залежать від складності предметної області і середовища.

У рамках попереднього архітектурного аналізу визначається початкова структура моделі (набір пакетів та їхніх залежностей) і розглядаються тільки верхні рівні (*Application* і *Business Services*).

? Запитання для самоперевірки

1. Дайте визначення актора.
2. Дайте визначення прецеденту.
3. Які Ви знаєте синоніми терміна “прецедент”?
4. Для чого використовують діаграму прецедентів?
5. Які відношення застосовують на діаграмі прецедентів?
6. Що таке моделювання контексту системи?
7. Що таке моделювання вимог до системи?
8. Для чого використовують браузер Rational Rose?
9. Для чого використовують вікно документації Rational Rose?
10. Для чого використовують вікно діаграми Rational Rose?
11. Для чого використовують журнал Rational Rose?
12. Як розмістити нового актора на діаграмі прецедентів?
13. Як розмістити новий прецедент на діаграмі прецедентів?
14. Що таке специфікація прецеденту?
15. Опишіть шаблон специфікації прецеденту.
16. Для чого використовують діаграму видів діяльності?
17. Дайте визначення стану дії.
18. Дайте визначення стану виду діяльності.
19. Дайте визначення переходу.
20. Дайте визначення точки галуження.
21. Дайте визначення смуги синхронізації.
22. Дайте визначення зони відповідальності.
23. Що містить попередній архітектурний аналіз системи?
24. Що визначають угоди моделювання?
25. Як утворити пакет?
26. Як утворити кооперацію реалізації прецеденту?

4. МОДЕЛЮВАННЯ КЛАСІВ

План викладу матеріалу:

1. Зображення класу.
2. Асоціації між класами.
3. Агрегація та композиція між класами.
4. Узагальнення та залежності між класами.
5. Розширення UML для моделей класів програмування і бізнесу.
6. Моделювання класів у Rational Rose.

Ключові терміни розділу

- *Класи*
- *Зображення класу*
- *Атрибути класу*
- *Операції класу*
- *Асоціації на діаграмах класів*
- *Багатополюсна асоціація*
- *Узагальнення між класами*
- *Клас керування*
- *Межовий клас*
- *Екземпляри класів* (*об'єкти*)
- *Кратність класу*
- *Опис атрибута класу*
- *Синтаксис операції*
- *Асоційований клас*
- *Агрегація та композиція*
- *Залежності між класами*
- *Клас-сутність*
- *Інтерфейс*

4.1. Зображення класу

Класи та їхні екземпляри (*об'єкти*) утворюють фундамент, на який опирається об'єктно-орієнтований підхід до проектування та розробки програмного забезпечення.

Класи – головні сутності при моделюванні програмних систем за допомогою UML. Моделювання класів дає змогу побудувати та представити систему в термінах об'єктно-орієнтованої концепції. Таке представлення системи має важливі переваги порівняно з іншими моделями. По-перше, модель класів відображає предметну область задачі, а, отже, є зрозумілою спеціалісту в цій предметній області, що дає змогу залучати до проектування системи вузькопрофесійних спеціалістів. По-друге, модель класів тривіально перетворюється в реалізацію на об'єктно-орієнтованих мовах з повним збереженням семантики моделі.

Моделі класів в UML можуть зображатися різними типами діаграм, кожен з яких відображає ті чи інші аспекти моделі і, від-

повідно, організації системи. Найчастіше моделі класів зображають за допомогою *діаграм класів* (*Class Diagram*).

Клас – головний елемент на діаграмах класів, що відображає сукупність *однотипних* об'єктів зі спільними атрибутами, операціями та семантикою. Зображення класу – прямокутник (див. § 2.2).

У зображенні класу виокремлюють три секції, в яких послідовно записують *назву*, *атрибути* та *операції*. Кожну секцію, крім першої, можна опустити. Назва класу має бути унікальною в межах пакета, в якому розташовано клас. Назви *абстрактних* класів в UML позначають курсивом.

Зазвичай, клас може налічувати довільну кількість екземплярів. Однак виникають ситуації, за яких число екземплярів класу необхідно обмежити. Найчастіше треба задавати клас, у якого:

- немає жодного екземпляра – службовий клас (*Utility*);
- рівно один екземпляр – синглетний клас (*Singleton*);
- строго визначена кількість екземплярів;
- довільна кількість екземплярів – варіант за домовленістю.

Кількість екземплярів класу називають *кратністю* класу (*multiplicity*), яку задають у специфікації класу. Кратність класу, залежно від інструментального засобу, може відображатися за допомогою спеціальної позначки (малий пунктирний квадратик) або числа у правому верхньому куті прямокутника, який зображає клас.

Атрибут – це позначене місце (*слот*), в якому може/можуть зберігатися значення. Загальний формат атрибутів такий:

```
[<видимість>] <назва> [<кратність>] [:<тип>]
   [=<початкове значення>][<властивості>]
```

Видимість атрибута визначає рівень доступу до атрибута і найчастіше набуває таких значень[1]:

- *відкритий* (*public*) – передбачає необмежений доступ до атрибута з боку інших класів і позначається символом "+";
- *захищений* (*protected*) – передбачає доступ до атрибута лише класам, які його наслідують, і позначається символом "#";

[1] Символи (або піктограми) позначення видимості в конкретних інструментальних засобах можуть відрізнятися від наведених

- *закритий* (*private*) – забороняє доступ до атрибута всім іншим класам і позначається символом "–".

Назва атрибута має бути унікальною для певного класу. *Тип* атрибута бажано задавати відповідно до типів даних у мові програмування, на якій реалізовуватимуть модель. *Кратність* визначає *атрибут-масив* для зберігання множини значень (здебільшого, кратність обмежується квадратними дужками). Для атрибута також може бути вказано його *початкове значення*.

Можна задавати такі *властивості* атрибута:

- `changeable` (змінюваний) – за домовленістю;
- `addOnly` (тільки долучення) – можна долучати нові значення для *атрибута-масиву*, без можливості наступних змін;
- `frozen` (заморожений) – після ініціалізації значення атрибута не змінюється (відповідає слову `const` у C++).

Статичні атрибути (або *атрибути класу*) на відміну від звичайних атрибутів належать не окремим об'єктам, а класу загалом, і позначаються на діаграмах класів підкресленням.

На рис. 4.1 зображено клас **`Customer`**, який має чотири атрибути: два загальні атрибути `name` і `birthDate`, закритий атрибут `password` і статичний захищений атрибут `count` з початковим значенням 1.

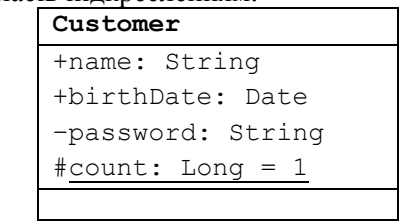


Рис. 4.1. Приклад класу **`Customer`**

Операції дають змогу закласти в клас та його об'єкти деяку поведінку. Загальний формат запису операцій наступний:

`[<видимість>]<назва>([список параметрів])[:<тип результату>]`

Видимість операції визначає рівень доступу до операції і позначається аналогічно до видимості атрибутів. *Назва* операції має бути унікальною для певного класу. *Тип результату* визначає тип значення, яке повертає операція, і повинен відповідати типам даних у мові програмування, яку обрано для реалізації моделі.

Список параметрів дає змогу специфікувати інтерфейс операції. Кількість параметрів у списку може бути довільною.

Кожен параметр у загальному випадку записують так:

[<напрям>] <назва> [:<тип>][= <значення за домовленістю>]

Напрям визначає характер використання параметра в операції і найчастіше набуває таких значень:

- `in` – параметр використовується як *вхідний*;
- `out` – параметр використовується як *вихідний*;
- `inout` – параметр використовується як вхідний і вихідний одночасно.

Назва параметра має бути унікальною для певної операції. *Тип* параметра варто визначати з огляду на типи даних в обраній мові програмування. *Значення за домовленістю* відображає значення параметра, яке буде прийняте в операції, якщо цей параметр буде опущено.

Приклади різних форм запису операцій наведено на рис. 4.2.

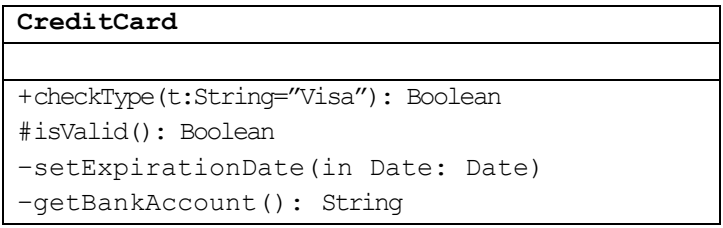


Рис. 4.2. Операції класу **CreditCard**

Абстрактні операції позначають на діаграмах класів курсивом. Якщо клас має хоча б одну абстрактну операцію, то його вважають абстрактним.

Статичні операції (або *операції класу*) належать цілому класу, працюють зі статичними атрибутами класу і позначаються на діаграмах підкресленням.

Відповідальності класу визначають його функції та роль в системі і формуються на початкових стадіях моделювання. Зазвичай відповідальності класів зазначають на загальних діаграмах, які містять лише концептуальні класи системи, і слугують для створення початкового уявлення про розподіл функціональності за класами. Кожен клас може мати одну або більше відповідальностей.

На рис. 4.3 зображено клас **Document**, який налічує чотири секції: *назви*, *атрибути*, *операції* та *відповідальності* класу.

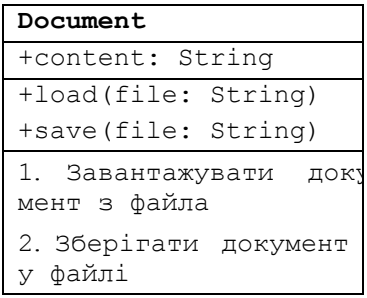


Рис. 4.3. Приклад класу **Document**

На діаграмі кожна відповідальність представляється одним або кількома чітко сформульованими реченнями. Існує практика виносити відповідальності класу за межі класу і зазначати їх у примітках, асоційованих з класом.

4.2. Асоціації між класами

Відношення *асоціації* на діаграмах класів трапляються найчастіше. Асоціацію позначають суцільною лінією (може завершуватися однією чи двома стрілками), що з'єднує класи. Асоціація означає, що екземпляри одного класу взаємодіють з екземплярами іншого класу під час виконання програми. Оскільки екземплярів класу може бути багато і кожен може взаємодіяти з декількома екземплярами іншого класу, то асоціація є *дескриптором*, що описує множину зв'язаних об'єктів (екземплярів асоціації).

Зв'язок між об'єктами у програмі може бути організований довільними способами. Можливість зв'язку означає, що об'єкт одного класу може надіслати повідомлення об'єкту іншого класу, зокрема, активізувати операцію або прочитати/змінити значення відкритого атрибута. Оскільки в об'єктно-орієнтованій програмі такі дії складають суть виконання програми, то виявлення асоціацій є однією з ключових задач під час її складання.

Базова нотація асоціації (суцільна лінія) засвідчує, що об'єкти асоційованих класів (*полюси асоціації*) взаємодіють між собою під час виконання програми. Для асоціацій в UML передбачено чимало *доповнень*. Доповнення не є обов'язковими: їх використовують за необхідності, у різних ситуаціях по-різному. Якщо викорис-

товувати усі доповнення одразу, то діаграма класів стає перевантаженою настільки, що її важко читати і розуміти. Розглянемо найважливіші доповнення:

- *Назва асоціації* (зазначається посередині над/під лінією) – ідентифікатор, який позначає асоціацію у моделі. Додаткового семантичного навантаження ця назва не несе, отож, зазвичай, не використовується (за винятком випадку, коли два класи зв'язані декількома асоціаціями).
- *Роль полюса асоціації* (або *специфікатор інтерфейсу*) – конкретизує асоціацію щодо певного класу. Роль зазначається поблизу лінії неподалік класу, набуває вигляду:

  `[<видимість>] <назва ролі> [: <тип>]`

  Використовується для ідентифікації полюсів у випадку самоасоціації (клас зв'язується сам з собою) або у випадку, коли два класи зв'язані декількома різними асоціаціями.
- *Кратність полюса асоціації* – визначає кількість об'єктів певного класу (з боку полюса), що беруть участь у зв'язку під час виконання програми. Кратність може задаватися конкретним числом, і тоді у кожному зв'язку з боку цього полюса беруть участь рівно стільки об'єктів, скільки задано. Найчастіше кратність задано у вигляді діапазону можливих значень:

  `<нижня_межа> .. <верхня_ межа>`

  Кратність полюса або верхню межу може зображати символ "*", який позначає довільну кількість екземплярів класу. Приклади допустимих діапазонів: 1..5, 1..*, *, 0..4.
- *Сортування на полюсі асоціації* – визначає тип упорядкування об'єктів на полюсі з кратністю понад 1. Можливі значення: *упорядковано* (`ordered` або `sorted`), *неупорядковано* (`unordered` або `unsorted`). За домовленістю – *неупорядковано*.
- *Змінюваність на полюсі асоціації* (`Is changeable`) – визначає `можливість/неможливість` зміни кількості екземплярів класу з боку полюса під час роботи програми (у межах специфікованої кратності).
- *Напрям навігації на полюсі асоціації* (`Is navigable`) – визначає `можливість/неможливість` доступу до екземплярів класу з

боку полюса за допомогою певної асоціації. Ті полюси, через які навігація можлива, на діаграмах позначають стрілками.

У деяких CASE-засобах відсутність стрілок означає можливість навігації у будь-якому напрямку.

Найчастіше використовують *бінарні асоціації* (*Binary Association*) – відношення між двома класами або рефлексивне відношення класу із самим собою. На рис. 4.4 зображено бінарну асоціацію з назвою *Містить* типу "один до багатьох" між класами *Авіарейс* і *Місце* (у літаку).

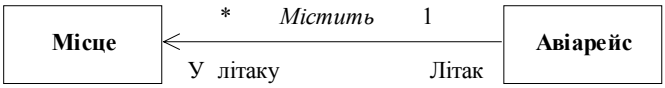


Рис. 4.4. Приклад бінарної асоціації між класами

Можлива ситуація, коли один екземпляр класу асоціюється з декількома екземплярами того ж самого класу. Нехай один екземпляр класу *ЧлениКафедри* представляє завідувача кафедри, а інші екземпляри цього класу – викладачів. Завідувач кафедри *керує* викладачами. Зв'язок, який виникає між екземплярами одного і того ж класу, називають *самоасоціацією* (рис. 4.5).

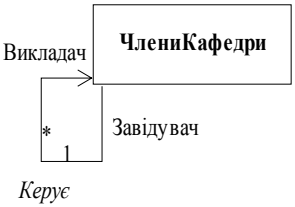


Рис. 4.5. Приклад самоасоціації

Здебільшого асоціація між класами *A* і *B* – це множина пар (*a*; *b*), де *a* – об'єкт класу *A*, а *b* – об'єкт класу *B*. Ці пари є *екземплярами* асоціації (зв'язками). Отже, асоціація є *класом*, який ще може володіти іншими додатковими властивостями, які залежать від асоціації.

Для визначення цих властивостей асоціації використовують *асоційований клас* (*associaton class*), який зображається, зазвичай, прямокутником і зв'язується з відповідною асоціацією штриховою лінією. Назви асоціації та асоційованого класу *мають збігатися*. Розглянемо, наприклад, постачання деталей для виготовлення деякого виробу (рис. 4.6).

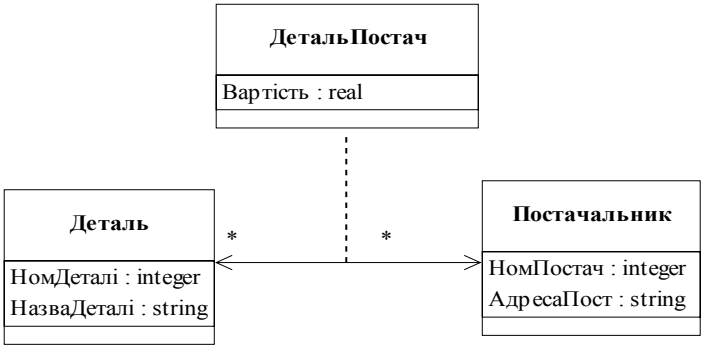


Рис. 4.6. Приклад асоційованого класу

Одну деталь можуть постачати різні постачальники, а один постачальник може постачати різні деталі. *Вартість* деталі залежить від асоціації (від конкретної деталі та конкретного постачальника) і може бути тільки атрибутом *асоційованого* класу.

Багатополюсна асоціація (*N-ary Association*) – це зв'язок між трьома та більше класами. Такий зв'язок графічно зображають ромбом, до якого підходять лінії, що з'єднують ромб із класами. Клас, що описує зв'язок, приєднується до ромба за допомогою штрихової лінії. Приклад такої асоціації зображено на рис. 4.7.

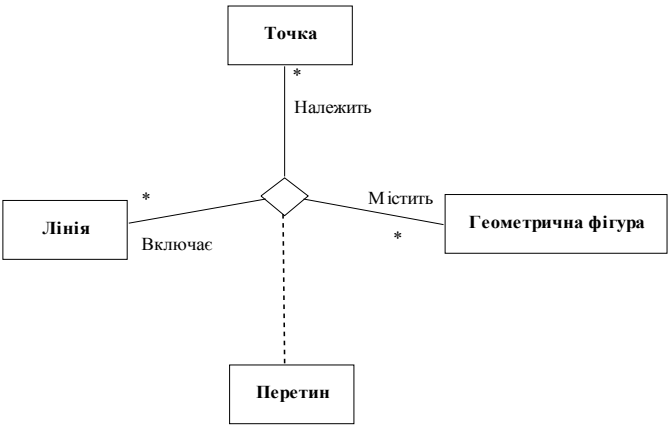


Рис. 4.7. Приклад багатополюсної асоціації

4.3. Агрегація та композиція між класами

У мові UML використовують два часткові й дуже важливі випадки відношення асоціації – *агрегацію* та *композицію*. В обох випадках йдеться про моделювання відношення типу “частина – ціле”. Відношення такого типу є відношеннями асоціації, оскільки частини і ціле, зазвичай, взаємодіють між собою.

Агрегація (*Aggregation*) від класу *A* до класу *B* означає, що об’єкти (один чи декілька) класу *A* входять до складу об’єкта класу *B*. На діаграмі класу це відзначається за допомогою спеціального графічного *доповнення*: на полюсі асоціації з боку “цілого” (у нашому випадку клас *B*) зображається порожній ромбик.

При агрегації жодних додаткових обмежень не накладають: об’єкт класу *A* (“частина”) може бути зв’язаний відношеннями агрегації з іншими об’єктами (тобто брати участь у декількох агрегаціях), може створюватися і знищуватися незалежно від об’єкта класу *B* (“цілого”). На рис. 4.8 зображено агрегацію між класом *Рисунок* (“ціле”) і класом *Фігури* (“частина”).

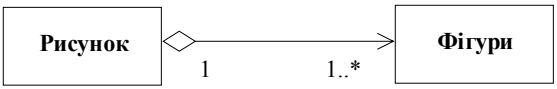


Рис. 4.8. Приклад агрегації

Композиція (*Composition*) є посиленою формою агрегації і створюється на основі бінарної асоціації. *Композиція* накладає дещо сильніші обмеження: композиційно “частина” може входити тільки в одне “ціле”, “частина” існує доти, доки існує “ціле”, і припиняє своє існування разом з “цілим”. Графічно відношення композиції зображають зафарбованим ромбиком.

“Ціле” у композиції слугує *класом-власником*, а “частина” – підпорядкованим класом. Підпорядковані класи, що належать до композиції, не можуть водночас налічувати декілька класів-власників. *Об’єкт-власник* і його складові частини (*підпорядковані об’єкти*) не можуть існувати окремо. Усі підпорядковані об’єкти змінюються разом із об’єктом-власником.

Графічно, композицію зображають як “дерево”, коренем якого є клас-власник, а листками – підпорядковані класи. На рис. 4.9 зображено частину композиції класу *Автомобіль*.

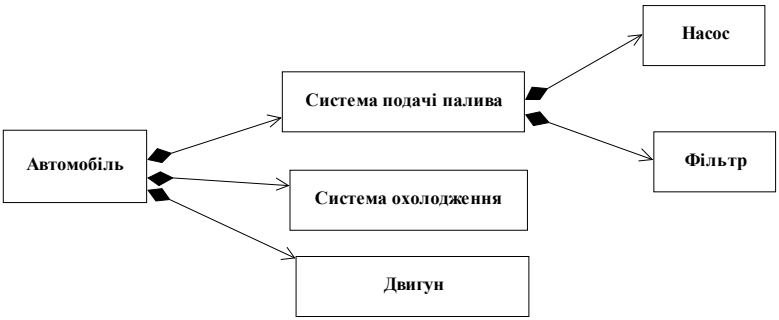


Рис. 4.9. Приклад композиції

У Rational Rose для перетворення відношення *агрегації* у відношення *композиції* необхідно вибрати опцію *Value* для властивості *Containment* на закладці *Role B Detail* у вікні специфікації відношення агрегації.

4.4. Узагальнення та залежності між класами

Узагальнення (*generalization*) – це відношення між двома сутностями, одна з яких є *частковим* (або *спеціалізованим*) випадком іншої. Відношення узагальнення передбачає виконання *принципу підстановки*:

> якщо сутність *A* – *загальне* (або *батько* – parent, *предок*) є узагальненням сутності *B* – часткове (або *дитина* – child, *нащадок*), то *B* може бути підставлене замість *A*.

Графічно відношення узагальнення зображають лінією з незафарбованою стрілкою, яка вказує на предка (рис. 4.10).

Відношення *успадкування* між класами в об’єктно-орієнтованому програмуванні є типовим прикладом узагальнення, за якого об’єкт спеціалізованого класу (нащадок) може бути підставлений замість об’єкта узагальненого класу (батька, предка).

Відношення *узагальнення* часто застосовують на діаграмі класів. Дійсно, важко уявити ситуацію, коли між об'єктами в одній системі немає нічого загального. Зазвичай, загальне є – і це загальне доцільно виокремити у деякий клас (*суперклас*). У цьому випадку загальні складові (атрибути та операції), зібрані в суперкласі, автоматично успадковуються підкласами.

Суперклас може бути *конкретним* (тобто мати власні екземпляри), або *абстрактним*, введеним саме для побудови відношення узагальнення. Узагальнення в моделі класів вводять довільно (за винятком вимоги відсутності циклів у ланцюжках узагальнень), зокрема, клас може бути підкласом декількох суперкласів (*множинне успадкування*); не потрібно, щоб у базових класів був загальний суперклас. На рис. 4.10 зображено узагальнення геометричної фігури.

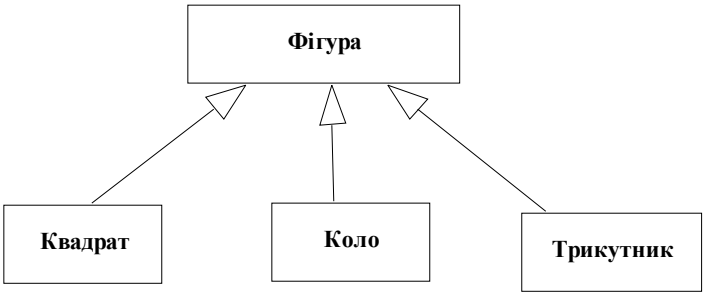


Рис. 4.10. Приклад узагальнення між класами

Залежність (*dependency*) – це найузагальненіше відношення між двома сутностями, яке вказує на те, що зміна *незалежної сутності* якось впливає на *залежну сутність*. Графічно відношення залежності зображають пунктирною стрілкою, спрямованою від залежної сутності до незалежної (рис. 4.11). Зазвичай, семантика конкретної залежності уточнюється в моделі за допомогою додаткової інформації.

На рис. 4.11 зображено залежність *Області малювання* від *Геометричної фігури*. Подібний зв'язок ніяк не регламентує тип відношення між об'єктами, а вказує лише на те, що залежний клас *Геометрична фігура* використовує якісь особливості реалізації іншого класу *Область малювання*.

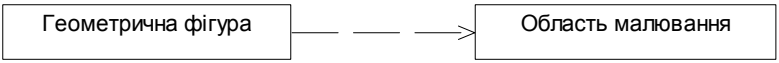


Рис. 4.11. Приклад залежності між класами

Під час моделювання класів залежності найчастіше використовують, щоб відобразити у сигнатурі операції той факт, що один клас використовує інший клас (незалежну сутність) як аргумент (рис. 4.12). У цьому випадку зміна одного класу знайде своє відображення при роботі іншого, оскільки незалежний клас може представити новий інтерфейс і/або поведінку.

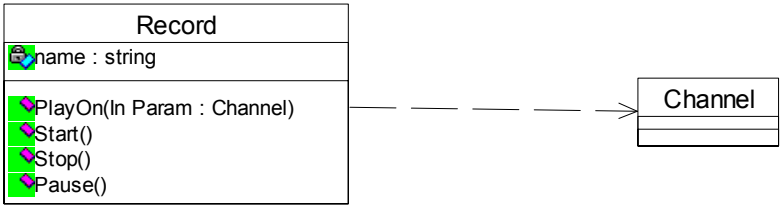


Рис. 4.12. Виклик класу залежним класом

Залежності можуть мати назви, хоча їх рідко використовують – у випадках, коли модель має чимало залежностей і доводиться на них посилатися. Здебільшого, для ідентифікації залежностей використовують стереотипи (див. вбудовану довідку засобу).

4.5. Розширення UML для моделей класів програмування і бізнесу

Побудова діаграми класів займає центральне місце у проектуванні складних систем. Зазвичай, програміст при розробці нових проектів прагне використовувати попередній досвід, накопичений ним і/або іншими програмістами (*стереотипний* підхід). Це зумовлено бажанням використовувати тільки перевірені фрагменти програмного коду з метою істотного скорочення термінів реалізації проекту.

До *стереотипного* підходу можна зачислити і механізми розширення UML, які дають змогу вводити додаткові графічні позначення, орієнтовані на розв'язування задач з певної предметної області. Важливе значення мають два спеціальні розширення: про-

філь розробки програмного забезпечення (*The UML Profile for Software Development Processes*) і профіль бізнес-моделювання (*The UML Profile for Business Modeling*).

У рамках профілю розробки програмного забезпечення запропоновано три спеціальні графічні позначення: для класу керування, класу-сутності та межового класу. Ці позначення використовують з метою уточнення семантики класів під час побудови різних діаграм. На рис. 4.13 наведені відповідні позначення: у формі піктограми (верхній рядок) і у формі класу зі стереотипом (нижній рядок).

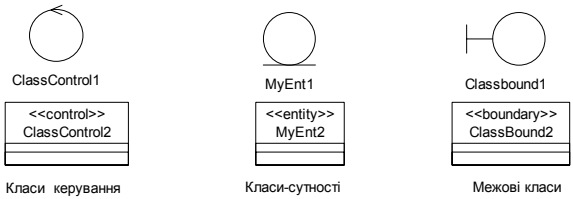


Рис. 4.13. Позначення профілю розробки програмного забезпечення

Клас керування (*control class*) відповідає за координацію поведінки об'єктів системи. У кожного прецеденту, зазвичай, є хоча б один клас керування, який контролює послідовність виконання дій цього прецеденту. Зазвичай, цей клас є активним та ініціює розсилання множини повідомлень іншим класам моделі. Клас керування може бути відсутнім у прецедентах, які здійснюють прості маніпуляції зі збереженими даними.

Клас-сутність (entity) містить інформацію, яка повинна зберігатися постійно і не знищуватися зі знищенням об'єктів певного класу чи припиненням роботи системи (вимиканням системи чи завершенням програми). Цей клас може відповідати окремій таблиці бази даних; у цьому випадку його атрибути є полями таблиці, а операції – збереженими процедурами. Зазвичай, цей клас є пасивним і лише приймає повідомлення від інших класів моделі. Класи-сутності є *ключовими абстракціями* (поняттями) ПО системи.

Межовий клас (*boundary*) розташовується на межі системи з зовнішнім середовищем, однак є складовою частиною системи.

Цей клас слугує посередником під час взаємодії зовнішніх об'єктів із системою. Зазвичай, для кожної пари *актор – прецедент* визначається один межовий клас. Типи межових класів: *інтерфейс користувача* (обмін інформацією з користувачем), *системний інтерфейс* і *апаратний інтерфейс* (протоколи використання, без деталей їхньої реалізації).

Інтерфейс (*interface*) у контексті мови UML є спеціальним випадком класу, який має тільки операції. Для позначення інтерфейсу використовують спеціальний графічний символ – коло чи прямокутник класу зі стереотипом "*interface*". Інтерфейси на діаграмах відображають ті елементи моделі, які є видимими ззовні, проте їхня структура є прихованою від користувачів.

Використання профілю розробки програмного забезпечення та інтерфейсу наведено на рис. 4.14 на прикладі побудови діаграми класів системи керування банкоматом.

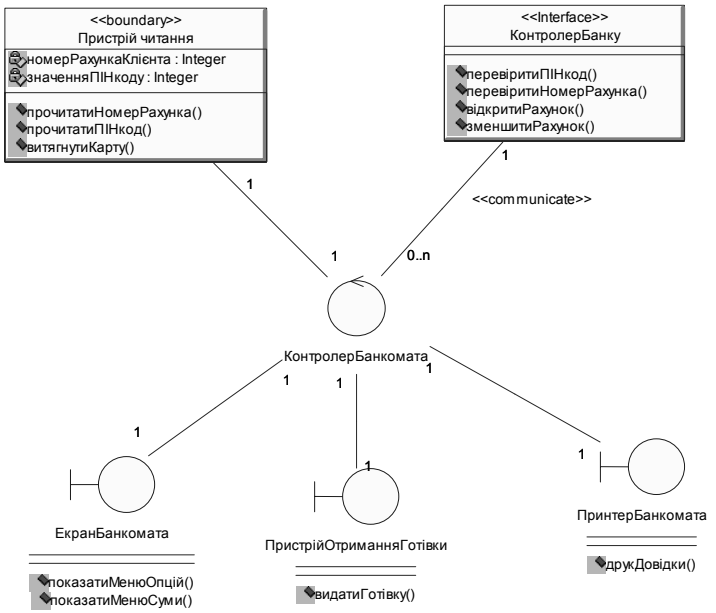


Рис. 4.14. Діаграма класів системи керування банкоматом

Зображена діаграма класів на рис. 4.14 є попередньою статичною моделлю системи керування банкоматом і може уточнюватися на наступних етапах роботи над проектом. Зокрема, може знадобитися змінити склад операцій окремих класів або їхню видимість, ввести додаткові атрибути або дещо детальніше розписати сигнатуру операцій.

У рамках профілю бізнес-моделювання також запропоновано три спеціальні графічні позначення:

- *співробітник* (*worker*) – клас опису співробітника, який є елементом бізнес-системи і взаємодіє з іншими співробітниками під час реалізації бізнес-процесу;
- *співробітник для зв'язків з оточенням* (*caseworker*) – клас опису співробітника, який є елементом бізнес-системи і взаємодіє з акторами під час реалізації бізнесу-процесу;
- *бізнес-сутність (business entity*) – спеціальний випадок класу-сутності, що також не ініціює жодних повідомлень.

4.6. Моделювання класів у Rational Rose

Активізувати робоче вікно *діаграми класів* у Rational Rose можна декількома способами:

- автоматично відображається після створення нового проекту;
- ❑ ЛКМ // на кнопці діаграми класів стандартної панелі;
- ⇩ Logical View ❑ ЛКМ (двічі) // на піктограмі Main;
- ➢ Browse ➢ Class Diagram …// задаємо нову діаграму.

У будь-якому випадку з'являється нове вікно із чистим робочим аркушем діаграми класів і *спеціальна панель інструментів*, що містить кнопки із зображенням графічних примітивів, необхідних для розробки діаграми класів.

На спеціальній панелі інструментів спочатку відображається тільки частина піктограм елементів, які можна використовувати для побудови діаграми класів. Додати кнопки з піктограмами інших графічних елементів (шаблон, бізнес-сутність, організаційний підрозділ тощо) або видалити непотрібні кнопки можна за допомогою команди контекстного меню *Customize* (*Настроювання*) при позиціонуванні курсора миші на спеціальній панелі інструментів.

Для додавання *класу* на діаграму класів потрібно:

- ❑ ЛКМ // на піктограмі класу спеціальної панелі інструментів;
- ❑ ЛКМ // на вільному місці робочого аркуша діаграми.

На діаграмі з'явиться зображення класу з маркерами зміни геометричних розмірів і стандартною назвою, яку варто змінити на мнемонічно відповідну поняттю ПО.

Додати *атрибут* до створеного раніше класу можна одним з перелічених способів:

- за допомогою команди *New Attribute* контекстного меню для класу, виокремленого на діаграмі класів;
- за допомогою команд *New* ➢ *Attribute* контекстного меню для класу, виокремленого у браузері проекту;
- за допомогою команди *Insert* контекстного меню, викликаного при позиціонуванні курсора в області відкритої *закладки атрибутів* у діалоговому вікні властивостей *Class Specification*.

Додати *операцію* до створеного раніше класу можна одним з наступних способів:

- за допомогою команди *New Operation* контекстного меню для класу, виокремленого на діаграмі класів;
- за допомогою команд *New* ➢ *Operation* контекстного меню для класу, виокремленого у браузері проекту;
- за допомогою команди *Insert* контекстного меню, викликаного при позиціонуванні курсору в області відкритої *закладки операцій* у діалоговому вікні властивостей *Class Specification*.

Додавання на діаграму класів *відношення між класами* виконується у такий спосіб:

- вибрати піктограму типу відношення на спеціальній панелі;
- якщо відношення *скероване*, то ❑ ЛКМ на першому елементі відношення (*джерело*, від якого відходить стрілка) на діаграмі класів і, не відпускаючи натиснуту ЛКМ, перемістити вказівник до другого елемента відношення (*приймач*);
- якщо ж відношення *нескероване*, то порядок вибору класів для цього відношення довільний;
- після переміщення до другого елемента кнопку миші необхідно відпустити (на діаграму класів буде додане нове відношення).

Увага! У Rational Rose для перетворення відношення *агрегації* у відношення *композиції* необхідно вибрати опцію *Value* для властивості *Containment* на закладці *Role B Detail* у вікні специфікації відношення агрегації.

Для відношення можна визначити кратність кожного з кінців, задати назву та стереотип, використовувати обмеження та ролі, а також деякі інші властивості.

Діаграми класів використовують на різних етапах моделювання програмної системи. На етапі *аналізу предметної області і формулювання вимог* діаграми класів використовують з метою *концептуалізації* (визначення об'єктів) предметної області. Цю роботу виконують аналітики спільно з фахівцями предметної області. Концептуальна діаграма класів, зазвичай, містить *класи-сутності* (entity) предметної області та відношення між ними.

Усі класи й діаграми класів, що зображають концептуальний рівень проекту, мають міститися у пакеті з назвою *Analysis Model*. На цьому рівні атрибути та операції класів, зазвичай, не вказують.

Здебільшого, на цьому етапі моделювання спочатку оформляють *словник* (або *глосарій*) предметної області, який містить текстовий опис термінів, сутностей, користувачів тощо. Словник ПО загального прикладу посібника може набути такого вигляду:

Курс	Навчальний курс, який читають на факультеті
Доступний курс (*Course Offering*)	Читання курсу в конкретному семестрі (курс можуть читати протягом декількох семестрів)
Каталог курсів	Каталог курсів, які читають на факультеті
Система розрахунку	Система обробки даних про оплату за курси
Оцінка	Оцінка студента за конкретний курс
Професор	Викладач університету
Табель успішності (*Report Card*)	Оцінки студента за курси у певному семестрі
Список курсу (*Roster*)	Список студентів, що записалися на запропонований курс
Студент	Особистість, яка навчається на факультеті
Навчальний графік (*Schedule*)	Курси, обрані студентом у поточному семестрі

На основі словника предметної області та специфікації вимог для системи реєстрації на курси можна попередньо ідентифікувати п'ять *класів-сутностей* ПО: *Student* (*Студент*); *Professor* (*Професор*); *Schedule* (*Навчальний графік студента*); *Course* (*Курс*); *CourseOffering* (*Доступний курс*). Створимо діаграму класів *Key Abstractions* у пакеті з назвою *Analysis Model*. На діаграмі класів *Key Abstractions* створимо класи-сутності системи реєстрації на курси та введемо деякі очевидні відношення (рис. 4.15).

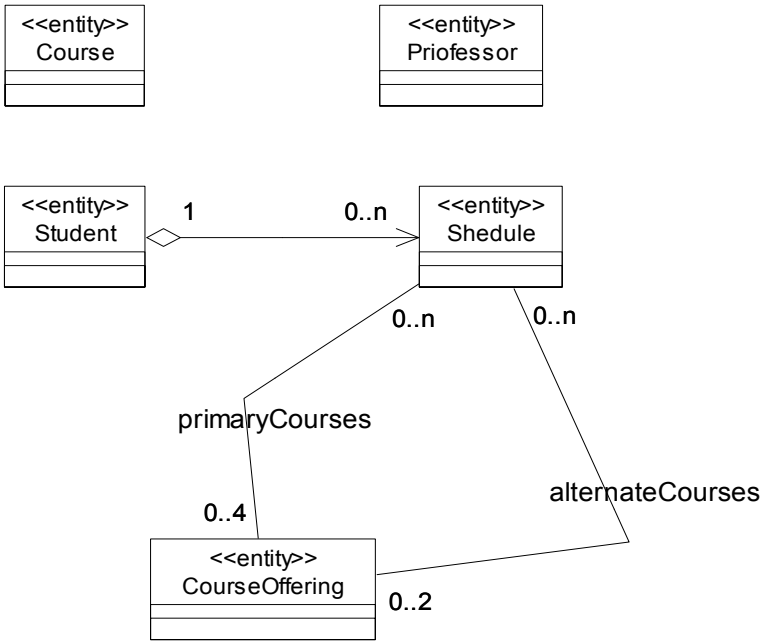


Рис. 4.15. Діаграма класів системи реєстрації на курси

Агрегація між *Student* і *Schedule* віддзеркалює той факт, що кожен графік є власністю конкретного студента. У системі зберігатимуться усі графіки студента за різні семестри.

Між класами *Schedule* і *CourseOffering* введено дві асоціації, оскільки конкретний курс може входити у графік студента як головний (не більше чотирьох курсів) і альтернативний (не більше двох курсів).

У пакеті *Analysis Model* для прецеденту "*Зареєструватися на курси*" створимо межові класи (*RegisterForCoursesForm* і *CourseCatalogSystem*) і клас керування (*RegistrationController*).

У пакеті *Analysis Model* створимо такі *пакети* з класами:

- *CoursesEntity* – містить класи *Course* і *CourseOffering*;
- *PersonsEntity* – містить класи *Student* і *Professor*;
- *OtherEntity* – містить клас *Schedule*;
- *FormBoundary* – містить класи *RegisterForCoursesForm* і *CourseCatalogSystem*;
- *Controllers* – містить клас *RegistrationController*.

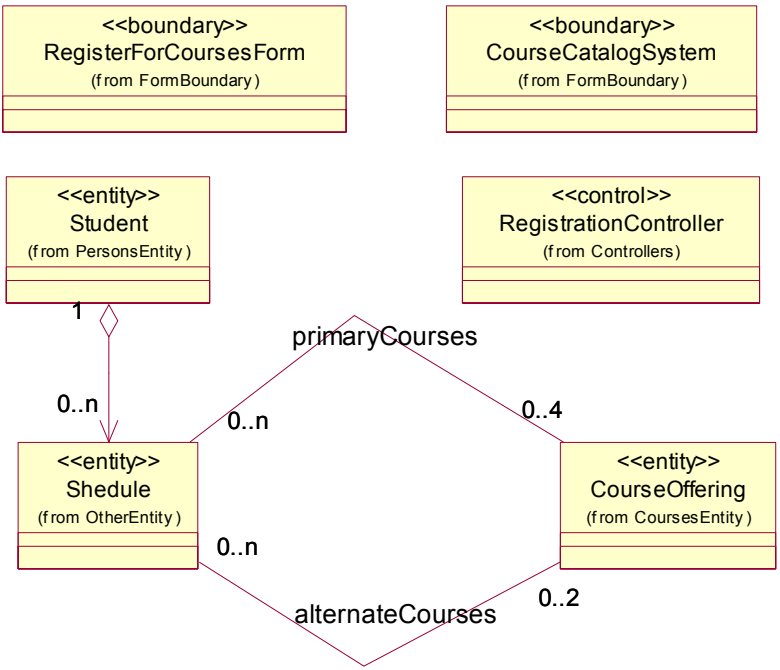


Рис. 4.16. Діаграма класів прецеденту "*Зареєструватися на курси*"

Далі створюємо діаграму класів, які беруть участь у реалізації прецеденту "*Зареєструватися на курси*":

⇩ Logical View ⇩ Design Model ⇩ Use-Case Realization

⇩ Use-Case Realization-Register for Courses

⇗ Register for Courses ❑ ПКМ // на кооперації у браузері
➢ New ➢ Class Diagram
Назва_діаграми:= ViewOfParticipatingClasses
// Відкрийте діаграму ViewOfParticipatingClasses і перетягніть
// на неї класи відповідно до рис. 4.16
// Створіть відношення відповідно до рис. 4.16.

Після розробки діаграми класів процес ООА і П може бути продовжено у двох напрямах. Якщо поведінка системи є простою, то можна приступити до створення діаграм послідовностей та кооперації. Однак для складних динамічних систем, зокрема для паралельних систем і систем реального часу, найважливішим аспектом їхнього функціонування є поведінка. Особливості поведінки таких систем можуть бути представлені на діаграмах станів і діяльності, розробка яких у загальних випадках є необов'язковою.

? Запитання для самоперевірки

1. Дайте визначення класу і об'єкта.
2. Як зображають класи на діаграмах?
3. З яких частин складається зображення класу?
4. Які відношення застосовують на діаграмі класів?
5. Що таке атрибут класу? Що таке операція класу?
6. Коротко охарактеризуйте специфікацію атрибута класу.
7. Коротко охарактеризуйте специфікацію операції класу.
8. Коротко охарактеризуйте можливі застосування відношення асоціації на діаграмах класів.
9. Що таке асоційований клас і багатополюсна асоціація?
10. Коротко охарактеризуйте можливі застосування відношення агрегації на діаграмах класів.
11. Коротко охарактеризуйте можливі застосування відношення композиції на діаграмах класів.
12. Коротко охарактеризуйте можливі застосування відношення узагальнення на діаграмах класів.
13. Що таке клас керування? Для чого його використовують?
14. Що таке межовий клас? Для чого його використовують?
15. Що таке клас-сутність? Для чого його використовують?
16. Що таке інтерфейс? Для чого його використовують?

5. МОДЕЛЮВАННЯ ВЗАЄМОДІЇ ТА ПОВЕДІНКИ ОБ’ЄКТІВ

План викладу матеріалу:

1. Загальні положення.
2. Діаграми послідовностей.
3. Діаграми кооперацій.
4. Приклад побудови діаграм взаємодії у Rational Rose.
5. Діаграми станів.

Ключові терміни розділу

- *Взаємодія*; *повідомлення*
- *Діаграма кооперації*
- *Лінія життя об’єкта*
- *Позначення об’єкта*
- *Рефлексивні повідомлення*
- *Стереотипи зв’язків*
- *Переходи*
- *Діаграма послідовностей*
- *Клієнти і сервери*
- *Фокус керування* (*активація*)
- *Головні повідомлення*
- *Кооперація*
- *Стани об’єкта*
- *Діаграми станів*

5.1. Загальні положення

У складних системах об’єкти взаємодіють один з одним, постійно обмінюючись повідомленнями. *Взаємодією* (*Interactions*) називають поведінку, яка виражається в обміні *повідомленнями* між об’єктами певної сукупності в заданому контексті, унаслідок чого досягають певної мети. *Повідомлення* (*Message*) – це специфікація обміну даними між об’єктами, за якого передається деяка інформація і передбачається, що у відповідь відбудеться певна дія.

За допомогою взаємодій моделюють потоки керування всередині операції, класу, компонента чи системи загалом. Взаємодії дають змогу моделювати такі потоки за двома критеріями:

- за часовою упорядкованістю повідомлень;
- за структурною організацією об’єктів, що обмінюються повідомленнями.

Моделі, які відображають часову упорядкованість повідомлень між об’єктами, називають *діаграмами послідовностей* (*sequence diagram*), а моделі, які відображають структурну організацію об’єктів, що обмінюються повідомленнями, – *діаграмами коопера-*

ції (*collaboration diagram*). Ці види діаграм є *ізоморфними*, тобто можуть вільно трансформуватися одна в одну.

Об'єкти, що посилають повідомлення, називають *клієнтами*, а об'єкти, що їх обробляють – *серверами*. У найпростішому випадку повідомлення можна розглядати як виклик методу деякого класу, у складніших випадках сервер опрацьовує черги повідомлень.

Діаграми послідовностей та кооперацій зображають взаємодію елементів моделі у контексті реалізації *окремих прецедентів*. *Діаграма станів* (*statechart diagram*) описує процес зміни стану системи чи її підсистеми за реалізації *усіх прецедентів*. Зміна стану окремих елементів системи, підсистем чи системи загалом може активізуватись подією, ініційованою з боку інших елементів, підсистем чи ззовні системи.

Головне призначення діаграми станів – описати можливі послідовності станів і переходів, які характеризують поведінку системи протягом усього її життєвого циклу. Системи/елементи, що реагують на зовнішні впливи від інших систем/елементів чи користувачів, називають *реактивними*. Якщо такі дії ініціюються у довільні випадкові моменти часу, то кажуть про *асинхронну* поведінку системи.

Хоча діаграми станів найчастіше описують поведінку окремих систем чи підсистем, вони також використовуються для моделювання усіх можливих змін станів конкретних об'єктів.

5.2. Діаграми послідовностей

Діаграма послідовностей (*Sequence Diagram*) призначена для відображення *часових* залежностей, що виникають у процесі взаємодії об'єктів. На верхньому краю діаграми *горизонтально* розміщують *об'єкти* (прямокутники із назвами об'єктів). З кожного об'єкта виходить *вертикальна* штрих-пунктирна лінія (*лінія життя* об'єкта), на якій умовно відкладено *час*.

Крайнім зліва на діаграмі зображають актора (або об'єкт), який є ініціатором взаємодії. Правіше зображають інший об'єкт, який безпосередньо взаємодіє з актором чи першим об'єктом і т.д. Отже, порядок розміщення об'єктів на діаграмі визначається міркуванням зручності візуалізації їхньої взаємодії один з одним.

Лінія життя об'єкта використовується для позначення періоду, протягом якого об'єкт існує в системі та може брати участь в усіх взаємодіях з іншими об'єктами.

Окремі об'єкти, виконавши свою роль у системі, можуть знищуватися, щоб звільнити ресурси, які вони займають. Для позначення моменту знищення об'єкта (виклик деструктора об'єкта) використовують спеціальну позначку у формі латинської літери "X".

У системі не обов'язково створювати усі об'єкти у початковий момент часу. Окремі об'єкти створюють у момент, коли виникає необхідність їхнього використання. Прямокутник такого об'єкта вертикально розміщується у тому місці діаграми, яке за віссю часу збігається з моментом його виникнення в системі.

Процес взаємодії об'єктів реалізується через повідомлення, які посилаються одними об'єктами іншим. Повідомлення, розташовано на діаграмі послідовності вище, передаються раніше за ті, які розташовано нижче. Щодо цього масштаб на осі часу не зазначають, оскільки діаграма послідовності моделює лише часову впорядкованість взаємодій типу "раніше-пізніше".

Окрім об'єктів та ліній життя об'єктів, на діаграмі послідовностей ще можуть міститися такі елементи:

- *фокус керування* (або *активація*) – тонкий вертикальний прямокутник, розташований уздовж лінії життя об'єкта (або прямо виходить з об'єкта), який позначає період активного життя об'єкта (тобто час, протягом якого об'єкт має фокус);
- *повідомлення* – позначається стрілкою з назвою дії, яка розміщується між лініями життя об'єктів (або між активаціями об'єктів); напрям стрілки задає напрям передачі даних;
- текстові позначки (оцінки часу, опис дій тощо).

Необхідність зображення зв'язків між окремими об'єктами може виникнути з різних причин. Найважливішою з них є зображення реалізації окремого прецеденту моделі. У ранніх версіях UML для цієї мети слугувала діаграма об'єктів, яку згодом замінили канонічною діаграмою кооперації.

Головними елементами, з яких утворюються діаграми взаємодії, є *об'єкти* – окремі екземпляри класів, які створюються на етапі реалізації моделі (виконання програми). Об'єкт може мати

назву вигляду `назва об'єкта:назва класу` і конкретні значення атрибутів.

У багатьох випадках `назва об'єкта` є відсутньою. Такий об'єкт називають *анонімним*, при цьому обов'язково записується двокрапка перед назвою відповідного класу. Може бути відсутньою і `:назва класу` (двокрапка відсутня також) – такий об'єкт називають *сиротою*.

У процесі функціонування системи об'єкт може перебувати в *активному* стані, виконуючи певні дії, або в стані *пасивного очікування* повідомлень від інших об'єктів. Для позначення станів активності об'єкта на діаграмах послідовностей застосовують спеціальну позначку – *фокус керування* (*focus of control*). У Rational Rose фокус керування можна увімкнути/вимкнути так:
➢ Tools ➢ Options… 🗀 Diagram ☑ Focus of control // або скинути.

Найуживаніші типи повідомлень: *Procedure Call* (виклик процедур, виконання операцій або перехід на вкладені потоки керування); *Object Message* (виклик методів об'єкта); *Return Message* (повернення з виклику процедури/методу; у процедурних потоках керування може не вказуватися, оскільки неявно передбачається наприкінці активізації об'єкта); *Asynchronous Message* (позначення простого асинхронного повідомлення, яке передається в довільний момент і не супроводжується отриманням фокуса керування об'єктом-приймачем).

Інколи об'єкт посилає повідомлення самому собі, ініціюючи *рефлексивні* повідомлення (*Message to Self*). Приклади: опрацювання натискання клавіш, набір цифр номера телефону тощо.

Лінії, що позначають передачу повідомлень (виклики методів), позначаються назвою виконуваної дії чи переданим повідомленням. Можуть бути відображені фактичні параметри, передані у точку виклику, чи результат, що повертається після виклику.

Найпростіші випадки галуження процесу взаємодії можна зобразити на одній діаграмі – альтернативні повідомлення зображаються паралельними прямими між лініями життів сусідніх об'єктів.

Кожен альтернативний потік керування може суттєво ускладнити розуміння моделі, отож рекомендують кожен потік керування зображати на окремій діаграмі послідовностей.

5.3. Діаграми кооперацій

Діаграма кооперації або діаграма співробітництва (*collaboration diagram*) описує *статичну структуру об’єктів*, що реалізують поведінку підсистеми.

Мета кооперації полягає у тому, щоб специфікувати особливості реалізації *прецедентів* та найважливіших *операцій* у системі. Кооперація визначає структуру поведінки системи у термінах взаємодії учасників цієї кооперації і може зображатися на рівні *специфікацій* (*specifications*) чи на рівні *прикладів* (*instances*).

Діаграма кооперації на рівні специфікації містить *класи*, *асоціації* та *кооперації* (зображаються пунктирним еліпсом, всередині якого записується назва кооперації). Такі діаграми кооперації, зазвичай, відображають особливості реалізації *прецедентів* у системі (див. пункт 3.7).

Діаграма кооперації на рівні прикладів містить *об’єкти* (екземпляри класів), *зв’язки* (екземпляри асоціацій) і *повідомлення* (зв’язки доповнюються стрілками повідомлень). На цьому рівні ілюструють тільки ті об’єкти та зв’язки, які мають безпосереднє відношення до реалізації певної кооперації. Для повідомлення, зазвичай, подають назву та порядковий номер у загальній послідовності повідомлень.

Одна і та ж сукупність об’єктів може брати участь у реалізації різних кооперацій. При цьому можуть змінюватись як зв’язки між окремими об’єктами, так і потік повідомлень між ними. Саме це відрізняє діаграму кооперації від діаграми класів, на якій зазначають усі без винятку класи, їхні атрибути й операції, а також усі асоціації та інші структурні відношення між елементами моделі.

Додати об’єкт на діаграму кооперації можна стандартним шляхом за допомогою відповідної кнопки на спеціальній панелі інструментів. Однак зручніше у браузері проекту виокремити необхідний клас і, утримуючи натиснуту ліву клавішу миші, перетягнути його на поле діаграми. У результаті на діаграмі кооперації відображається об’єкт з назвою класу і маркерами зміни його геометричних розмірів. За домовленістю кожен об’єкт, що додається, вважають анонімним. Задати назву об’єкта можна у вікні специфікації.

На діаграмі кооперації може зображатися *мультиоб'єкт* (установлений прапорець *multiple instances* у специфікації об'єкта) – множина об'єктів, які створюються на базі одного класу. Мультиоб'єкт використовують для того, щоб показати операції та сигнали, які адресовано всій множині об'єктів.

Об'єкти – учасники зв'язку на діаграмі кооперації – можуть мати стереотипи, які вказують на їхню роль у реалізації цього зв'язку. Найчастіше використовують такі стереотипи:

- `<<unspecified>>` – невизначений (за домовленістю);
- `<<parameter>>` – об'єкт є параметром деякої операції;
- `<<local>>` – локальна змінна (область видимості обмежена сусіднім об'єктом);
- `<<global>>` – глобальна змінна (область видимості поширюється на всю діаграму кооперації);
- `<<self>>` – рефлексивний зв'язок об'єкта з самим собою, який передбачає передачу об'єктом повідомлень самому собі.

Увага! У Rational Rose замість текстових стереотипів ролей об'єктів використовуються спеціальні позначки (квадратик з літерою).

Додати повідомлення на діаграму кооперації можна за допомогою кнопки із піктограмою повідомлення на спеціальній панелі інструментів. Однак зручніше додавати повідомлення за допомогою закладки *Messages* у вікні специфікації зв'язку. Після виконання команди контекстного меню *Insert To* (вставити у напрямі) відображається випадаючий список з пропозицією вибору операції відповідного класу.

Заміняємо стандартну назву операції на мнемонічно зрозумілішу назву. Ця назва буде також і назвою доданого повідомлення. У результаті цих дій на діаграмі кооперації поруч із лінією зв'язку відображається стрілка з номером і назвою повідомлення.

Увага! Зв'язок на діаграмі кооперації може мати декілька повідомлень. Повідомлення у цьому випадку можуть зливатися в одну потовщену лінію з декількома стрілками.

На рис.5.1 зображено діаграму кооперації моделі початку телефонної розмови.

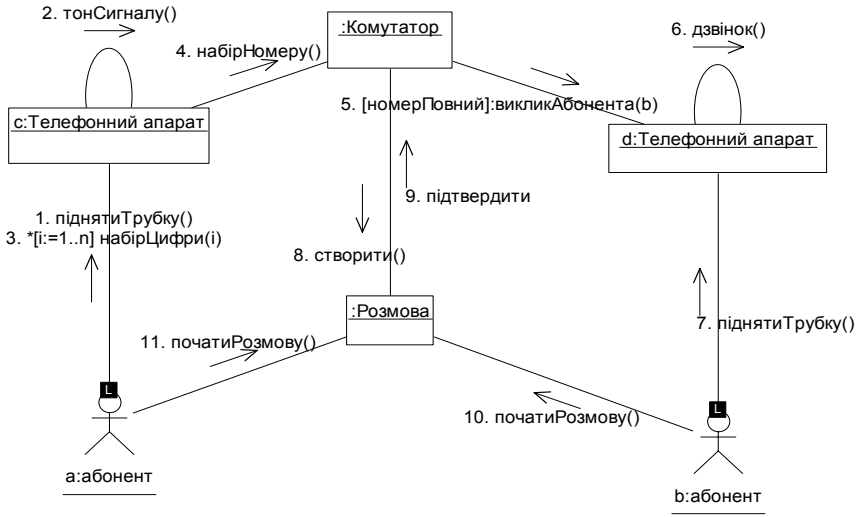


Рис. 5.1. Діаграма кооперації моделі початку телефонної розмови

5.4. Приклад побудови діаграм взаємодії у Rational Rose

У попередньому розділі створено діаграму класів (*ViewOf-ParticipatingClasses*), які беруть участь у реалізації прецеденту “*Зареєструватися на курси*” (див. рис. 4.16). Зараз побудуємо діаграми взаємодії, які відображатимуть особливості поведінки об’єктів, які реалізують цей прецедент. Передусім будуємо діаграму (одну чи більше), яка описує головний потік подій і його підлеглі потоки. Для кожного альтернативного потоку будуємо окрему діаграму.

Створення діаграми послідовностей основного потоку подій:

⇩ Logical View ⇩ Design Model ⇩ Use-Case Realization

⇩ Use-Case Realization-Register for Courses

⇗ Register for Courses ❑ ПКМ // на кооперації у браузері

➢ New ➢ Sequence Diagram

Name:= *Register for Courses-Basic Flow* // Назва діаграми

Аналогічно створюємо діаграми послідовностей *альтернативних підпотоків* подій: Register for Courses-Basic Flow (Create Shedule), Register for Courses-Basic Flow (Update Shedule), Register for Courses-Basic Flow (Delete Shedule).

Налаштування опцій для діаграм взаємодії:

➢ Tools ➢ Options … 🗀 Diagram ☑ Sequence Numbering ☑ Collaboration Numbering ☑ Focus of Control (зняти)

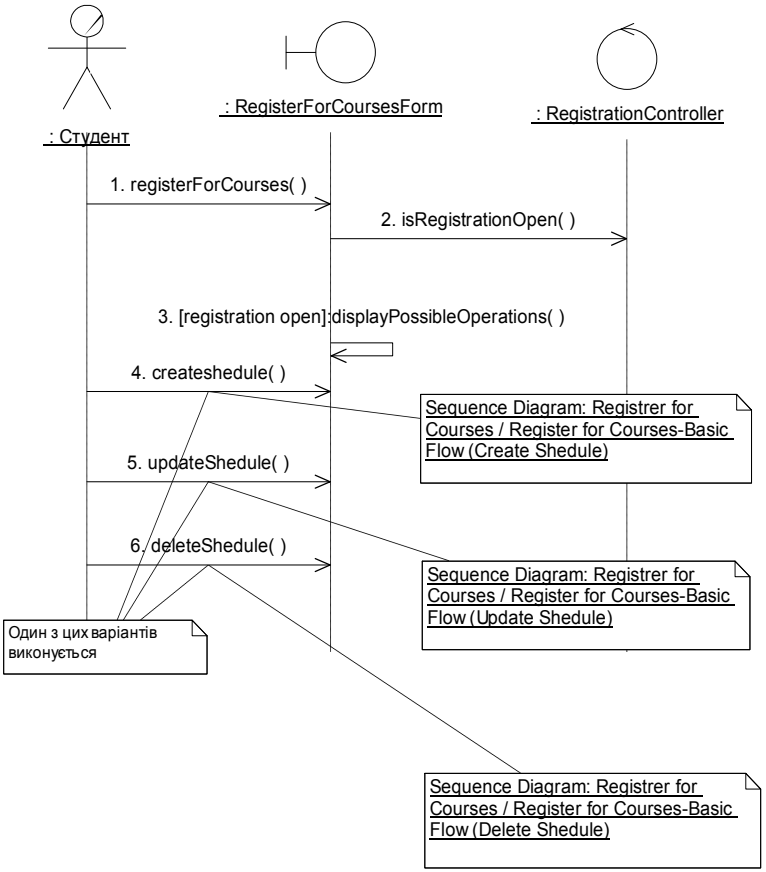


Рис. 5.2. Діаграма послідовностей Register for Courses-Basic Flow

Заповнення діаграми послідовностей Register for Courses-Basic Flow:

⇩ Logical View ⇩ Design Model ⇩ Use-Case Realization

⇩ Use-Case Realization-Register for Courses

⇩ Register for Courses ⇗ Register for Courses-Basic Flow

❑ ЛКМ (двічі) // відкриття порожньої діаграми

// Перетягніть актора *Студент* із браузера на діаграму
// Перетягніть класи *RegisterForCoursesForm* і *Registration-*
// *Controller* із браузера на діаграму
1) ⇗ Object Message // фіксуємо ЛКМ і проводимо мишею від
// лінії життя *Студент* до лінії життя *RegisterForCoursesForm*
2) ⇗ Номер_повідомлення ❑ ПКМ
3) ➢ <*new operation*> Name:= *registerForCourses*( ) ❑ OK

Повторюємо дії 1 – 3 доти, доки не введемо усі інші повідомлення (див. рис. 5.2). Для рефлексивного повідомлення 3 використовуємо кнопку Message to Self.

Розміщення на діаграмі приміток:

⇗ Note // на спеціальній панелі інструментів
❑ ЛКМ // на місці розміщення примітки
⇗ *Нову_примітку* та уведіть туди текст
⇗ Anchor Notes To Item // на спеціальній панелі інструментів
❑ ЛКМ (і зафіксуйте) // проведіть покажчик від примітки до
// елемента діаграми, з яким вона зв'язується (штрихова лінія).
// Щоб створити примітку – *посилання* на іншу діаграму, ство-
// ріть порожню примітку і перетягніть на неї з браузера потріб-
// ну діаграму.

Заповнену діаграму послідовностей Register for Courses-Basic Flow з примітками – посиланнями на інші діаграми – зображено на рис. 5.2. Для створення відповідної кооперації (рис. 5.3) достатньо відкрити діаграму послідовностей Register for Courses-Basic Flow і натиснути клавішу F5.

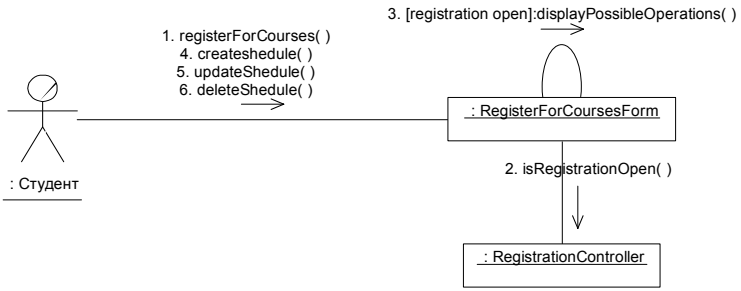


Рис. 5.3. Діаграма кооперації Register for Courses-Basic Flow

Налаштуємо тепер опції для діаграм взаємодії так:

➢ Tools ➢ Options … 🗀 Diagram ☑ Sequence Numbering ☑ Collaboration Numbering ☑ Focus of Control

Отриману при цьому діаграму послідовностей Register for Courses-Basic Flow з фокусами керування зображено на рис. 5.4.

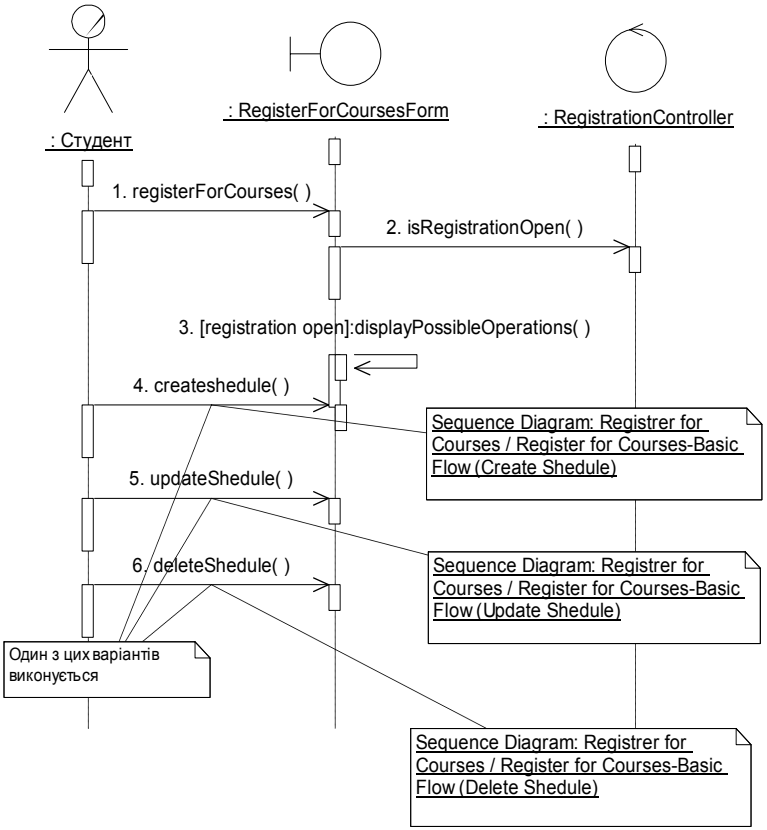


Рис. 5.4. Діаграма послідовностей Register for Courses-Basic Flow

Створюємо діаграму Register for Courses-Basic Flow (Submit Shedule) для відображення роботи з підпорядкованим журналом:

⇩ Logical View ⇩ Design Model ⇩ Use-Case Realization ⇩ Use-Case Realization-Register for Courses

⇗ Register for Courses ❑ ПКМ // на кооперації у браузері
➢ New ➢ Sequence Diagram
Name:= Register for Courses-Basic Flow (Submit Shedule).

Аналогічно заповненню діаграми послідовностей Register for Courses-Basic Flow заповнюємо діаграми послідовностей *альтернативних потоків* подій:

Register for Courses-Basic Flow (Create Shedule) – рис. 5.5;
Register for Courses-Basic Flow (Update Shedule) – рис. 5.6;
Register for Courses-Basic Flow (Delete Shedule) – рис. 5.7,

а також *допоміжну* діаграму Register for Courses-Basic Flow (Submit Shedule) – рис. 5.9.

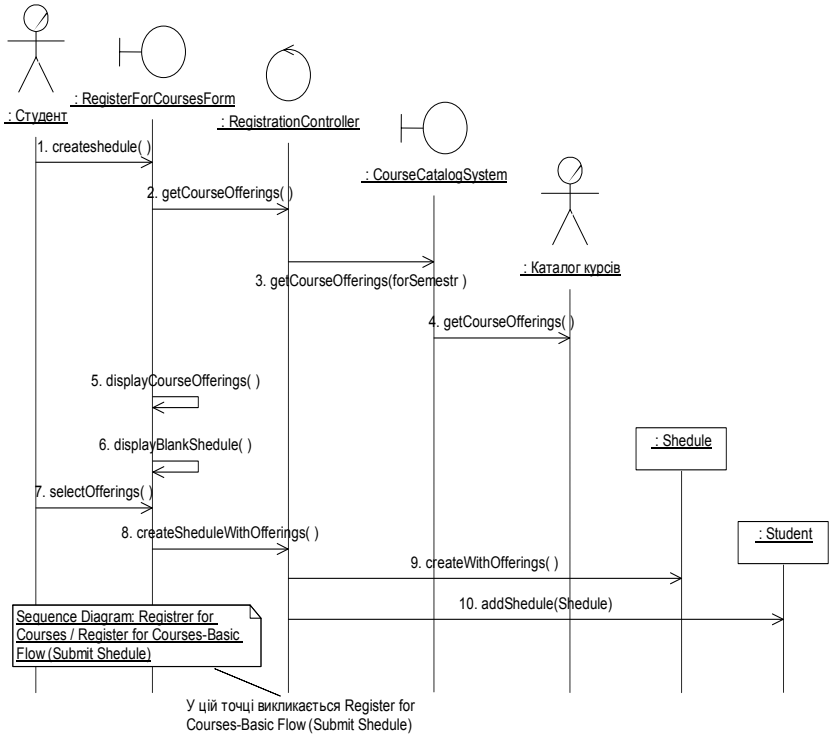


Рис. 5.5. Діаграма Register for Courses-Basic Flow (Create Shedule)

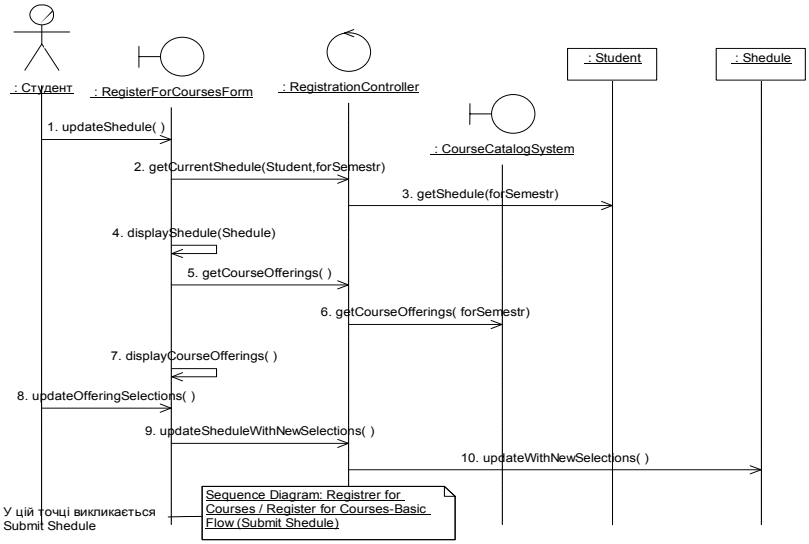


Рис. 5.6. Діаграма Register for Courses-Basic Flow (Update Shedule)

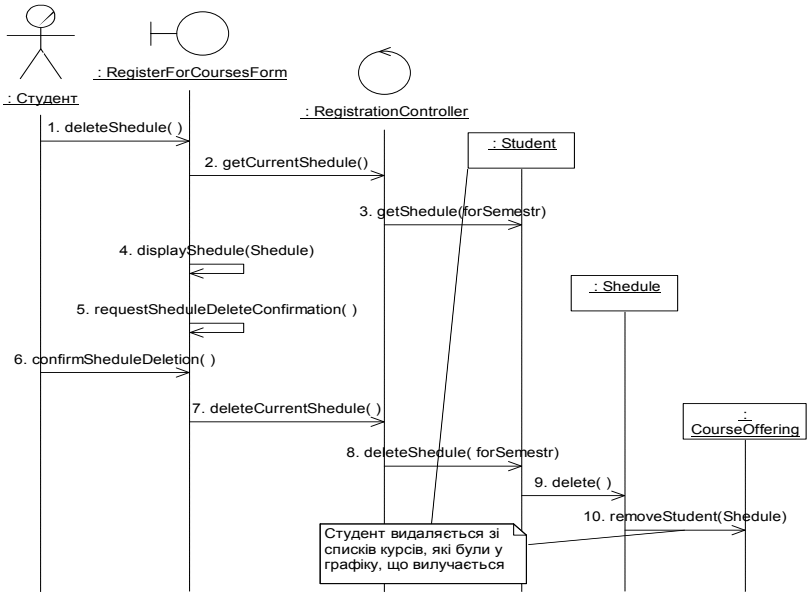


Рис. 5.7. Діаграма Register for Courses-Basic Flow (Delete Shedule)

Для створення допоміжної діаграми *Register for Courses-Basic Flow* (*Submit Schedule*) необхідно попередньо побудувати *асоційовані* класи *ScheduleOfferingInfo* і *PrimaryScheduleOfferingInfo* (див. рис. 5.8), які відображатимуть властивості зв'язків між класами *Schedule* і *CourseOffering*. Розмістимо ці класи у пакеті *OtherEntity*, а діаграму класів *CourseOfferingInfo* – у пакеті *Analysis Model*.

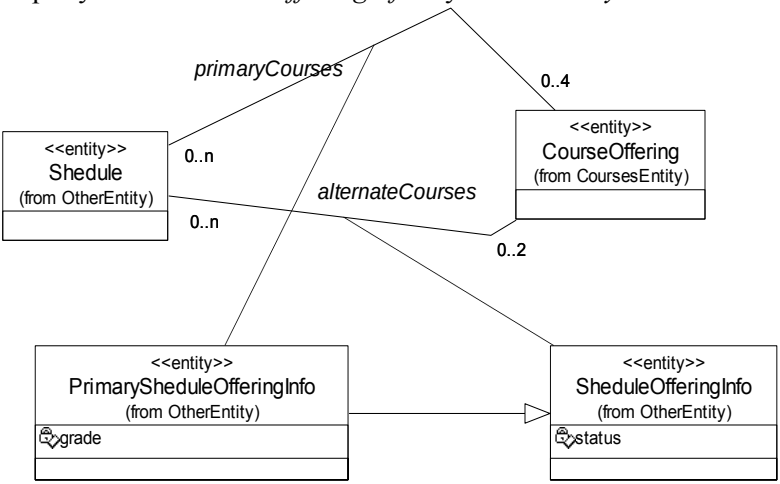


Рис. 5.8. Діаграма класів *CourseOfferingInfo*

Асоційований клас *ScheduleOfferingInfo*, що зв'язує графік студента (*Schedule*) і асоціацію альтернативного курсу, має тільки один атрибут **`status`** (статус курсу в конкретному графіку), що може набувати значення: "*позначений для долучення до графіка*" (*selected*) або "*скасований*" (*cancelled*).

Асоційований клас *PrimaryScheduleOfferinglnfo*, що зв'язує графік студента (*Schedule*) і асоціацію основного курсу, має атрибути **`grade`** (*оцінка за курс*) і **`status`** (*статус курсу*). Атрибут **`status`** може набувати значення: "*внесений у графік*" (*enrolled*) і "*зафіксований у графіку*" (*committed*). Якщо курс у процесі закриття реєстрації переходить з альтернативного в головний, то до відповідної асоціації додається атрибут **`grade`** (*оцінка*).

Отже, клас *PrimaryScheduleOfferinglnfo* успадковує властивості класу *ScheduleOfferinglnfo* і додає свої власні (рис. 5.9).

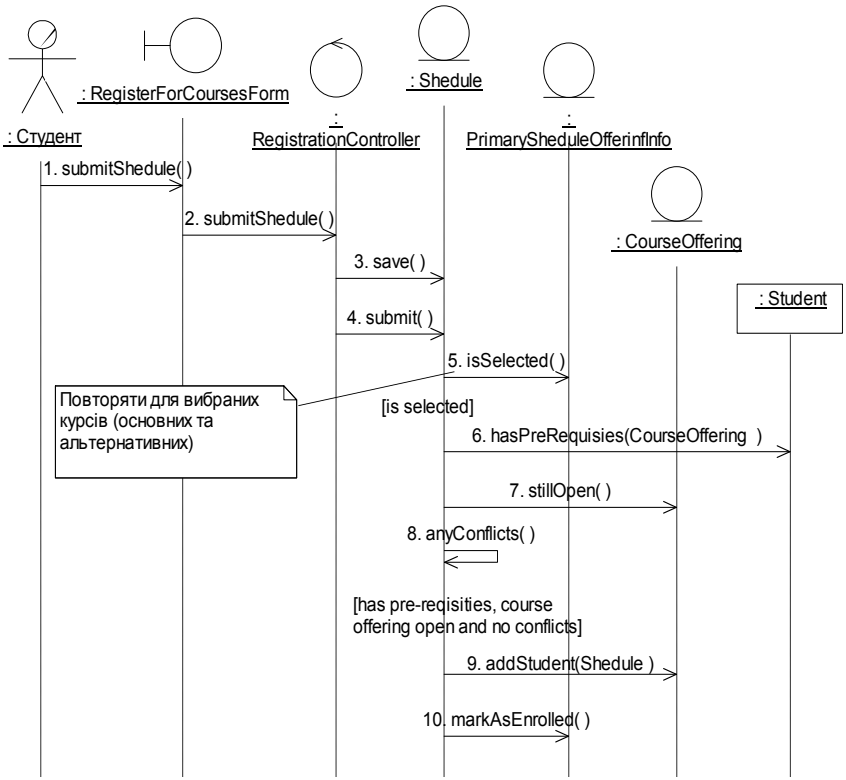


Рис. 5.9. Діаграма *Register for Courses-Basic Flow* (*Submit Schedule*)

Початковий набір зв'язків між класами уточнюється на основі аналізу діаграм кооперації. Якщо два об'єкти взаємодіють (обмінюються повідомленнями), то між ними на діаграмі кооперації має існувати зв'язок (шлях взаємодії), який перетворюється у двонаправлену асоціацію між відповідними класами. Якщо повідомлення між деякою парою об'єктів передаються тільки в одному напрямі, то для відповідної асоціації вводять напрям навігації.

5.5. Діаграми станів

Діаграма станів (*State chart diagram*) визначає усі можливі стани (*state*), у яких може знаходитися конкретний об’єкт під час свого існування, а також процес зміни станів цього об’єкта у результаті настання деяких подій (*event*).

На діаграмі виокремлюють два спеціальних стани – початковий (*start*) і кінцевий (*stop*). Початковий стан виокремлюють чорною крапкою, він відповідає тільки що створеному стану об’єкта. Кінцевий стан виокремлюють чорною крапкою в білому крузі, він відповідає стану об’єкта безпосередньо перед його знищенням. На діаграмі станів може бути тільки один початковий стан і декілька кінцевих станів (або кінцевих станів може не бути взагалі).

Усі інші стани зображають прямокутником із заокругленими краями. Кожен стан має дві частини. У верхній частині відображають назву стану. Назва стану може бути порожньою (*анонімний* стан). Нижню частину стану призначено для відображення внутрішніх дій (*actions*), які виконуються у відповідь на визначені події, що виникають у цьому стані. Наприклад, діаграма станів об’єкта класу *Account* (*Рахунок*) може набувати вигляду як на рис. 5.10.

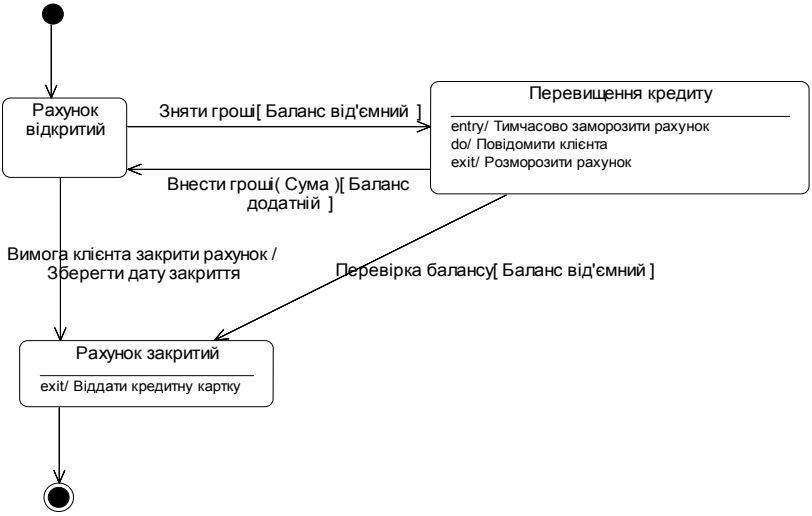


Рис. 5.10. Діаграма станів об’єкта класу *Account*

Для стану об'єкта можна задати *вхідну подію* (**On Entry**), *вихідну подію* (**On Exit**), *довільну подію* (**On Event НазваПодії**) і діяльність (**Do**). У відповідь на виникнення події чи під час реалізації діяльності можна задати просту дію (**Action**) чи надіслати повідомлення/активізувати подію (**Send Event**) іншому об'єкту.

Опис простих дій має такий формат:

```
entry | exit | do / дія  // одна з подій (entry, exit) чи do
event НазваПодії [(СписокПараметрів)] [умова] / дія
```

Опис активізації події має такий формат:

```
entry | exit | do /^Об'єкт.НазваПодії [(СписокПараметрів)]
event НазваПодії [(СписокПараметрів)] [умова] /
                    ^Об'єкт.НазваПодії [(СписокПараметрів)]
```

Для реалізації реакції на подію стану об'єкта чи реалізації діяльності стану об'єкта необхідно виконати таке:

⇗ Стан_об'єкта ➢ Open Specification …

🗀 Actions ❑ ПКМ // на області розміщення дій

➢ Insert ⇗ Нову_подію ❑ ПКМ ➢ Specification …

⇩ When // уточнити подію чи обрати діяльність

// Для **On Event** відкриваються поля введення: Event (назви

// події); Arguments (списку параметрів); Condition (умови)

⇩ Type ⇗ **Action** (чи **Send Event**)

// Для **Send Event** відкриваються поля введення: Name (назви

// події); Send Arguments (списку параметрів); Send target

// (назва об'єкта)

// описати дію чи сформувати повідомлення

❑ ОК ❑ ОК

Переходом (*Transition*) називають переміщення з одного стану об'єкта в інший внаслідок виникнення певної події (**Event**). На діаграмі перехід зображають стрілкою, що починається на первісному стані і закінчується наступним станом. Переходи можуть бути рефлексивними (*Transition To Self*). Найпростіший перехід має таку специфікацію (відображається поруч зі стрілкою):

НазваПодії [**(СписокПараметрів)**] [**[умова]**]

Найпростіший перехід зі стану 1 у стан 2 ілюструє, що об'єкт, який перебуває у стані 1, перейде у стан 2 тоді, коли відбудеться визначена для переходу подія і справдиться задана **умова** (на діаграмі обмежується квадратними дужками).

Подія, що визначається **НазвоюПодії**, може задаватися назвою операції або звичайною фразою (як на рис. 5.10). Наприклад, замість фрази "*Вимога клієнта закрити рахунок*" можна задати назву операції **RequestClosure**. У подій можуть бути параметри. Наприклад, подія "*Внести* гроші" має параметр **Сума**.

Під час виконання переходу можна задати просту дію (**Action**) чи надіслати повідомлення/активізувати подію (**Send Event**) іншому об'єктові. Опис переходу матиме такий формат:

НазваПодії [**(СписокПараметрів)**] [**[умова]**] **/ дія**

НазваПодії [**(СписокПараметрів)**] [**[умова]**] **/**

^Об'єкт.НазваПодії[**(СписокПараметрів)**]

У будь-якому випадку для опису специфікації переходу необхідно виконати таке:

⇗ Перехід ➢ Open Specification … 🗀 General

Event := **НазваПодії**; Arguments := **СписокПараметрів**

🗀 Detail; Guard Condition := **умова**

// За потреби описати дію у полі Action чи сформувати

// повідомлення у полях Send Event (**НазваПодії**),

// Send Arguments (**СписокПараметрів**); Send target (**Об'єкт**)

❑ OK

Немає потреби створювати діаграми станів для кожного класу, їх використовують тільки у складних випадках. Однак, якщо об'єкт деякого класу може існувати у декількох станах і у кожному з них по-різному поводиться, то для нього така діаграма може бути дуже корисною.

? Запитання для самоперевірки

1. Дайте визначення взаємодії та повідомлення.
2. Як зображають повідомлення на діаграмах?
3. Що таке клієнт? Що таке сервер?
4. Що відображає діаграма кооперацій?
5. Що відображає діаграма послідовностей?
6. Що відображає діаграма станів?
7. Що таке життєва лінія об'єкта?
8. Які форми зображення назви об'єкта використовують на діаграмах взаємодії?
9. Що відображає фокус керування? Як його активізувати/деактивізувати?
10. Коротко охарактеризуйте найуживаніші типи повідомлень на діаграмах взаємодії.
11. Коротко охарактеризуйте найвживаніші типи ролей об'єктів на діаграмах кооперацій.
12. Коротко охарактеризуйте найвживаніші стереотипи зв'язків на діаграмах кооперацій.
13. Що таке стан об'єкта? Як його зображають?
14. Що таке анонімний стан об'єкта?
15. Що таке початковий стан об'єкта? Як його зображають?
16. Що таке кінцевий стан об'єкта? Як його зображають?
17. Які події використовують для характеристики поведінки об'єкта у деякому стані?
18. Як активізують просту внутрішню дію для стану об'єкта?
19. Як активізують долучення події зовнішнього об'єкта для даного стану об'єкта?
20. Опишіть специфікацію найпростішого переходу між об'єктами.
21. Як активізують дію для переходу між об'єктами?

6. ПРОЕКТУВАННЯ АРХІТЕКТУРИ ПРОГРАМНОЇ СИСТЕМИ

План викладу матеріалу:

1. Загальні положення.
2. Діаграми компонентів.
3. Діаграми розміщення.

Ключові терміни розділу

- *Діаграми реалізації*
- *Діаграма компонентів*
- *Діаграма розміщення*
- *Експортований інтерфейс*
- *Компоненти*
- *Вузли мережі*
- *Імпортований інтерфейс*
- *Фізичні канали зв'язку*

6.1. Загальні положення

Важливою властивістю об'єктно-орієнтованого моделювання програмних систем є *погодженість* моделей системи від стадії аналізу до програмних модулів. Ця погодженість моделей забезпечується завдяки застосуванню принципів абстрагування, модульності та поліморфізму на всіх стадіях розробки. Моделі аналізу можуть безпосередньо порівнюватися з моделями реалізації.

Парадигма об'єктно-орієнтованого моделювання полегшила процес взаєморозуміння між розробниками, експертами і замовниками системи.

Стандартною нотацією для моделювання великих програмних систем на основі об'єктно-орієнтованої методології слугує уніфікована мова моделювання (UML). На базі її нотації описуються етапи роботи над проектом, ставляться задачі аналітикам, проектувальникам, програмістам, тестувальникам, системним адміністраторам тощо.

Об'єктно-орієнтована методологія опирається на систему діаграм – одиничних описів фрагментів системи. Різні типи діаграм відображають різні аспекти системи. У кожної діаграми є своя мета і своя аудиторія. Моделювання здійснюється як "порівневий спуск" від *концептуальної* моделі до *логічної*, а потім до *фізичної* моделі програмної системи.

Концептуальну модель зображають як діаграму *прецедентів* (*Use Case diagram*), які слугують для виконання ітераційного циклу загальної постановки задачі разом із замовником. Діаграми прецедентів є основою для досягнення взаєморозуміння між програмістами-професіоналами і "бізнесменами" – замовниками проекту.

Логічна модель дає змогу визначати два різних погляди на системи: статичний і динамічний. Статична модель виражається діаграмами *класів* (*Class diagram*), які є основою для генерації програмного коду цільовою мовою програмування.

Для опису динаміки систем використовуються діаграми *поведінки* (*behavior diagrams*), що підрозділяються на:

- діаграми *станів* (*statechart diagrams*);
- діаграми *активності* (*activity diagrams*);
- діаграми *взаємодії* (*interaction diagrams*), які складаються з:
  - діаграм *послідовностей* (*sequence diagrams*);
  - діаграм *кооперацій* (*collaboration diagrams*).

Фізична модель задається діаграмами *реалізації* (*implementation diagrams*), за допомогою яких описують архітектуру програмної системи. Вони складаються з діаграм *компонентів* (*component diagrams*) і діаграм *розміщення* (*deployment diagrams*).

Компонент – фізичний модуль коду. Ним можуть бути виконавчі модулі коду, бібліотеки, таблиці баз даних, файли і документи. Діаграми компонентів ілюструють вигляд моделі на фізичному рівні. На ній зображають типи компонентів програмного забезпечення системи і залежності між ними, що виникають на етапі компіляції чи у процесі виконання програми.

Діаграми розміщення ілюструють фізичне розміщення різних компонентів системи у *вузлах* мережі (процесорах, фізичних і логічних пристроях). *Процесор* – машина, що має обчислювальну потужність з обробки даних (сервери, робочі станції, мікрочіпи тощо). *Пристрій* – апаратура, що не володіє обчислювальною потужністю (термінали введення/виведення, принтери, сканери тощо).

6.2. Діаграми компонентів

Діаграма компонентів забезпечує узгоджений перехід від логічного уявлення до конкретної реалізації проекту у формі програмного коду. Одні компоненти можуть існувати тільки на етапі компіляції коду, інші – на етапі його виконання.

У багатьох середовищах розробки компонент (або модуль) відповідає окремому файлові. Головними графічними елементами діаграми компонентів є *компоненти*, *інтерфейси* і *залежності* між ними (означення та базові позначення для компонентів та інтерфейсів див. у пункті 2.2). Відношення залежності розглянуто у пунктах 2.3 і 4.4.

Нагадаємо, що відношення залежності слугує для зображення спеціальної форми зв'язку між двома елементами моделі, коли зміна одного елемента моделі впливає на інший елемент моделі або спричинює до його зміни.

На діаграмах компонентів залежності можуть відображати зв'язки окремих файлів програмної системи на етапі компіляції та генерації об'єктного коду. В інших випадках залежність може відображати наявність у *незалежному* компоненті описів класів, які використовуються у *залежному* компоненті для створення відповідних об'єктів.

На діаграмах компонентів залежності можуть також зв'язувати компоненти та імпортовані інтерфейси. Залежність компонента-клієнта від імпортованого інтерфейсу (перетягується з браузера на діаграму) означає, що компонент використовує відповідний інтерфейс у процесі свого виконання.

Водночас на цій же діаграмі може бути й інший компонент, який реалізує цей інтерфейс. Відношення реалізації інтерфейсу сформується, якщо перетягнути інтерфейс з браузера на відповідний компонент.

Наприклад, зображений нижче фрагмент діаграми компонентів (рис. 6.1) відображає інформацію про те, що компонент **`Main`** залежить від *імпортованого* інтерфейсу **`IDialog`**, який реалізується компонентом **`Library`**. У цьому випадку для компонента **`Library`** інтерфейс **`IDialog`** є *експортованим*.

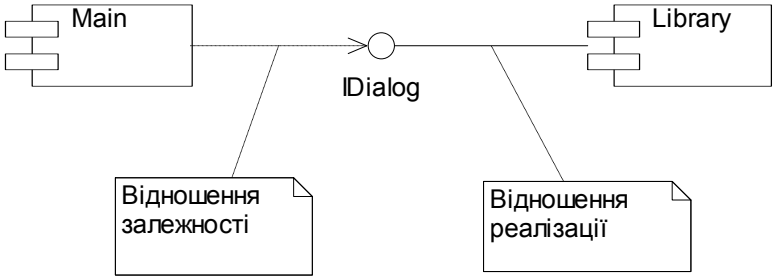


Рис. 6.1. Фрагмент діаграми компонентів

На рис. 6.2 зображено компонент (або модуль) головної програми (у формі піктограми та стереотипу), який позначає файл, що містить кореневу програму. У C++, наприклад, це відповідає файлові з розширенням **.cpp**, що містить привілейовану функцію **main**. Зазвичай, існує один такий модуль на програму.

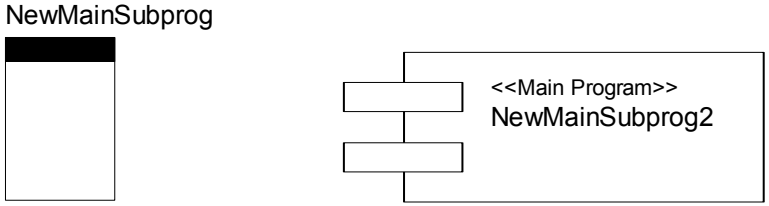


Рис. 6.2. Зображення компонента головної програми

На рис. 6.3 зображено компоненти підпрограм і пакетів (у формі піктограми), які позначать файли, що містять окремі підпрограми чи пакети підпрограм.

Позначка опису (*Spec*) і позначка тіла (*Body*) позначають файли, що містять, відповідно, описи та реалізації. У C++, наприклад, модуль описів відповідає заголовному файлові з розширенням **.h**, а модуль тіла – файлові тексту програми з розширенням **.cpp**.

Кожен модуль повинен мати назву; зазвичай, це назва відповідного фізичного файла у каталозі проекту. Кожна повна назва файла має бути унікальною у каталозі його розміщення. Відповідно до правил конкретного середовища розробки, на назви накладаються й інші обмеження. Отже, кожен модуль містить або опис, або визначення класів та об’єктів, а також інші конструкції мови.

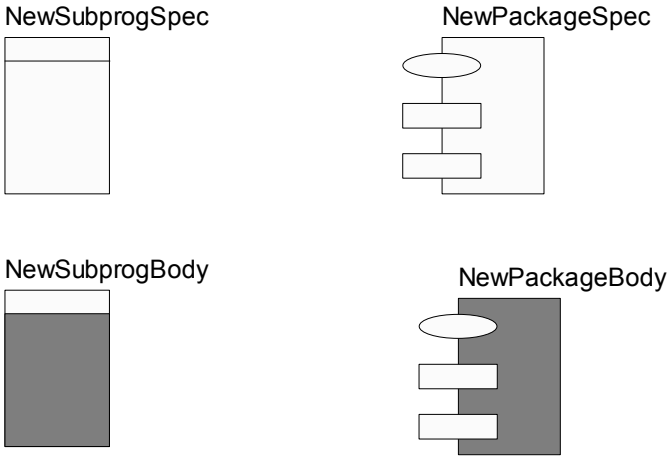


Рис. 6.3. Зображення компонентів підпрограми і пакета

Зазначимо, що в нашому випадку з діаграми компонентів не випливає, що класи реалізовано зазначеним компонентом. Якщо необхідно наголосити, що деякий компонент реалізує окремі класи, то для позначення компонента використовують розширений символ прямокутника. Щодо цього прямокутник компонента ділять на дві секції горизонтальною лінією. Верхня секція слугує для запису імені компонента і, можливо, додаткової інформації, а нижня секція – для вказівки класів, що реалізовуються цим компонентом.

Усередині символу компонента допускається зображати інші елементи графічної нотації, такі як класи (компонент рівня типів) або об'єкти (компонент рівня прикладів). У цьому випадку символ компонента зображають так, щоб вмістити ці додаткові символи.

Об'єкти, які перебувають в окремому компоненті-екземплярі, зображаються вкладеними у символ зазначеного компонента. Подібна вкладеність означає, що виконання компонента вабить виконання операцій відповідних об'єктів. Що стосується доступу до цих об'єктів, то він може бути додатково специфікований за допомогою кванторів видимості, подібно до видимості пакетів. Змістовний сенс видимості може відрізнятися для різних видів пакетів.

6.3. Діаграми розміщення

Діаграма *розміщення* (*deployment diagram*) віддзеркалює фізичні взаємозв'язки між програмними й апаратними компонентами проектованої системи. Ця діаграма є гарним засобом для представлення маршрутів переміщення об'єктів і компонентів у розподіленій системі.

Кожен вузол на діаграмі розміщення є певним обчислювальним пристроєм (здебільшого самостійна частина апаратури). Ця апаратура може бути як простим пристроєм чи датчиком, так і мейнфреймом.

Наприклад, на цій діаграмі, може зображатися персональний комп'ютер, з'єднаний з Unix-сервером за допомогою протоколу TCP/IP. З'єднання між вузлами демонструють фізичні канали зв'язку, за допомогою яких здійснюються взаємодії в системі.

Хоча діаграми розміщення і діаграми компонентів можна зображати окремо, також допускається накладати діаграму компонентів на діаграму розгортання. Це доцільно робити, щоб проілюструвати, які компоненти виконуються і на яких вузлах

Чимало проектувальників творчо підходять до використання цього виду інформації, однак інші будують ці діаграми формально з метою відповідності стандартам мови UML.

? Запитання для самоперевірки

1. Дайте визначення компонента.
2. Як зображають компоненти на діаграмах?
3. Що таке клієнт? Що таке сервер?
4. Що відображає діаграма компонентів?
5. Що відображає діаграма розміщення?
6. Чи можна поєднувати діаграми компонентів і розміщення?
7. Чи можна відображати об'єкти у діаграмі компонентів?
8. Які форми зображення класів використовують на діаграмах компонентів?
9. Як відображають підпрограми та головну програму?
10. Коротко охарактеризуйте типи інтерфейсів на діаграмах компонентів.