МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
Національний аерокосмічний університет ім. М. Є. Жуковського
"Харківський авіаційний інститут"

І. В. Шевченко, Ю. А. Кузнецова

ПРОЕКТУВАННЯ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ. ОСНОВИ ПОБУДОВИ UML-ДІАГРАМ

Навчальний посібник

Харків "ХАІ" 2019

УДК 004.4 (076.5)
Ш37

Рецензенти: канд. техн. наук, доц. О. О. Мазурова,
канд. техн. наук Т. В. Філімончук

Шевченко, І. В.

Ш37 Проектування програмного забезпечення. Основи побудови UML-діаграм [Електронний ресурс] : навч. посіб. / І. В. Шевченко, Ю. А. Кузнецова. – Харків : Нац. аерокосм. ун-т ім. М. Є. Жуков-ського "Харків. авіац. ін-т", 2019. – 82 с.

Наведено опис мови візуального моделювання UML, розробленої для специфікації, візуалізації, проектування та документування компонентів програмного забезпечення, бізнес-процесів та ін. Мова UML одночасно є простим і потужним засобом моделювання, який можна ефективно застосовувати для побудови концептуальних, логічних і графічних моделей складних систем самого різного цільового призначення. Ця мова увібрала в себе найкращі якості методів програмної інженерії, що з успіхом використовувалися протягом останніх років при моделюванні великих і складних систем.

Для студентів усіх форм навчання спеціальності 121 «Інженерія програмного забезпечення».

Іл. 82. Бібліогр.: 10 назв

УДК 004.4 (076.5)

ВСТУП

UML (Unified Modeling Language – уніфікована мова моделювання) – це стандарт системи позначень для побудови діаграм об'єктно-орієнтованого аналізу і проектування (ООАП). Потрібно не тільки освоїти цю систему позначень, але набагато важливіше навчитися «мислити об'єктами». Мова UML – це не принцип або метод ООАП, це всього лише система позначень. Можна досконало освоїти синтаксис діаграм UML і спеціалізовані CASE-засоби, але не навчитися при цьому розробляти нові ефективні програми або модифікувати існуючі.

Однак для ООАП потрібна своя мова, що дозволяє формулювати думки і спілкуватися з іншими розробниками. Ось такою мовою і є UML.

Як мовний засіб UML надає словник і правила комбінування слів у цьому словнику. Мова диктує, як створити і прочитати модель, проте не містить ніяких рекомендацій про те, яку модель системи необхідно створити, – це виходить за рамки UML і є прерогативою процесу розроблення програмного забезпечення.

UML – це мова візуалізації. Написання моделей мовою UML переслідує одну просту мету – полегшення процесу передачі інформації про систему. За кожним символом UML стоїть певна семантика, що дозволяє уникати помилок інтерпретації.

UML – це мова специфікацій і точних визначень. Тому моделювання мовою UML є побудовою моделей, які є точними, недвозначними і повними.

UML – це мова конструювання. UML не є візуальною мовою програмування, але моделі в термінах UML можуть бути відображені на певний набір об'єктно-орієнтованих мов програмування. UML надає можливості прямого (існуюча модель → новий код) і зворотного (існуючий код → нова модель) проектування. Досить часто засоби UML-моделювання реалізують відображення UML-моделей в коди мовами Java, C++, C# та ін.

UML – це мова документування. UML надає засоби відображення вимог до системи, побудови документації, тестів, моделювання необхідних дій для планування проекту та для управління поставленими кінцевому користувачеві релізами.

В UML всі моделі поділяють на такі:

- структурні моделі;
- моделі поведінки.

Структурні моделі призначені для опису статичної структури

сутностей та елементів деякої системи, зокрема класів, інтерфейсів, атрибутів, відношень.

Моделі поведінки – моделі, які призначені для опису процесу функціонування елементів системи, зокрема їх методів та їх взаємодії, а також процесу зміни станів окремих елементів і системи в цілому.

На рис. В.1 показано діаграми мови моделювання UML 1.0.

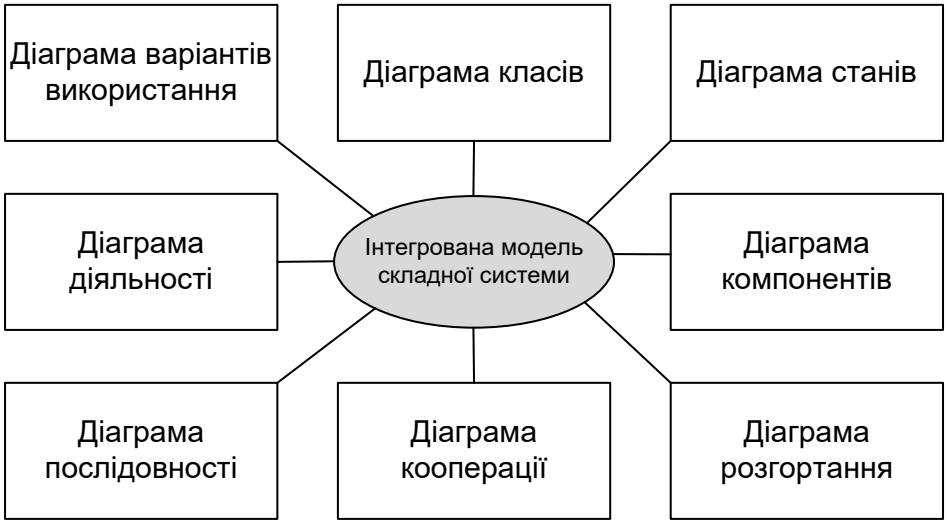


Рисунок В.1 – Діаграми UML 1.0

На рис. В.2 зображено діаграми мови моделювання UML 2.0.

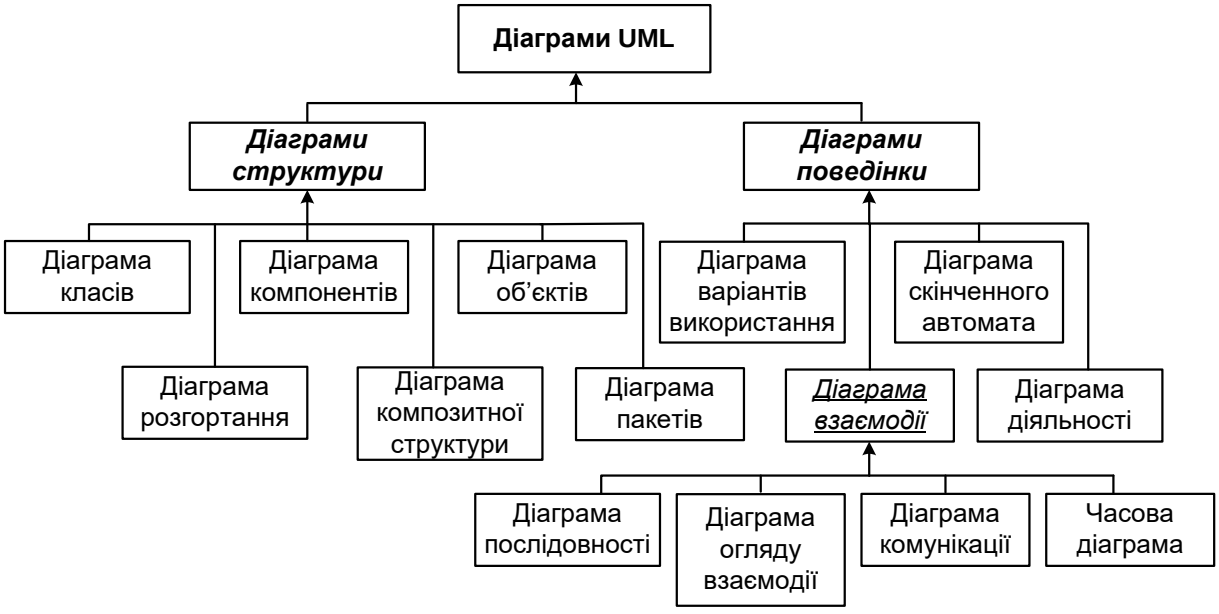


Рисунок В.2 – Діаграми UML 2.0

У рамках цього посібника будуть розглянуті такі діаграми UML, а саме:

діаграма варіантів використання (use case diagram)
діаграма класів (class diagram)
діаграми поведінки (behavior diagrams):
діаграма діяльності (activity diagram)
діаграма станів (statechart diagram)
діаграми взаємодії (interaction diagrams):
діаграма послідовності (sequence diagram)
діаграма кооперації (collaboration diagram)
діаграми реалізації (implementation diagrams):
діаграма компонентів (component diagram)
діаграма розгортання (deployment diagram)

Що ж таке Rational Rose? Це програмний інструментальний засіб від компанії Rational Software для візуального об'єктно-орієнтованого моделювання систем на основі класів і їх взаємодії, або можна сформулювати більш спрощено: це візуальний редактор, що дозволяє моделювати програмні системи будь-якої складності на основі графічних діаграм мови UML.

Розглянемо переваги від застосування Rational Rose:

- скорочення циклу розроблення програми «замовник – програміст – замовник» (тепер замовнику немає необхідності чекати першої альфа-версії, щоб переконатися, що все робиться зовсім не так, як він очікував);
- збільшення продуктивності роботи програмістів (менше ручного кодування → менше помилок, менше помилок → менше налагодження, менше налагодження → більша продуктивність);
- поліпшення споживчих якостей програм, які створюються з орієнтацією на кінцевих користувачів і бізнес;
- здатність вести великі проекти і групи проектів;
- можливість повторного використання вже створеного програмного забезпечення (ПЗ) шляхом акцентування на детальному розбиранні їх архітектури і компонентів;
- здатність мови UML бути універсальним «містком» між різними учасниками процесу створення програмного забезпечення.

Зауважимо, що Rational Rose не створює готовий вихідний код. Rational Rose може допомогти створити основу для системи, наприклад, заготовки класів разом з їх взаємодією (підтримка прямого проектування), а наповнювати методи змістом має програміст. Але, виправивши щось навіть у структурі класів, програміст завжди зможе

отримати візуальне відображення цих змін у Rational Rose (підтримка зворотного проектування).

Даний посібник присвячено опису синтаксису наведених вище діаграм UML, які є базовими серед великої їх кількості, а також допоможе навчитися будувати їх за допомогою інструментального засобу Rational Rose.

Для побудови UML-діаграм можна використовувати й інші програмні засоби. Все більш популярними стають графічні UML онлайн-редактори. Але треба зазначити, що на відміну від Rational Rose вони дозволяють тільки будувати UML-діаграми і не підтримують процес кодогенерації та зворотного процесу проектування (від коду до моделей).

Наведемо найбільш популярні засоби для побудови UML-діаграм: Draw.io, Gliffy, LucidChart, Cacoo, Creately, GenMyModel, Microsoft Visio, Diagramo, Visual Paradigm, Umbrello. Деякі з них також підтримують роботу в режимі реального часу і дозволяють створювати колективні UML-діаграми.

1 РОЗРОБЛЕННЯ ДІАГРАМИ ВАРІАНТІВ ВИКОРИСТАННЯ

Основне призначення *діаграми варіантів використання* (use case diagram) – *графічне подання функціональних вимог, які були надані замовником, до ПЗ, що розробляється.*

Діаграма варіантів використання відноситься до статичного подання проектованої системи і показує відношення (зв'язки) між варіантами використання (прецедентами) і дійовими особами.

Синтаксис діаграми варіантів використання є досить простим, вона легко може бути прочитана замовником ПЗ, а отже, її можна використовувати для узгодження функціональних вимог до системи, адже кожний варіант використання (прецедент) є окремою функціональною вимогою.

Варіант використання (прецедент)

Варіант використання (прецедент) є послідовністю (сценарієм) дій (транзакцій), які виконуються системою у відповідь на подію, що ініціює деякий зовнішній об'єкт (дійова особа).

Варіант використання описує типову взаємодію користувача і системи. Наведемо приклад двох типових варіантів використання звичайного текстового редактора: «Зробити виділений текст напівжирним» або «Видалити абзац тексту». Навіть на такому

простому прикладі можна виділити ряд властивостей варіанта використання:

- він охоплює деяку очевидну для користувачів функцію, яка може бути як невеликою, так і досить великою;
- вирішує для користувача деяку дискретну задачу.

У найпростішому випадку варіант використання визначають у процесі обговорення з користувачем тих функцій, які він хотів би реалізувати за допомогою програмного забезпечення, що розробляється.

Окремий варіант використання позначається еліпсом, усередині якого міститься його коротка назва або ім'я (рис. 1.1). Варіанти використання зазвичай називають, починаючи з дієслова або віддієслівного іменника, описуючи при цьому, що користувач бачить як кінцевий результат процесу.

Варіанти використання описують, що має буде робити система. Щоб фактично розробити систему, однак, будуть потрібні більш конкретні деталі. Ці деталі описуються в документі, званому "потік подій" (flow of events). Метою потоку подій є документування процесу оброблення даних, що реалізується в рамках варіанта використання. Цей документ детально описує, що будуть робити користувачі системи, і що – сама система.

Рисунок 1.1 – Приклад варіантів використання

Хоча потік подій і описується детально, він також не має залежати від реалізації. Мета формулювання потоку подій – описати, що буде робити система, а не як вона буде робити це.

Зазвичай потік подій містить:

- короткий опис;
- передумова (pre-conditions);
- основний потік (сценарій) подій;
- альтернативний потік (сценарій) подій;
- постумова (post-conditions).

Послідовно розглянемо ці складові частини потоку подій.

Опис. Кожен варіант використання повинен мати пов'язаний з ним короткий опис того, що він буде робити.

Передумова. Передумова варіанта використання – це такі умови, які мають бути виконані, перш ніж варіант використання почне виконуватися сам. Наприклад, такою умовою може бути виконання іншого варіанта використання або наявність у користувача прав доступу, необхідних для запуску цього. Не у всіх варіантів використання бувають попередні умови.

Діаграми варіантів використання не повинні відображати порядок їх виконання. За допомогою передумов, проте, можна документувати і таку інформацію. Наприклад, передумовою одного варіанта використання може бути те, що в цей час має виконуватися інший.

Основний і альтернативний потоки подій. Конкретні деталі варіантів використання описуються в основному і альтернативних потоках подій. Потік подій поетапно описує, що має відбуватися під час виконання закладеної у варіанти використання функціональності. Потік подій акцентує увагу на тому, що буде робити система, а не як вона буде робити це, причому описує все це з точки зору користувача. Основний і альтернативний потоки подій містять такий опис:

- спосіб запуску варіанта використання;
- різні шляхи виконання варіанта використання;
- основний потік подій варіанта використання;
- відхилення від основного потоку подій (так звані альтернативні потоки);
- потоки помилок;
- спосіб завершення варіанта використання.

Постумова. Постумовою називають такі умови, які завжди мають бути виконані після завершення варіанта використання. Наприклад, в кінці варіанти використання можна помітити прапорцем якийсь перемикач. Інформація такого типу входить до складу постумови. Як і для передумов, за допомогою постумов можна вводити інформацію про порядок виконання варіантів використання системи. Якщо, наприклад, після одного варіанта використання має завжди виконуватися інший, це можна описати як постумову. Такі умови є не у кожного варіанта використання.

Наведемо приклад опису основного і альтернативного сценаріїв для варіанта використання «Зняти гроші з рахунку».

Передумова

Варіант використання починається, коли клієнт вставляє свою картку в банкомат.

Основний потік

1. На екрані банкомата виводиться вітання і пропонується клієнту ввести свій PIN-код.
2. Клієнт вводить PIN-код.
3. Виконується процедура підтвердження введеного PIN-коду. Якщо PIN-код не підтверджений, виконується альтернативний потік подій А1.
4. На екрані банкомата виводиться список доступних дій.
5. Клієнт вибирає пункт "Зняти гроші з рахунку".
6. На екрані банкомата виводиться повідомлення із запитом суми грошей для зняття.
7. Клієнт вводить необхідну суму.
8. Виконується процедура визначення, чи є на рахунку достатньо грошей. Якщо грошей недостатньо, виконується альтернативний потік А2.
9. Необхідна суму грошей віднімається з рахунку клієнта.
10. Банкомат видає клієнтові необхідну суму готівкою.
11. На екрані банкомата виводиться повідомлення із запитом на друкування квитанції. Якщо клієнт вибирає друкування квитанції, то банкомат друкує квитанцію.
12. Банкомат повертає клієнту його картку.
13. Варіант використання завершується.

Альтернативний потік А1. *Введення неправильного PIN-коду*

1. Банкомат інформує клієнта, що PIN-код введено неправильно.
2. Банкомат повертає клієнту його картку.
3. Варіант використання завершується.

Альтернативний потік А2. *Недостатньо грошей на рахунку*

1. Банкомат інформує клієнта, що грошей на його рахунку недостатньо.
2. Банкомат повертає клієнту його картку.
3. Варіант використання завершується.

Треба зазначити, що сформульовані потоки подій стануть основою для побудови системних тестів, що будуть перевіряти розроблене ПЗ на відповідність функціональним вимогам.

Дійова особа (актор)

Дійова особа (актор, виконавець) – це роль, яку користувач відіграє відносно системи. Дійові особи являють собою ролі, а не конкретних людей або найменування робіт. Графічне позначення діючої особи показано на рис. 1.2.

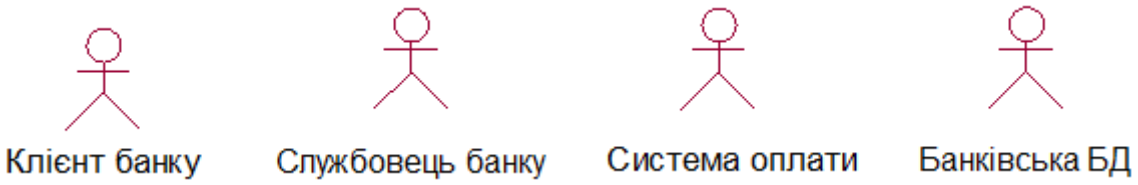


Рисунок 1.2 – Приклад дійових осіб

Незважаючи на те, що дійова особа зображується у вигляді стилізованої людської фігурки, дійовою особою може також бути зовнішня система, якій необхідна деяка інформація від даної системи («Система оплати», «Банківська БД»). Показувати на діаграмі дійових осіб слід тільки в тому випадку, коли їм дійсно необхідні деякі варіанти використання.

Дійові особи ділять на три основних типи:

- користувачі системи;
- інші системи, які взаємодіють із даною;
- час.

Перший тип дійових осіб – це фізичні особи або користувачі системи. Вони найбільш типові і є практично в кожній системі. Наприклад, до цього типу належать клієнти і обслуговуючий персонал. Називаючи дійових осіб, використовують їх рольові імена, а не ті, що відповідають їх посаді. Конкретна людина може відігравати безліч ролей. Рольові, а не посадові імена дозволяють отримати більш стабільну картину дійових осіб. Посади можуть змінюватися час від часу, при цьому ролі і відповідальності можуть переміщатися від однієї посади до іншої. Використовуючи ролі для назви дійових осіб, не доведеться оновлювати модель кожен раз при появі нової посади або при зміні розподілу обов'язків між ними.

Другим типом дійових осіб є інша система. Припустимо, що у банку є програмне забезпечення «Система оплати», яку

використовують для роботи з інформацією про рахунки клієнтів. Система роботи банкомата, що проектується, повинна мати можливість взаємодіяти з «Системою оплати», в такому випадку остання стає дійовою особою.

Зробивши систему дійовою особою, ми припускаємо, що вона не буде змінюватися взагалі (за винятком її інформаційного наповнення). Такі дійові особи знаходяться поза сферою дії того, що ми розробляємо і, таким чином, не підлягають контролю з нашого боку. Якщо ми збираємося змінювати або розробляти також і «Систему оплати», вона потрапляє при цьому в наш проект і, таким чином, не може бути показана як дійова особа.

Третій найбільш поширений тип дійової особи – це час. Час стає дійовою особою, якщо від нього залежить запуск будь-яких подій в системі. Система, наприклад, може кожного дня опівночі виконувати будь-які службові процедури з налаштування і узгодження своєї роботи. Через те, що час не підлягає нашому контролю, він є дійовою особою.

Види відношень

Для побудови діаграми варіантів використання необхідно встановити зв'язки (відношення) між акторами і варіантами використання.

В UML є кілька стандартних видів відношень між акторами і варіантами використання:

- асоціації (association relationship);
- узагальнення (generalization relationship);
- залежності:
    - включення (include relationship);
    - розширення (extend relationship).

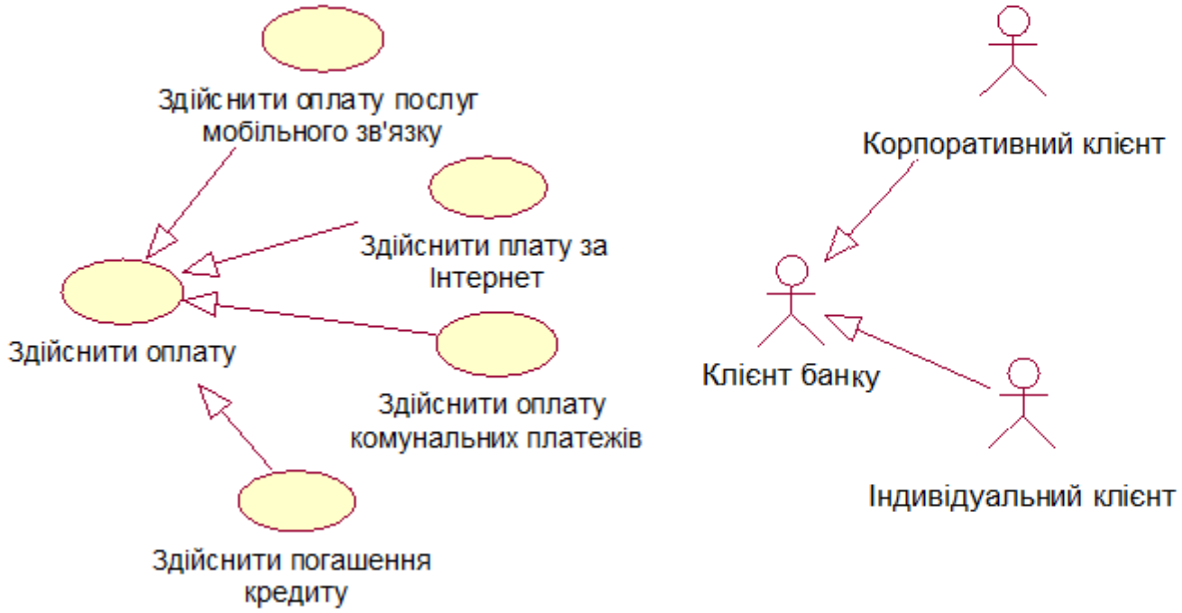
Відношення асоціації. Відношення асоціації вказує на наявність зв'язку між дійовою особою і варіантом використання (рис. 1.3). Для встановлення зв'язку асоціації використовують безперервну лінію з простим наконечником (або зовсім без нього).

Рисунок 1.3 – Відношення асоціації

Відношення узагальнення. Відношення узагальнення між двома варіантами використання (або двома діючими особами) вказує на те, що у кількох варіантах використання (або дійових осіб) є загальні риси (рис. 1.4).

Для встановлення зв'язку узагальненння використовують безперервну лінію з трикутним наконечником.

Зазначимо, що відношення узагальнення є єдиним видом відношення, яке може бути показано між акторами на діаграмі варіантів використання. *Всі інші види відношень між акторами є неприпустимими!*



а *б*

Рисунок 1.4 – Відношення узагальнення:
а – між варіантами використання, б – між акторами

Відношення включення. Відношення включення між двома варіантами використання вказує на те, що один варіант використання *обов'язково (тобто завжди)* використовує функціональність іншого варіанта використання.

Для встановлення зв'язку включення використовують пунктирну лінію з простим наконечником.

На рис. 1.5 показано приклад такого відношення. Тут варіант використання «Зняти гроші з рахунку» має аутентифікувати (розпізнати) клієнта банку, перш ніж дозволить виконання транзакції зі зняття грошей з рахунку.

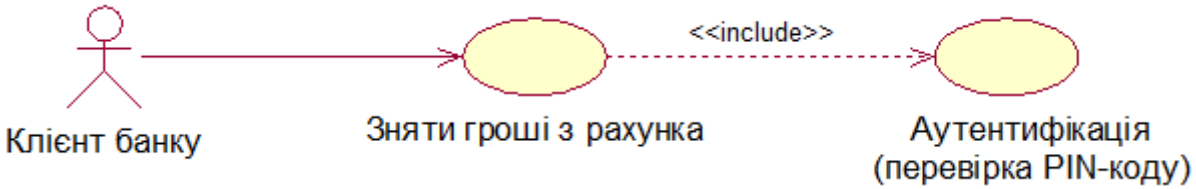


Рисунок 1.5 – Відношення включення

Відношення розширення. Відношення розширення між двома варіантами використання дозволяє *тільки з необхідності* одному варіанту використання задіяти функціональні можливості, що надаються іншим варіантом використання.

Для встановлення зв'язку включення використовують пунктирну лінію з простим наконечником.

На рис. 1.6 показано приклад такого відношення. Тут варіант використання «Зняти гроші з рахунка» іноді застосовує функціональні можливості, що надаються варіантом використання «Друк квитанції». Це відбувається тоді і тільки тоді, коли клієнт банку вибирає пункт «Друк квитанції» під час роботи варіанта використання «Зняти гроші з рахунка».

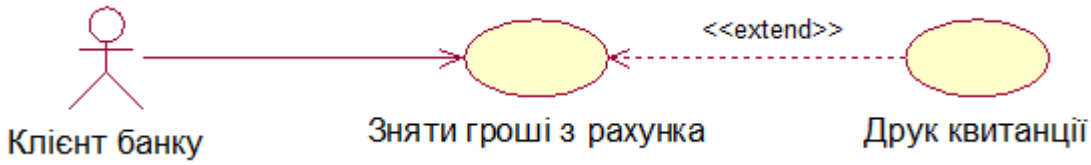


Рисунок 1.6 – Відношення розширення

Зверніть увагу на напрямок стрілок для відношення включення та розширення. Напрямок відповідних стрілок – різний! Легко не заплутатися, якщо пам'ятати, що напрямок стрілок збігається з напрямком "читання" зв'язків між варіантами використання. "Щоб зняти гроші, необхідно **обов'язково** пройти аутентифікацію, тобто ввести PIN-код". Саме слово "обов'язково" свідчить про те, що між двома варіантами використання необхідно поставити відношення включення. "Роздрукування квитанції є **можливим**, але не обов'язковим, при знятті грошей з рахунка". Слово "можливим" свідчить про те, що між двома варіантами використання необхідно поставити відношення розширення.

При встановленні відношення включення два варіанти використання мають вхідну стрілку. При встановленні відношення розширення один із варіантів використання (що розширює функціональність основного варіанта використання) не має вхідної стрілки.

Правила побудови діаграми варіантів використання

Розробляючи діаграми варіантів використання, необхідно дотримуватися таких правил (рис. 1.7):

- не моделювати зв'язки між діючими особами (за визначенням дійові особи знаходяться поза сферою дії системи);
- не з'єднувати стрілкою два варіанти використання безпосередньо (крім випадків зв'язків включення, розширення та узагальнення), тому що діаграми даного типу описують тільки, які варіанти використання доступні системі, а не порядок їх виконання (для відображення порядку виконання варіантів використання застосовують діаграми інших типів);
- кожен варіант використання має бути ініційований дійовою особою, тобто завжди має бути стрілка, що починається на діючій особі, і закінчується на варіанті використання (винятком є розглянуті зв'язки включення, розширення, узагальнення).

2 РОЗРОБЛЕННЯ ДІАГРАМИ ДІЯЛЬНОСТІ

При моделюванні поведінки системи, що проектується або аналізується, виникає необхідність деталізувати особливості алгоритмічної і логічної реалізації виконуваних системою операцій. Традиційно для цієї мети використовували блок-схеми алгоритмів.

Діаграма діяльності (activity diagram) в UML дуже схожа на блок-схему алгоритму, вони мають однакову мету: обидві відображають певний алгоритм, тобто послідовність дій, виконання яких приводить до отримання бажаного результату.

Але діаграма діяльності – це щось більше, ніж просто блок-схема, вона має суттєві переваги. Головним недоліком блок-схем є неможливість відобразити паралельність виконання операцій, вони можуть показати тільки послідовний процес виконання операцій.

Графічно діаграму діяльності подають у формі графа діяльності. Граф діяльності є різновидом скінченного автомата, вершинами якого є певні діяльності (дії), а переходи відбуваються по закінченні цих діяльностей (дій).

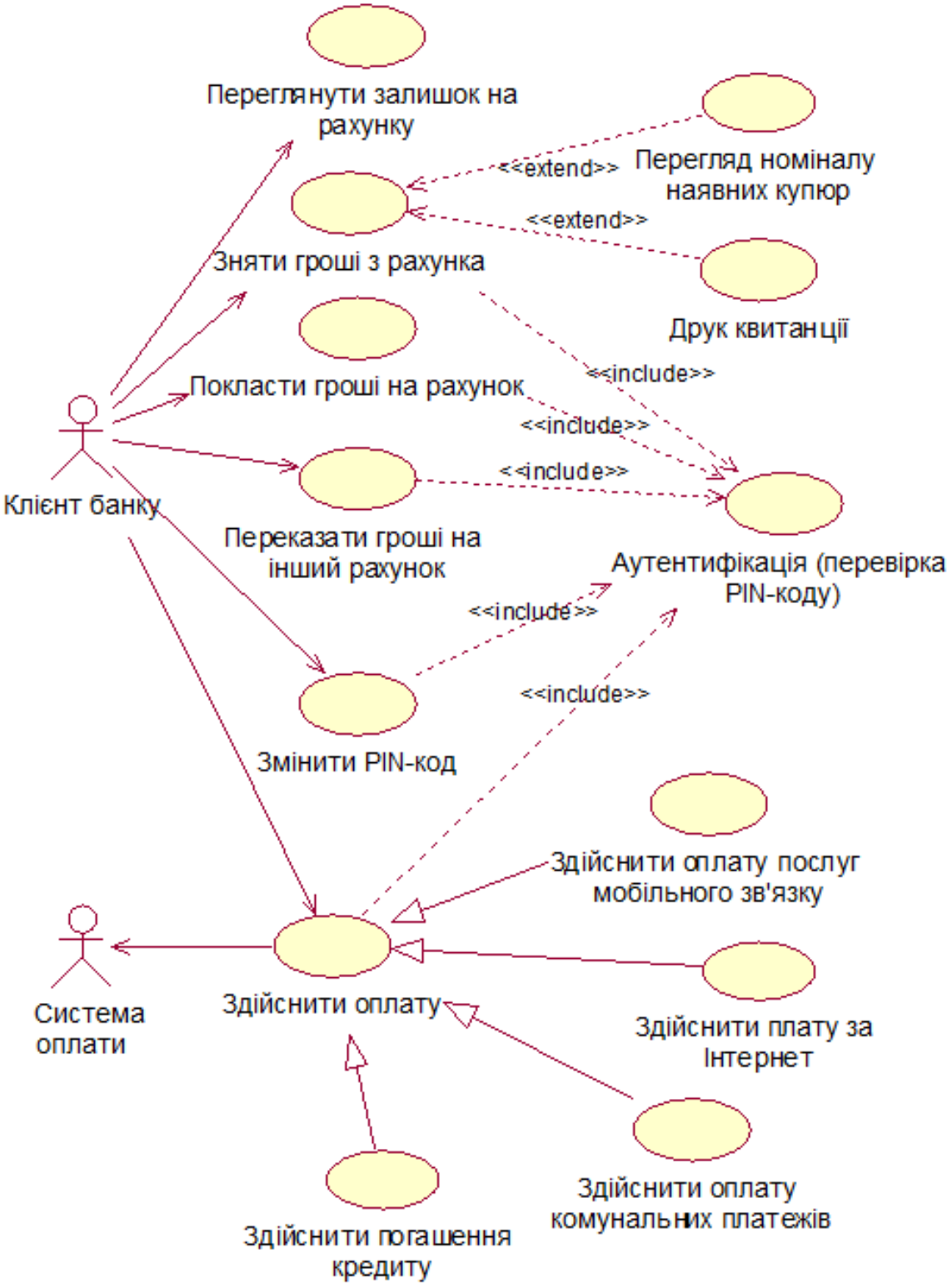


Рисунок 1.7 – Приклад діаграми варіантів використання для програмної системи "Банкомат"

Дія (англ. action) є фундаментальною одиницею поведінки. Дія

отримує множину вхідних сигналів і перетворює їх на множину вихідних сигналів. Одна із цих множин, або обидві водночас, можуть бути порожніми.

Діяльність містить сукупність дій. Кожна дія в діяльності може виконуватися один, два або більше разів під час одного виконання діяльності.

В контексті мови UML діяльність (activity) являє собою деяку сукупність окремих обчислень, виконуваних автоматом. При цьому окремі елементарні обчислення можуть приводити до деякого результату або дії (action).

На діаграмі діяльності відображається логіка або послідовність переходу від однієї діяльності до іншої, при цьому увагу фіксують на результаті діяльності. Сам же результат може приводити до зміни стану системи або повернення деякого значення.

Стан діяльності (дії)

Діаграма діяльності є скінченним автоматом, тому така діаграма повинна мати скінченну множину станів і переходів між ними.

Стани на діаграмі діяльності можуть бути наведені у вигляді станів діяльності або станів дій.

Чим відрізняється дія від діяльності? І дія, і діяльність – це різновиди активності (поведінки).

Дія – це атомарна (тобто неподільна) одиниця поведінки, для якої характерне таке:

- вона відбувається миттєво;
- її не можна перервати;
- вона завжди закінчується.

Діяльність – це неатомарна (тобто може мати внутрішню структуру) одиниця поведінки, для якої характерне таке:

- відбувається деякий час;
- її можна перервати;
- вона завжди закінчується.

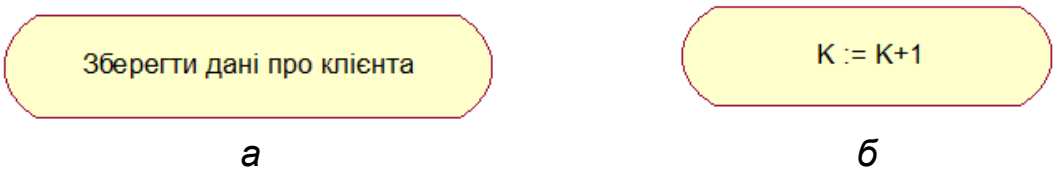
Діяльність – це складена одиниця поведінки, яка в свою чергу може складатися з інших більш простих діяльностей і т.д. Найпростішими діяльностями, які не можна поділити (деталізувати), є елементарні діяльності, тобто дії.

Стан дії (Action state) або стан діяльності (activity state) є спеціальним випадком стану, вони повинні мати одну вхідну дію і, принаймні, одну вихідну дію.

Знаходження в стані дії або стані діяльності неявно передбачає,

що вхідна дія вже завершилася.

На діаграмі діяльності стани позначають округленими прямокутниками (рис. 2.1).



а *б*

Рисунок 2.1 – Графічне зображення станів:
а – стан діяльності, б – стан дії

Кожна діаграма діяльності повинна мати єдиний початковий і єдиний кінцевий стани (рис. 2.2). Кожна діаграма діяльності починається в початковому стані і закінчується в кінцевому стані.

Рисунок 2.2 – Початковий та кінцевий стани

Саму діаграму діяльності прийнято розташовувати таким чином, щоб дії слідували зверху вниз. Для зручності візуального подання на діаграмі діяльності допускається зображати декілька кінцевих станів (тоді їх вважають еквівалентними один одному).

Переходи

Перехід переводить діяльність (дію) в подальший стан відразу, як тільки закінчиться діяльність (дія) в попередньому стані.

Якщо зі стану виходить єдиний перехід, то він може існувати не позначеним. Якщо ж таких переходів декілька, то спрацювати може тільки один із них. Саме в цьому випадку для кожного з таких переходів має бути явно записана в прямих дужках умова спрацьовування (охоронна умова). При цьому для всіх переходів, що виходять з деякого стану, має виконуватися вимога істинності тільки для одного з них. Подібний випадок зустрічається тоді, коли діяльність, що виконується послідовно, має розділитися на альтернативні гілки залежно від

значення деякого проміжного результату. Така ситуація отримала назву розгалуження (рис. 2.3).

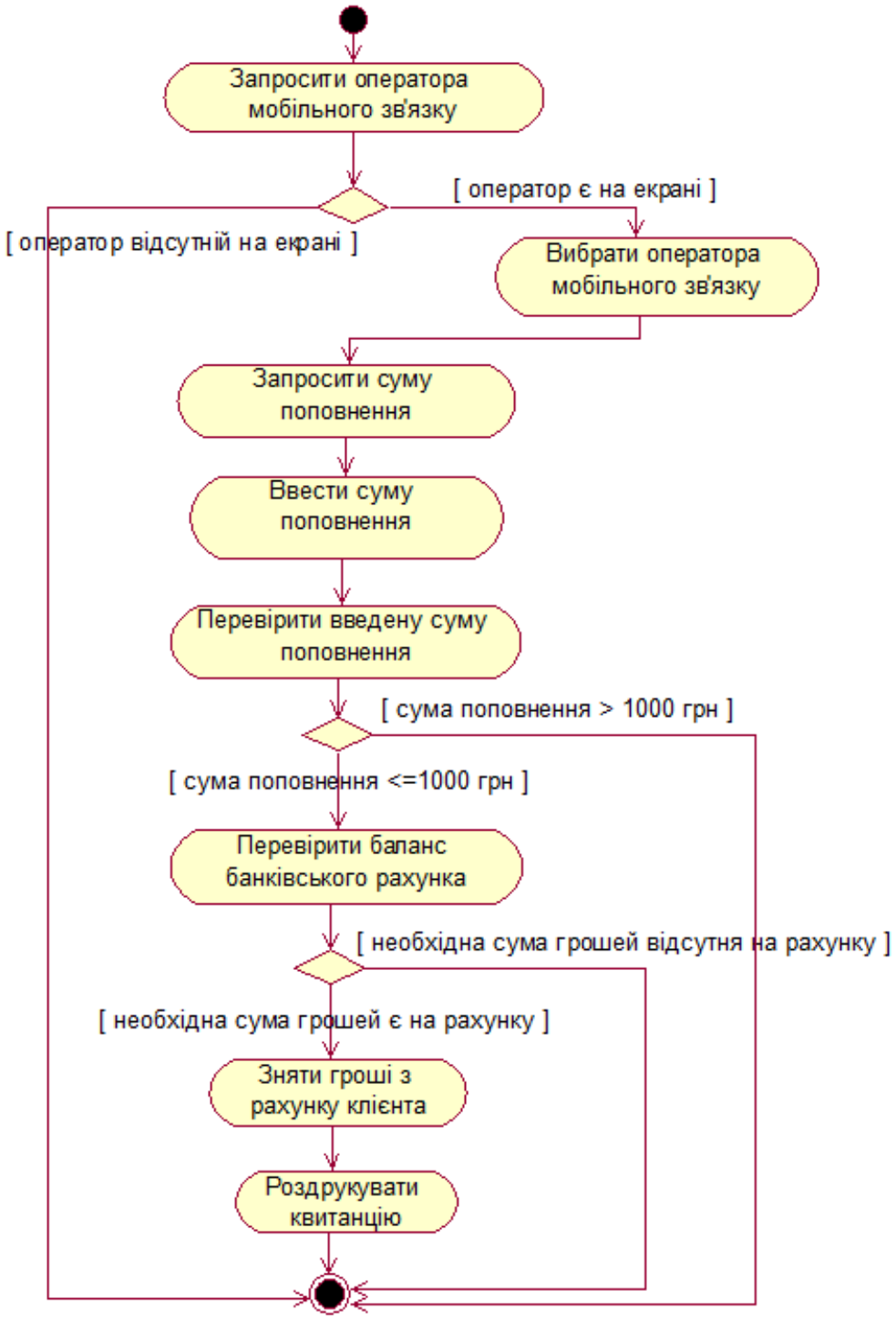
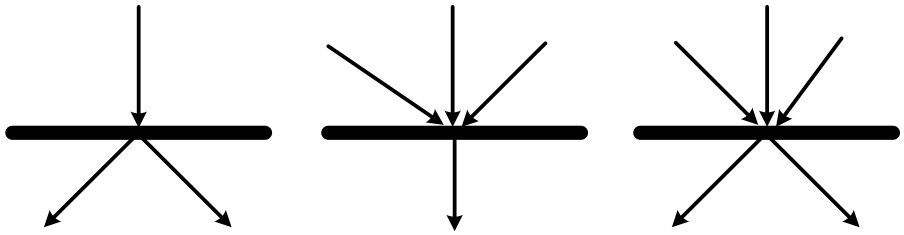


Рисунок 2.3 – Приклад розгалужень на діаграмі діяльності

Лінії синхронізації

Один із найбільш значущих недоліків звичайних блок-схем або структурних схем алгоритмів пов'язаний з проблемою зображення паралельних гілок окремих обчислень. Розпаралелювання обчислень істотно підвищує загальну швидкодію програмних систем. У мові UML для цієї мети використовують спеціальний символ для розділення і злиття паралельних обчислень або потоків управління.

При цьому розділення має один вхідний перехід і кілька вихідних, злиття, навпаки, має кілька вхідних переходів і один вихідний (рис. 2.4, а, б). Зазначимо, що лінія синхронізації не може мати декілька вхідних і одночасно декілька вихідних переходів (рис. 2.4, в).



а *б* *в*

Рисунок 2.4 – Лінії синхронізації: а – розділення; б – злиття; в – помилкове використання

На рис. 2.5 показано, що після виконання дії А запускаються два паралельні потоки. Дії С і D виконуються паралельно з дією В. Перш ніж виконається дія Е, паралельні потоки мають бути завершені.

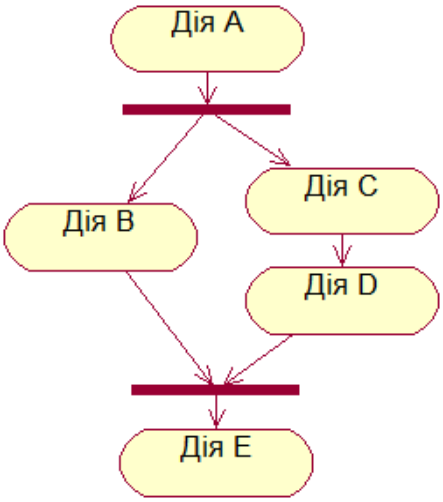


Рисунок 2.5 – Графічне зображення розподілу і злиття паралельних потоків управління

Доріжки

Діаграми діяльності можуть бути використані не тільки для специфікації алгоритмів обчислень або потоків управління в програмних системах. Не менш важливою областю їх застосування є моделювання бізнес-процесів.

Дійсно, діяльність будь-якої компанії (фірми) також є не що інше, як сукупність окремих активностей (робіт, операцій), спрямованих на досягнення необхідного результату.

Однак для аналізу бізнес-процесів бажано виконання кожної активності асоціювати з конкретним підрозділом компанії. В цьому випадку підрозділ несе відповідальність за реалізацію окремих активностей (робіт, операцій), а сам бізнес-процес подається у вигляді переходів активностей від одного підрозділу до іншого.

Для моделювання цих особливостей в мові UML використовується спеціальна конструкція, яка отримала назву доріжки (swimlanes).

Мається на увазі візуальна аналогія з плавальними доріжками в басейні, якщо дивитися на відповідну діаграму. При цьому всі стани дій на діаграмі діяльності ділять на окремі групи, які відділяють одна від одної вертикальними лініями. Дві сусідні лінії і утворюють доріжку, а група станів між цими лініями виконується окремим підрозділом (відділом, групою, відділенням, філією) компанії (рис. 2.6).

3 РОЗРОБЛЕННЯ ДІАГРАМИ КЛАСІВ

Діаграма класів (class diagram) служить для подання статичної структури моделі системи в термінології класів об'єктно-орієнтованого програмування.

Діаграма класів може відображати, зокрема, різні взаємозв'язки окремих сутностей предметної області, такими, як об'єкти і підсистеми, а також описує їх внутрішню структуру і типи відношень.

На даній діаграмі не вказують інформацію про тимчасові аспекти функціонування системи. З цієї точки зору діаграма класів є подальшим розвитком концептуальної моделі системи, що проектується.

Діаграма класів є графом, вершинами якого є елементи типу "класифікатор", які пов'язані різними типами структурних відношень. Слід зауважити, що діаграма класів може також містити інтерфейси, пакети, відношення і навіть окремі екземпляри, такі, як об'єкти і зв'язки. Коли говорять про цю діаграму, то мають на увазі статичну структурну

модель проектованої системи. Тому діаграму класів прийнято вважати графічним поданням таких структурних взаємозв'язків логічної моделі системи, що не залежать від часу (є інваріантними відносно часу).

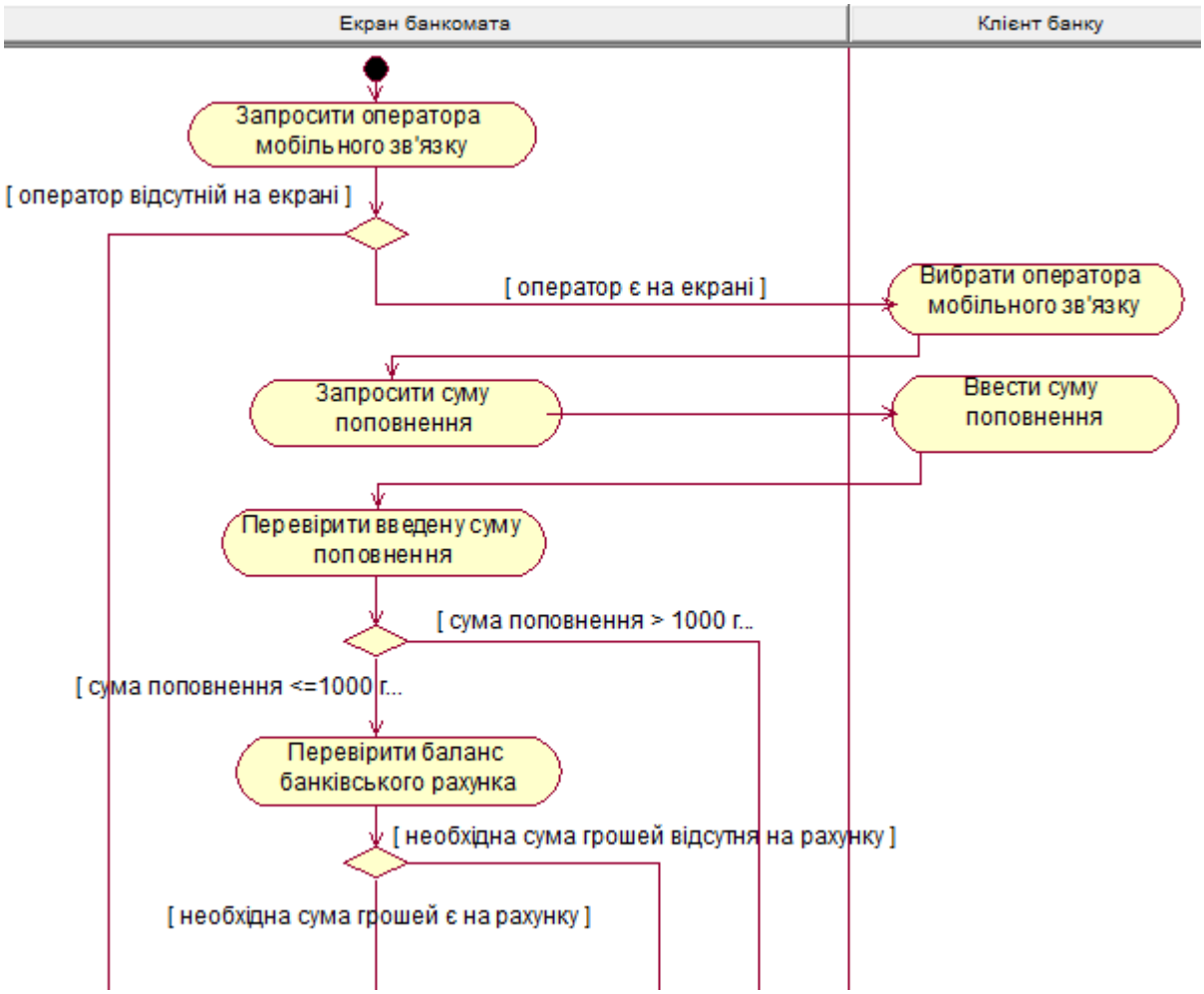
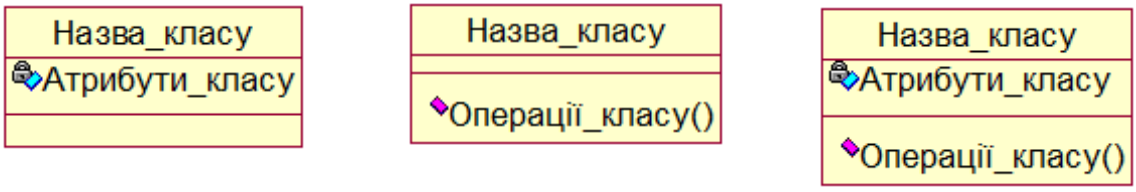


Рисунок 2.6 – Графічне зображення фрагмента діаграми діяльності з доріжками

Клас

Клас (class) у UML служить для позначення множини об'єктів, які мають однакову структуру, поведінку і відношення з об'єктами інших класів.

Графічно клас зображується у вигляді прямокутника, який додатково може бути розділений горизонтальними лініями на розділи або секції (рис. 3.1). У цих розділах можуть зазначати назву класу, атрибути (змінні) і операції (методи).



а б в

Рисунок 3.1 – Графічне зображення класу на діаграмі класів

Обов'язковим елементом позначення класу є його назва. На початкових етапах розроблення діаграми окремі класи можуть позначатися простим прямокутником із зазначенням тільки назви відповідного класу (рис. 3.2, а). Відповідно до опрацювання окремих компонентів діаграми опис класів доповнюється атрибутами (рис. 3.2, б) і операціями (рис. 3.2, в).

Передбачається, що остаточний варіант діаграми містить найбільш повний опис класів, які складаються із заповнених трьох розділів або секцій (рис. 3.2, г).

Навіть якщо секція атрибутів і операцій є зайвою, в позначенні класу вона виділяється горизонтальною лінією, щоб відразу відрізнити клас від інших елементів мови UML.

а б в

г

Рисунок 3.2 – Приклади графічного зображення класів на діаграмі

Назва класу. Рекомендується як назви класів використовувати іменники, записані без пробілів. Необхідно пам'ятати, що саме назви класів утворюють словник предметної області при ООАП. Назва класу має бути унікальним у рамках пакета, в якому розташований даний клас.

Клас може не мати екземплярів або об'єктів. У цьому випадку його називають абстрактним класом, а для позначення його імені використовують похилий шрифт (курсив). У мові UML прийнято спільну угоду про те, що будь-який текст, який відноситься до абстрактного елемента, записується курсивом. Ця обставина є семантичним аспектом опису відповідних елементів мови UML.

У деяких випадках необхідно явно вказати, до якого пакета належить той чи інший клас. Для цієї мети використовують спеціальний символ роздільник – подвійна двокрапка "::". Синтаксис рядка назви класу в цьому випадку буде таким:

<назва_пакета> :: <назва_класу>

Іншими словами, перед назвою класу має бути явно вказана назва пакета, до якого його слід віднести. Наприклад, якщо визначено пакет з назвою "Банк", то клас "Банківський_рахунок" у цьому банку може бути записаний у вигляді: "Банк :: Банківський_рахунок ".

Атрибути класу. У другій секції прямокутника класу записують його атрибути (attributes) або властивості. У мові UML прийнято певну стандартизацію запису атрибутів класу, яка підпорядковується деяким синтаксичним правилам. Кожному атрибуту класу відповідає окремий рядок тексту, яка складається з квантора видимості атрибута, назви атрибута, його кратності, типу значень атрибута і, можливо, його початкового значення:

<квантор_видимості> <назва_атрибута> [кратність]:
<тип_атрибута> = <початкове значення> {рядок-властивість}

Квантор видимості може набувати одного з трьох можливих значень і, відповідно, відображається за допомогою спеціальних символів:

- символ "+" позначає атрибут з областю видимості типу загальнодоступний (public), такий атрибут доступний або видний з будь-якого іншого класу пакета, в якому визначено діаграму;
- символ "#" позначає атрибут з областю видимості типу захищений (protected), такий атрибут недоступний або не видний для всіх класів, за винятком підкласів даного класу;

− символ "-" позначає атрибут з областю видимості типу закритий (private), такий атрибут недоступний або не видний для всіх класів без виключення.

Квантор видимості може бути опущений. У цьому випадку його відсутність просто означає, що видимість атрибута не вказується. Ця ситуація відрізняється від прийнятих за замовчуванням угод у традиційних мовах програмування, коли відсутність квантора видимості трактується як public або private. Однак замість умовних графічних позначень можна записувати відповідне ключове слово: public, protected, private.

На рис. 3.3 показано позначення кванторів видимості в Rational Rose.

Назва атрибута є рядком тексту, який використовують як ідентифікатор відповідного атрибута, і тому має бути унікальним у межах даного класу. Назва атрибута є єдиним обов'язковим елементом синтаксичного позначення атрибута.

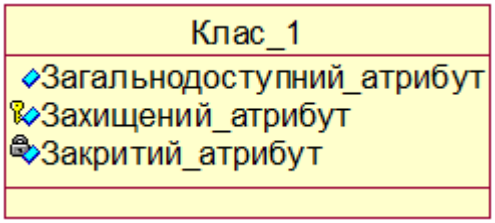


Рисунок 3.3 – Відображення кванторів видимості в Rational Rose

Кратність атрибута характеризує загальну кількість конкретних атрибутів даного типу, що входять до складу окремого класу. У загальному випадку кратність записується в формі рядка тексту в квадратних дужках після назви відповідного атрибута:

[Нижня_межа1 .. верхня_межа1, нижня_межа2 .. верхня_межа2, ..., нижня_межаk .. верхня_межаk],

де нижня_межа і верхня_межа є додатними цілими числами.

Як приклад розглянемо такі варіанти завдання кратності атрибутів:

[0..1] – кратність атрибута може набувати значення 0 або 1 (0 означає відсутність значення для даного атрибута);

[0 .. *] – кратність атрибута може набувати будь-якого додатного цілого значення, яке більше нуля або дорівнює йому (ця кратність може бути записана коротше у вигляді простого символу – [*]);

[1 .. *] – кратність атрибута може набувати будь-якого додатного цілого значення, яке більше одиниці або дорівнює їй;

[1..5] – кратність атрибута може набувати будь-якого значення з чисел: 1, 2, 3, 4, 5;

[1..3,5,7] – кратність атрибута може набувати будь-якого значення з чисел: 1, 2, 3, 5, 7;

[1..3,7..10] – кратність атрибута може набувати будь-якого значення з чисел: 1, 2, 3, 7, 8, 9, 10;

[1..3,7 .. *] – кратність атрибута може набувати будь-якого значення з чисел: 1, 2, 3, а також будь-якого додатного цілого значення, яке більше 7 або дорівнює йому.

Якщо кратність атрибута не вказана, то за замовчуванням приймають її значення, яке дорівнює 1..1, тобто в точності 1.

Тип атрибута є виразом, семантика якого визначається мовою специфікації відповідної моделі. В нотації UML тип атрибута іноді визначають залежно від мови програмування, яку передбачається використовувати для реалізації даної моделі. У найпростішому випадку тип атрибута вказується рядком тексту, що має осмислене значення в межах пакета або моделі, до яких відноситься розглянутий клас.

Початкове значення служить для завдання деякого початкового значення для відповідного атрибута в момент створення окремого екземпляра класу (рис. 3.4). Тут необхідно дотримуватися правил належності значення типу конкретного атрибута. Якщо початкове значення не вказано, то значення відповідного атрибута не визначено на момент створення нового екземпляра класу. З іншого боку, конструктор відповідного об'єкта може перевизначати вхідне значення в процесі виконання програми, якщо в цьому виникає необхідність.

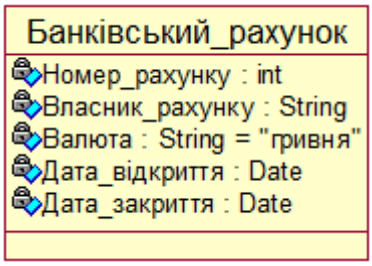


Рисунок 3.4 –Приклад завдання атрибутів класу в Rational Rose

Операція. У третій секції прямокутника записуються операції або методи класу.

Операція (operation) – це деякий сервіс, що надає кожний

екземпляр класу за певними вимогами.

Сукупність операцій характеризує функціональну поведінку об'єкта класу. Запис операцій класу мовою UML також стандартизована і підпорядковується певним синтаксичним правилам. При цьому кожній операції класу відповідає окремий рядок, яка складається з квантора видимості операції, назви операції, списку параметрів, виразу типу значення, що повертається операцією і, можливо, рядка-властивості даної операції:

<Квантор_видимості> <назва_операції> (список параметрів):
<Вираз_типу_значення_що_повертається> {рядок-властивість}

Квантор видимості, як і у випадку атрибутів класу, може набувати одного із трьох можливих значень і, відповідно, відображається за допомогою спеціального символу: "+" означає загальнодоступний (public), "#" – захищений (protected), "-" – закритий (private).

Квантор видимості для операції може бути опущений. В цьому випадку його відсутність просто означає, що видимість операції не визначена. Замість умовних графічних позначень також можна записувати відповідне ключове слово: public, protected, private.

Назва операції являє собою рядок тексту, який використовують як ідентифікатор відповідної операції, і тому має бути унікальним у межах даного класу. Назва операції є єдиним обов'язковим елементом синтаксичного позначення операції (рис. 3.5).

Список параметрів є переліком розділених комою формальних параметрів, кожен з яких може бути поданий в такому вигляді:

<Вид_параметра> <назва_параметра>:
<Вираз_типу> = <значення_параметра_за_замовчуванням>.

Тут вид параметра – це одне з ключових слів in, out або inout зі значенням in за замовчуванням у разі, якщо тип параметра не вказується.

Назва параметра є ідентифікатором відповідного формального параметра. Вираз типу є залежною від конкретної мови програмування специфікацією типу значення, що повертається для відповідного формального параметра. Нарешті, значення за замовчуванням у загальному випадку є виразом для значення формального параметра, синтаксис якого залежить від конкретної мови програмування і підпорядковується прийнятим у ньому обмеженням.

Вираз типу значення, що повертається, також є залежною від мови реалізації специфікацією типу або типів значень параметрів, які

повертаються об'єктом після виконання відповідної операції. Двокрапка і вираз типу значення, що повертається, можуть бути опущені, якщо операція не повертає ніякого значення. Для вказівки кратності значення, що повертається, дана специфікація може бути записана у вигляді списку окремих виразів.

Рядок-властивість служить для зазначення значень властивостей, які можуть бути застосовані до даного елемента. Рядок-властивість не є обов'язковою, він може бути відсутнім, якщо ніякі властивості не специфіковані.

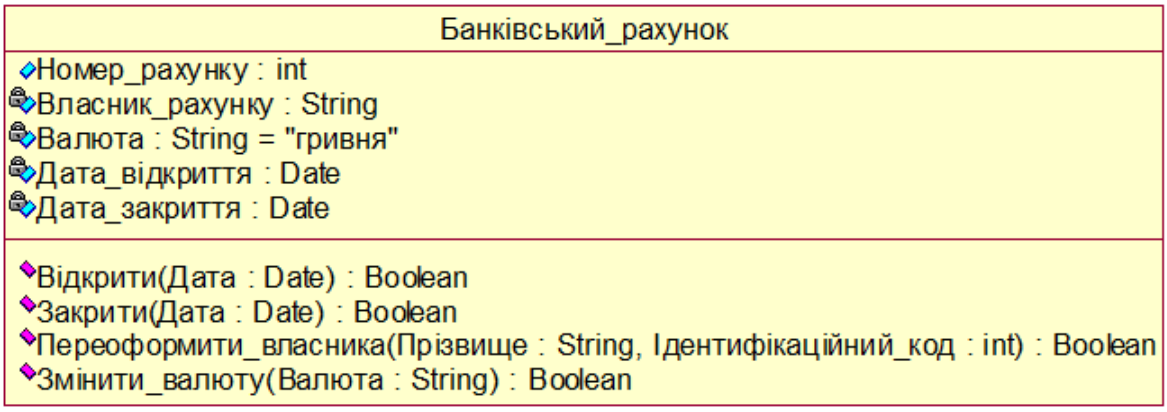


Рисунок 3.5 – Приклад завдання операцій класу в Rational Rose

Відношення між класами

Крім внутрішнього устрою або структури класів на відповідній діаграмі вказують різні відношення між класами. При цьому сукупність типів таких відношень фіксована у UML і зумовлена семантикою цих типів відношень.

Базовими відношеннями (зв'язками) між класами в мові UML є такі:

- відношення асоціації (association relationship);
- відношення залежності (dependency relationship);
- відношення узагальнення (generalization relationship);
- відношення реалізації (realization relationship).

Кожне з цих відношень має власне графічне подання на діаграмі, яке відображає взаємозв'язки об'єктів відповідних класів.

Відношення асоціації

Відношення асоціації відповідає наявності деякого семантичного зв'язку між класами.

Після того, як класи будуть пов'язані відношенням асоціації, об'єкти цих класів можуть передавати повідомлення один одному. Отже, відношення асоціації дає можливість дізнаватися про загальнодоступні атрибути і операції іншого класу.

Асоціації можуть бути односпрямованими і двоспрямованими. На рис. 3.6, а зображено приклад асоціації між класами, який показує, що об'єкт класу А передає повідомлення об'єкту класу В.

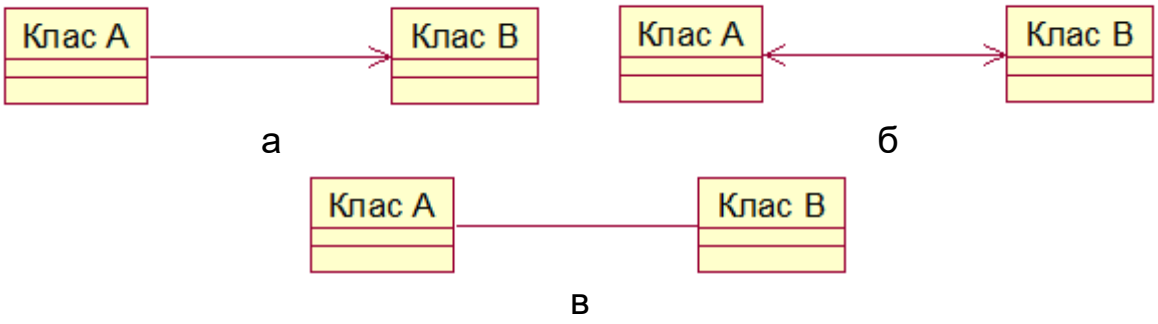


Рисунок 3.6 – Відношення асоціації між класами:
а – односпрямоване; б, в – двоспрямоване

При генерації коду в Rational Rose в описі класів, пов'язаних асоціацією, будуть додані атрибути відповідних класів. На рис. 3.7 показано код, який відповідає односпрямованій асоціації, що зображена на рис. 3.6, а.

```
#include "Class_B.h"

class Class_A {
 public
 Class_A ();

 Class_B * the_Class_B;
}
```

```
class Class_B {
 public

 Class_B ();
}
```

Рисунок 3.7 – Приклад реалізації на С ++ односпрямованої асоціації

Відношення агрегації

Відношення агрегації є більш сильною формою асоціації.

Агрегацією називають зв'язок між цілим і його частинами.

На рис. 3.8 цілим є клас А, а частинами – класи В і С. Передбачається існування складових частин поза цілого, тобто ціле і частини можуть бути знищені в різний час.

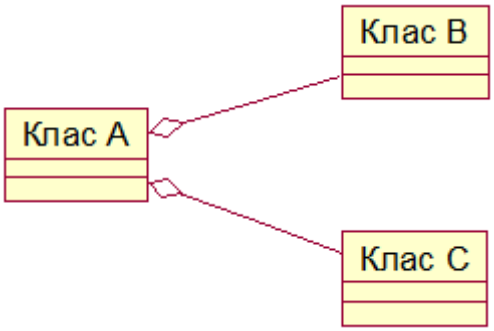


Рисунок 3.8 – Відношення агрегації між класами

При генерації коду для агрегації автоматично створюються підтримуючі її додаткові атрибути (рис. 3.9).

```
#include "Class_B.h"
#include "Class_C.h"

class Class_A {
 public
 Class_A ();
 Class_B the_Class_B;
 Class_C the_Class_C;
}

class Class_B {
 public

 Class_B ();
}

class Class_C {
 public

 Class_C ();
}
```

Рисунок 3.9 – Приклад реалізації на C ++ агрегації

Відношення композиції

Відношення композиції є окремим випадком відношення агрегації (рис. 3.10).

У цій спеціальній формі відношення "частина-ціле" частини в деякому сенсі перебувають всередині цілого, частини не можуть виступати у відриві від цілого, тобто зі знищенням цілого знищуються і всі його складові частини.

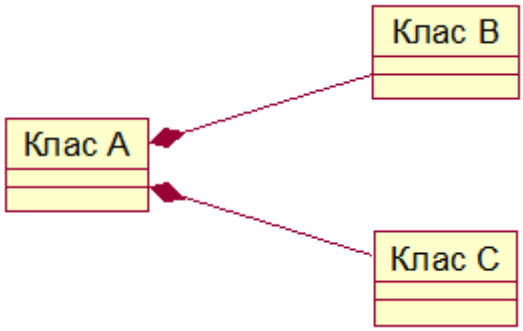


Рисунок 3.10 – Відношення композиції між класами

Відношення залежності

Відношення залежності завжди односпрямоване і показує, що один клас залежить від визначень, які зроблені в іншому класі. Зміни в одному класі вплинуть на інший клас (рис. 3.11).

Рисунок 3.11 – Відношення залежності між класами

При генерації коду в Rational Rose не генерується жодних додаткових атрибутів у цих класах, але створюються специфічні для мови програмування оператори, необхідні для підтримки цього зв'язку (для мови С ++ це оператор #include, рис. 3.12).

Таким чином, клас А має дізнаватися про клас В в інший спосіб, ніж це відбувається при асоціації класів. Тут передбачається три способи. По-перше, клас В можна зробити глобальним, і тоді клас А знатиме про його існування. По-друге, клас В можна інстанціювати як локальну змінну всередині методу класу А. По-третє, клас В можна передавати методам класу А як параметр. За наявності зв'язку залежності необхідно слідувати одному з наведених підходів. Вибір підходу може вплинути на всю модель.

```
class Class_A {
 public
 int метод_класу_A (Class_B * b);

 Class_A ();
 }
```

Рисунок 3.12 – Приклад реалізації на С ++ залежності

Відношення узагальнення

Відношення узагальнення є зв'язком успадкування між двома класами.

Дане відношення дає можливість успадковувати загальнодоступні і захищені атрибути і методи іншого класу. Крім успадкованих кожен підклас має свої власні унікальні атрибути, операції і зв'язки. Зв'язки узагальнення дозволяють економити час і зусилля як при розробленні, так і при подальшій підтримці програми.

На рис. 3.13 клас А називають класом-предком, а класи В і С – класами-нащадками. Загальні для класів В і С елементи знаходяться в класі А (рис. 3.14).

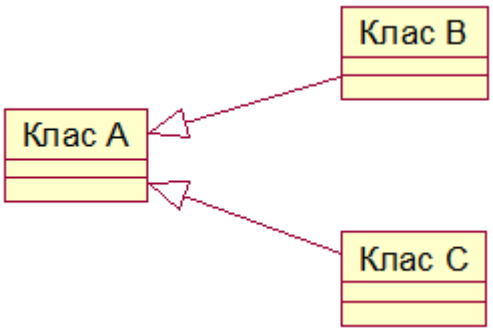


Рисунок 3.13 – Відношення узагальнення між класами

```
#include "Class_A.h"

class Class_B: public Class_A {
 public
 Class_B ();

 // власні аргументи і методи
 ...
}
```

```
#include "Class_A.h"

class Class_C: public Class_A {
 public
 Class_C ();

 // власні аргументи і
 методи
 ...
}
```

Рисунок 3.14 – Приклад реалізації на С ++ залежності

Об'єкти

Об'єкт (object) є окремим екземпляром класу, який створюється на етапі виконання програми.

Об'єкт має своє власне ім'я і конкретні значення атрибутів (рис. 3.15). У силу різних причин може виникнути необхідність показати

взаємозв'язки не тільки класів моделі, але й окремих об'єктів, що реалізують ці класи. В даному випадку може бути розроблена діаграма об'єктів, яка, хоча і не є канонічною в метамоделі мови UML, але має самостійне призначення.

Рисунок 3.15 – Приклад графічного зображення об'єктів класів

Шаблони або параметризовані класи

Шаблон (template) або параметризований клас (parametrized class) – спеціальний тип класів, який призначений для створення групи класів, кожен з яких може бути отриманий зв'язуванням цих параметрів з дійсними значеннями (рис. 3.16).

Наприклад, якщо є параметризований клас *Список*, то за допомогою реалізації відповідного класу можна створити такі класи, як *Список_співробітників_банка*, *Список_банківських_рахунків*.

Шаблон не може бути безпосередньо використаний як клас, оскільки містить невизначені параметри. Найчастіше як шаблон виступає деякий суперклас, параметри якого уточнюються в його класах-нащадках.

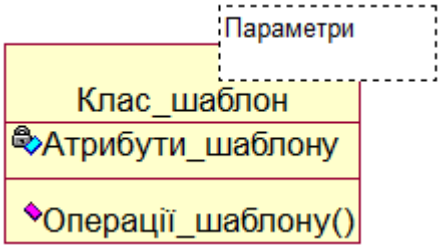


Рисунок 3.16 – Приклад графічного зображення класу-шаблону

Клас-наповнювач

Клас-наповнювач (instantiate class) є параметризованим класом, аргументи якого мають фактичні значення. Відповідно до нотації UML назву класу-наповнювача необхідно брати в кутові дужки <> (рис. 3.17).

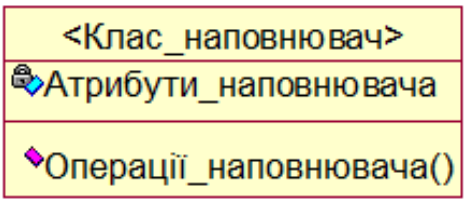


Рисунок 3.17 – Приклад графічного зображення класу-наповнювача

Утиліта класу

Утиліта класу (class utility) є сукупністю логічно пов'язаних операцій (рис. 3.18).

Наприклад, в утиліті класу можна об'єднати операції, що реалізують різні економічні функції. Таким чином, утиліти застосовують для зберігання загальних елементів функціональності багаторазового використання, необхідних в інших системах.

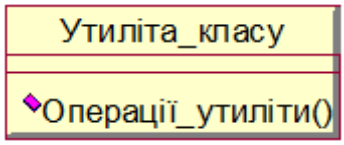


Рисунок 3.18 – Приклад графічного зображення утиліти класу

Утиліта параметризованого класу

Утилітою параметризованого класу (parametrized class utility) є параметризований клас, що складається тільки з набору операцій. Шаблон для створення утиліт класу показано на рис. 3.19.

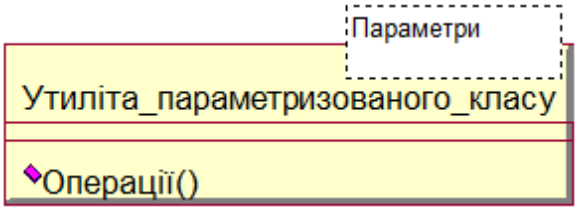


Рисунок 3.19 – Приклад графічного зображення утиліти параметризованого класу

Кваліфікатори

Кваліфікатори (qualifiers) застосовують для того, щоб зменшити область дії асоціації.

Припустимо, що між класами *Клієнт_банку* і *Банк* встановлено зв'язок асоціації і для даного значення атрибута Ідентифікаційний_номер клієнта існують два банки, які взаємодіють з клієнтом. Це можна показати на діаграмі класів за допомогою кваліфікатора (рис. 3.20).

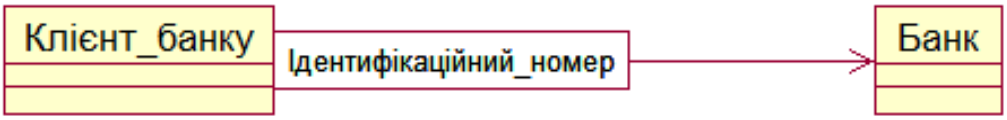


Рисунок 3.20 – Приклад графічного зображення кваліфікатора класу

Клас асоціації

Клас асоціації (association class) є класом, який зберігає атрибути, які відносяться до асоціації, тобто зв'язку між класами.

Припустимо, у нас є два класи *Співробітник_банку* і *Посада*. І необхідно додати на діаграму атрибут Рік_призначення. Виникає запитання: до якого класу додати цей атрибут? Цей атрибут більшою мірою відноситься до зв'язку між двома класами *Співробітник_банку* і *Посада*, ніж до якогось класу конкретно. Тому ми його поміщаємо в так званий клас асоціацій (рис. 3.21).

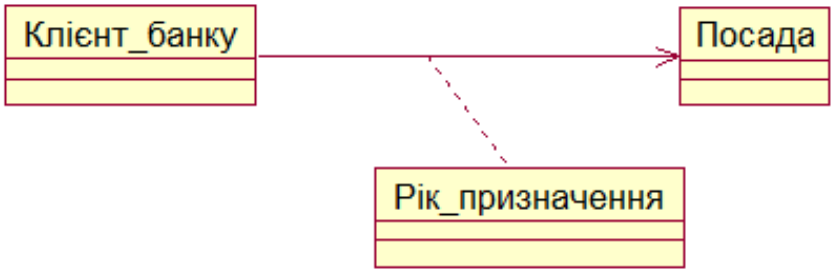


Рисунок 3.21 – Приклад графічного зображення класу асоціації

Кратність зв'язку

Для зв'язку між класами можна задати кратність (множинність) зв'язку, яка показує, скільки екземплярів одного класу взаємодіє з допомогою зв'язку з одним екземпляром іншого класу в даний момент.

Правила завдання кратності були розглянуті вище при описі завдання кратності атрибутів класу (рис. 3.22).

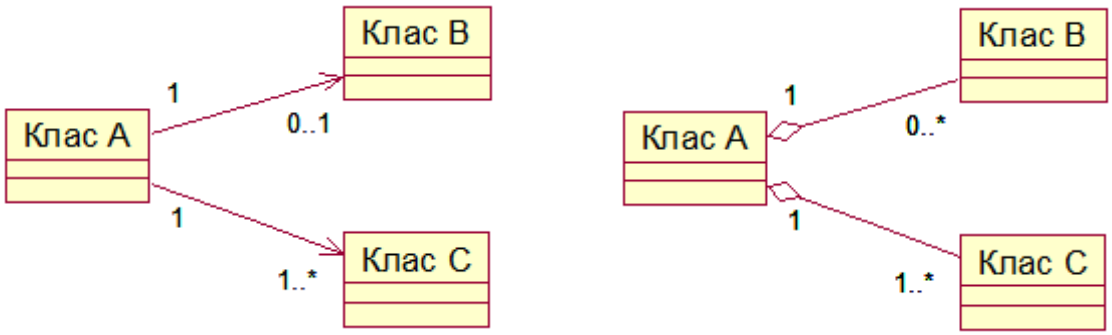


Рисунок 3.22 – Приклад графічного зображення кратності класу

На рис. 3.23 показано приклад побудови діаграми класів.

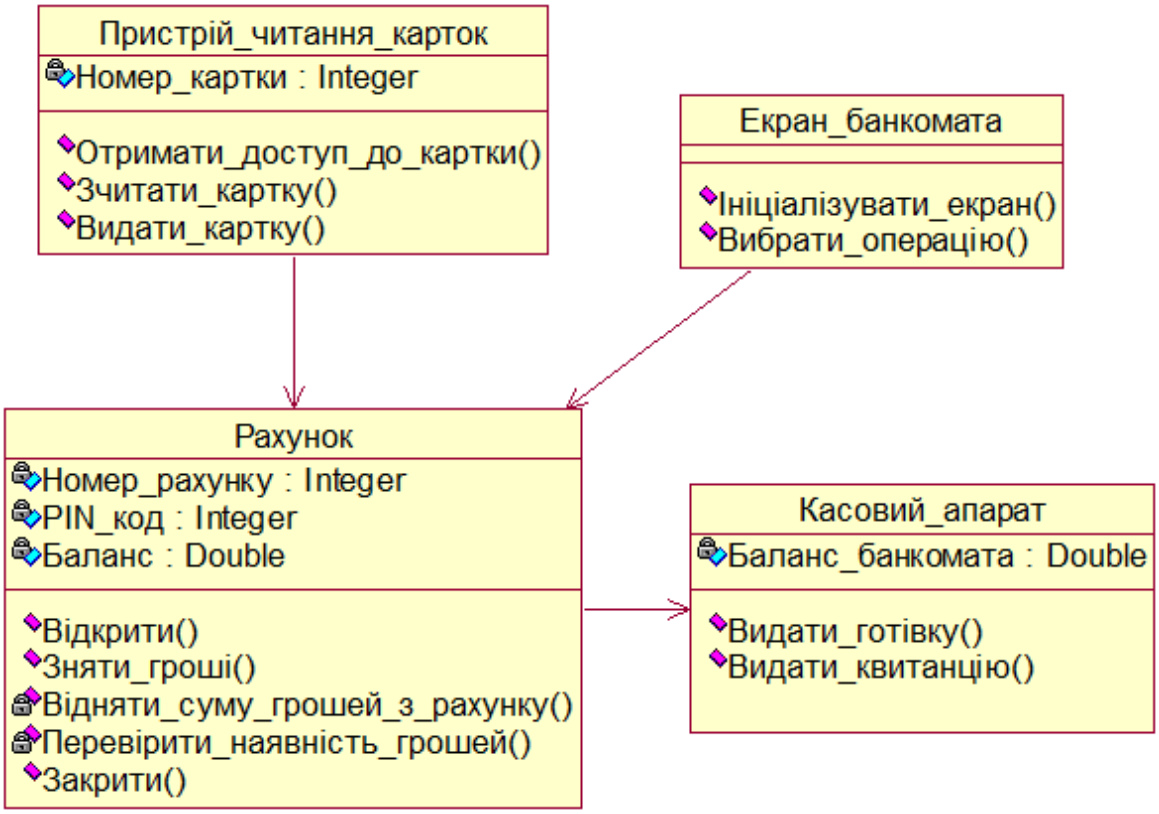


Рисунок 3.23 – Приклад побудови діаграми класів

Пакети

Пакети (packages) застосовують для групування класів, що мають деяку спільність.

Можна об'єднати класи за їх функціональністю. Наприклад, пакет *Безпека* буде містити всі класи, що відповідають за безпеку програмного забезпечення. Інші пакети можуть називатися *Робота_зі_співробітниками*, *Підготовка_звітів*, *Оброблення_помилок* (рис. 3.24).

Якщо уважно підійти до групування класів, можна отримати пакети, які практично не залежать один від одного. Пакети можуть бути повторно використані в інших програмних системах.

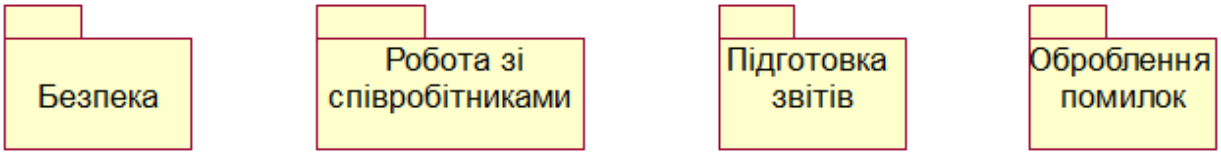


Рисунок 3.24 – Приклад побудови діаграми пакетів

4 РОЗРОБЛЕННЯ ДІАГРАМ ВЗАЄМОДІЇ

Діаграми взаємодії

Діаграми взаємодії (interaction diagrams) є моделями, що описують поведінку взаємодіючих груп об'єктів.

Як правило, діаграма взаємодії охоплює поведінку об'єктів у рамках тільки одного варіанта використання. На такій діаграмі відображається ряд об'єктів і ті повідомлення, якими вони обмінюються між собою. Якщо варіант використання має не тільки основний сценарій, але і альтернативні, то діаграми взаємодії мають бути розроблені для кожного сценарію.

Існує два типи діаграм взаємодії: діаграма послідовності і діаграма кооперації.

Обидві діаграми відображають події, які беруть участь у процесі оброблення інформації варіантів використання, і повідомлення, якими обмінюються об'єкти, тобто обидві діаграми відображають одну і ту ж інформацію.

Однак події на діаграмі послідовності впорядковані за часом, а діаграма кооперації організована навколо самих об'єктів.

Діаграма послідовності

Діаграма послідовності (sequence diagrams) – це впорядкована за часом діаграма взаємодії, читати її слід зверху вниз.

Розглянемо елементи діаграми послідовності (рис. 4.1). На діаграмі послідовності відображають об'єкти, які беруть участь у потоці подій. Кожен об'єкт має свою лінію життя, яка відображається пунктирною лінією під об'єктом.

На діаграмі також відображають повідомлення, якими обмінюються об'єкти. Повідомлення позначають між лініями життя об'єктів. Повідомлення показує, що один об'єкт потребує від іншого виконання будь-яких функцій. При генерації коду повідомлення перетворюються у виклики методів відповідних класів. Для цього необхідно в Rational Rose обов'язково співвіднести повідомлення з методами класів.

Повідомлення можуть бути *рефлексивними*, що свідчить про виклик об'єктом власної операції. Момент знищення (деструкції) об'єкта зображується хрестом на лінії життя.

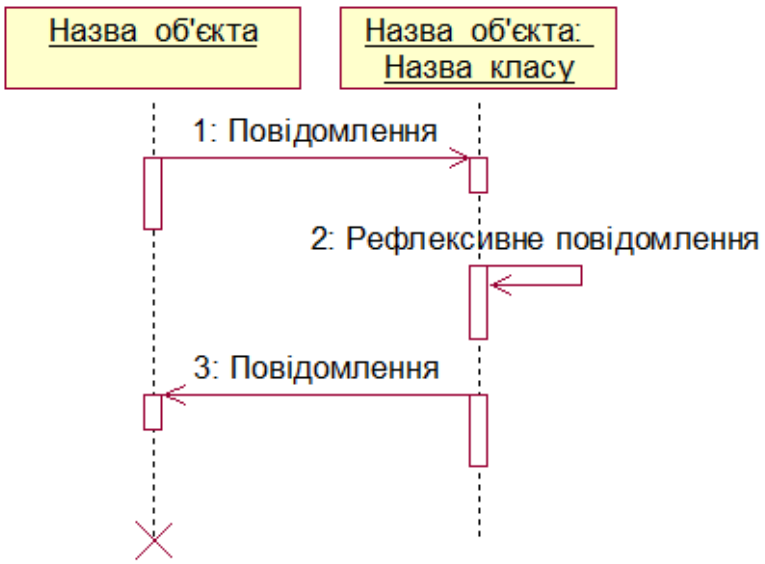


Рисунок 4.1 – Графічне зображення діаграми послідовності

Для повідомлень, які посилаються, можна визначити тип синхронізації (рис. 4.2).

Просте повідомлення. Значить, що всі повідомлення надсилаються в одному потоці управління (використовується за замовчуванням).

Синхронне повідомлення. Його застосовують, коли об'єкт-клієнт

посилає повідомлення і чекає відповіді.

Повідомлення з відмовою ставати в чергу. Об'єкт-клієнт посилає повідомлення об'єкту-серверу. Якщо сервер не може негайно прийняти повідомлення, воно скасовується.

Повідомлення з лімітованим часом очікування. Об'єкт-клієнт посилає повідомлення об'єкту-серверу, а потім чекає вказаний час. Якщо протягом цього часу сервер не приймає повідомлення, воно скасовується.

Асинхронне повідомлення. Об'єкт-клієнт посилає повідомлення об'єкту-серверу і продовжує свою роботу, не чекаючи підтвердження про отримання.

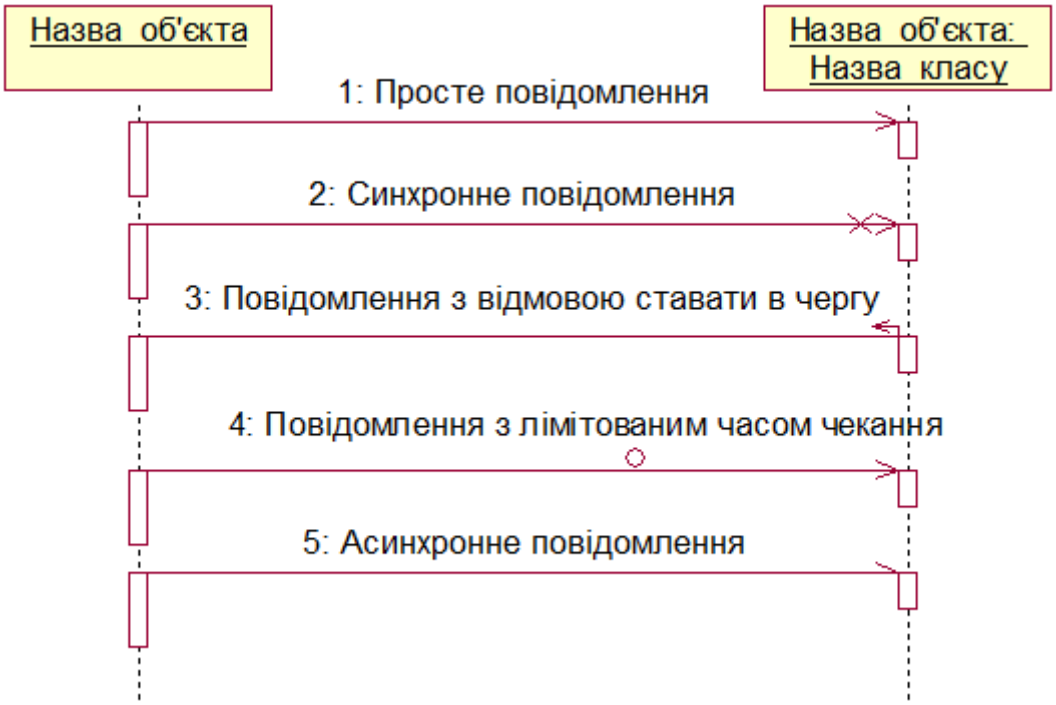


Рисунок 4.2 – Графічне зображення синхронізації повідомлень

Розглянемо варіант використання "Зняти гроші з рахунку", який передбачає декілька можливих сценаріїв (послідовностей) подій:

1) спроба зняття грошей при введенні правильного PIN-коду та їх достатньої кількості на рахунку;
2) спроба зняття грошей при введенні неправильного PIN-коду;
3) спроба зняття грошей при введенні правильного PIN-коду та відсутності їх достатньої кількості на рахунку.

На рис. 4.3 показано діаграму послідовності для першого (основного) сценарію: зняття 500 грн з рахунку за відсутності таких проблем, як неправильний PIN-код або брак грошей на рахунку.

Діаграма кооперації

Іншим видом діаграми взаємодії є діаграма кооперації (collaboration diagrams). Подібно до діаграми послідовності діаграма кооперації відображає потік подій в конкретному сценарії варіанта використання. У діаграмі кооперації більша увага приділяється саме зв'язкам між об'єктами. На рис. 4.4 показано приклад діаграми кооперації, яка відповідає діаграмі послідовності на рис. 4.3.

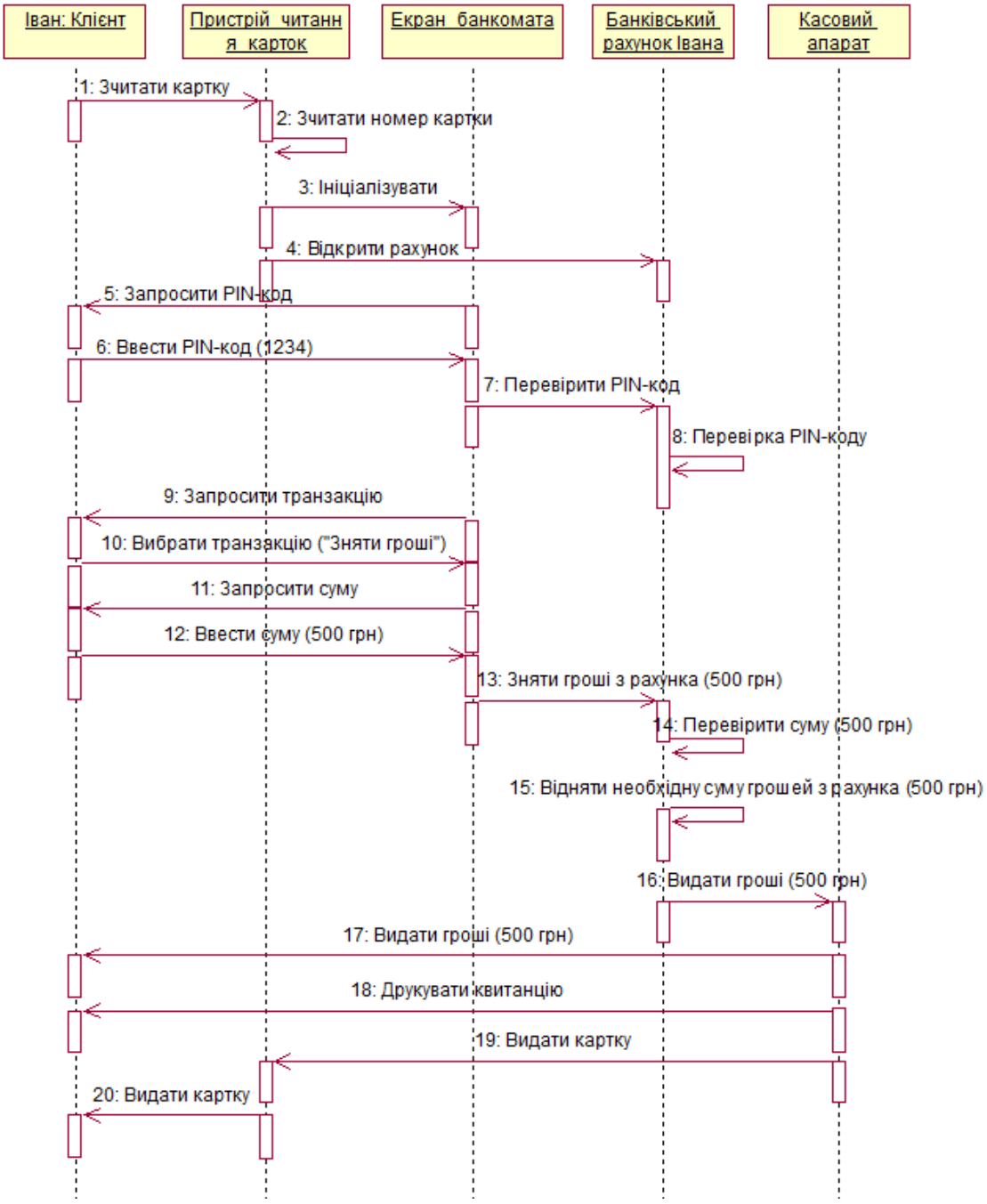


Рисунок 4.3 – Приклад побудови діаграми послідовності

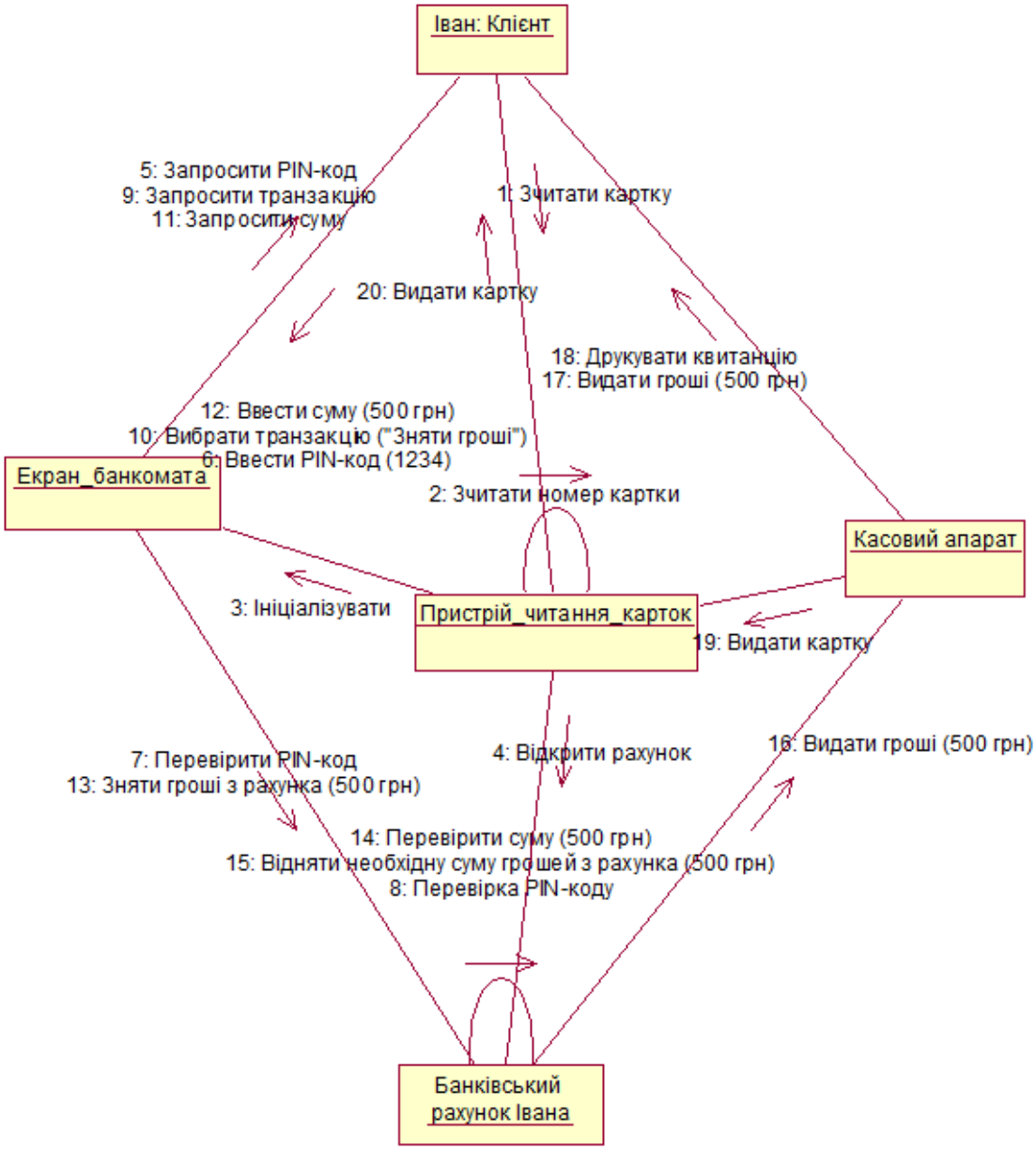


Рисунок 4.4 – Приклад побудови діаграми кооперації

Як видно з рис. 4.4, на діаграмі кооперації показано всю ту інформацію, яка була і на діаграмі послідовності, але діаграма кооперації по-іншому описує потік подій. З неї легше зрозуміти відношення між об'єктами, однак важче усвідомити послідовність подій.

З цієї причини часто для будь-якого сценарію створюють діаграми обох типів, хоча вони і містять одну і ту ж інформацію, але подають її з різних точок зору.

На діаграмі кооперації так само, як і на діаграмі послідовності, стрілки позначають повідомлення, обмін якими здійснюється в рамках даного варіанта використання. Але їх часова послідовність, однак,

відображається шляхом нумерації повідомлень.

Нумерація повідомлень робить сприйняття їх послідовності більш важкою, ніж у випадку розташування ліній на листі зверху вниз. З іншого боку, таке просторове розташування дозволяє більш легко показати взаємозв'язок об'єктів або іншу інформацію.

5 РОЗРОБЛЕННЯ ДІАГРАМИ СТАНІВ

На діаграмі класів зображують тільки взаємозв'язки структурного характеру, які не залежать від часу або реакції системи на зовнішні події. Однак для більшості фізичних систем, крім найпростіших і очевидних, статичних уявлень абсолютно недостатньо для моделювання процесів функціонування подібних систем як в цілому, так і їх окремих підсистем і елементів.

Кожна прикладна система характеризується не тільки структурою складових її елементів, але й деякою поведінкою або функціональністю.

Для загального подання функціональності модельованої системи призначено діаграму варіантів використання, яка на концептуальному рівні описує поведінку системи в цілому і яку було розглянуто у розділі 1.

Для моделювання поведінки на логічному рівні в мові UML можуть використовуватися відразу декілька канонічних діаграм: діяльності (див. розділ 2), послідовності, кооперації (див. розділ 4) і станів, кожна з яких фіксує увагу на окремому аспекті функціонування системи.

На відміну від інших діаграм діаграма станів описує процес зміни станів тільки одного класу, а точніше – одного екземпляра певного класу, тобто моделює всі можливі зміни в стані конкретного об'єкта. При цьому зміна стану об'єкта може бути викликана зовнішніми впливами з боку інших об'єктів або ззовні. Саме для опису реакції об'єкта на подібні зовнішні впливи і використовують діаграму станів (statechart diagram).

Головне призначення діаграми станів – описати можливі послідовності станів і переходів, які в сукупності характеризують поведінку об'єкта протягом його життєвого циклу (починаючи з моменту його створення і до його знищення).

Діаграма станів по суті є графом спеціального виду, який являє собою деякий автомат. Вершинами цього графа є стани, які зображаються відповідними графічними символами. Дуги графа служать для позначення переходів зі стану в стан. Діаграми станів можуть бути вкладені одна в одну, утворюючи вкладені діаграми для

більш детального подання окремих елементів моделі.

У Rational Rose на основі діаграми станів не генерується ніякий код. Вони необхідні для того, щоб задокументувати динаміку поведінки об'єкта класу, яка дозволить розробникам отримати про нього більш чітке уявлення.

Стан

Станом (state) називають одну з можливих умов, в якій може існувати об'єкт (рис. 5.1). Для виявлення станів об'єкта необхідно дослідити значення атрибутів об'єкта і зв'язки з іншими об'єктами.

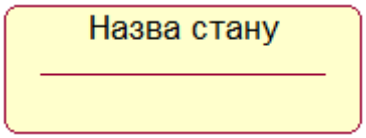


Рисунок 5.1 – Графічне зображення стану

Назва стану. Назва стану задається рядком тексту, який розкриває зміст даного стану. Оскільки стан системи є складовою частиною процесу її функціонування, рекомендується як назву імені використовувати дієслова в теперішньому часі (дзвенить, друкує, очікує) або відповідні дієприкметники (зайнятий, вільний, передано, отримано). Назва стану може бути відсутньою, тобто вона є необов'язковою для деяких станів. У цьому випадку стан є анонімним, і якщо на діаграмі таких станів декілька, то всі вони мають відрізнятися між собою.

Список внутрішніх дій. Зі станом можна пов'язувати дані п'яти типів: діяльність, вхідну дію, вихідну дію і історію станів.

Діяльністю (Activity) називається поведінка, яка реалізується об'єктом, коли він знаходиться в даному стані. Наприклад, якщо рахунок знаходиться в стані «Закрито», то відбувається повернення кредитної картки клієнта. Опису діяльності всередині стану передує мітка «do» (рис. 5.2).

Вхідною дією (Entry action) називається поведінка, яка виконується, коли об'єкт переходить у даний стан. Наприклад, при переході рахунка в стан «Перевищено рахунок» виконується дія «Тимчасово заморозити рахунок» незалежно від того, звідки об'єкт переходить у даний стан. Опису діяльності всередині стану передує мітка «entry» (рис. 5.2).

Вихідна дія (Exit action) подібна до вхідної. Однак вона

здійснюється як складова частина процесу виходу об'єкта з даного стану. Наприклад, при виході рахунка зі стану «Перевищено рахунок» незалежно від того, куди він переходить, виконується дія «Розморозити рахунок». Опису діяльності всередині стану передує мітка «exit» (рис. 5.2).

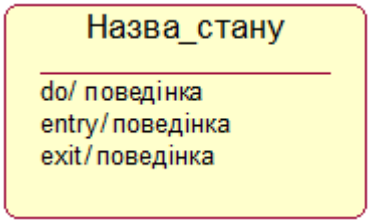
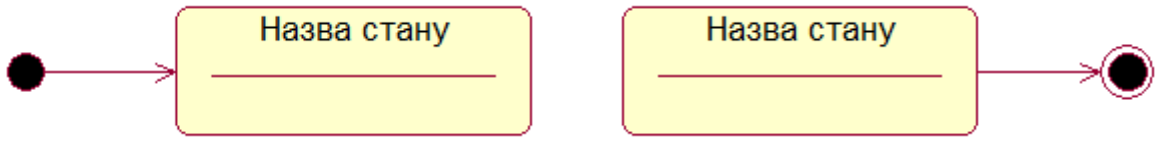


Рисунок 5.2 – Графічне зображення стану зі списком внутрішніх дій

Початковий і кінцевий стани

Початковим (Start) називається стан, в якому об'єкт знаходиться відразу після свого створення (рис. 5.3). Початковий стан обов'язковий. Наприклад, при створенні рахунок має стан «Відкрито».

Кінцевим (Stop) називається стан, в якому об'єкт знаходиться безпосередньо перед своїм знищенням (рис. 5.3). Кінцевий стан не є обов'язковим, їх може бути кілька.



а б

Рисунок 5.3 – Графічне зображення початкового (а) і кінцевого (б) станів

Перехід

Переходом (Transition) називається переміщення з одного стану в інший. Сукупність переходів діаграми показує, як об'єкт може переходити з одного стану в інший. Переходи можуть бути рефлексивними: об'єкт переходить в той же стан, в якому він в даний час знаходиться (рис. 5.4).

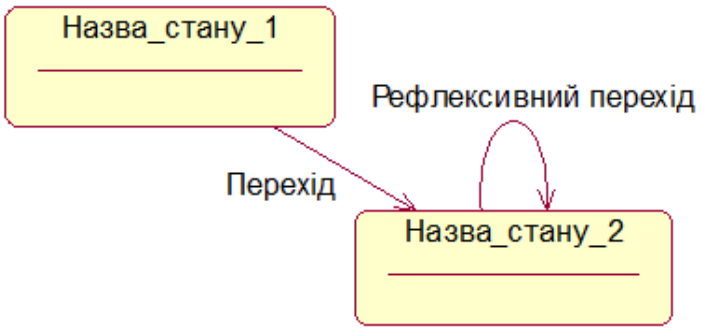


Рисунок 5.4 – Графічне зображення переходів

Для переходу існує кілька специфікацій: події, огороджувальні умови і дії (рис. 5.5).

Подія (Event) – це те, що викликає перехід об'єкта з одного стану в інший. Наприклад, подія «Клієнт вимагає закрити рахунок» викликає перехід рахунку зі стану «Відкрито» в стан «Закрито». Більшість переходів повинні мати події, за винятком автоматичних переходів з одного стану в інший.

Охоронні умови (Guard conditions) визначають, коли перехід може бути виконаний, а коли ні. Наприклад, подія «Зробити внесок» переведе рахунок зі стану «Перевищення рахунку» в стан «Відкрито», але тільки якщо баланс більше нуля. Охоронні умови ставити необов'язково.

Дією (Action) називається неперервна поведінка, що виконується як частина переходу. Вхідні і вихідні дії показують всередині стану, оскільки вони визначають, що відбувається, коли об'єкт входить або виходить зі стану. Інші дії зображують уздовж лінії переходу, оскільки вони не повинні виконуватися при вході в стан або на виході з нього. Наприклад, при переході рахунка з одного стану в інший виконується дія «Зберегти дату закриття рахунку». Ця дія виконується тільки при переході зі стану «Відкрито» в стан «Закрито».

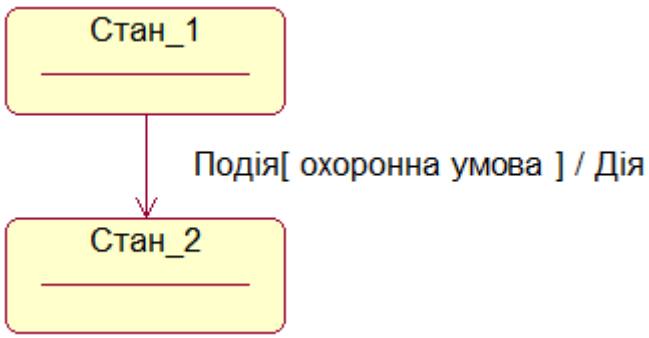


Рисунок 5.5 – Графічне зображення елементів переходу

Складений стан і підстани

Для зменшення безладу на діаграмі можна вкладати одні стани в інші.

Складений стан (Composite state) – це складний стан, який складається з інших вкладених у нього станів. Останні будуть виступати відносно до першого як підстани (substate).

Складений стан може містити два або більше паралельних підавтоматів або кілька послідовних підстанів. Кожний складений стан може уточнюватися тільки одним із зазначених способів. При цьому будь-який з підстанів, в свою чергу, може бути складеним станом і містити в собі інші вкладені підстани. Кількість рівнів вкладеності складених станів не фіксована у UML (рис. 5.6).

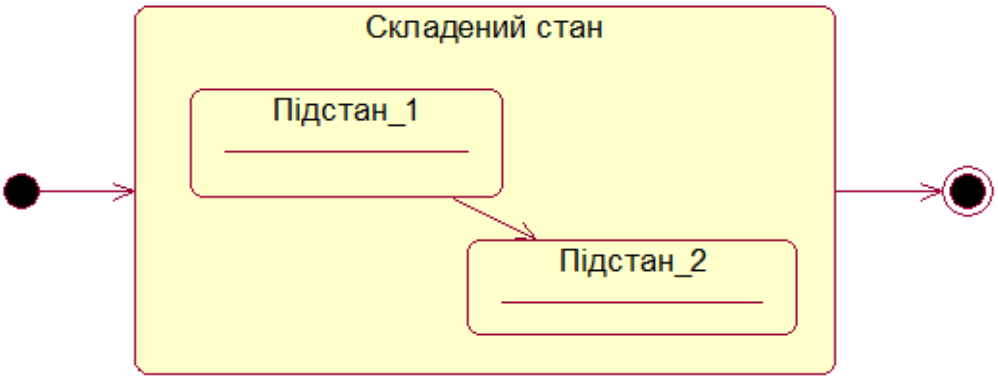


Рисунок 5.6 – Графічне зображення складеного стану з двома вкладеними в нього послідовними підстанами

Історія станів

Бувають ситуації, коли система має пам'ятати, в яких станах вона була в минулому. Наприклад, якщо реалізується вихід зі складеного стану з трьома підстанами, то може знадобитися, щоб система запам'ятала, з якої точки складеного стану стався вихід. Для вирішення цієї проблеми можна в складений стан включити початковий стан. Тоді буде відомо, де знаходиться стартова точка в складеному стані. І саме там виявиться об'єкт при вході до складеного стану. Для того, щоб запам'ятати, де знаходився об'єкт, можна використовувати історію станів (state history). У цьому випадку об'єкт може вийти зі складеного стану, а потім повернутися точно в те місце, звідки вийшов. Позначення історії станів показано на рис. 5.7. Крім того, можна відстежувати і історію вкладених підстанів.

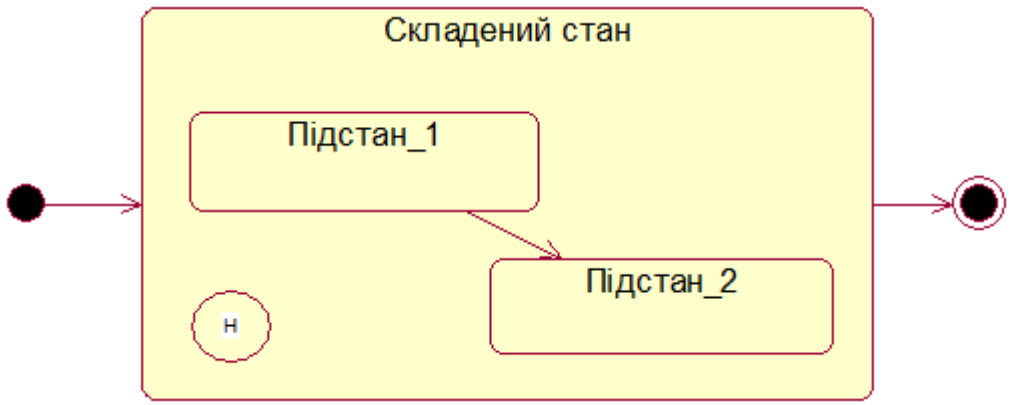


Рисунок 5.7 – Графічне зображення історії складеного стану

На рис. 5.8 показано приклад діаграми станів для об'єкта класу «Рахунок».

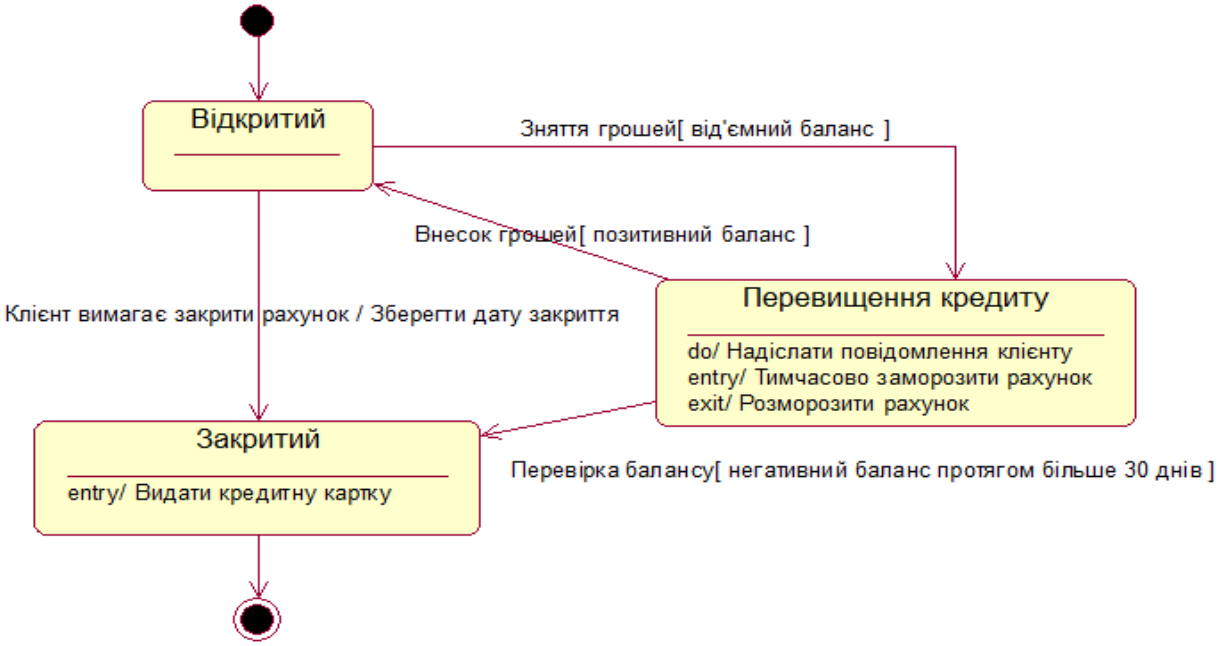


Рисунок 5.8 – Приклад побудови діаграми станів

6 РОЗРОБЛЕННЯ ДІАГРАМИ КОМПОНЕНТІВ

Усі розглянуті раніше діаграми відображали концептуальні аспекти побудови моделі системи і відносилися до логічного рівня подання. Особливість логічного подання полягає в тому, що воно оперує поняттями, які не мають самостійного матеріального втілення. Іншими словами, різні елементи логічного подання, такі, як класи, асоціації, стани, повідомлення, не існують матеріально або фізично.

Вони лише відображають наше розуміння структури фізичної системи або аспекти її поведінки.

Основне призначення логічного подання полягає в аналізі структурних і функціональних відносин між елементами моделі системи. Однак для створення конкретної фізичної системи необхідно певним чином реалізувати всі елементи логічного подання в конкретні матеріальні сутності. Для опису таких реальних сутностей призначений інший аспект модельного подання, а саме фізичне подання моделі.

Щоб пояснити відміну логічного подання від фізичного, розглянемо в загальних рисах процес розроблення деякої програмної системи. Її вихідним логічним поданням можуть бути структурні схеми алгоритмів і процедур, описи інтерфейсів і концептуальні схеми баз даних. Однак для реалізації цієї системи необхідно розробити вихідний текст програми деякою мовою програмування (наприклад, C++, C#, Java та ін.). При цьому вже в тексті програми передбачається така організація програмного коду, яка передбачає його розбиття на окремі модулі.

Проте вихідні тексти програми ще не є остаточною реалізацією проекту, хоча вони і служать фрагментом його фізичного подання. Очевидно, програмна система може вважатися реалізованою в тому випадку, коли вона буде здатна виконувати функції свого цільового призначення. А це можливо, тільки якщо програмний код системи буде реалізований у формі виконуваних модулів, бібліотек класів і процедур, стандартних графічних інтерфейсів, файлах баз даних. Саме ці компоненти є необхідними елементами фізичного подання системи.

Таким чином, повний проект програмної системи – це сукупність моделей логічного і фізичного подання, які мають бути узгоджені між собою. У мові UML для фізичного подання моделей систем використовують так звані діаграми реалізації (implementation diagrams), які містять дві окремі діаграми: діаграму компонентів і діаграму розгортання.

Діаграма компонентів (Component diagram) описує особливості фізичного подання системи. Ця діаграма дозволяє визначити архітектуру розроблюваної системи, встановивши залежності між програмними компонентами, в ролі яких може виступати вихідний, бінарний і виконуваний код. У багатьох середовищах розроблення програмних систем модуль або компонент відповідає файлу.

Компоненти

Для подання фізичних сутностей у UML застосовують спеціальний термін – компонент (component).

Компонентом називається фізичний модуль коду. Компонентами бувають як бібліотеки вихідного коду, так і виконувані файли. Наприклад, у мові C++ окремими компонентами будуть файли * .h і *.cpp. Виконуваний файл *.exe, що виходить при компіляції, також є компонентом системи. На рис. 6.1 показано графічне зображення компонента як узагальненого поняття.

Перед початком генерації коду необхідно співвіднести кожен із класів з відповідними компонентами. Мовою C++ кожен із класів співвідноситься з двома компонентами, один з яких відповідає *.cpp файлу цього класу, а інший – *.h файлу. При генерації коду Rational Rose використовує інформацію про компоненти для створення відповідних файлів бібліотек коду.

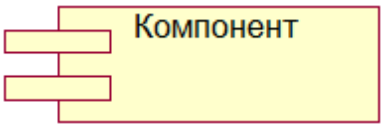


Рисунок 6.1 – Графічне зображення компонента

Типи компонентів

Специфікація і тіло підпрограми. На рис. 6.2 показані позначення для зображення видимої специфікації програми і тіла її реалізації. Зазвичай підпрограма складається з колекції стандартних програмних компонентів і не містить визначень класу.

Рисунок 6.2 – Графічне зображення специфікації (а) і тіла (б) підпрограми

Головна програма. На рис. 6.3 показано позначення файла, який є коренем програми.

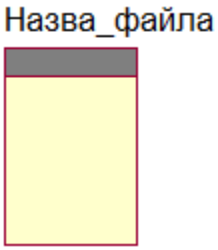


Рисунок 6.3 – Графічне зображення головної програми

Специфікація і тіло пакета. В даному випадку пакет – це реалізація класу. Специфікацією пакета є заголовки з відомостями про прототипи функцій класу. Мовою С++ це файл із розширенням *.h. Тіло пакета містить код операцій класу. У С++ це файл з розширенням *.cpp.

а б

Рисунок 6.4 – Графічне зображення специфікації (а) і тіла (б) пакета

*Файл *.DLL.* Для зображення файла динамічної бібліотеки використовують позначення, яке показано на рис. 6.5.

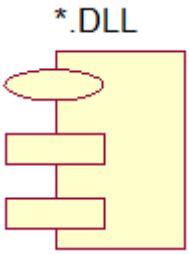


Рисунок 6.5 – Графічне зображення файла * .DLL

Специфікація і тіло задачі. Позначення на рис. 6.6 застосовують

для пакетів, які мають незалежні потоки управління. Виконуваний файл зазвичай подають як специфікацію задачі з розширенням *.exe.

Рисунок 6.6 – Графічне зображення специфікації (а) і тіла (б) задачі

Іншим способом завдання типу компонента є явна вказівка його стереотипу (рис. 6.7).

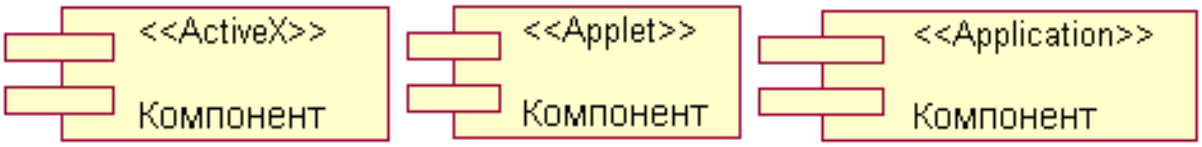


Рисунок 6.7 – Графічне зображення стереотипу компонентів

Зв'язки між компонентами

Єдиний можливий тип зв'язків між компонентами – це залежність, яка означає, що один компонент залежить від іншого. На рис. 6.8 зображено залежність між компонентами A і B. Тут компонент A залежить від компонента B. Інакше кажучи, в компоненті A існує певний клас, який залежить від якогось класу компонента B. Ці зв'язки мають значення при компіляції. Оскільки A залежить від B, то A не може бути скомпільовано до B. Аналізуючи діаграму на рис. 6.8, можна зрозуміти, що спочатку компілюється B, а потім вже A.

Рисунок 6.8 – Графічне зображення залежності компонентів

Слід уникати циклічних залежностей. Якщо A залежить від B, а B залежить від A, то жоден із них не можна скомпілювати, поки не

скомпільовано інший. Таким чином, два компоненти мають бути розглянуті як один великий компонент. Всі циклічні залежності необхідно видалити до початку генерації коду. Залежності пов'язані також із проблемами управління системою. Якщо А залежить від В, то будь-які зміни у В вплинуть на А. За допомогою діаграми компонентів можна оцінити наслідки всіх змін, які планується внести.

Крім того, залежності дають можливість зрозуміти, які частини системи можна використовувати повторно, а які не можна. Виходячи з діаграми на рис. 6.8 компонент А буде важко застосувати вдруге. Оскільки він залежить від В, то зробити це можна тільки спільно з В. З іншого боку, В легко використовувати повторно, тому що він не залежить від жодного компонента. Чим від меншого числа компонентів залежить даний компонент, тим легше його використовувати повторно.

На рис. 6.9 показано приклад побудови діаграми компонентів.

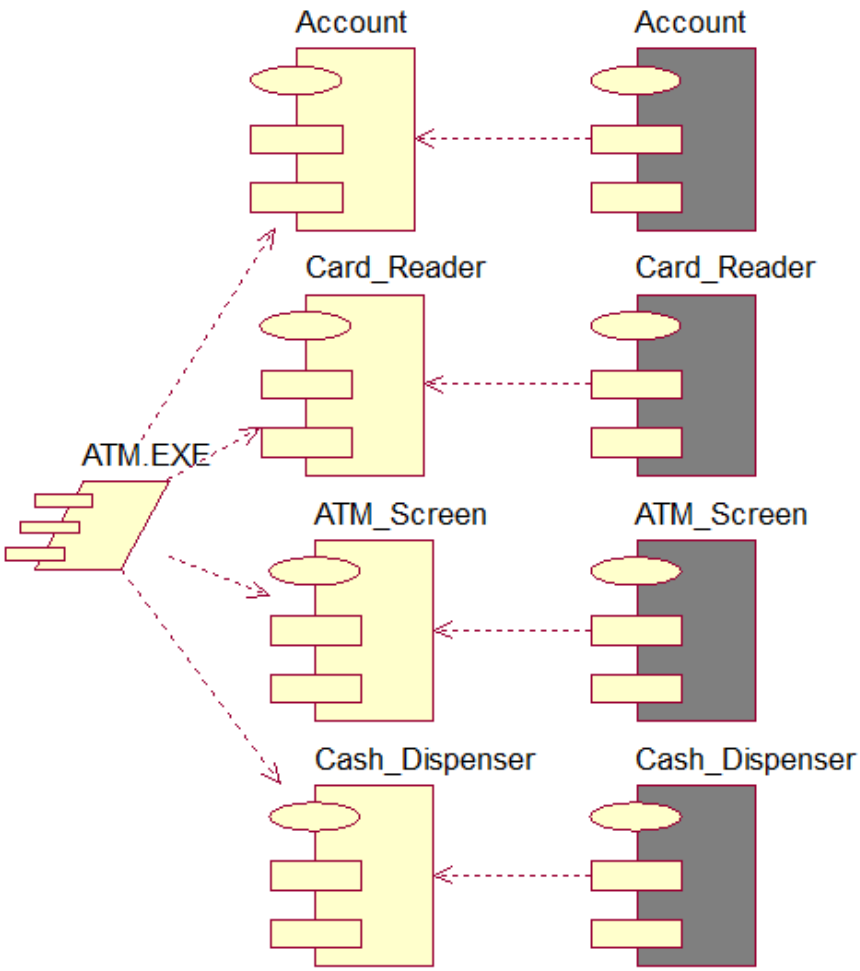


Рисунок 6.9 – Приклад побудови діаграми компонентів

7 РОЗРОБЛЕННЯ ДІАГРАМИ РОЗГОРТАННЯ

Фізичне подання програмної системи не може бути повним, якщо відсутня інформація про те, на якій платформі і на яких обчислювальних засобах вона реалізована. Звичайно, якщо розробляється проста програма, яка може виконуватися локально на комп'ютері користувача, що не використовує ніяких периферійних пристроїв і ресурсів, то в цьому випадку немає необхідності в розробленні додаткових діаграм. Однак при розробленні корпоративних систем спостерігається зовсім інша ситуація.

По-перше, складні програмні системи можуть реалізовуватися в мережному варіанті на різних обчислювальних платформах і технологіях доступу до розподілених баз даних. Наявність локальної корпоративної мережі потребує вирішення цілого комплексу додаткових завдань щодо раціонального розміщення компонентів по вузлах цієї мережі, що визначає загальну продуктивність програмної системи.

По-друге, інтеграція програмної системи з Інтернетом визначає необхідність вирішення додаткових питань при проектуванні системи, таких, як забезпечення безпеки, криптозахищеності і стійкості доступу до інформації для корпоративних клієнтів. Ці аспекти в чималому ступені залежать від реалізації проекту в формі фізично існуючих вузлів системи, таких, як сервери, робочі станції, брандмауери, канали зв'язку і сховища даних.

Нарешті, технології доступу і маніпулювання даними в рамках загальної схеми "клієнт-сервер" також потребують розміщення великих баз даних у різних сегментах корпоративної мережі, їх резервного копіювання, архівування, кешування для забезпечення необхідної продуктивності системи в цілому. Ці аспекти також потребують візуального подання з метою специфікації програмних і технологічних особливостей реалізації розподілених архітектур.

Було зазначено, що першою з діаграм фізичного подання є діаграма компонентів. Другою формою фізичного подання програмної системи є діаграма розгортання (deployment diagram) або діаграма розміщення. ЇЇ застосовують для подання загальної конфігурації і топології розподіленої програмної системи, і вона містить розподіл компонентів по окремих вузлах системи. Крім того, діаграма розгортання показує наявність фізичних з'єднань – маршрутів передачі інформації між апаратними пристроями, задіяними в реалізації системи.

Діаграма розгортання призначена для візуалізації елементів і компонентів програми, які існують лише на етапі її виконання (runtime). При цьому подаються тільки компоненти-екземпляри програми, які є

виконуваними файлами або динамічними бібліотеками. Ті компоненти, які не використовують на етапі виконання, на діаграмі розгортання не відображаються. Так, компоненти з вихідними текстами програм можуть бути присутніми тільки на діаграмі компонентів. На діаграмі розгортання їх не вказують.

Діаграма розгортання містить графічні зображення процесорів, пристроїв, процесів і зв'язків між ними. На відміну від діаграм логічного подання діаграма розгортання є єдиною для системи в цілому, оскільки вона має повністю відображати особливості її реалізації. Ця діаграма по суті завершує процес об'єктно-орієнтованого аналізу і проектування для конкретної програмної системи, і її розроблення, як правило, є останнім етапом специфікації моделі.

Вузол

Вузол (Node) – це деякий фізично існуючий елемент системи, що має деякий обчислювальний ресурс. Як обчислювальний ресурс вузла може розглядати наявність щонайменше деякого об'єму електронної або магнітооптичної пам'яті і/або процесора. В останній версії мови UML поняття вузла розширено і може містити не тільки обчислювальні пристрої (процесори), але й інші механічні або електронні пристрої, такі, як датчики, принтери, модеми, цифрові камери, сканери і маніпулятори.

Вузли поділяють на процесори і пристрої.

Процесор

Процесором (Processor) називають будь-яку машину, що має обчислювальну потужність, тобто здатну здійснювати оброблення даних (рис. 7.1). У цю категорію потрапляють сервери, робочі станції та інші пристрої, що містять фізичні процесори.

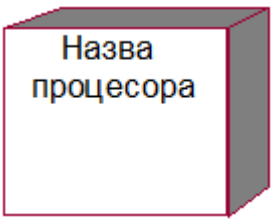


Рисунок 7.1 – Графічне зображення процесора

Для класифікації процесорів можна використовувати стереотипи. На рис. 7.2 показано приклад завдання стереотипу процесорів.

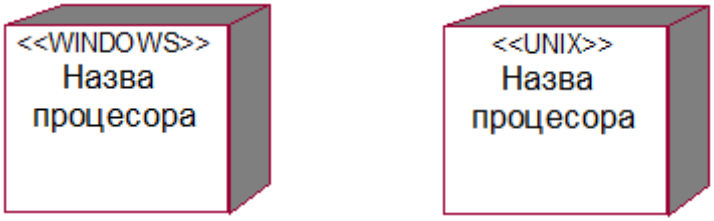


Рисунок 7.2 – Графічне зображення стереотипу процесора

Процесом (Process) називають потік оброблення інформації, що виконується на процесорі. Процесом, наприклад, вважають виконуваний файл. На рис. 7.3 показано зображення процесора із зазначенням процесу, який на ньому виконується.

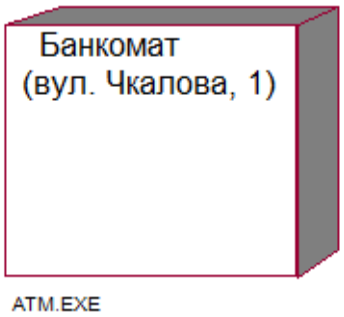


Рисунок 7.3 – Графічне зображення процесу

Пристрої

Пристроєм (Device) називають апаратура, яка не має обчислювальної потужності (рис. 7.4). Наприклад, до пристроїв можна віднести монітори, принтери, сканери та ін.

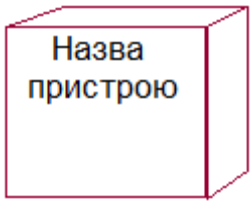


Рисунок 7.4 – Графічне зображення пристроїв

З'єднання

З'єднанням (Connection) називають фізичний зв'язок між двома процесорами, двома пристроями або процесором і пристроєм (рис. 7.5). З'єднання є різновидом асоціації. Наявність з'єднання вказує на необхідність організації фізичного каналу для обміну інформацією між відповідними вузлами.

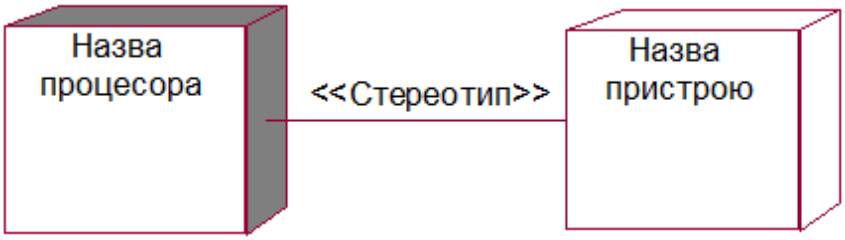


Рисунок 7.5 – Графічне зображення зв'язку між процесором і пристроєм

На рис. 7.6 показано приклад побудови діаграми розгортання.

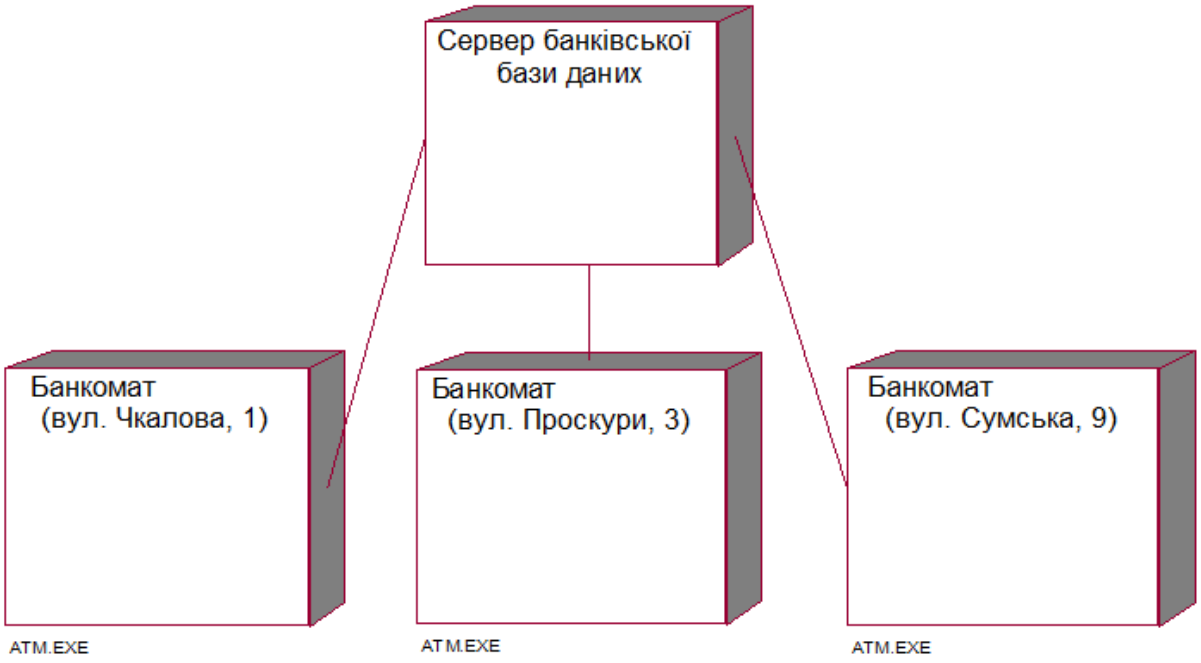


Рисунок 7.6 – Приклад побудови діаграми розгортання

8 РОЗРОБЛЕННЯ ДІАГРАМ UML У RATIONAL ROSE

Вступ до Rational Rose

Rational Rose – потужний інструмент аналізу і проектування об'єктно-орієнтованих програмних систем. Він дозволяє моделювати системи до написання коду. За допомогою готової моделі недоліки проекту легко виявити на стадії, коли їх виправлення не потребує ще значних витрат (часових, фінансових та ін.).

Модель Rose – це картина системи. Вона містить усі діаграми UML і детально описує, що система містить і як функціонує. Тому розробники можуть її використовувати як ескіз або креслення створюваної системи.

Rational Rose дозволяє генерувати «скелетний» код великою кількістю різних мов. Крім того, можна виконувати зворотне проектування коду і створювати, таким чином, моделі вже існуючих систем. Rose при зміні моделі внесе зміни в код, а також при змінах коду Rose дозволить внести зміни в модель. Тим самим вдається підтримувати відповідність між моделлю і кодом, зменшуючи ризик «старіння» моделі відносно коду.

На рис. 8.1 показано головне вікно Rational Rose. Основними елементами інтерфейсу є: 1 – браузер, 2 – вікно документації, 3 – панелі інструментів, 4 – вікно діаграми і 5 – журнал.

Браузер. Організовує подання моделі у вигляді ієрархічної структури, яка спрощує навігацію і дозволяє відшукати будь-який елемент моделі в проекті. При цьому будь-який елемент, який розробник додає в модель, відразу відображається у вікні браузера. Відповідно, вибравши елемент у вікні браузера, можна його візуалізувати у вікні діаграми або змінити його специфікацію. Браузер дозволяє також організовувати елементи моделі в пакети і переміщати елементи між різними поданнями моделі.

Вікно документації. Це вікно призначено для документування елементів подання моделі. У ньому можна записувати найрізноманітнішу інформацію. Така інформація в подальшому перетворюється в коментарі і ніяк не впливає на логіку виконання програмного коду.

У вікні документації активізується та інформація, яка відноситься до окремого виділеного елемента діаграми. При цьому виділити елемент можна або у вікні браузера, або у вікні діаграми. При додаванні нового елемента на діаграму (наприклад, класу) автоматично генерується документація до нього, яка є порожньою (No

documentation). У подальшому розробник самостійно вносить необхідну пояснювальну інформацію, яка запам'ятовується і може бути змінена в ході роботи над проектом.

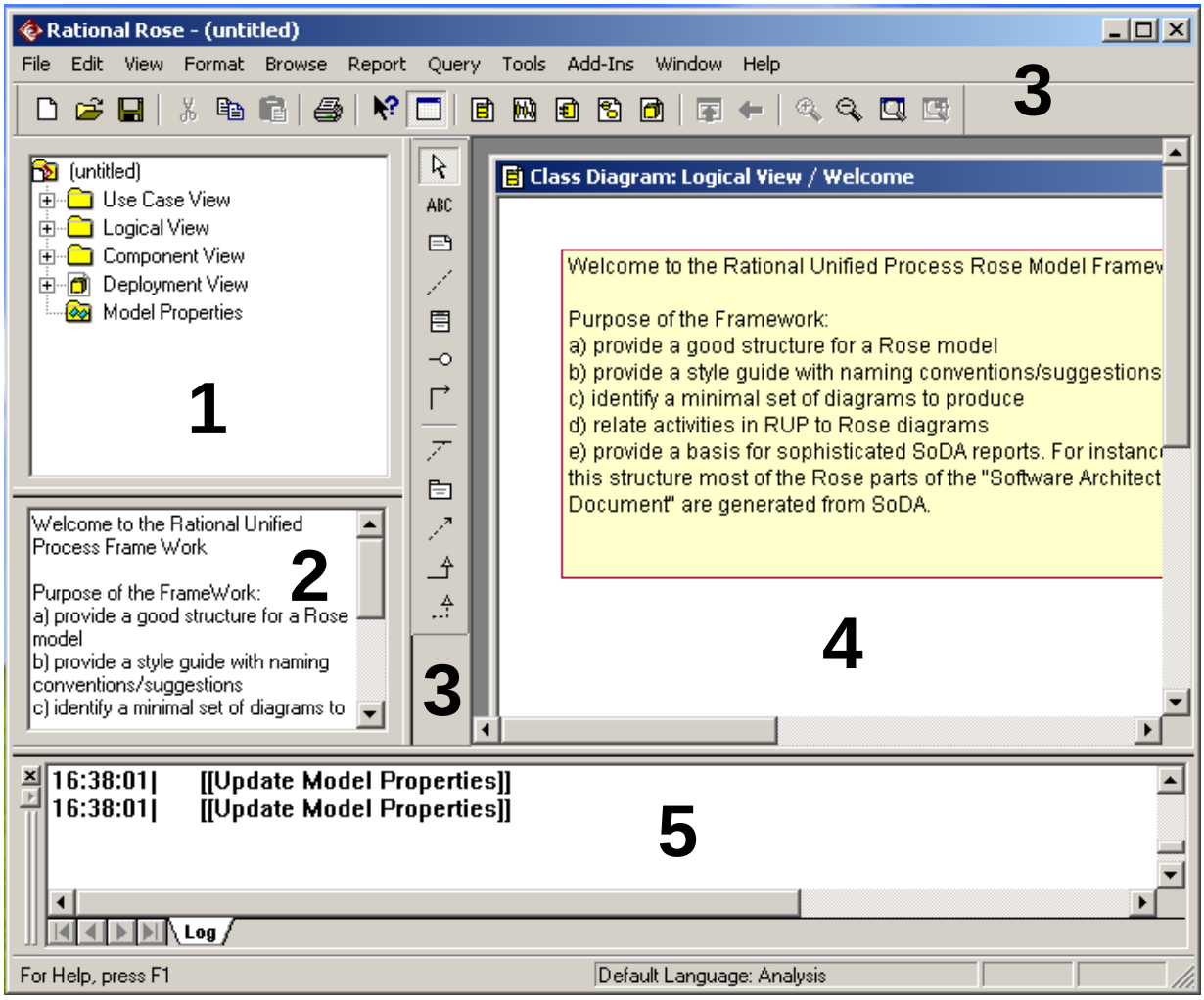


Рисунок 8.1 – Головне вікно Rational Rose

Панелі інструментів. Окремі пункти головного меню, призначення яких зрозуміло з їх назв, об'єднують подібні операції, відносяться до всього проекту в цілому. Деякі з пунктів меню містять добре знайомі функції (відкриття проекту, друк діаграм, копіювання в буфер і вставка з буфера різних елементів діаграм). Інші є настільки специфічними, що можуть потребувати додаткових зусиль на вивчення (опції генерації програмного коду, перевірка узгодженості моделей, підключення додаткових модулів).

Стандартна панель інструментів розташовується нижче головного меню програми. Деякі з інструментів недоступні (новий проект не має ніяких елементів). Стандартна панель інструментів

забезпечує швидкий доступ до тих команд меню, які виконуються розробниками найбільш часто.

Спеціальна панель інструментів розташовується між вікном браузера і вікном діаграми в середній частині робочого інтерфейсу. На ній розташовуються елементи для побудови відповідних діаграм. За замовчуванням пропонується панель інструментів для побудови діаграми класів моделі.

Вікно діаграми. Це вікно є основною робочою областю інтерфейсу, в якій візуалізуються різні подання моделі проекту. Його використовують для перегляду і редагування однієї або декількох діаграм UML.

Перемикання між діаграмами можна здійснити вибором потрібного подання на стандартній панелі інструментів або через пункт меню Window (Вікно). При активізації конкретної діаграми змінюється зовнішній вигляд спеціальної панелі інструментів, яка налаштовується під конкретний вид діаграми.

Журнал. Вікно журналу призначено для автоматичного запису різної службової інформації, що утворюється в ході роботи з програмою. У журналі фіксується час і характер виконуваних розробником дій, таких, як оновлення моделі, настроювання меню і панелей інструментів, а також повідомлень про помилки, що виникають при генерації програмного коду.

Подання в середовищі Rational Rose

Модель Rose підтримує чотири подання (views): подання варіантів використання, логічне подання, подання компонентів і подання розгортання.

Подання варіантів використання (Use case view). Це подання містить усіх дійових осіб, всі варіанти використання та їх діаграми для конкретної системи. Воно може також містити деякі діаграми послідовності і діаграми кооперації. Подання варіантів використання – це погляд на систему, незалежний від її реалізації. Основна увага тут приділяється тому, що буде робити система, а не як вона буде це робити.

Логічне подання (Logical view). Це подання концентрується на тому, як система буде реалізовувати поведінку, яка описана у варіантах використання. Воно дає докладну картину складових частин картини і описує їх взаємодію. Логічне подання містить, крім іншого, конкретні необхідні класи, діаграми класів і діаграми станів. За допомогою цього подання можна сконструювати детальний проект системи.

Подання компонентів (Component view). Це подання містить інформацію про бібліотеки коду, виконувані файли, динамічні бібліотеки та інші компоненти моделей. Містить компоненти і діаграми компонентів.

Подання розгортання (Deployment view). Це подання відповідає фізичному розміщенню системи, яке може відрізнятися від її логічної архітектури. Містить вузли і єдину діаграму розгортання.

Доступ до подання варіантів використання, логічного подання, подання компонентів і подання розгортання здійснюється через браузер.

Розроблення діаграми варіантів використання

Робота над проектом у середовищі Rational Rose починається із загального аналізу проблеми і побудови діаграми варіантів використання, яка відображає функціональне призначення програмної системи, що проектується. Загальні рекомендації з побудови діаграми варіантів використання наведені в розділі 1.

Для розроблення діаграми варіантів використання в середовищі Rational Rose необхідно активізувати відповідну діаграму у вікні діаграми. Для цього слід розкрити подання варіантів використання в браузері (Use Case View) і двічі клацнути на піктограмі Main або скористатися пунктом меню Browse / Use Case Diagram. Виникне спеціальна панель інструментів, що містить графічні примітиви, характерні для розроблення діаграми варіантів використання (рис. 8.2).

На цій панелі інструментів існують всі необхідні для побудови діаграми варіантів використання елементи. Про призначення окремих кнопок панелі можна дізнатися зі спливаючих підказок. Щоб додати елемент, слід натиснути кнопку із зображенням відповідного примітиву, після чого клацнути мишею на вільному місці діаграми. На діаграмі з'явиться зображення вибраного елемента з маркерами зміни його геометричних розмірів і запропонованою середовищем назвою за замовчуванням.

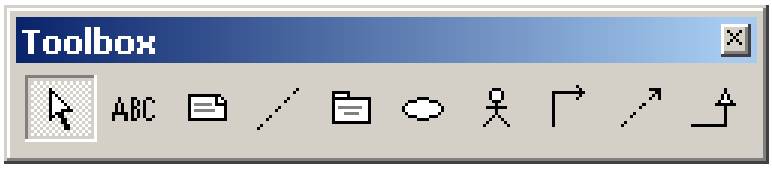


Рисунок 8.2 – Панель інструментів для діаграми варіантів використання

Назву елемента може бути змінено розробником або відразу після розміщення елемента на діаграмі, або в ході подальшої роботи над проектом. Після клацання правою кнопкою миші на вибраному елементі викликається контекстне меню елемента, серед опцій якого є пункт Open Specification (Відкрити специфікацію). У цьому випадку активізується діалогове вікно зі спеціальними вкладками, в поля яких можна занести всю інформацію щодо даного елемента.

На рис. 8.3 показано приклад побудованої таким способом діаграми варіантів використання.

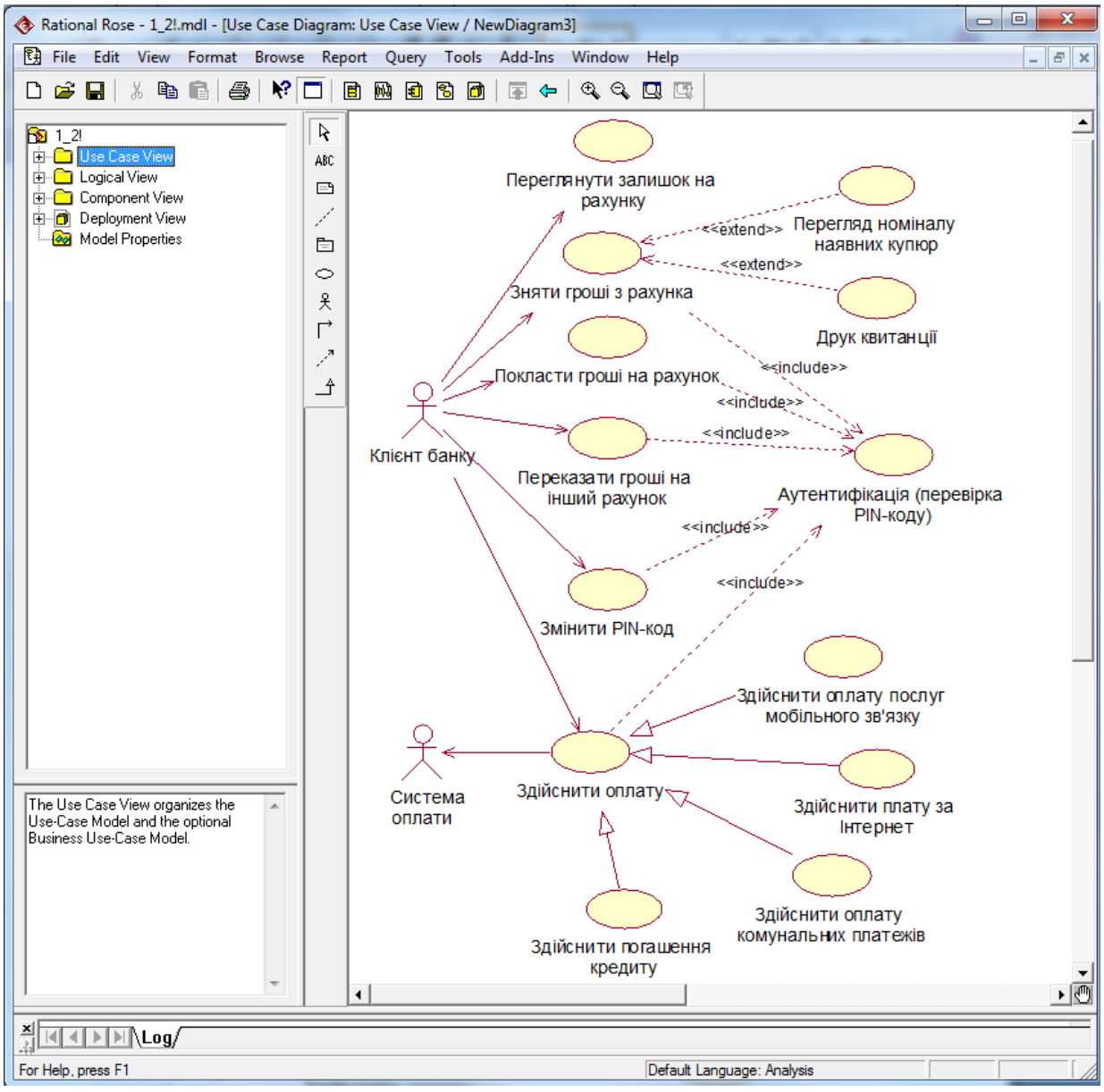


Рисунок 8.3 – Приклад розроблення діаграми варіантів використання в середовищі Rational Rose

Слід пам'ятати, що діаграма варіантів використання є високорівневим поданням моделі, тому вона не повинна містити занадто багато варіантів використання і акторів. У подальшому побудована діаграма може бути змінена додаванням нових елементів, таких, як варіанти використання і актори, або їх видаленням. Для видалення елемента не тільки з діаграми, але й з моделі в цілому необхідно виділити цей елемент на діаграмі і скористатися пунктом меню Edit / Delete from Model (за допомогою пункту меню Edit / Delete виділений елемент буде видалено тільки з діаграми, але він залишиться в моделі і його можна побачити через браузер).

При роботі зі зв'язками на діаграмі варіантів використання слід пам'ятати про призначення відповідних зв'язків. Йдеться про те, що якщо для двох елементів вибраний вид зв'язку не є допустимим, то середовище повідомить про це розробнику, і такий зв'язок не буде доданий на діаграму.

Розроблення діаграми класів

Діаграма класів є основним логічним поданням моделі і містить найдетальнішу інформацію про внутрішній устрій об'єктної програмної системи.

Ця діаграма відображається за замовчуванням у вікні діаграми після створення нового проекту. Викликати її можна також, клацнувши на кнопці із зображенням діаграми класів на стандартній панелі інструментів, або якщо розкрити логічне подання в браузері (Logical View) і двічі клацнути на піктограмі Main. Можна скористатися пунктом меню Browse / CIass Diagram.

Після активізації діаграми класів спеціальна панель інструментів набуде вигляду, показаного на рис. 8.4. Додавання і видалення елементів відбувається аналогічно, однак у кожного класу є велика специфікація, що містить інформацію про його атрибути і операції.

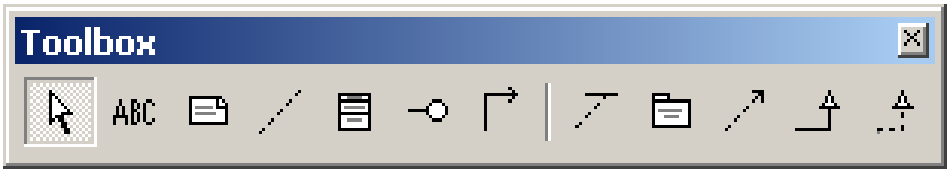


Рисунок 8.4 – Панель інструментів для діаграми класів

Для окремих атрибутів виділеного класу можна задати видимість, тип даних і початкові значення атрибута, а також призначити стереотип через пункт контекстного меню Open Specification. При цьому

пропонується вибір відповідних значень із списку. Для окремих операцій вибраного класу можна задати видимість і тип результату, що повертається, додати аргументи до операцій, задати виняткові ситуації і цілий ряд додаткових властивостей. Ці властивості операції доступні через пункт контекстного меню Open Specification і вкладку Operations. При подвійному натисканні на вибраній операції відкривається додаткове вікно зі вкладками, які відповідають окремим із зазначених раніше властивостям.

Додавання на діаграму класів відносин (зв'язків) між класами типу асоціацій, залежностей, агрегацій і узагальнень виконується таким чином. На спеціальній панелі інструментів вибирають необхідний тип зв'язку клацанням по кнопці з відповідним зображенням. Якщо зв'язок спрямований, то на діаграмі класів треба виділити перший елемент зв'язку (джерело, від якого виходить зв'язок) і, не відпускаючи натиснуту ліву кнопку миші, перемістити її покажчик до другого елемента зв'язку (приймач, до якого спрямований зв'язок). Після переміщення до другого елемента кнопку миші слід відпустити, а на діаграму класів буде додано новий зв'язок. Якщо ж зв'язок ненаправлений (або двонаправлений), то порядок вибору класів для цього зв'язку є довільним.

Для зв'язків можна визначити кратність кожного з кінців зв'язку, задати назву і стереотип, використати обмеження і ролі, а також деякі інші властивості. Доступ до специфікації зв'язку можна отримати після виділення зв'язку на діаграмі і виклику контекстного меню клацанням правої кнопки миші.

На рис. 8.5 показано приклад побудованої таким способом діаграми класів.

Розроблення діаграми станів

Переходячи до розгляду діаграми станів, слід зазначити, що в середовищі Rational Rose цей тип діаграм відноситься тільки до окремого класу. Для того, щоб побудувати діаграму станів для класу, його спочатку необхідно створити і уточнити. Після цього виділити на діаграмі класів або в браузері. Почати побудову діаграми станів для вибраного класу можна, якщо розкрити логічне подання в браузері (Logical View), виділити розглянутий клас і вибрати пункт контекстного меню New / Statechart Diagram, що розкривається після клацання правою кнопкою миші. Крім того, можна скористатися пунктом меню Browse / State Diagram.

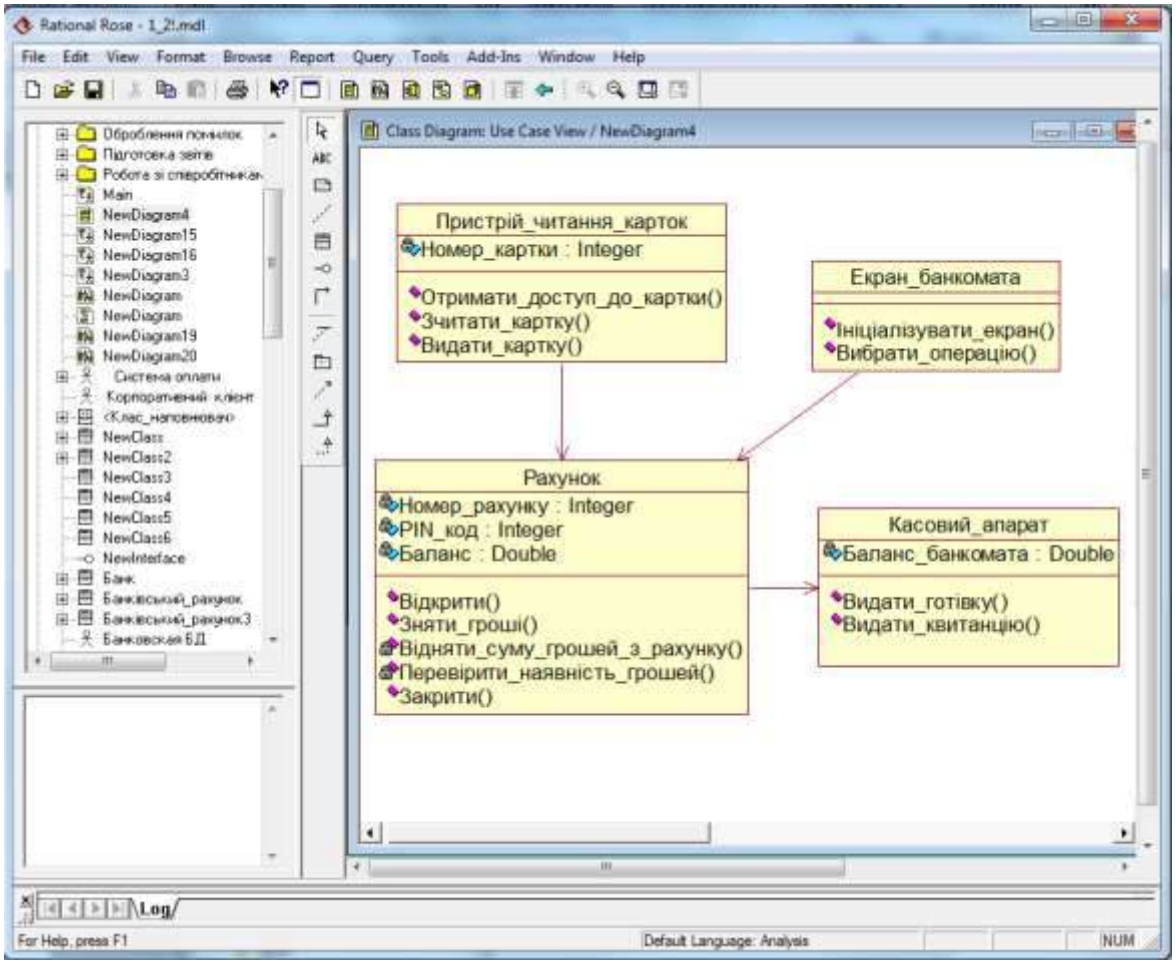


Рисунок 8.5 – Приклад розроблення діаграми класів
у середовищі Rational Rose

Після виконання зазначених дій у вікні діаграми з'явиться чисте зображення для розміщення елементів цієї діаграми, які вибирають за допомогою спеціальної панелі інструментів (рис. 8.6).

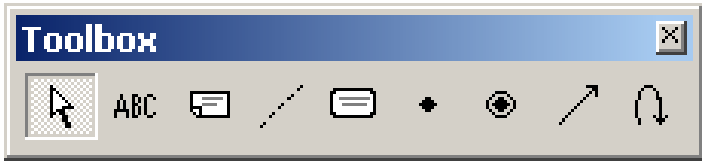


Рисунок 8.6 – Панель інструментів для діаграми станів

Як видно з рисунка, в середовищі відсутні деякі з розглянутих раніше елементів діаграми станів. Процес додавання і видалення станів і переходів на діаграму станів аналогічний цим діям з елементами інших діаграм.

На рис. 8.7 показано приклад побудованої діаграми станів.

Після додавання стану або переходу на діаграму станів можна відкрити специфікацію вибраних елементів і визначити їх специфічні риси, доступні на відповідних вкладках. За необхідності можна візуалізувати вкладеність станів і підключити історію окремих станів.

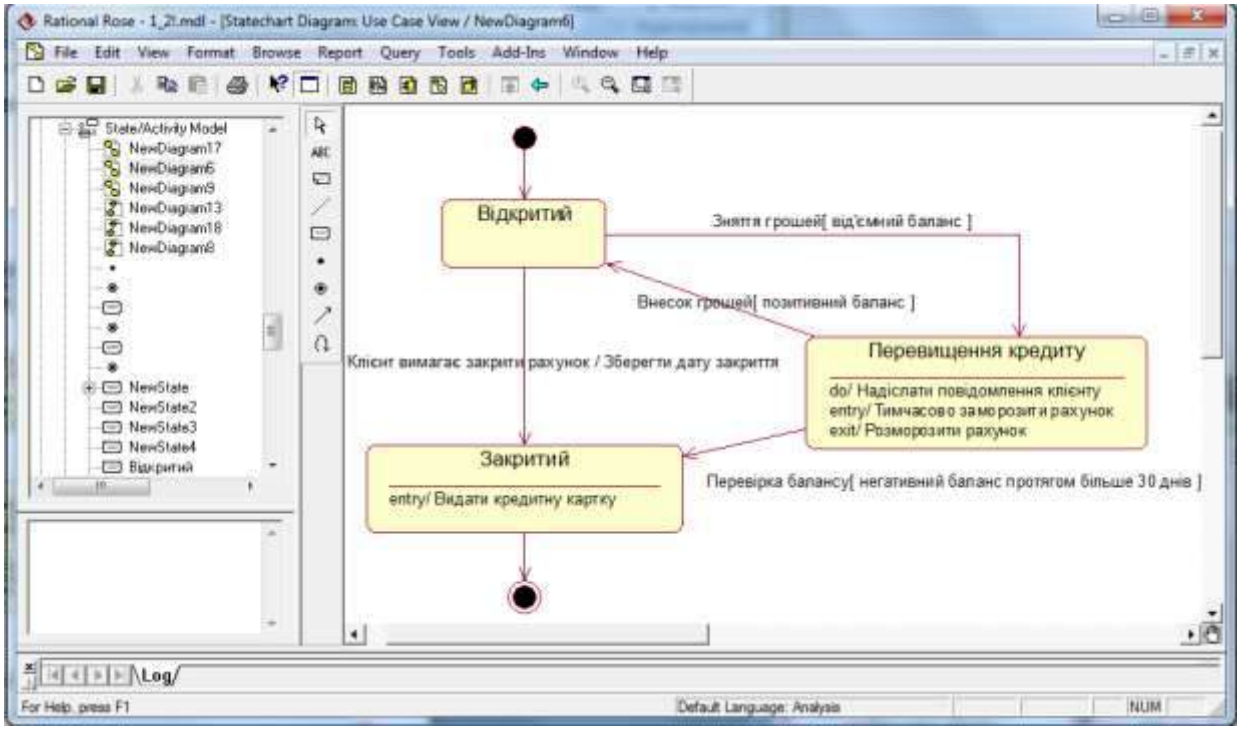


Рисунок 8.7 – Приклад розроблення діаграми станів
у середовищі Rational Rose

Розроблення діаграми діяльності

Почати побудову діаграми діяльності для вибраного класу можна, якщо розкрити логічне подання в браузері (Logical View), виділити розглянутий клас і вибрати пункт контекстного меню New / Activity Diagram, що розкривається після клацання правою кнопкою миші. Дана діаграма не викликається через пункт меню Browse.

Після виконання зазначених дій у вікні діаграми з'явиться чисте зображення для розміщення елементів цієї діаграми, які вибираються за допомогою спеціальної панелі інструментів (рис. 8.8).

Після додавання дії або переходу на діаграму діяльності можна відкрити специфікацію вибраних елементів і визначити їх специфічні риси, доступні на відповідних вкладках.

На рис. 8.9 показано приклад побудованої діаграми діяльності.

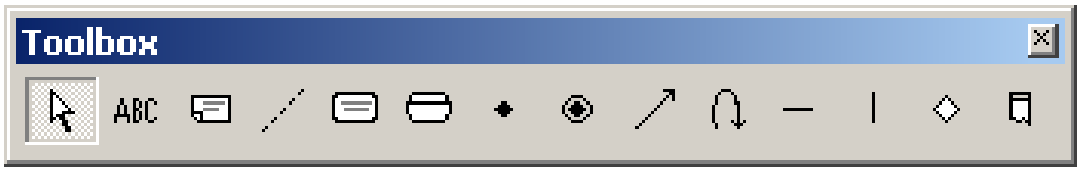


Рисунок 8.8 – Панель інструментів для діаграми діяльності

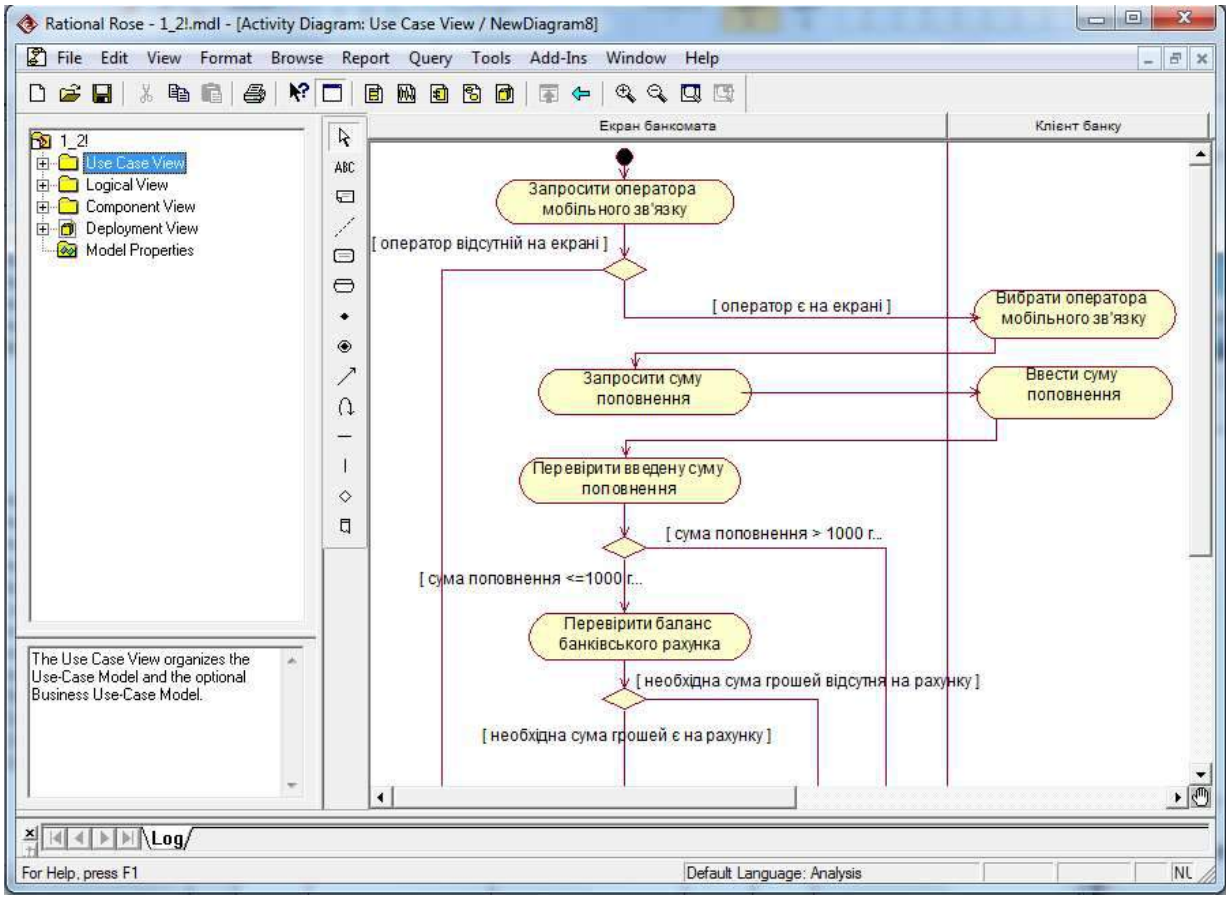


Рисунок 8.9 – Приклад розроблення діаграми діяльності в середовищі Rational Rose

Розроблення діаграми послідовності

Діаграма послідовності може бути активізована за допомогою кнопки із зображенням діаграми послідовності на стандартній панелі інструментів або через пункт меню Browse / Interaction Diagram.

Після виконання зазначених дій у вікні діаграми з'явиться чисте зображення для розміщення елементів діаграми послідовності, які вибирають за допомогою спеціальної панелі інструментів (рис. 8.10).

Побудова діаграми послідовності зводиться до додавання або видалення окремих об'єктів і повідомлень, а також до їх специфікації.

Доступ до специфікації цих елементів організований або через контекстне меню, або через пункт меню Browse / Specification. При додаванні повідомлень на діаграму послідовності вони отримують за замовчуванням свій номер у послідовності.

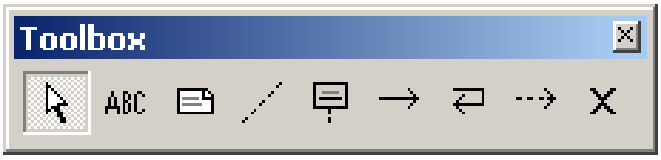


Рисунок 8.10 – Панель інструментів для діаграми послідовності

На рис. 8.11 показано приклад побудованої діаграми послідовності.

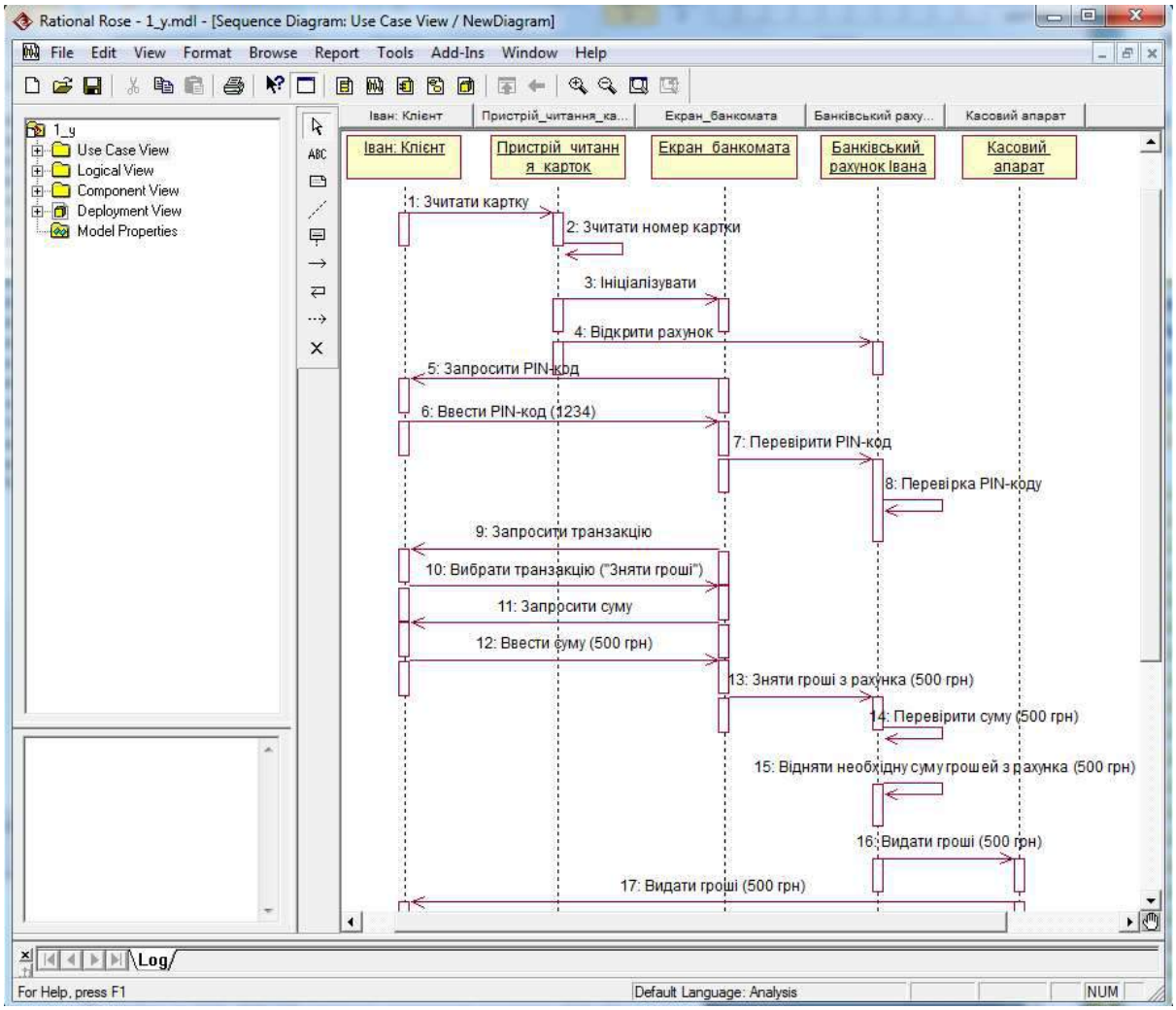


Рисунок 8.11 – Приклад розроблення діаграми послідовності в середовищі Rational Rose

З необхідності можна змінити порядок проходження повідомлень та їх специфікацію, а також зіставити повідомлення з операціями. Додатково можна встановлювати синхронізацію повідомлень, зв'язати з повідомленням примітки (коментари) за допомогою скриптів.

Розроблення діаграми кооперації

Діаграма кооперації є іншим способом візуалізації взаємодії в моделі і, як і діаграма послідовності, оперує об'єктами і повідомленнями. Особливість роботи в середовищі Rational Rose полягає в тому, що цей вид канонічної діаграми створюється автоматично після побудови діаграми послідовності і натискання клавіші <F5>. За допомогою цієї ж клавіші здійснюється перемикання між діаграмами послідовності і кооперації.

Після того, як діаграма кооперації активізована, спеціальна панель інструментів набуває вигляду, показаного на рис. 8.12.

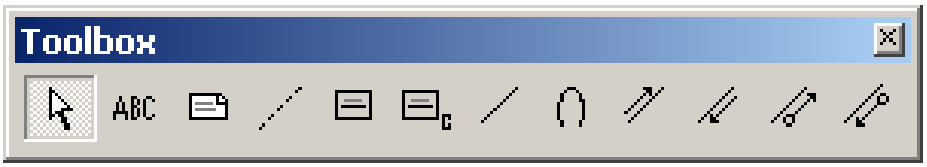


Рисунок 8.12 – Панель інструментів для діаграми кооперації

На цій панелі є кнопки з піктограмами об'єктів і різних типів повідомлень. Робота з діаграмою кооперації полягає в додаванні або видаленні об'єктів і повідомлень, а також заповненні їх специфікації. При цьому зміни, що вносяться до діаграми кооперації, автоматично вносяться і в діаграму послідовності, що можна побачити, активізувавши останню натисканням клавіші <F5>.

На рис. 8.13 показано приклад діаграми кооперації, яка була автоматично згенерована середовищем Rose після побудови діаграми послідовності (рис. 8.11).

Як і для діаграми послідовності, для діаграми кооперації можна змінювати порядок проходження повідомлень, додавати потоки даних, визначати стійкість об'єктів на основі активізації відповідних специфікацій.

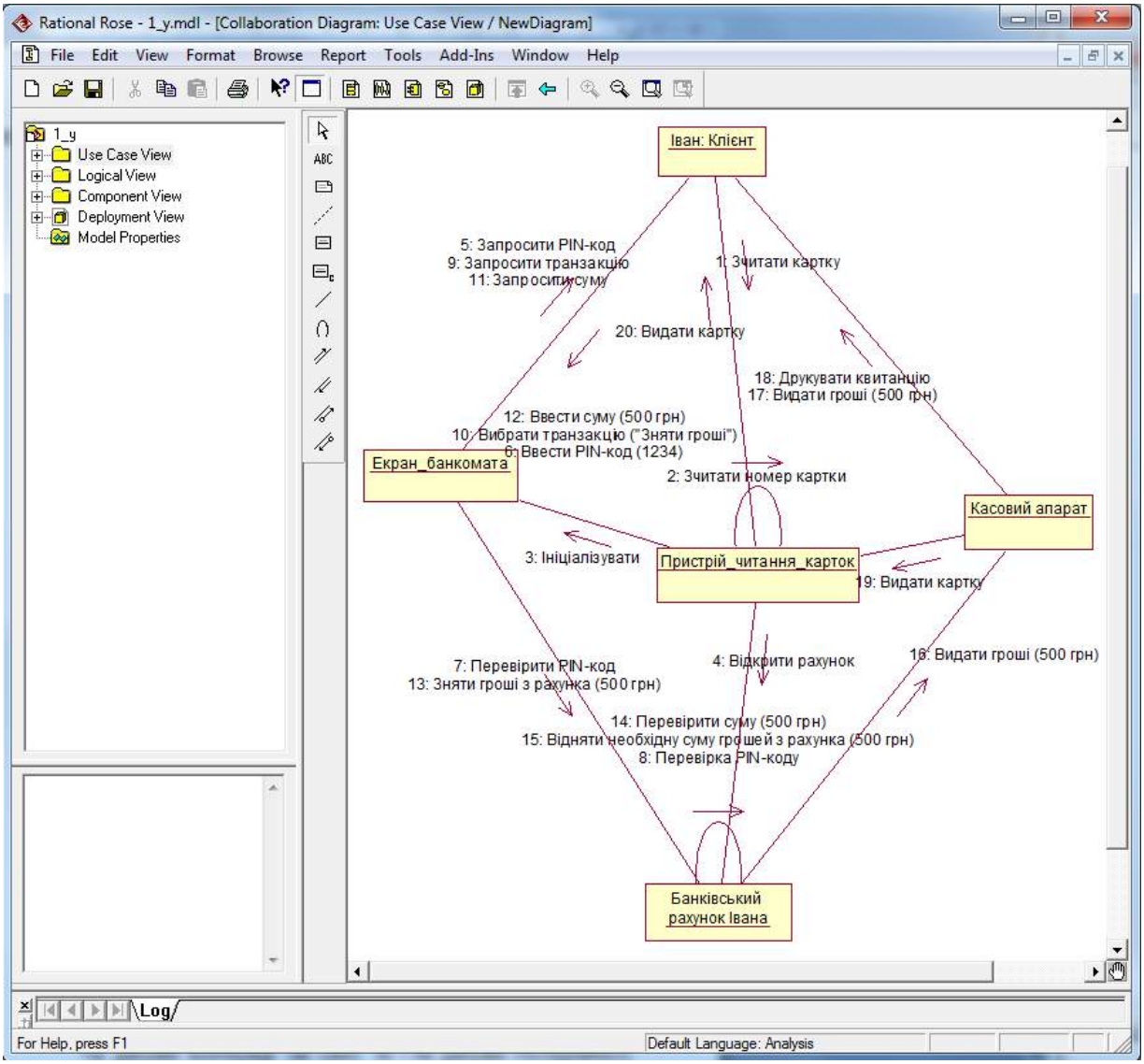


Рисунок 8.13 – Приклад розроблення діаграми кооперацій в середовищі Rational Rose

Розроблення діаграми компонентів

Діаграма компонентів є частиною фізичного подання моделі і відіграє важливу роль у процесі ООАП. Активізація діаграми компонентів може бути виконана, якщо клацнути на кнопці із зображенням діаграми компонентів на стандартній панелі інструментів. Можна також розкрити компонентне подання у браузері (Component View) і двічі клацнути на піктограмі Main або вибрати пункт меню Browse / Component Diagram.

Після активізації діаграми компонентів спеціальна панель інструментів набуде вигляду, показаного на рис. 8.14.

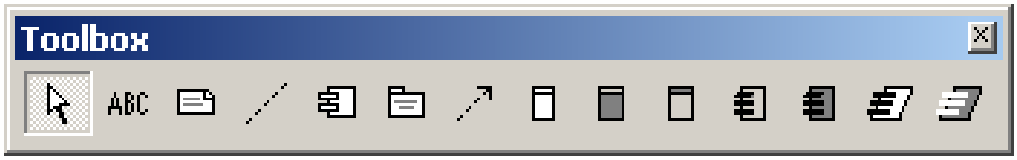


Рисунок 8.14 – Панель інструментів для діаграми компонентів

Додавання і видалення елементів відбувається аналогічно, однак для кожного компонента можна визначити різні деталі, такі, як стереотип, мова програмування, декларації, класи. Робота з цими деталями компонентів здійснюється через специфікацію компонента, яка стає доступною після виклику контекстного меню.

На рис. 8.15 показано приклад графічного зображення діаграми компонентів.

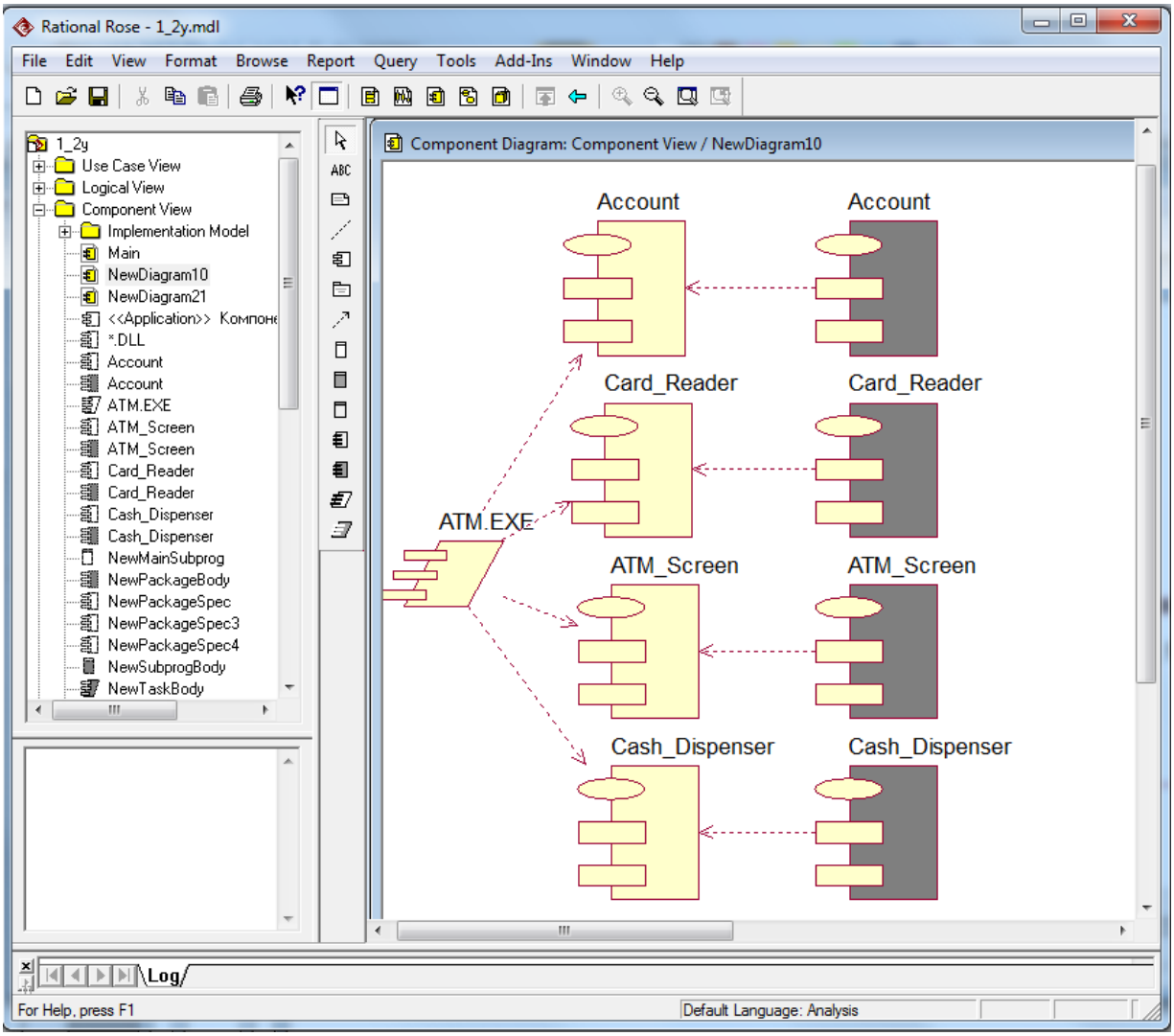


Рисунок 8.15 – Приклад розроблення діаграми компонентів у середовищі Rational Rose

При роботі з діаграмою компонентів можна створювати пакети і компоненти, змінювати їх специфікацію і залежності між різними елементами діаграми. При встановленні реалізації класів на компоненті можна виділити клас у браузері і перетягнути його на потрібний компонент діаграми.

Розроблення діаграми розгортання

Діаграма розгортання є другою складовою частиною фізичного подання моделі. Активізація діаграми розгортання може бути виконана, якщо клацнути на кнопці із зображенням діаграми розгортання на стандартній панелі інструментів. Можна також клацнути на піктограмі подання розгортання в браузері (Deployment View) або вибрати пункт меню Browse / Deployment Diagram.

Після активізації діаграми розгортання спеціальна панель інструментів набуде такого вигляду, показаного на рис. 8.16.

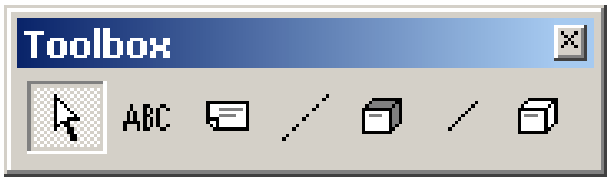


Рисунок 8.16 – Панель інструментів для діаграми розгортання

Робота з діаграмою розгортання полягає в створенні процесорів і пристроїв, їх специфікації, встановленні зв'язків між ними, а також додаванні і специфікації процесів. Що стосується окремих процесорів, то для них можна використовувати стереотипи.

На рис. 8.17 показано приклад графічного зображення діаграми розгортання.

Генерація коду на C++

Одним із найбільш потужних властивостей середовища Rational Rose є можливість генерації програмного коду після побудови моделі.

У процесі кодогенерації генерується «скелетний» код для описаних на відповідній діаграмі класів. Таким чином, під процесом кодогенерації розуміють процес класогенерації.

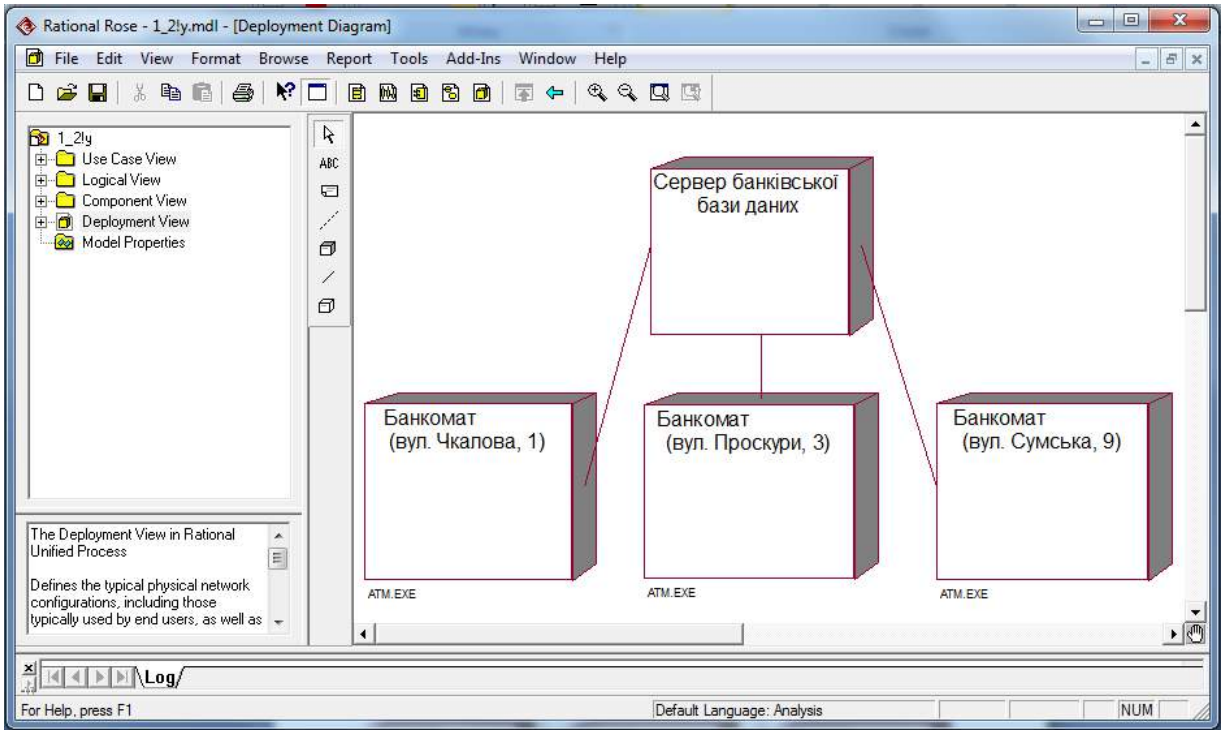


Рисунок 8.17 – Приклад розроблення діаграми розгортання в середовищі Rational Rose

Для початку процесу генерації коду мають бути описані всі класи, а також для кожного класу на діаграмі компонентів мають бути вказані компоненти опису цих класів. Так, для генерації коду мовою C++ для кожного класу з діаграми класів на діаграмі компонентів мають бути вказані заголовки (*.h) і файл реалізації (*.cpp). У свою чергу, на діаграмі компонентів у кожного компонента має бути встановлена прив'язка до конкретного класу з діаграми класів. Прив'язка виконується через пункт контекстного меню Open Specification і вкладку Realizes: необхідно вказувати клас і по правій кнопці миші вибрати Assign.

Далі на діаграмі компонентів потрібно виділити компонент, для якого буде генеруватися код, вибрати пункт меню Tools / C++ (тим самим вибирається мова для опису спроектованого класу) і вибрати Code Generation.

У результаті виконання зазначених вище дій над кожним компонентом з діаграми компонентів для кожного класу будуть згенеровані файли *.h і *.cpp.

ДОДАТОК

Словник термінів

Актор або дійова особа – зовнішня відносно проектованої системи сутність (користувач, інша система, час), яка взаємодіє з нею для отримання і надання будь-якої інформації.

Асоціація – семантичне відношення між класами або об'єктами класів.

Агрегація – асоціація між цілим і його частиною або частинами.

Варіант використання або прецедент – опис на «високому рівні» фрагмента функціональності (набору сценаріїв), яку забезпечує система, що моделюється.

Взаємодія – обмін інформацією між елементами системи, що моделюється.

Вузол – частина апаратних засобів проектованої системи.

Діяльність – спільна поведінка об'єктів системи, що моделюється, спрямована на вирішення конкретної задачі.

Залежність – відношення, яке показує, що зміна в одному елементі відношення спричинить зміни і в іншому елементі відношення.

Інтерфейс – семантична і синтаксична конструкція для специфікації послуг, які надаються класом чи компонентом.

Клас – опис множини однорідних об'єктів реального світу (шаблон однорідних об'єктів).

Композиція – різновид агрегації: об'єкт-частина створюється і знищується разом з об'єктом-цілим.

Компонент – фізичний модуль програмного коду системи, що моделюється.

Кооперація – опис деякої спільної поведінки.

Узагальнення – відношення між класами, що дозволяє успадковувати класу-нащадку вміст (дані і поведінку) класу-предка.

Об'єкт – конкретна реалізація (екземпляр) класу.

Відношення – довільний, що не залежить від часу, зв'язок між елементами моделі.

Пакет – засіб угруповання класів та інших програмних ресурсів (у тому числі і самих пакетів) єдиного призначення.

Роль – певний контекст поведінки, асоційований зі зв'язком між об'єктами класів (об'єкти класів відіграють відносно один до одного відповідні ролі).

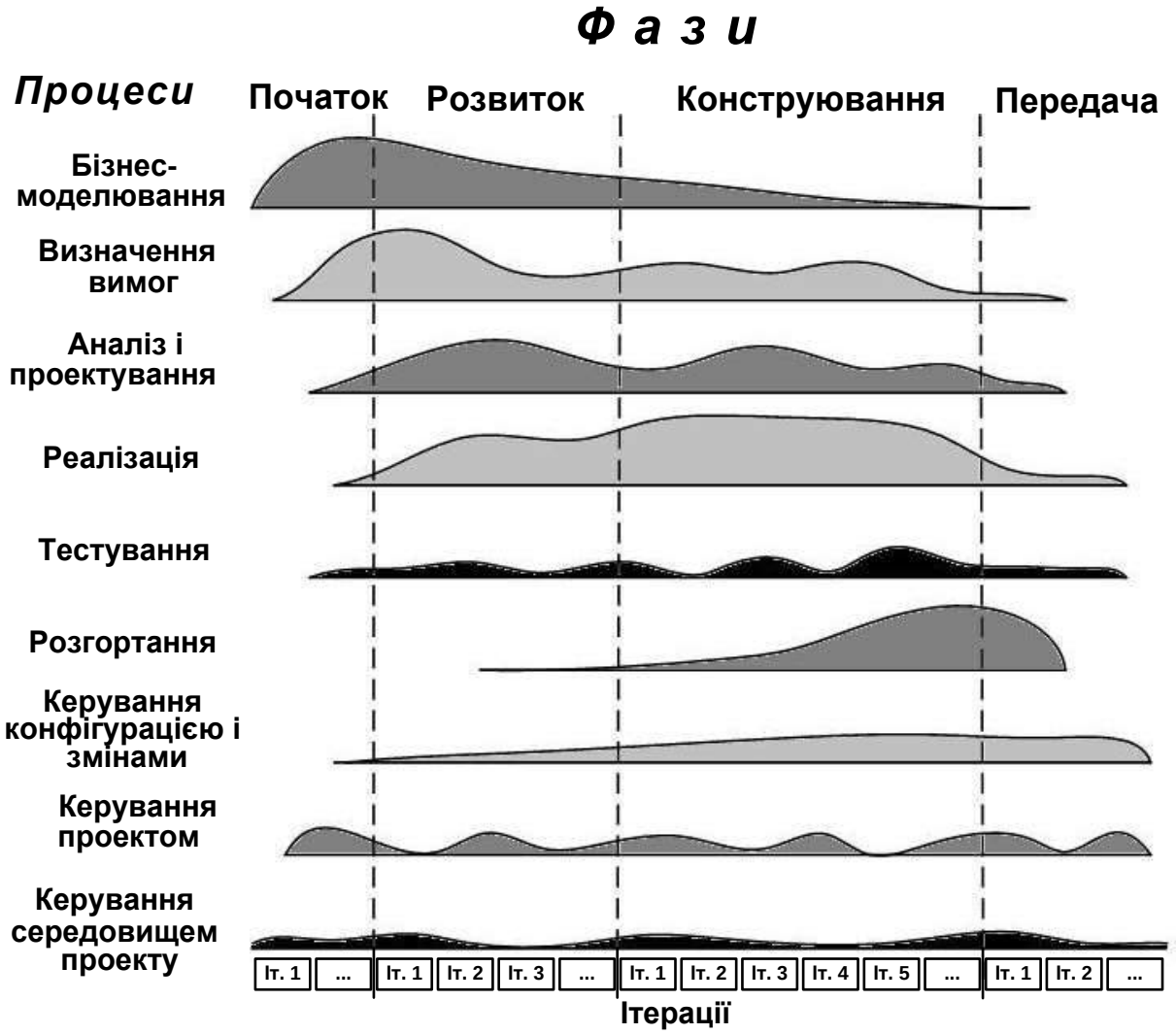
Повідомлення – зв'язок між об'єктами, в якому один з них запитує в іншого виконання якихось дій (функцій).

Стан – одна з можливих ситуацій існування об'єкта, в якій він може виконувати деяку діяльність або чекати на деяку подію.

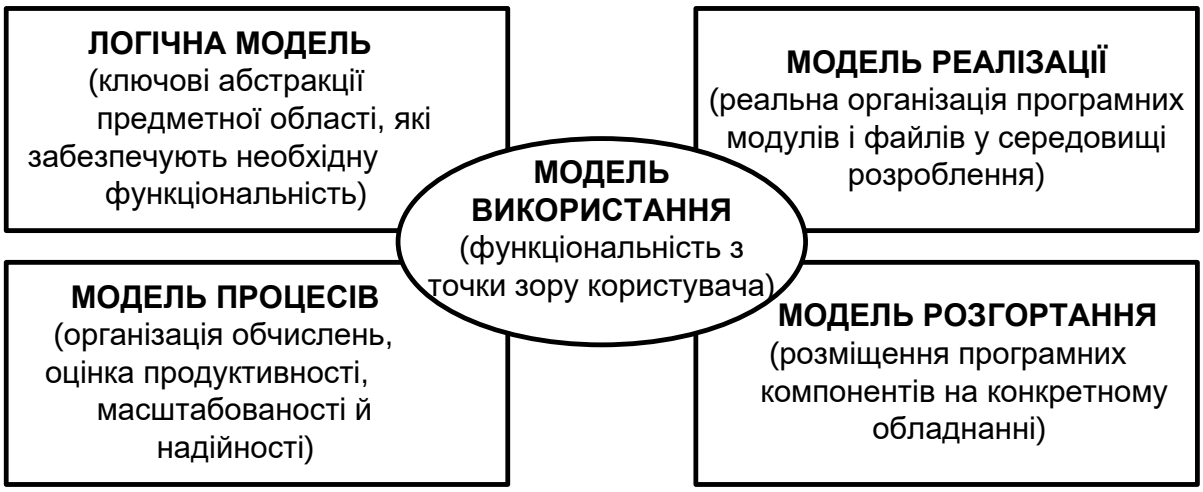
Стереотип – механізм, що дозволяє розширювати семантику мови UML.

Сценарій – послідовність кроків, що описує взаємодію актора і системи або об'єктів.

RUP (Rational Unified Process)

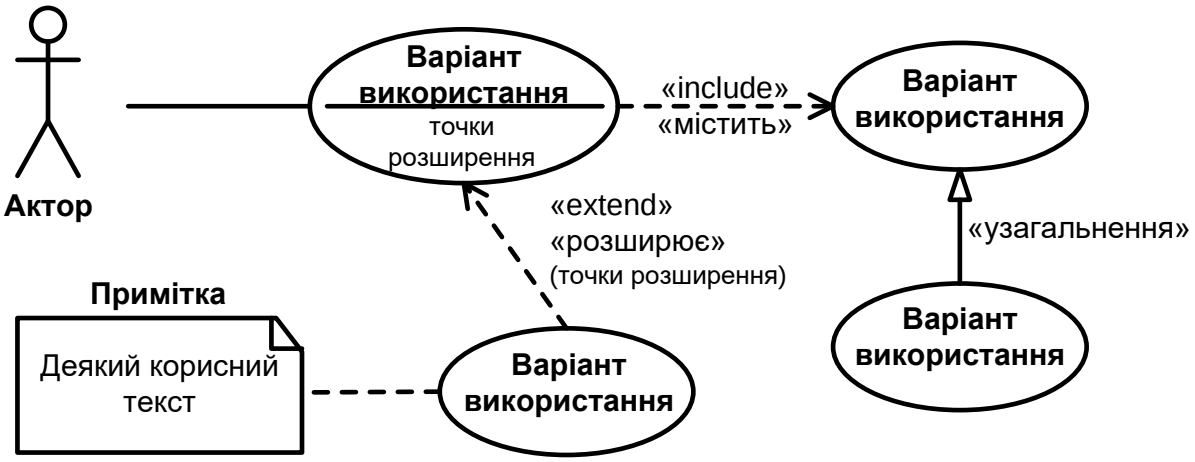


Специфікація ПЗ, що розробляється, при використанні UML



Діаграма варіантів використання (use case diagram)

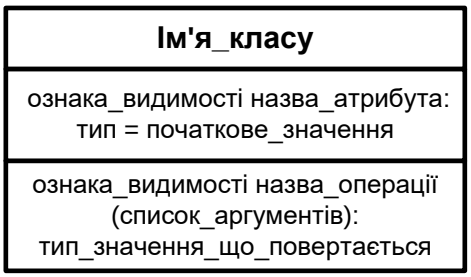
Показує набір варіантів використання, акторів і їх зв'язки (відноситься до статичного подання системи).



Діаграма класів (class diagram)

Показує набір класів та інтерфейсів, а також їх зв'язки (відноситься до статичного подання системи).

Клас



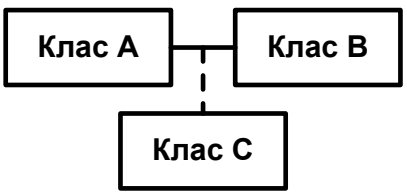
Ознаки видимості атрибутів

\+ загальний (public)

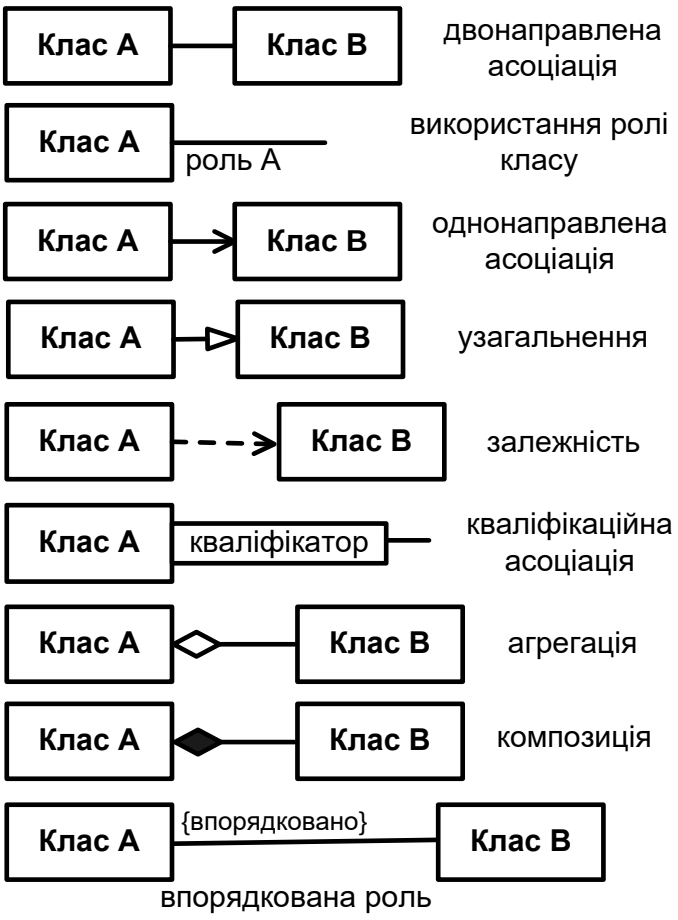
\- прихований (private)

\# захищений (protected)

Відношення класу асоціації

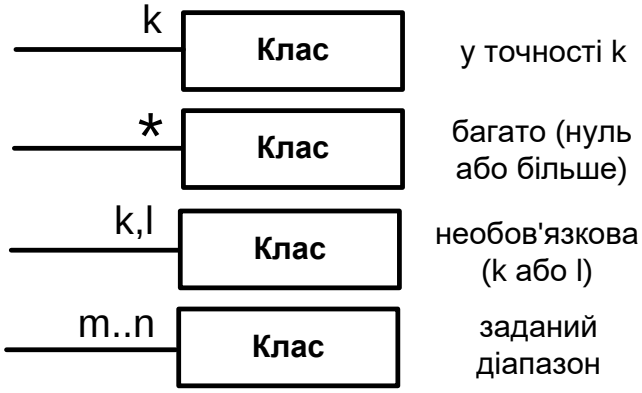


Відношення між класами



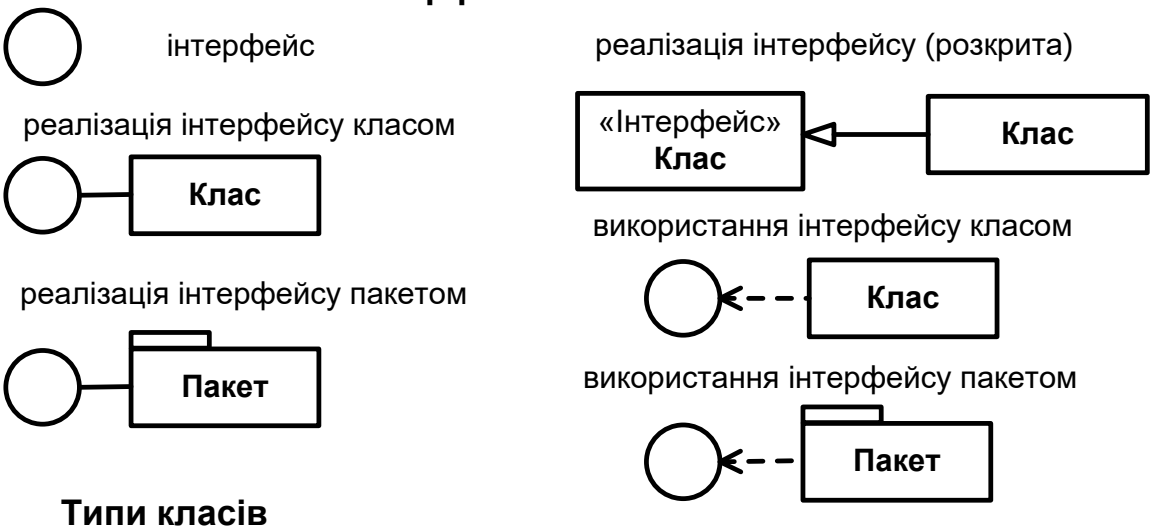
Діаграма класів (class diagram)

Кратності

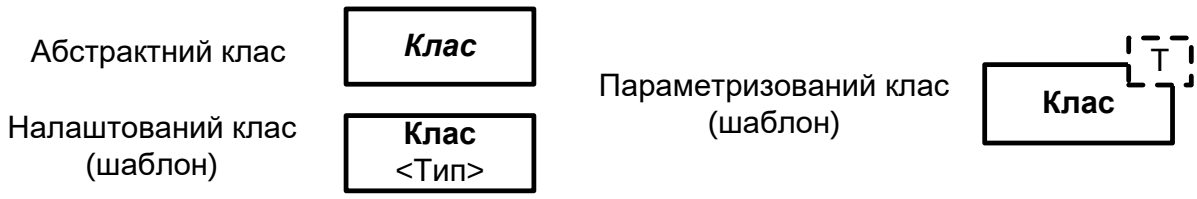


Стереотипи класів

Інтерфейси

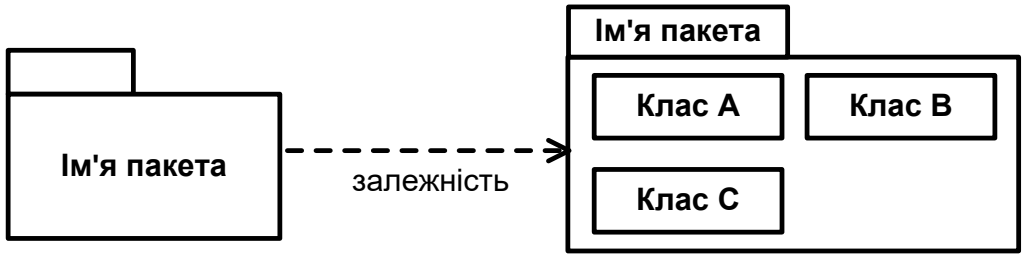


Типи класів



Діаграма пакетів (package diagram)

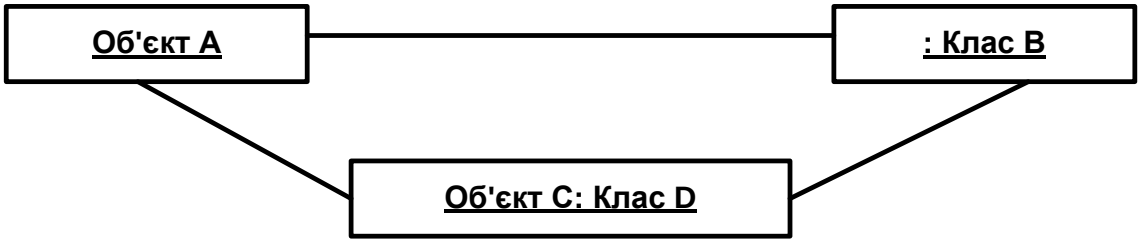
Є різновидом діаграми класів, показує об'єднання класів у пакети і зв'язки між пакетами (відноситься до статичного подання системи).



Діаграма об'єктів (object diagram)

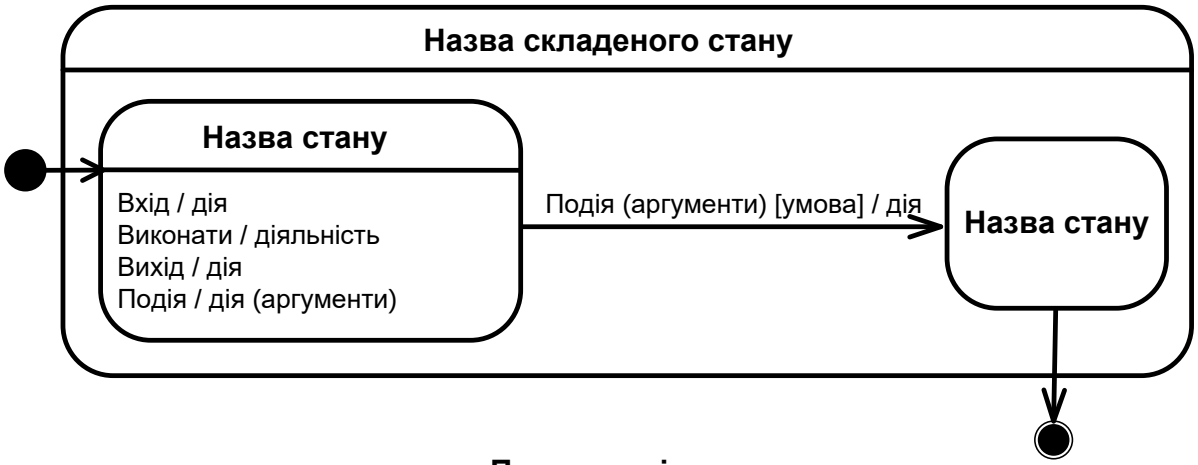
Показує набір об'єктів і їх зв’язки в деякий момент часу (відноситься до статичного подання системи).

Позначення об'єктів класів і зв'язків між ними

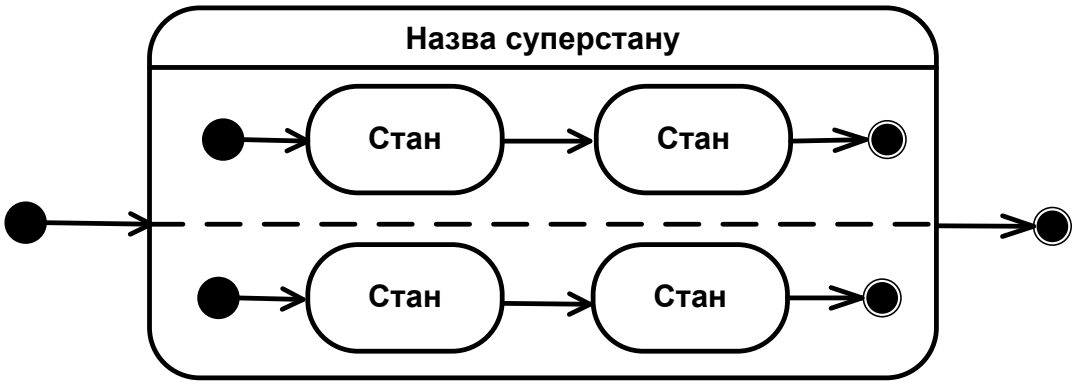


Діаграма станів (statechart diagram)

Показує життєвий цикл одного об'єкта: набір станів, починаючи з моменту його створення і закінчуючи знищенням, а також умови переходу з одного стану в інший (відноситься до динамічного подання системи).

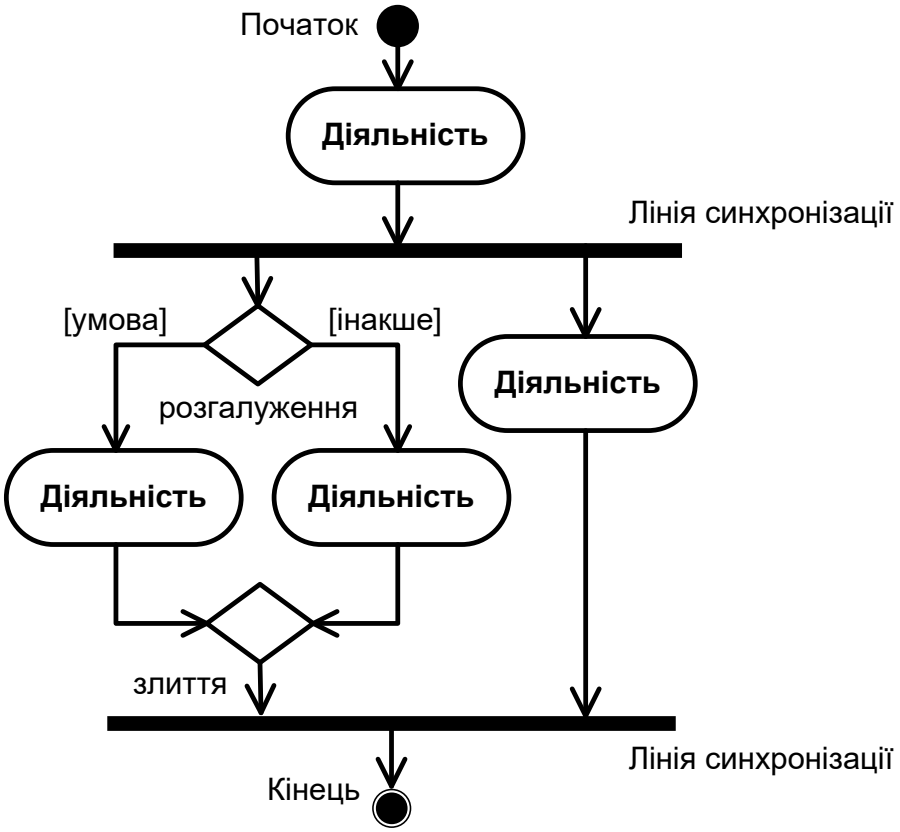


Паралельні стани



Діаграма діяльності (activity diagram)

Показує переходи від одного виду діяльності до іншого для вирішення деякої задачі (відноситься до динамічного подання системи).



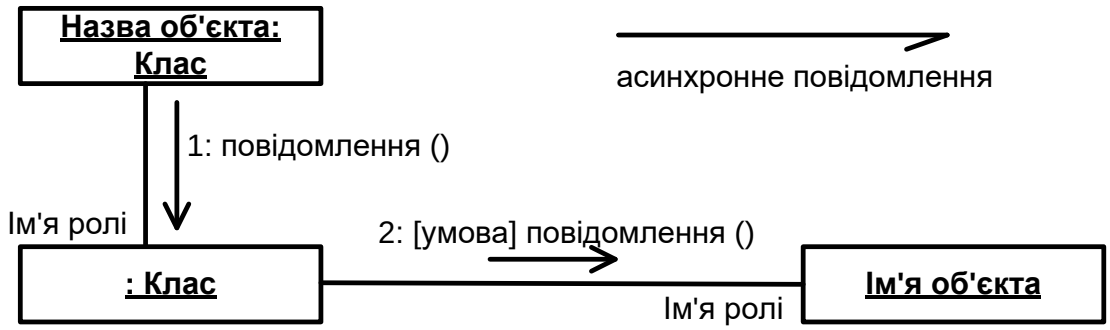
Діаграма взаємодії (interaction diagram)

Показує поведінку взаємодіючих груп об'єктів (відноситься до динамічного подання системи), містить два типи діаграм:

- діаграму послідовності;
- діаграму кооперації.

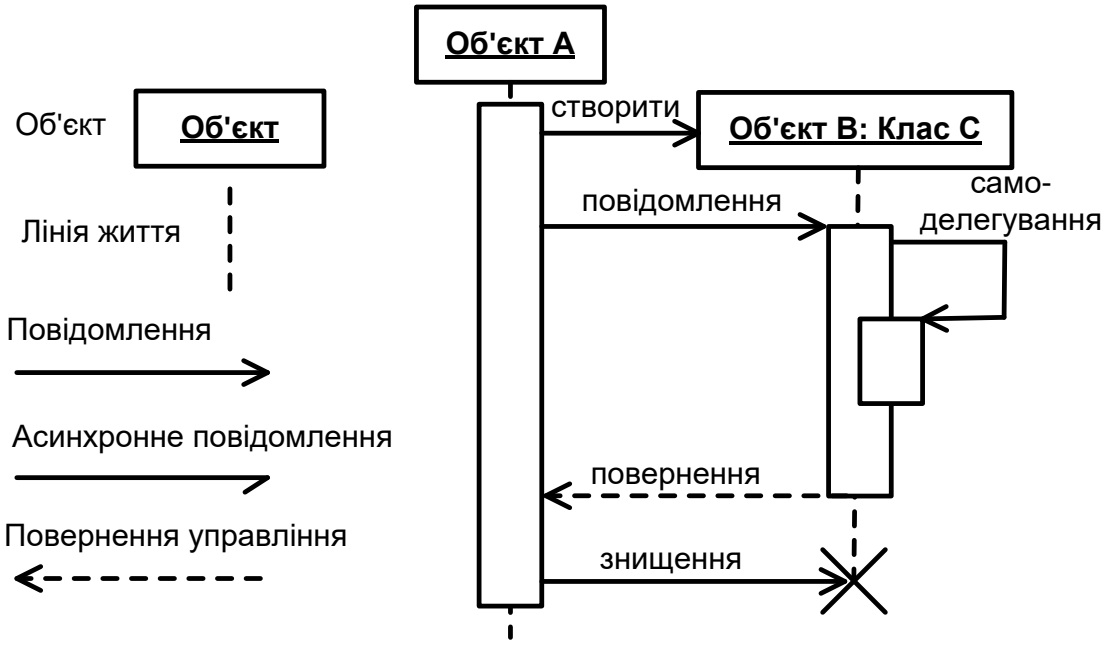
Діаграма кооперації (collaboration diagram)

Є різновидом діаграми взаємодії, показує потоки даних між взаємодіючими об'єктами, що дозволяє уточнити зв'язок між ними.



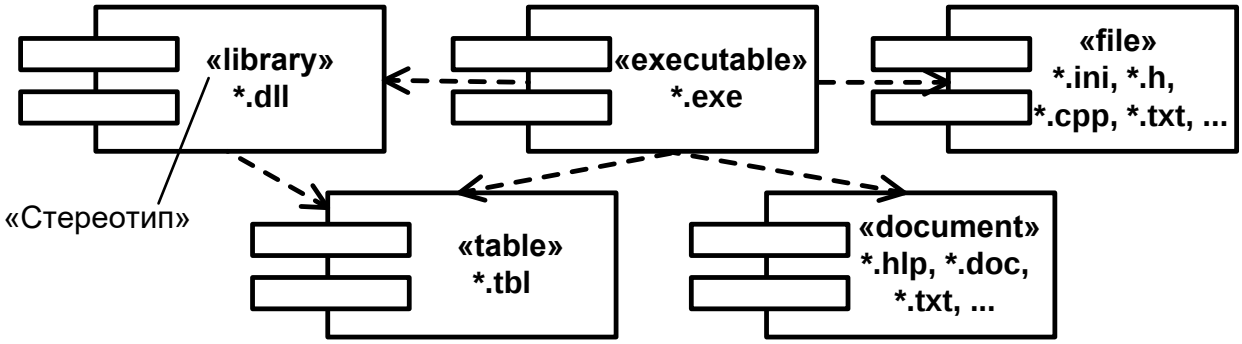
Діаграма послідовності (sequence diagram)

Є різновидом діаграми взаємодії, показує впорядковану за часом взаємодію об'єктів.



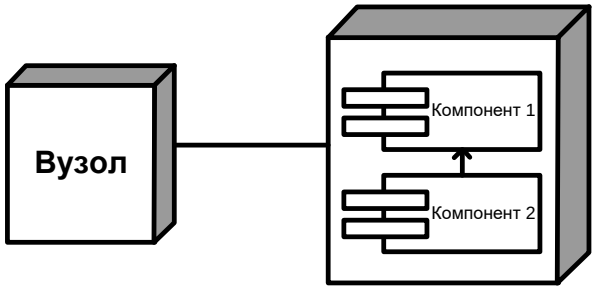
Діаграма компонентів (component diagram)

Показує набір програмних компонентів проектованої системи і зв'язки між ними (відноситься до статичного подання системи).

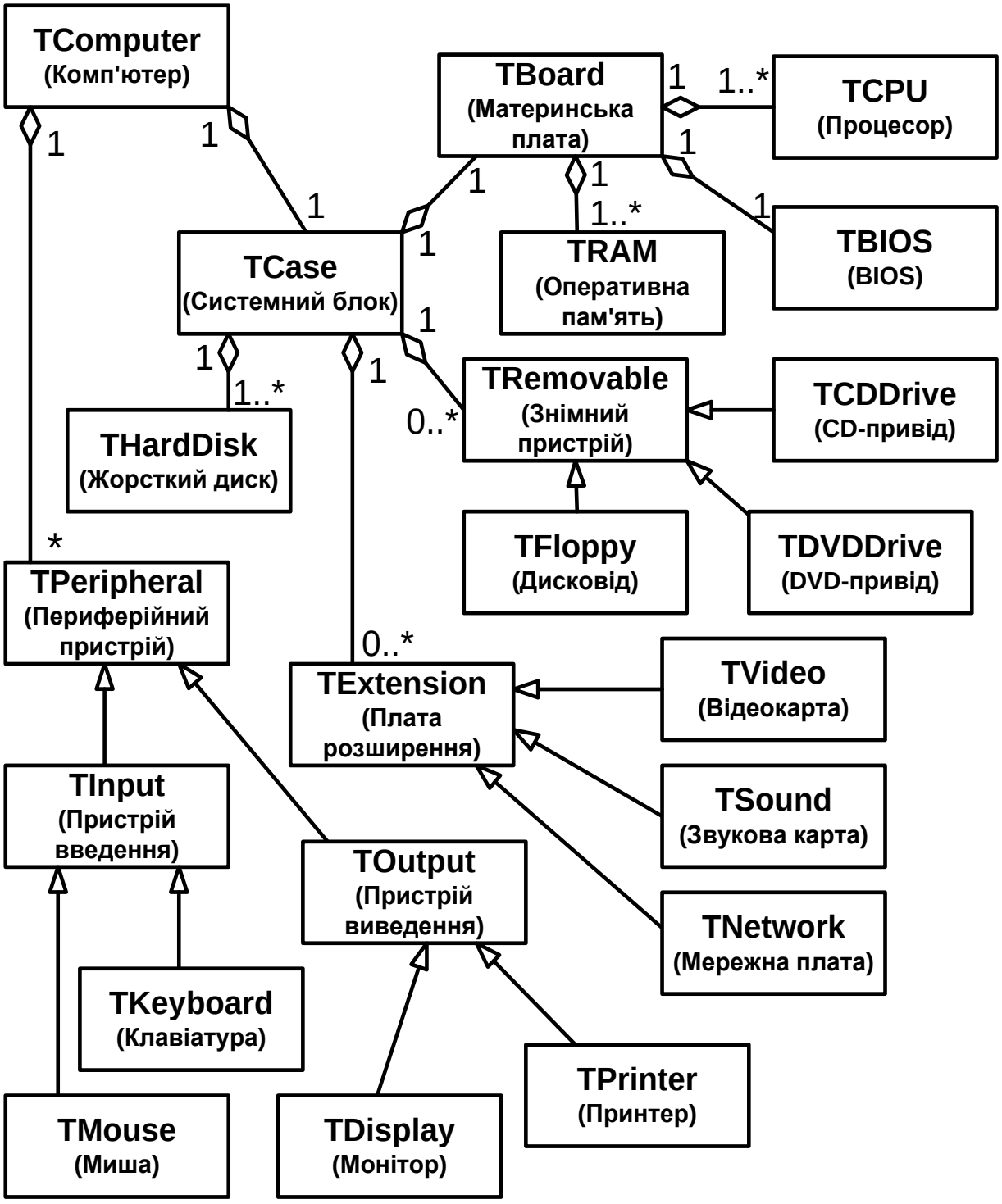


Діаграма розгортання (deployment diagram)

Показує фізичні взаємозв'язки програмних й апаратних елементів (вузлів) проектованої системи (відноситься до статичного подання системи).



Приклад діаграми класів, яка описує пристрій «Комп'ютер»



БІБЛІОГРАФІЧНИЙ СПИСОК

Боггс, У. UML и Rational Rose 2002 / У. Боггс, М. Боггс. – М.: Лори, 2004. – 528 с.

Буч, Г. UML. Руководство пользователя / Г. Буч, Дж. Рамбо, А. Джекобсон. – М.: ДМК, 2000. – 432 с.

Дудзяний, І. М. Об'єктно-орієнтоване моделювання програмних систем: навч. посіб. – Львів: Видавничий центр ЛНУ імені Івана Франка, 2007. – 108 с.

Кватрани, Т. Rational Rose 2000 и UML. Визуальное моделирование / Т. Кватрани. – М.: ДМК Пресс, 2001. – 176 с.

Ларман, К. Применение UML 2.0 и шаблонов проектирования / К. Ларман. – М.: ООО «Вильямс», 2007. – 736 с.

Леоненков, А. В. Самоучитель UML / А. В. Леоненков. – СПб.: BHV, 2004. – 432 с.

Трофимов, С. А. CASE-технологии: практическая работа в Rational Rose / С. А. Трофимов. – М.: Бином-Пресс, 2002. – 288 с.

Фаулер, М. UML. Основы / М. Фаулер, К. Скотт. – СПб.: Символ-Плюс, 2002. – 192 с.

Федотова, Д. Э. CASE-технологии: практикум / Д. Э. Федотова, Ю. Д. Семенов, К. Н. Чижик. – М.: Горячая линия-Телеком, 2005. – 160 с.

Шмуллер, Дж. Освой самостоятельно UML за 24 часа / Дж. Шмуллер. – М.: Изд. дом «Вильямс», 2002. – 352 с.

ЗМІСТ

Навчальне видання

Шевченко Ілона Володимирівна
Кузнецова Юлія Анатоліївна

ПРОЕКТУВАННЯ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ. ОСНОВИ ПОБУДОВИ UML-ДІАГРАМ

Редактор Т. Г. Кардаш

Зв. план, 2019
Підписано до видання 04.10.2019
Ум. друк. арк. 4,6. Обл.-вид. арк. 5,13. Електронний ресурс

Видавець і виготовлювач
Національний аерокосмічний університет ім. М. Є. Жуковського
«Харківський авіаційний інститут»
61070, Харків-70, вул. Чкалова, 17
http://www.khai.edu
Видавничий центр «ХАІ»
61070, Харків-70, вул. Чкалова, 17
izdat@khai.edu

Свідоцтво про внесення суб'єкта видавничої справи
до Державного реєстру видавців, виготовлювачів і розповсюджувачів
видавничої продукції сер. ДК № 391 від 30.03.2001