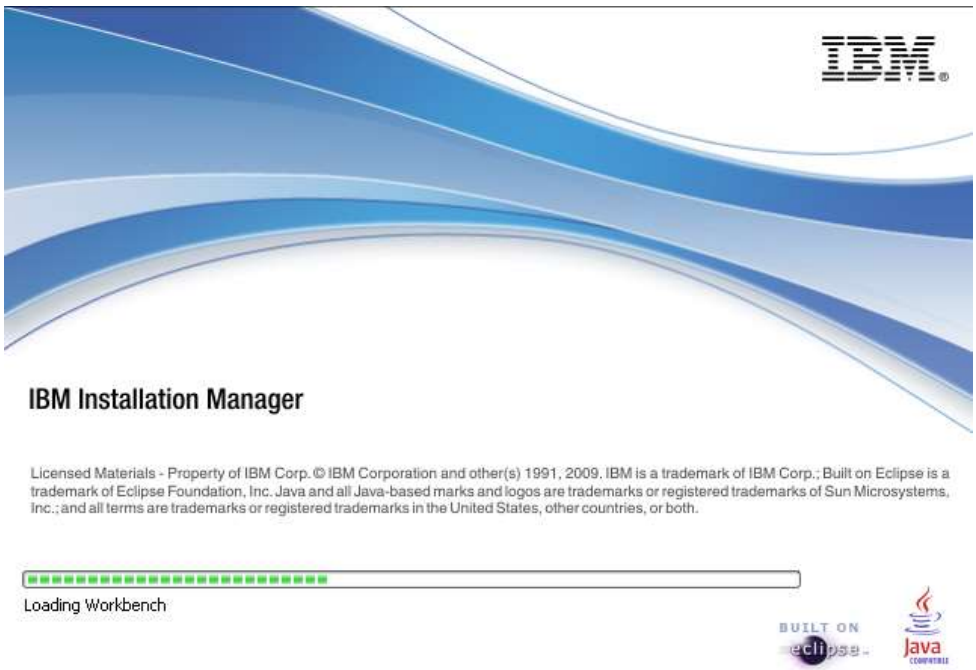
Міністерство освіти і науки України
Тернопільський національний технічний університет імені Івана Пулюя

кафедра програмної інженерії

Петрик М.Р., Мудрик І.Я.

Проєктування програмного забезпечення на основі об'єкнто-орієнтованого аналізу вимог та інструментальних засобів розробки IBM Rational Software Architect

(від Вимог до коду)



Науково-методичний посібник щодо змістовного наповнення основного розділу (**Проєктування**) кваліфікаційної роботи (другий освітній кваліфікаційний рівень «Магістр»)

спеціальність 121 – інженерія програмного забезпечення

Тернопіль 2022

Петрик М.Р. Проєктування програмного забезпечення на основі аналізу вимог та інструментальних засобів розробки IBM Rational Software Architect (від Вимог до коду)
Науково-методичний посібник. Тернопіль: Вид-во ТНТУ ім. Івана Пулюя.-2022.- 560с.

Рекомендовано до видання засіданням кафедри програмної інженерії (Протокол №1 від 01.09.2022) та рішенням НМР факультету комп'ютерно-інформаційних систем і програмної інженерії ТНТУ (Протокол №1 від 19.01.2022)

Рецензенти:

Федасюк Д.В., докт. техн. наук, професор, завідувач кафедри програмного забезпечення, Національний університет «Львівська політехніка»

Дудар З.В., канд. техн. наук, професор, завідувач кафедри кафедри програмної інженерії, Харківський національний університет радіоелектроніки

В посібнику викладені основи науково-методичного забезпечення із застосуванням новітніх інструментальних засобів розробки IBM Rational Software Architect для аналізу вимог, моделювання, проєктування та конструювання програмного забезпечення, що використовується в міждисциплінарному контексті для низки професійно-орієнтованих дисциплін «Аналіз вимог», «Об'єктно-орієнтоване програмування», «Моделювання та аналіз програмного забезпечення», «Архітектура та проєктування програмного забезпечення», «Конструювання програмного забезпечення», «Проєктний практикум програмної інженерії», «Менеджмент проєктів програмного забезпечення» та включає положення і практичні рекомендації щодо змістовного наповнення основного розділу (Проєктування) кваліфікаційної ї роботи (другий освітній кваліфікаційний рівень «Магістр») спеціальності 121 – інженерія програмного забезпечення.

Метою посібника є освоєння студентами сучасних підходів і методам проєктування програмного забезпечення (ПЗ), їх практичного застосування з використанням інструментальних засобів побудови моделей, що використовуються на різних етапах життєвого циклу ПЗ, від вимог до автоматизованого генерування коду. Він може бути корисним також для студентів суміжних спеціальностей, зокрема, «комп'ютерні науки», «комп'ютерна інженерія» та інших ІТ- спеціальностей.

Вступ

Стандарт вищої освіти ступеня – Магістрр, галузі знань 12 Інформаційні технології, спеціальності 121 Інженерія програмного забезпечення є нормативним документом, в якому узагальнюється зміст вищої освіти. Документ установлює галузеві кваліфікаційні вимоги до діяльності випускників вищого навчального закладу зі спеціальності 121 Інженерія програмного забезпечення та освітньої програми «Інженерія програмного забезпечення» освітньо-кваліфікаційного рівня «Магістр» і державні вимоги до властивостей та якостей особи, яка здобула певний освітній рівень.

Освітньо-професійна програма є нормативним документом, у якому визначається нормативний термін (90 ECTS) та зміст навчання, нормативні форми державної атестації, встановлюються вимоги до змісту, обсягу і рівня освіти та професійної підготовки фахівця за спеціальністю 121 Інженерія програмного забезпечення.

Загальні вимоги до властивостей і якостей випускників закладу вищої освіти (ЗВО) подаються у вигляді переліків компетентностей щодо вирішення певних проблем і задач соціальної діяльності, інструментальних, загальних і професійних компетенцій та системи умінь, що забезпечують наявність цих компетенцій.

Компетентності випускника за освітньої програми «Інженерія програмного забезпечення»

Інтегральні компетентності - Здатність особи розв’язувати складні задачі і проблеми у певній галузі професійної діяльності або у процесі навчання, що передбачає проведення досліджень та/або здійснення інновацій та характеризується невизначеністю умов і вимог.

Загальні компетентності - Здатність до абстрактного мислення, аналізу та синтезу; здатність спілкуватися іноземною мовою як усно, так і письмово; здатність проводити дослідження на відповідному рівні; здатність спілкуватися з представниками інших професійних груп різного рівня (з експертами інших галузей знань/видів економічної діяльності); Здатність генерувати нові ідеї (креативність).

Спеціальні (**фахові, предметні**) **компетентності :**

СК01. Здатність аналізувати предметні області, формувати, класифікувати вимоги до програмного забезпечення.
СК02. Здатність розробляти і реалізовувати наукові та/або прикладні проєкти у сфері інженерії програмного забезпечення.
СК03. Здатність проєктувати архітектуру програмного забезпечення, моделювати процеси функціонування окремих підсистем і модулів.
СК04. Здатність розвивати і реалізовувати нові конкурентоспроможні ідеї в інженерії програмного забезпечення.
СК05. Здатність розробляти, аналізувати та застосовувати специфікації, стандарти, правила і рекомендації в сфері інженерії програмного забезпечення.

СК06. Здатність ефективно керувати фінансовими, людськими, технічними та іншими проєктними ресурсами у сфері інженерії програмного забезпечення.
СК07. Здатність критично осмислювати проблеми у галузі інформаційних технологій та на межі галузей знань, інтегрувати відповідні знання та розв'язувати складні задачі у широких або мультидисциплінарних контекстах.
СК08. Здатність розробляти і координувати процеси, етапи та ітерації життєвого циклу програмного забезпечення на основі застосування сучасних моделей, методів та технологій розроблення програмного забезпечення.
СК09. Здатність забезпечувати якість програмного забезпечення.

Результати навчання:

РН01 Знати і застосовувати сучасні професійні стандарти і інші нормативно-правові документи з інженерії програмного забезпечення
РН02 Оцінювати і вибирати ефективні методи і моделі розроблення, впровадження, супроводу програмного забезпечення та управління відповідними процесами на всіх етапах життєвого циклу.
РН03 Будувати і досліджувати моделі інформаційних процесів у прикладній області.
РН04 Виявляти інформаційні потреби і класифікувати дані для проєктування програмного забезпечення.
РН05 Розробляти, аналізувати, обґрунтовувати та систематизувати вимоги до програмного забезпечення.
РН06 Розробляти і оцінювати стратегії проєктування програмних засобів; обґрунтовувати, аналізувати і оцінювати варіанти проектних рішень з точки зору якості кінцевого програмного продукту, ресурсних обмежень та інших факторів.
РН07 Аналізувати, оцінювати і застосовувати на системному рівні сучасні програмні та апаратні платформи для розв'язання складних задач інженерії програмного забезпечення.
РН08 Розробляти і модифікувати архітектуру програмного забезпечення для реалізації вимог замовника.
РН09 Обґрунтовано вибирати парадигми і мови програмування для розроблення програмного забезпечення; застосовувати на практиці сучасні засоби розроблення програмного забезпечення.
РН10 Модифікувати існуючі та розробляти нові алгоритмічні рішення детального проєктування програмного забезпечення.
РН11 Забезпечувати якість на всіх стадіях життєвого циклу програмного забезпечення, у тому числі з використанням релевантних моделей та методів оцінювання, а також засобів автоматизованого тестування і верифікації програмного забезпечення.
РН12 Приймати ефективні організаційно-управлінські рішення в умовах невизначеності та зміни вимог, порівнювати альтернативи, оцінювати ризики.
РН13 Конфігурувати програмне забезпечення, керувати його змінами та розробленням програмної документації на всіх етапах життєвого циклу.
РН14 Прогнозувати розвиток програмних систем та інформаційних технологій.
РН15 Здійснювати реінжиніринг програмного забезпечення відповідно до вимог замовника.
РН16 Планувати, організовувати та здійснювати тестування, верифікацію та валідацію програмного забезпечення.
РН17 Збирати, аналізувати, оцінювати необхідну для розв'язання наукових і прикладних задач інформацію, використовуючи науково-технічну літературу, бази даних та інші джерела.

Об'єктом діяльності Магістра ІПЗ, якому мають відповідати теми кваліфікаційних робіт, є програмне забезпечення, моделі і методи проєктування, процеси, інструментальні засоби та ресурси розробки, супроводження та забезпечення якості програмного забезпечення.

Загальні Вимоги до змісту, структури, оформлення та обсягу атестаційної роботи магістра спеціальності 121 – Інженерія програмного забезпечення освітньої програми «Інженерія програмного забезпечення» визначені відповідними методичними вказівками, розроблених кафедрою. Цей посібник є додатком до цих вказівок в частині змістовного наповнення розділу Проєктування.

Основи проєктування ПЗ на основі IBM RSA

1. Дослідження вимог до ООП-системи

Проєктування та реалізація обє'ктно-орієнтованого програмного забезпечення (на прикладі проєкту розробки інтелектуального скарбника (**smart treasurer**) системи функцінальної діяльності українських кредитних спілок (CUST, Smart Treasurer of Ukranian Credit Unions).

Етапи об'єктно-орієнтованого проектування (OOD) з використанням UML

Специфікація вимог

Аналіз вимог – початковий етап проєктування ПЗ, що включає формування документу «Специфікації вимог» та формулює загальне призначення системи, що вона повинна робити для конкретного користувача. Весь процес проєктування ПЗ супроводжується посиланням на **Специфікацію вимог** для визначення того, якими функціональними властивостями повинна мати система.

Загальні рекомендації щодо оформлення та змісту документу «Специфікація вимог»

Специфікація вимог (СВ) є основним документом, яким керується команда розробників ПЗ в продовж усього циклу розробки. Обсяг цього документу у пояснювальній записці до кваліфікаційної роботи ОКР «Магістр» для спеціальності 121 – інженерія програмного забезпечення повинен бути від 2,5 до трьох сторінок. У СВ повинна бути викладена мета розробки і який фунціонал має бути розроблений для конкретного користувача. В стислій формі повинен бути опис функціоналу та короткі технічні вимоги до реалізації системи (до 0,5 сторінки).

Коротко має бути охарактеризована структура самої системи, який вигляд мали б мати її основні складові елементи. Ні в якому разі система не повинна розглядатись як щось монолітне і непорушне, а динамічне що може піддаватися змінам. Практика «моноліту» це вже давно відійшла в історію, бо це шлях до неуспіху. З системою, яка добре структурована, з добре окресленими зв'язками дуже легко працювати, вносити зміни і розвивати її.

Важливим при скададанні СВ є окреслення в загальному вигляді інтерфейсу користувача, який повинен бути легкодоступним і неперевтомлюючим.

Текс опис функціоналів системи треба писати чітко і зв'язно, щоб аналізуючи СВ можна визначити класифікувати основні сутності предметної області і відношення між ними (структурні та асоціативні зв'язки, успадкування, залежності та використання та ін.). Такі сутності допоможуть деталізувати та реалізувати основні сценарії взаємодії користувача з системою. Задача аналізу і класифікації предметної області є дуже важливою. Словник реальних (не надуманих) класифікованих сутностей є основою створення класів системи з відповідними збалансованими повноваженнями для реалізації функціоналу системи, від чого напряму залежить якість створюваного програмного коду.

Приклад формування Специфікації вимог

Цей приклад формування Специфікації вимог нами зроблено не в ускладненій, а в достатньо простій формі з метою досягнення доступності його розуміння студентами. В

стислій формі викладені змістовні описи функціоналу, яким повинна володіти розроблювальна програмна система з можливістю її подальшого вдосконалення. Водночас цей приклад є також достатньо оригінальним, наповнений цікавими корисними ідеями на основі сучасних підходів розробки ПЗ, є відсутній у такому вигляді у літературі і перевірений на антиплагіат. Він наглядно моделює типову систему предметної області, що розглядається не як монолітна (такий собі програмний «монстр»), а має певну структуру, яка складається з окремих елементів, яким делеговано реалізацію конкретних функціоналів (сервісів) та їх частин і взаємодію між собою. Даний приклад може бути легко ускладнений та розвинутий на суміжні та інші предметні області, надаючи студентам дієві практичні навички проєктування ПЗ, розуміння цілісності системи, принципи декомпозиції («*розділяй і володарюй*») та ін., що є визначальними критеріями відбору програмних інженерів провідними IT-роботодавцями.

Українські кредитні спілки (Ukrainian Credit Unions, UCU), започатковані ефективними українськими підприємцями ще в 19 столітті в Україні (головно в Галичині) та в еміґрації (США, Канада, Західна Європа, Австралія), забезпечили стрімкий розвиток українського бізнесу на світовій арені. UCU планує для зручності та ефективності роботи ґрупи українських підприємців – користувачів, що є членами UCU (здійснення ними основних фінансових операцій або транзакцій) реалізувати програмний додаток CUST App (рис. 1.1), який буде інтелектуалізовувати роботу скарбника спілки, допомагаючи автоматизувати його функціональні операції з по роботі з банком по кредитуванню проєктів та отриманню прибутків від них (дебітуванню користувачів), перевірки їх рахунків по наявності коштів, їх спроможності щодо кредитування, внесення коштів для кредитування перспективних проєктів/депозит та ін. Кожен користувач (підприємець-член UCU) може мати рахунок у банку, що обслуговує конкретну UCU. Користувач CUST App повинен мати можливість переглядати баланс свого рахунку, отримувати кошти від успішних вкладень (знімати гроші з рахунку) та депозитувати кошти під проєкти (вносити їх на рахунок).

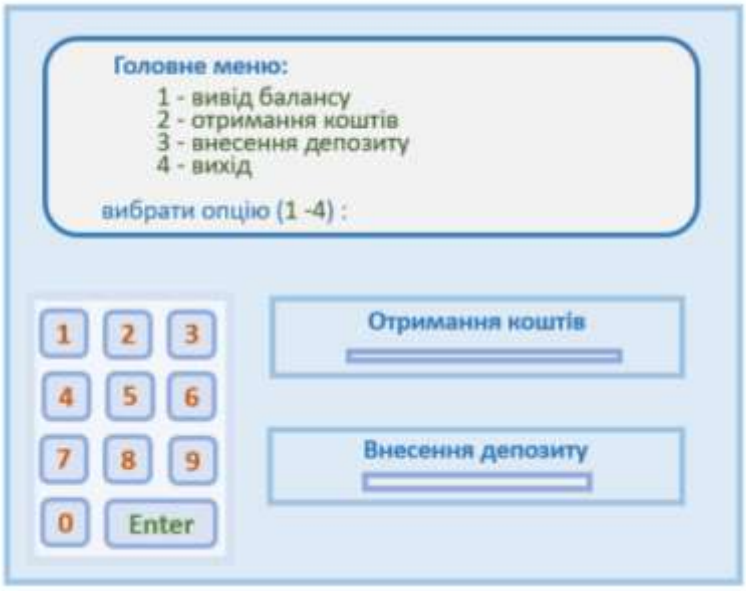


Рис. 1.1.Модель віджету «Інтерфейс користувача» (Головне меню) додатку CUST App

Смарт-девайси та інші апаратні елементи та інтерфейсу CUST App:

- віртуальний монітор для виведення повідомлень користувачу;
- віртуальна кнопкова панель для отримання від користувача цифрових даних;

• смарт-девайс отримання коштів (ДОК) користувачем,

• смарт-девайс внесення депозитів (ДВД) (пакетів з вкладами) на рахунок.

Проєктований CUST App просить користувача набрати номер рахунку на кнопковій панелі. крім того, в кінці сеансу друкує квитанцію та ін., наш все виводить на монітор. Це зроблено для спрощення реалізації проєкту).

Мета розробки: Створюване на замовлення користувача ПЗ CUST App, повинно проводити через CUST фінансові операції, що проводяться клієнтами UCU. ПЗ має інкапсулювати функцію смарт-девайсів (для отримання коштів у вигляді прибутків від реалізованих проєктів користувачів членів UCU та внесення коштів на депозит – для кредитування нових проєктів та ін.) всередині програмних компонентів та не повинно залежати від фізичних операцій, як ці смарт-девайси виконують (принцип багаторівневості).

Це перша (пілотна) версія ПЗ для моделювання та дослідження цілісності виконання основного функціоналу CUST App на ПК, яке в подальшому буде імплементовано в систему реального CUST-об'єкту. Емуляцію роботи віртуального монітора CUST планується реалізувати через меню візуалізації дій користувача, подане у верхній частині віджету «Інтерфейс користувача» (Головне меню) (рис.1), використовуючи монітор ПК, для віртуальної кнопкової панелі – кнопкове меню вводу, що справа внизу на віджеті.

Сеанс з CUST App полягає в ідентифікації користувача за номером його рахунку та особистому ідентифікаційному коду (PIN), за якою слідує оформлення та виконання фінансової операції. Для ідентифікації користувача та виконання транзакцій CUST має взаємодіяти з базою даних (БД), що містить банківські рахунки користувачів.

БД (в наших припущеннях) - організований набір даних на комп'ютері, в де для кожного банківського рахунку користувача зберігається його **номер рахунку**, PIN і **баланс**, що показує суму грошей на рахунку. З метою відпрацювання та налагодження режимів і сценаріїв взаємодії «клієнт- CUST», розглядається робота системи з одним CUST-об'єктом (одночасне звернення декількох CUST-об'єктів до БД в цій версії не розглядається і є предметом наступних реалізацій). Припускається, що банк не здійснюватиме жодних змін в інформації БД у той час, коли клієнт користується CUST App. Складні проблеми безпеки комерційних CUST App тут не розглядаються (для простоти реалізації вважається, що доступ та маніпуляції з інформацією БД здійснюються без спеціальних заходів безпеки).

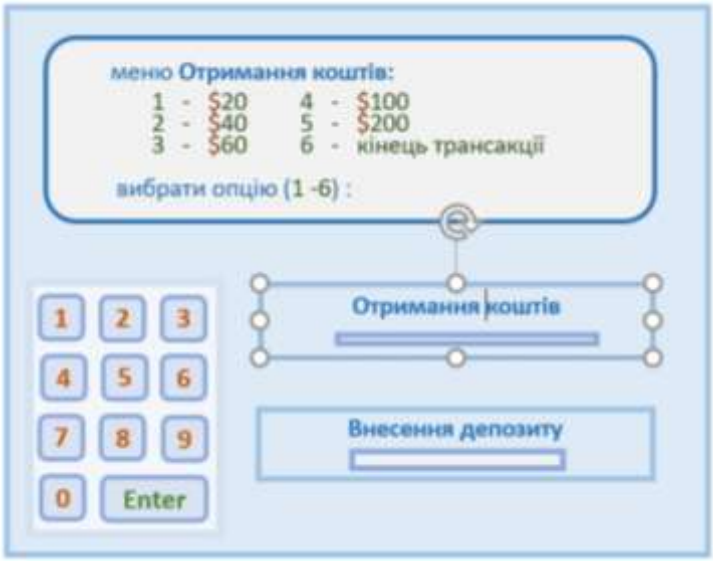


Рис. 1.2. Меню для отримання коштів CUST App

Вимоги до послідовності дій користувача (спрощені) у взаємодії з CUST App:

1. Віртуальний монітор, показуючи вітальне повідомлення, пропонує користувачеві ввести номер рахунку.

2. Користувач вводить 5-значний номер рахунку, набираючи його на кнопковій панелі.

3. Монітор пропонує ввести PIN (особистий ідентифікаційний код), асоційований з цим номером рахунку.

4. Користувач вводить 5-значний PIN-код кнопковій панелі (рис. 1.1).

5. Якщо користувач вводить дійсний номер рахунку та правильний PIN для цього рахунку, Монітор відображає головне меню (рис. 1.1). Якщо користувач вводить недійсний номер рахунку або неправильний PIN, Монітор показує відповідне повідомлення і CUST App повертається до кроку 1, щоб розпочати процес ідентифікації користувача на предмет чи є він клієнтом банку.

Після того, як користувача ідентифіковано, головне меню (рис. 1.1) показує перенумеровані опції для кожного з 3 типів транзакцій: довідки про баланс (опція 1), зняття (опція 2) та внесення грошей (опція 3) та опцію, яка дозволяє користувачеві вийти із системи (опція 4).

Користувач (як клієнт) потім вибирає або виконання транзакції (вводячи **1**, **2** або **3**) або вихід із системи (опція 4). Якщо користувач вводить недійсну опцію, Монітор показує повідомлення про помилку і знову виводить головне меню.

Якщо користувач вводить 1 для отримання довідки про баланс, Монітор показує **баланс рахунку** користувача. Для цього CUST-об'єкт має отримати баланс рахунку із БД банку.

Коли користувач вибирає опцію 2 для отримання коштів, виконуються дії:

6. Монітор показує меню для отримання коштів (рис. 1.2) зі списком стандартних сум, що знімаються з рахунку: 20 $ (опція 1), 40 $ (опція 2), 60 $ (опція 3), 100 $ (опція 4), 200 $ (опція 5) та опцію, що дозволяє користувачеві скасувати операцію (опція 6). ДОК містить певну кількість коштів (щодня поповнюється до 500 20$ банкнот). (Реальні CUST App можуть мати певні відмінності: зчитувати номер рахунку користувача з банківської, карти, включати один смарт-девайс одночасно для отримання коштів та внесення депозитів та ін., що може бути предметом подальших обговорень вдосконалення системи

7. На віртуальній кнопковій панелі користувач вводить обрану опцію (1-6).

8. Якщо вибрана сума, що знімається, перевищує баланс рахунку користувача, Монітор показує повідомлення, що констатує це і пропонує користувачеві вибрати меншу суму. Потім CUST App повертається на крок 6. Якщо вибрана сума, що знімається, менша або дорівнює балансу рахунку користувача (доступна сума), CUST App переходить до кроку 9.

Якщо користувач вирішує скасувати операцію (опція 6), CUST App показує головне меню (рис. 1.1) і чекає на введення користувача .

9. Якщо ДОК містить достатньо готівки для задоволення запиту, CUST App переходить до кроку 10. В іншому випадку Монітор показує повідомлення, що вказує на проблему і що пропонує користувачеві вибрати меншу суму. Потім CUST App повертається на крок 6.

10. CUST-об'єкт дебітує (віднімає) суму, що знімається з балансу рахунку користувача в БД банку.

11. ДОК видає користувачеві бажану суму грошей.

12. Монітор показує повідомлення, яке нагадує користувачеві, щоб він забрав гроші.

Якщо користувач вводить опцію 3 (внесення або коштів на рахунок (депозитування)) головного меню, то виконуються наступні дії:

13. Монітор пропонує ввести суму депозиту або натиснути 0 (нуль), щоб скасувати операцію.

14. За допомогою віртуальної кнопкової панелі користувач вводить суму депозиту або 0 (згідно вимог користувача на даній на кнопковій панелі.

15. Якщо користувач вводить суму депозиту, CUST App переходить до кроку 16. Якщо він вибрав скасування операції (ввівши 0), CUST App показує головне меню і чекає на введення даних користувача.

16. Монітор показує повідомлення, яке пропонує користувачеві внести депозит у ДВД.

17. Якщо протягом 2 хвилин ДВД отримує пакет з депозитом, CUST App кредитує (додає) депоновану суму на **баланс рахунку** користувача в БД банку (кошти не стають миттєво доступними для зняття з рахунку. Спочатку банк має фізично перевірити суму готівки у пакеті депозиту. Якщо ДВД не отримує пакета протягом зазначеного часу, Монітор показує повідомлення про те, що система скасувала операцію через відсутність дії користувача).

Потім CUST App показує головне меню і чекає на введення користувача. Після того, як система успішно здійснить транзакцію, вона повинна знову показати головне меню (рис. 1.1), щоб користувач міг зробити додаткові операції. При виході користувача із системи (опція 4), CUST App має надіслати йому слова подяки та показати вітальне повідомлення для наступного користувача.

Аналіз предметної області (**системи CUST**)

Вказаний приклад Специфікації вимог є типовим, який пишеться представником Замовника, щоб як найдетальніше викласти інформацію розробнику ПЗ, що він хоче отримати від ПЗ розроблюваного на замовлення ПЗ у його розумінні.

Звичайно, первинна інформація що подана замовником у його Вимогах, потребує уточнення і детального обговорення, поки вона не набуде підписаного сторонами документа. Представники Замовника, як правило, за великим винятком не фахівцями з інформатики і програмної інженерії та мають дуже віддалені і дотичні поняття в галузі системного аналізу. Написаний ними тексти Вимог можуть включати багато повторень, неоднозначностей у формулюваннях завдань, нечіткостей опису сценаріїв реалізації функціоналу майбутнього ПЗ, невизначеностей, помилок, технічних описок тощо.

Специфікація вимог має бути результатом процесу детального збору вимог, який може додатково включати опитування як потенційних користувачів системи так і фахівців в предметних областях, пов'язаних з її функціями. Напр., системний аналітик, якого запросили, щоб підготувати специфікацію вимог для ПЗ CUST App, міг би опитати експертів з фінансової діяльності, щоб отримати точніше уявлення про те, що має робити ПЗ та використати цю інформацію для складання списку вимог, якими повинні керуватися розробники системи.

Життєвий цикл системи. Моделі ЖЦ

Процес збору вимог є ключовим завданням першого етапу життєвого цикл (ЖЦ) ПЗ. ЖЦ ПЗ характеризує етапи, які ПЗ проходить починаючи з моменту ідеї його задуму і до моменту, коли воно вже перестає використовуватись. Ці етапи включають:

- Аналіз,
- Проєктування,
- Реалізація,
- Тестування і налагодження,
- Розгортання (передача замовнику),
- Супровід і вилучення з обігу.

Існують різні моделі ЖЦ, кожна зі своїми уподобаннями та приписами щодо того, коли і як часто розробники ПЗ повинні виконувати кожен із цих етапів.

Моделі водоспаду складаються з одноразової послідовності вказаних етапів розробки.

Ітеративні моделі ЖЦ ПЗ декілька разів повторюють один чи декілька етапів. Існує багато нових сучасних процесів розробки, з яких найбільш відомим є Rational Unified Process™ (RUP), розроблений IBM Rational Software Corporation. RUP – потужний процес, призначений для проєктування ПЗ різної складності, включаючи моделювання варіантів використання та забезпечення оптимальної ієрархії класів предметної області та архітектури системи, що забезпечують їх реалізацію, керування процесом розробки і внесення змін, автоматичне ґенерування програмного коду, розробку усієї необхідної документації та управлінням проєктом та ін. (рис. 1.3).

Раціональний уніфікований процес (RUP) розробки ПЗ на основі UML

RUP : керований варіантами використання, архітектурно-центрований, ітеративний, інкрементний; включає 4 фази (5 робочих процесів (РП) на кожну):

- **Inception** (Аналіз і визначення вимог(ВВ);
- **Elaboration** (Проектування);
- **Consruction** (Побудова);
- **Transition** (Впровадження).

Фази	Основні завдання
Inception	виявлення загальних можливостей ПЗ, його здійсненності, схвалення проекту
Elaboration	уточнення вимог, створення основних моделей та архітектури ПЗ
Consruction	- детальне проектування підсистем і класів, реалізація коду
Transition	тестування системи, впровадження і передача користувачу.

Ітерації: фази розбиті на послідовність ітерацій (міні-проектів), кожна з яких відповідає певним завданням (варіантам виористання).

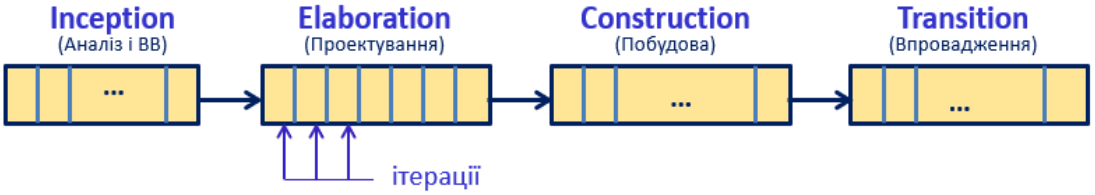


Рис. 1.3. Загальна схема RUP

Етап (**фаза**) **Аналізу** ЖЦ ПЗ зосереджує увагу на формулюванні завдання, що розв'язується. Розробляючи будь-яку систему, потрібно правильно вирішити завдання, але не менш важливо вирішити **правильне завдання**. Системні аналітики збирають вимоги, що формулюють конкретне завдання. Специфікація вимог визначає систему CUST App з достатньою деталізацією, в розгорнутий аналіз (це вже з нашою допомогою зроблено у Вимогах замовника у попередньому розділі. Подальша робота над Вимогами потребує їх строгої формалізації та побудови спеціальних моделей вимог, що є основою для проєктування майбутнього ПЗ.

Моделі вимог. Методологія моделювання варіантів використання (ВВ) є однією з найефективніших методик розробки моделей вимог, як дозволяє чітко зрозуміти, що має робити розроблювана система для конкретного користувача.

Цей процес проєктування ПЗ визначає ВВ системи, кожен з яких представляє одну з можливостей, які система надає своїм клієнтам (користувачам). CUST App є декілька ВВ, пов’язаних з виконанням фінансових трансакцій, таких, як:

"Переглянути баланс рахунку",
"Отримати кошти",
"Депонувати кошти",
"Перевести кошти з рахунку на рахунок".

Спрощена система CUST App, яку створюємо, допускає **три перші** з перерахованих ВВ (рис. 1.4).

ВВ - типовий сценарій використання системи клієнтом (основні сценарії в загальному (ще не у формалізованому) вигляді описані у специфікації вимог). Перелік кроків, необхідних для виконання транзакцій кожного типу (перевірки балансу, зняття та внесення грошей) описують такі ВВ CUST App : «Переглянути баланс рахунку», «Отримати кошти», «Депонувати кошти».

Основні моделі UML, необхідні для створення проєкту ПЗ

Діаграми варіантів використання (**ВВ**)

Першою з необхідних для проєктування ПЗ моделей UML, що складають основні артефакти проєктованої ООП-системи – діаграма варіантів використання, що моделює взаємодії між клієнтами та системою. Метою цього етапу проєктування ПЗ є представлення усіх видів взаємодій користувачів з системою без опису їх деталей (*спочатку основне, деталі даються в інших діаграмах UML, створюваних у процесі розробки*).

Діаграми ВВ супроводжуються детальними описами ВВ – більш формалізовані і структуровані описи, що формуються на основі текстових описів функуціоналу у Специфікації вимог. На сьогоднішній день багато сучасних інструментальних систем розробки ПЗ комплектуються спеціальними додатками автоматизованого формування строго формалізованих моделей вимог на основі Специфікації вимог замовника.

У ЖЦ ПЗ діаграми ВВ створюються на етапі Аналізу вимог. Для проєктування великих і складних системах діаграми ВВ є простим і необхідним інструментом допомоги системним аналітикам і програмним інженерам зосередитися на задоволенні вимог користувачів.

На рис. 1.4 показана діаграма ВВ для проєктованої CUST App. Тут «чоловічок» представляє актора, що визначає ролі, які зовнішній об'єкт (користувач або інша система) - може виконувати в процесі взаємодії з CUST App. У діаграмі ВВ може бути декілька акторів (діаграма ВВ для CUST App реальної UCU могла б також включати актора з ім'ям Адміністратор, який щодня поповнює запас готівки у ДОК та ін.). Необхідний довідковий матеріал, щодо використання IBM RSA для побудови цього класу UML діаграм та усіх наступних, поданий у Додатку.

Рис. 1.4. Діаграма ВВ для системи CUST App з погляду користувача

На перспективу будем використовувати дещо оновлені версії діаграми ВВ (згідно змін у Вимогах, внесених замовником), зокрема, що також дозволяє користувачам перераховувати кошти між рахунками (рис. 1.5.).

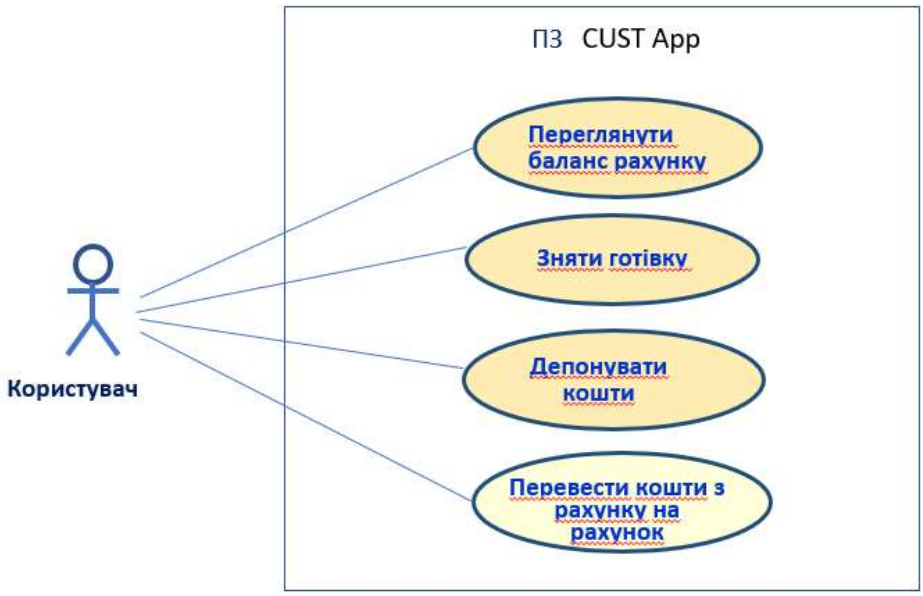


Рис. 1.5. Діаграма ВВ модифікованої версії CUST App з урахуванням ВВ переведення користувачами коштів між рахунками

Актори. У проєктованій системі акторів знаходимо, вивчаючи Специфікацію вимог, у якій, зокрема, сказано: «Користувачі CUST App повинні мати можливість переглядати баланс свого рахунку, отримувати готівку та депонувати кошти». Актором у кожному з трьох ВР є користувач (клієнт), що взаємодіє з CUST App. При фінансових операціях роль Користувача відіграє зовнішній об'єкт – реальна людина (екземпляр Користувача). На діаграмі ВВ (рис. 1.4) показаний один актор, під яким стоїть його ім'я (Користувач).

ВВ. У UML кожен ВВ візуалізується овалом, з'єднаним з актором суцільною лінією асоціації. Перш ніж програмні інженери реалізують систему на конкретній ООП-мові, системні аналітики, проектувальники ПЗ (разом з архітектором) повинні проаналізувати її специфікацію вимог, розробивши сукупність варіантів використання та на основі цього

розробити інші моделі проєкту, що є підґрунтям для написання правильного програмного коду. Досвідчений програмний інженер також може успішно виконувати ці види робіт проєктування (*поганий той програмний інженер, який не прагне стати професійним проєктувальником ПЗ, а ще краще архітектором ПЗ*).

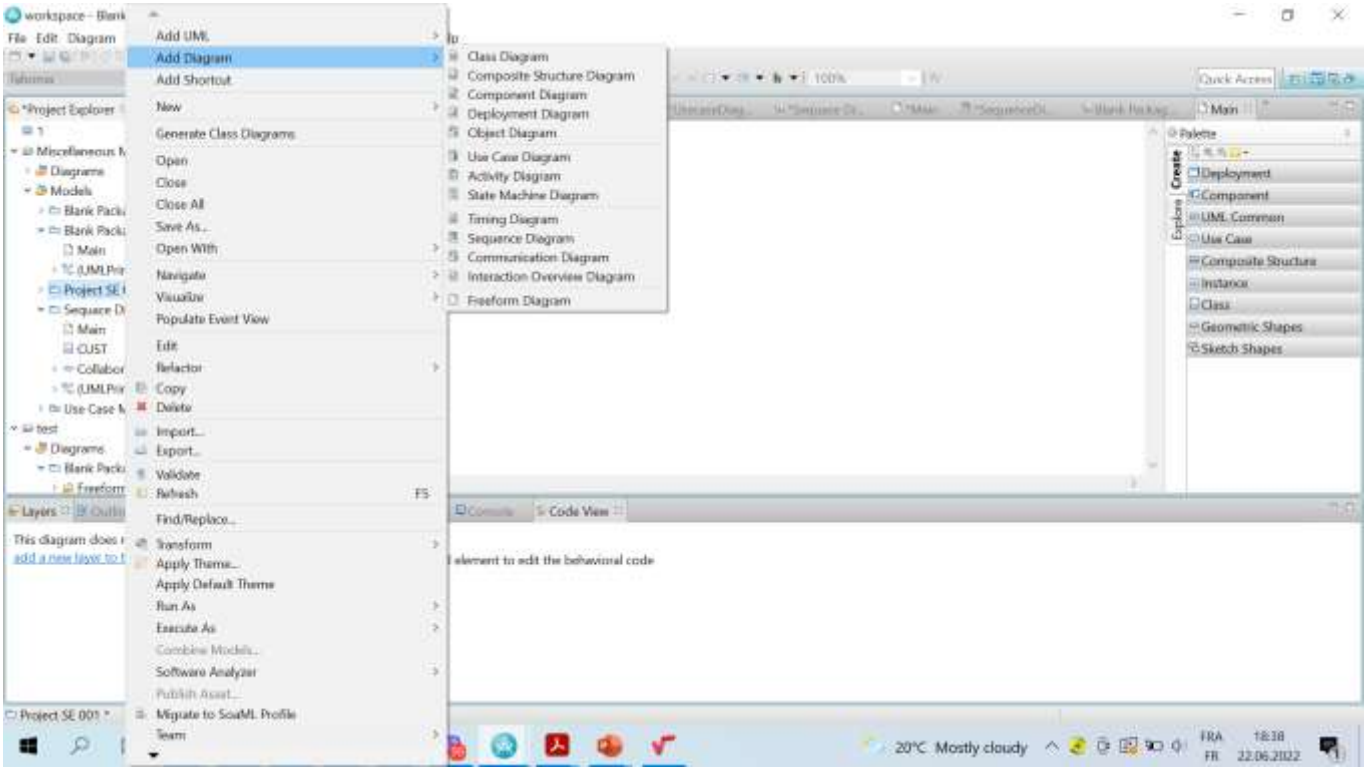


Рис.1.5.1. Виклик через Project Explorer IBM RSA із власного проєкту і вкладку ADD Diagram меню вибору необхідних для проектування типів діаграм UML

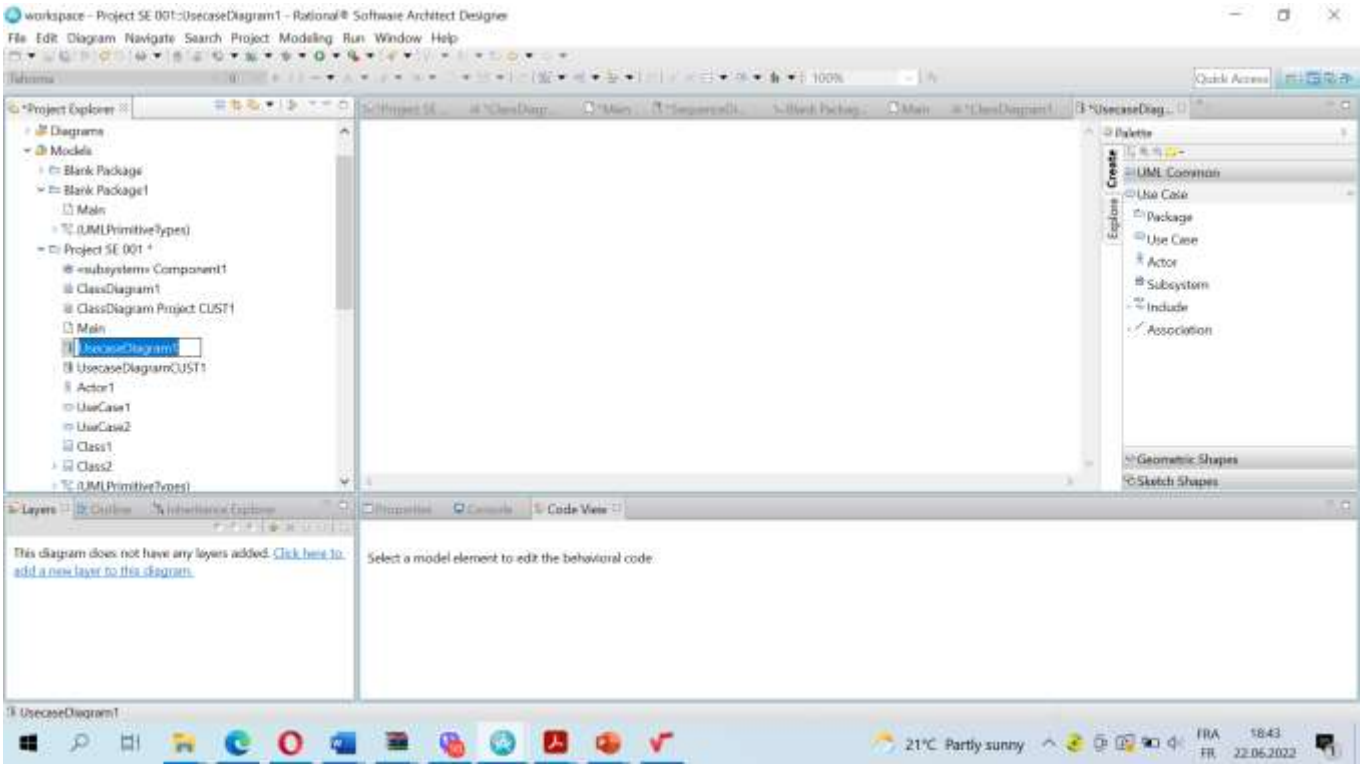


Рис. 1.5.2. Меню IBM RSA (v. 10.07) з необхідними робочими полями для побудови діаграм варіантів використання Use Case Diagram UML та палітрою інструментів для швидкого набору основних пікторгам діаграми (актори, ВВ, асоціації)

Описи ВВ (Приклади)

1.ВВ "**Отримати кошти**". Щоб отримати кошти/готівку, виконуються такі дії:

1. Монітор показує меню (рис. 1.3) зі списком стандартних сум, що знімаються з рахунку: 20 $ (опція 1), 40 $ (опція 2), 60 $ (опція 3), 100 $ (опція 4) і 200 $ (опція 5). Меню містить також опцію, яка дозволяє користувачеві скасувати операцію (опція 6).

2. На віртуальній кнопковій панелі користувач вводить обрану опцію (1-6).

3. Якщо вибрана сума, що знімається, перевищує баланс рахунку користувача, Монітор показує повідомлення, що констатує це і пропонує користувачеві вибрати меншу суму. Потім CUST App повертається на крок 1. Якщо вибрана сума, що знімається, менша або дорівнює балансу рахунку користувача (це прийнятна сума), CUST App переходить до кроку 4.

4. Якщо ДОК містить достатньо готівки для задоволення запиту, система CUST App переходить до кроку 5. В іншому випадку Монітор показує повідомлення, що вказує на проблему і що пропонує користувачеві вибрати меншу суму. Потім CUST App повертається на крок 6.

5. CUST-об'єкт дебітує (віднімає) суму, що знімається з балансу рахунку користувача в БД банку.

6. ДОК видає користувачеві бажану суму грошей.

7. Монітор показує повідомлення, яке нагадує користувачеві, щоб він забрав гроші.

2.ВВ "**Депонувати кошти** (**поставити кошти на депозит**)". Якщо користувач вводить опцію 3 (на головному меню), виконуються такі дії:

1. Віртуальний монітор пропонує ввести суму депозиту або натиснути 0 (нуль), щоб скасувати операцію.

2. За допомогою віртуальної кнопкової панелі користувач вводить суму депозиту або 0.

3. Якщо користувач вводить суму депозиту, CUST App переходить до кроку 4. Якщо він вибрав скасування операції (ввівши 0), CUST App показує головне меню і чекає на введення користувача.

5. Монітор показує повідомлення, яке пропонує користувачеві внести з депозит ДВД.

3.ВВ "Переглянути баланс рахунку"

1.Користувач вводить опцію 1 для отримання довідки на Віртуальному моніторі про баланс рахунку користувача.

2.Отримавши значення введеної користувачем опції про перевірку балансу, CUST App надсилає повідомлення БД виконати дії про підготовку користувачеві довідки про баланс.

3. Виконавши перевірку балансу рахунку зареєстрованого користувача, БД поверне CUST App інформацію про готівковий баланс рахунку користувача та загальний баланс депозиту рахунку користувача.

3.Обидва баланси виводяться на Монітор. Цими діями виконання транзакції завершується.

На етапі аналізу необхідно вникнути у вимоги до системи, щоби скласти специфікацію високого рівня, що система повинна робити.

2. Проєктування системи CUST. Ідентифікація класів предметної області

Структура системи визначає об'єкти (компоненти) системи та їх взаємовідношення. Компоненти системи взаємодіють між собою в цілях розв'язання задачі. Задача декомпозиції системи на елементи (компоненти) є є складним завданням і потребує певних підходів щоб рівномірно збалансувати між усіма елементами усі обов’язки системи щодо їх реалізації, визначених у моделі вимог ВВ, встановлення на цій основі найважливіших зв’язків (структурні звязки, асоціації, успадкування (з метою уникнення повторів), зв’язки використання та ін.). В подальшому аналізі та уточненні Вимог використання таких тверджень як, напр, «Система виконує таку-то функцію чи здійснює таку-то таку операцію ..» є недопустимо в проєктуванні, бо це тяготіє невизначеністю і монолітністю системи (одна із грубих помилок проєктувальників, студенти в цьому не вийняток). Потрібно чітко розподіляти обовязки між її елементами і писати приблизно так: «Такий то клас чи компонент Системи виконує таку-то функцію чи здійснює таку-то таку операцію ...».

інтерфейс користувача. Система повинна мати інтерфейс користувача (рис. 1.1), який визначає порядок реалізації основних сценарії ПЗ, що реалізації ним усіх фінансових та та аналітичних операцій (Оп) і взаємодіє з БД з інформацією про банківські рахунки.

Поведінка системи визначає її зміни у процесі взаємодії її об'єктів мж собою. Розробники ПЗ системи повинні специфікувати обидва ці її аспекти (структуру і поведінку).

Типи використовуваних діаграм для моделювання ПЗ у ООП-проєкті. UML визначає необхідні типи діаграм для документування моделей проєктованої системи. Кожна з них є

важливою для її цілісного проєктування та описує окремі характеристики структури або поведінки системи.

1 . **Діаграми ВВ** (use case diagrammes) моделюють взаємодії між системою та її зовнішніми об'єктами (акторами) у термінах ВР (можливостей системи, таких як «Переглянути баланс рахунку», «Отримати кошти» та «Депонувати кошти»).

2. **Діаграми класів** (class diagrammes) моделюють класи та зв'язки між ними. Класи - «будівельні блоки» для побудови системи. Кожен іменник, описаний у специфікації вимог, є кандидатом у класи системи (рахунок, кнопкова панель).

3. **Діаграми станів** (mashine state, **MS** diagrammes) моделюють зміни станів об'єктів класів системи. Стан об'єкта описується значеннями всіх атрибутів в певний момент часу. Після підтвердження PIN-коду користувача CUST переходить зі стану "*користувач не ідентифікований*" в стан "*користувач ідентифікований"*.

4. **Діаграми діяльності** (activity diagrammes, **AD**) моделюють діяльність об'єкта – послідовність подій у ході виконання сценарію. **AD** моделює дії об'єкта дії та специфікує порядок, у якому їх виконує. CUST App має отримати баланс рахунку користувача (з БД з інформацією про банківські рахунки), перш ніж Монітор зможе показати користувачеві баланс.

5. **Діаграми комунікації** (communication diagrammes, **CD**) (у перших версіях UML це діаграми кооперації), моделюють взаємодії між об'єктами в системі, з акцентом на тому, які відбуваються взаємодії.

6. **Діаграми послідовностей** (sequances diagrammes, **SD**) моделюють взаємодії об'єктів, але, на відміну від **CD**, акцентують увагу на часові послідовності цих взаємодій. **SD** дозволяють показати порядок, у якому відбуваються взаємодії під час виконання фінансової Оп.

Нами не ставиться мету формального вивчення студентом усіх цих різних типів UML діаґрам та багато інших UML засобів, не приведених у цьому посібнику, як також і окремих панелей і меню IBM RSA. Основним є навчити студентів застосування вказаних інструментальних засобів для створення ефективного проєкту, результатом якого є правильний програмний код з правильною архітектурою, що дозволяє його повторне використання, швидке внесення змін, його розвиток.

Ідентифікація класів предметної області згідно специфікації вимог

Класи визначаються зі Специфікації вимог, проаналізувавши **іменники** та **іменні словосполучення** (конструкції) тексту вимог на предмет їх відношення до реалізації сценаріїв тих чи інших ВВ та відношень з іншими сутностями системи.. При цьому вирішується, що деякі з цих іменників та іменних конструкцій є атрибутами інших класів у системі. Під час цього процесу можуть виявитися додаткові класи.

На рис. 2.1 перераховані іменники та іменні конструкції, виявлені на основі аналізу специфікації вимог до системи CUST App. Вони перераховані зліва направо у порядку появи у тексті Специфікації вимог.

Основні іменники та іменні конструкції в Специфікації вимог		
банк	гроші/кошти	номер рахунку
CUST	Монітор	PIN
користувач	кнопкова панель	база даних банку
клієнт	ДОК (смартдевайс отримання	перевірка балансу

	коштів)	
трансакція	20 $ банкнота	Отримання коштів
рахунок	ДВД (смартдевайс внесення депозиту)	внесення депозиту
баланс	пакет з депозитом	

Рис. 2.1. Наявні **іменники** та **іменні конструкції** у Специфікації Вимог

Спочатку вирішується **завдання верхнього рівня**: ідентифікувати класи та зв'язки між ними. Атрибути – на етапі детального проєктування класів. Операції також приписуються класам на пізніших стадія проектування.

Класи проєктованої ООП-системи CUST App. Кожен іменник з рис. 2.1 пов'язаний з одним або декількома пунктами списку:

- **CUST**
- **Віртуальний Монітор**
- **Віртуальна Кнопкова Панель**
- **ДОК**
- **ДВД**
- **Рахунок**
- **База даних банківських рахунків**
- **Перевірка балансу**
- **Отримання коштів**
- **Внесення коштів на депозит**

Від іменників до класів. Імена класів пишуться англійською мовою з великої літери, як це слід робити при написанні правильного коду на C++, що реалізує проєкт (Б. Стауструп). Список англійських і назв класів (класи проектування) зберігаємо у строгій відповідності з виявленими у ході аналізу вимог українськими назвами класів (класи аналізу), щоб не втратити зв'язки з фізичними сутностями предметної області. Якщо ім'я класу містить декілька слів чи словосполучень, пишемо їх разом, починаючи кожне з великої літери.

	Класи проєктування (англ. назви)	**Класи аналізу** (укр. назви абстракцій предметної області зі спец. **Вимог**)
1	**CUST**	Електронний Смарт Скарбник UCU
2	**VirtMonitor**	Віртуальний Монітор
3	**VirtKeypad**	Віртуальна Кнопкова Панель
4	**CashReceiptDevice**	смарт- девайс отримання коштів (ДОК)
5	**DepositDevice**	Смарт-девайс внесення коштів на депозит (ДВД)
7	**BankDataBase**	База даних банківських рахунків користувачів
8	**Account**	Рахунок користувача
9.	**ReceivingCash**	Трансакція «Отримати кошти »
10	**Deposit**	Трансакція «Внести кошти на депозит»
11.	**BalanceRequest**	Трансакція «Перевірити баланс рахунку»

Моделювання класів системи і зв'язків між ними

UML дозволяє за допомогою діаграм класів моделювати класи в системі **CUST App** та їх відношення. Рис. 2.2 представляє клас CUST. У UML кожен клас зображується прямокутником із трьома відділеннями. Верхнє відділення (купе) містить ім'я класу, написане жирним шрифтом із вирівнюванням по центру. Середнє відділення містить атрибути класу. Нижнє відділення містить операції класу (р.5). На рис. 2.2 середнє та нижнє відділення порожні, оскільки ще не визначено атрибути та дії (операції) цього класу.

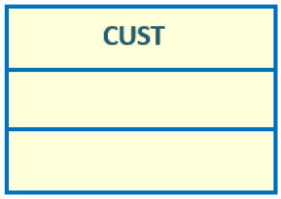


Рис. 2.2. Подання класу **CUST** на діаграмі класів UML

На діаграмі класів (рис. 2.3) показано, як співвідносяться один з одним класи **CUST** та **ReceivingCash**. Тут прямокутники, які мають тільки ім'я класу, немає інших відділень. UML допускає опущення атрибутів та операцій для читабельності. Це неповна діаграма, деяка інформація з міркувань компакності (вміст другого і третього відділень) у класі не зображується. До цих відділень інформація буде поміщена по мірі необхідності, пізніше (р.3, 5). Як буде показано далі, IBM RSA надає широкі можливості інкапсуляції різних частин класу, забезпечуючи високу читабельність діаграм.

Рис. 2.3. Діаграма класів, що показує асоціацію між 2 класами

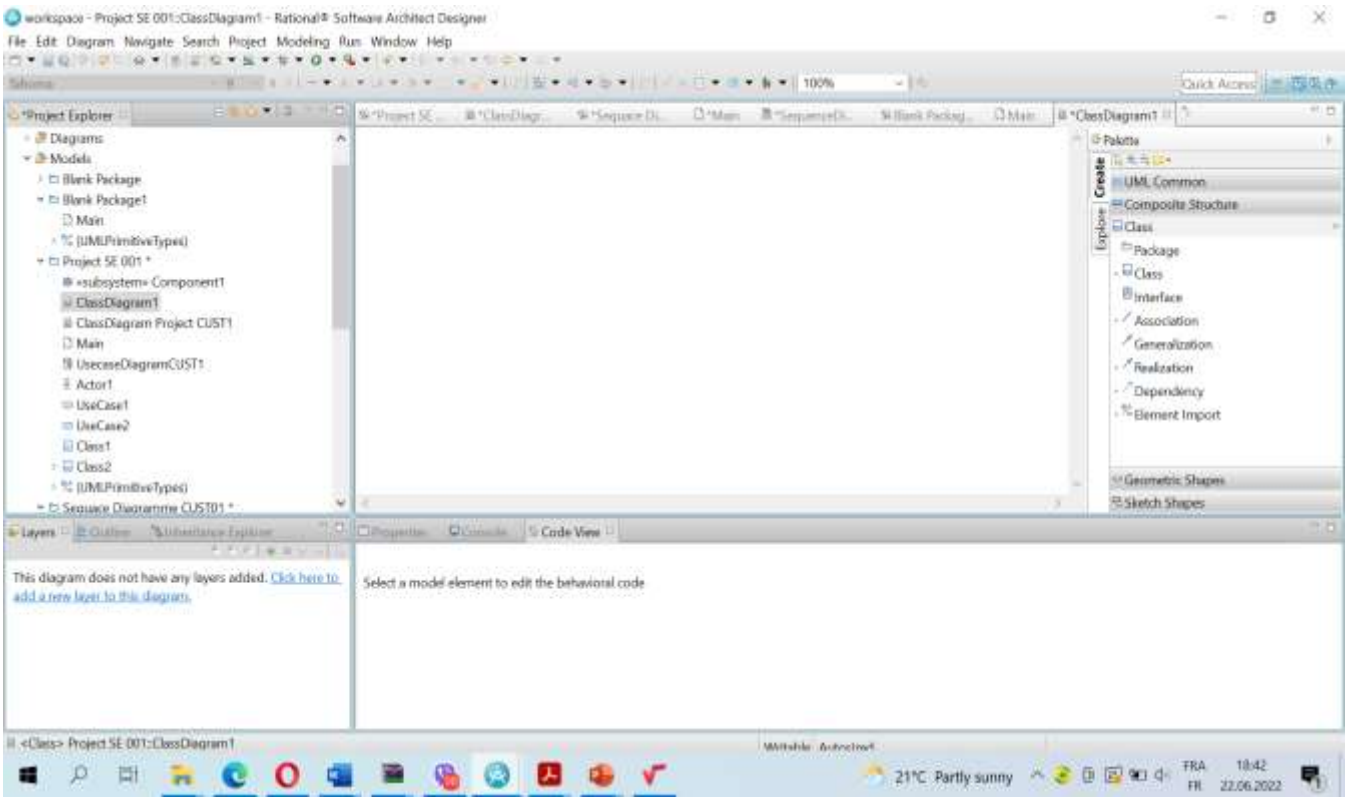


Рис. 2.3.1. Основне меню IBM RSA для побудови діаграм класів Class Diagram UML. зліва: підменю-палітра інструментів для швидкого і полегшеного набору класів, інтерфейсів та різних типів зв'язків між ними

UML може моделювати багато типів кратності (рис. 2.4) .

Символ	Зміст
0	нуль
1	один
n	ціле значення
0..1	нуль або один
m, n	m або n
m..n	від m до n
*	довільне невід'ємне ціле (0 або більше 0)
0..*	0 або більше 0 (еквівалентно *)
1..*	один або декілька

Рис. 2.4. Типи кратності

Композиція. Крім вказання простих відношень асоціації можуть специфікувати складніші, напр., те, що об'єкти одного класу складаються з об'єктів інших класів. Згідно зі специфікації **вимог**, об'єкт **CUST** складається з Монітору, кнопкової панелі, ДОК та ДВД. Суцільні ромби на рис. 2.5 вказують, що клас **CUST** знаходиться із класами **VirtMonitor, VirtKeypad, ReceivingCash, DepositDevice** у відношеннях композиції. Композиція - відношення «ціле-частина».

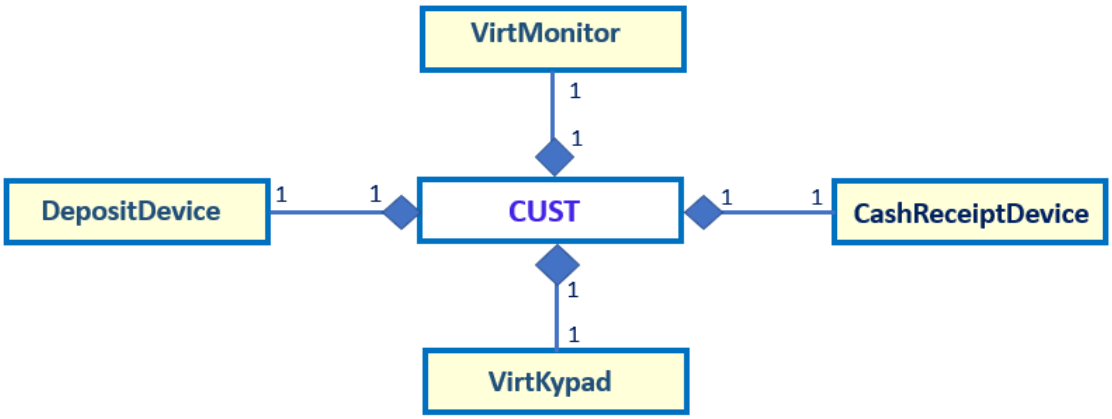


Рис. 2.5. Діаграма класів CUST App з відношеннями композиції

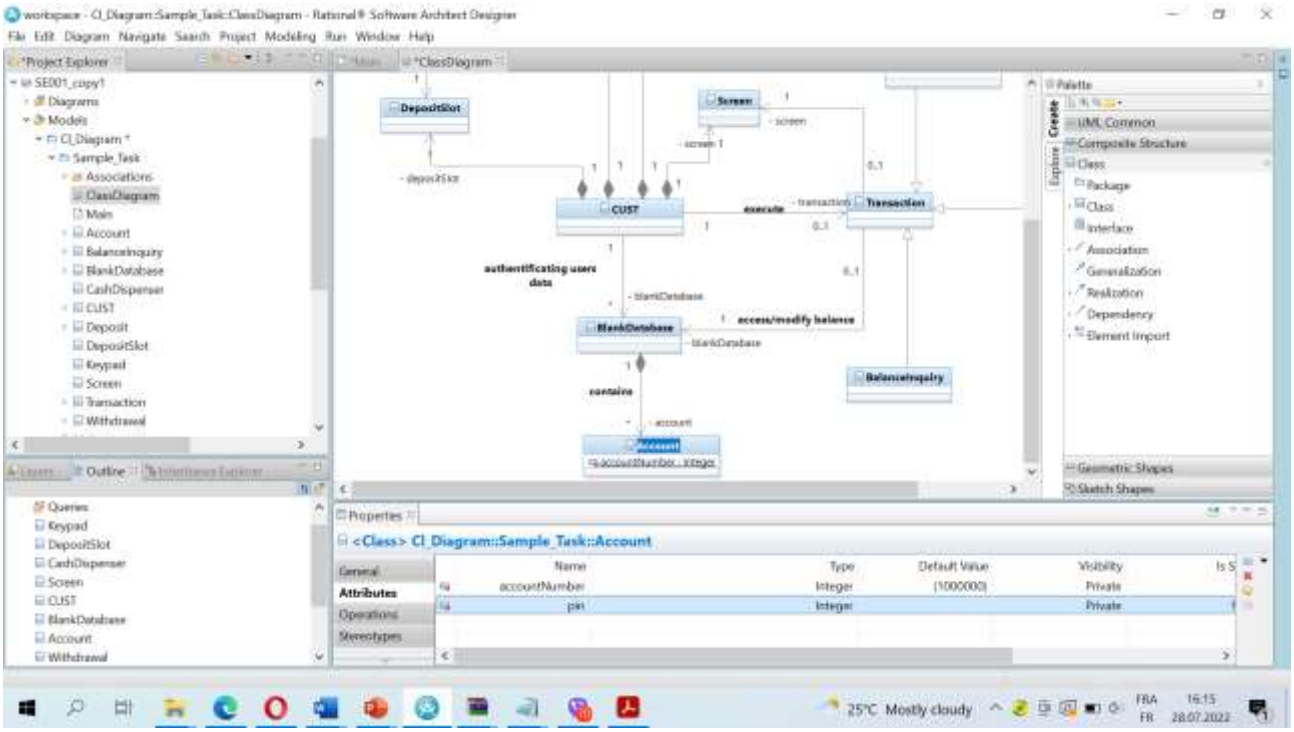


Рис. 2.5. 1. Діаграма ієрархії класів UML, побудована IBM RSA для системи CUST (додавання атрибутів та їх параметризація). Знизу підменю Властивості – для швидкої параметризації атрибутів та операцій

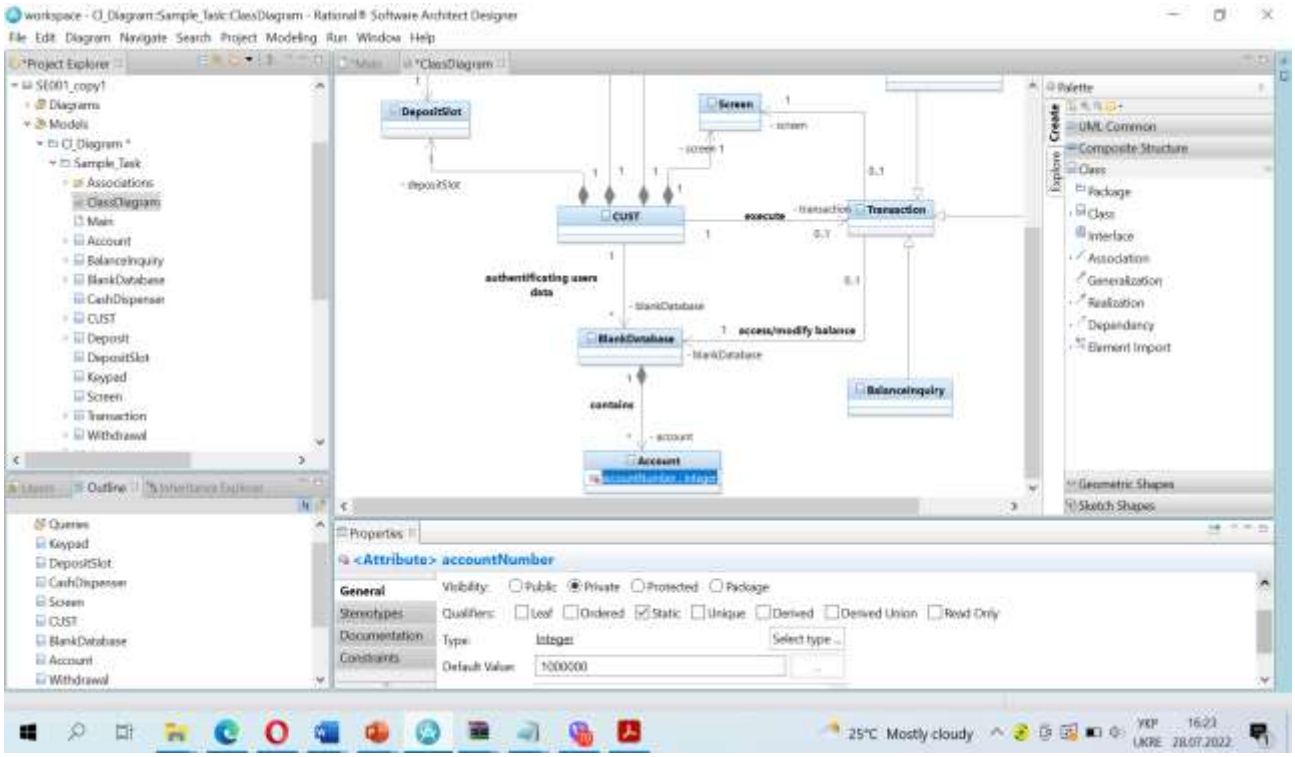


Рис.2 .5.2. Діаграма ієрархії класів UML в IBM RSA для системи CUST (додавання атрибутів: вибір класу/типу атрибута)

На рис. 2.6 показана діаграма класів системи CUST App, яка моделює більшість класів, ідентифікованих раніше, а також асоціації між ними, які отримуємо зі специфікації вимог.

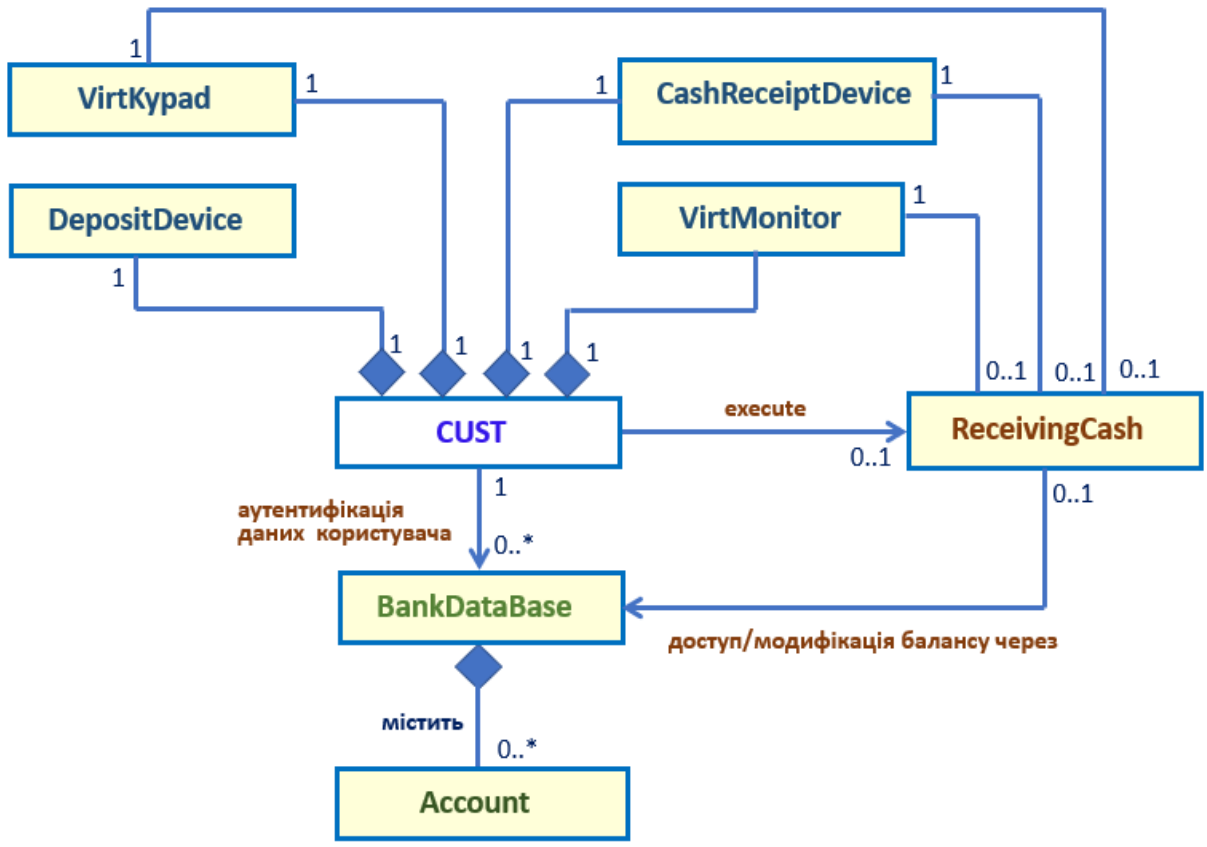


Рис. 2.6. Розширена діаграма класів для ООП-моделі системи CUST App для ВВ «Отримати кошти»

Подібно до трансакції «Отримання коштів», транзакція «Внесення депозиту на рахунок» вимагає використання Віртуального Монітору і Віртуальної Кнопкової Панелі відповідно для виведення інструкції користувачу на ввід даних та прийому системою введення. Для внесення депозиту об'єкт **:Deposit** звертаєтеся до об’єкта ДВД :**DepositDevice** виконати його дію - внести депозит.

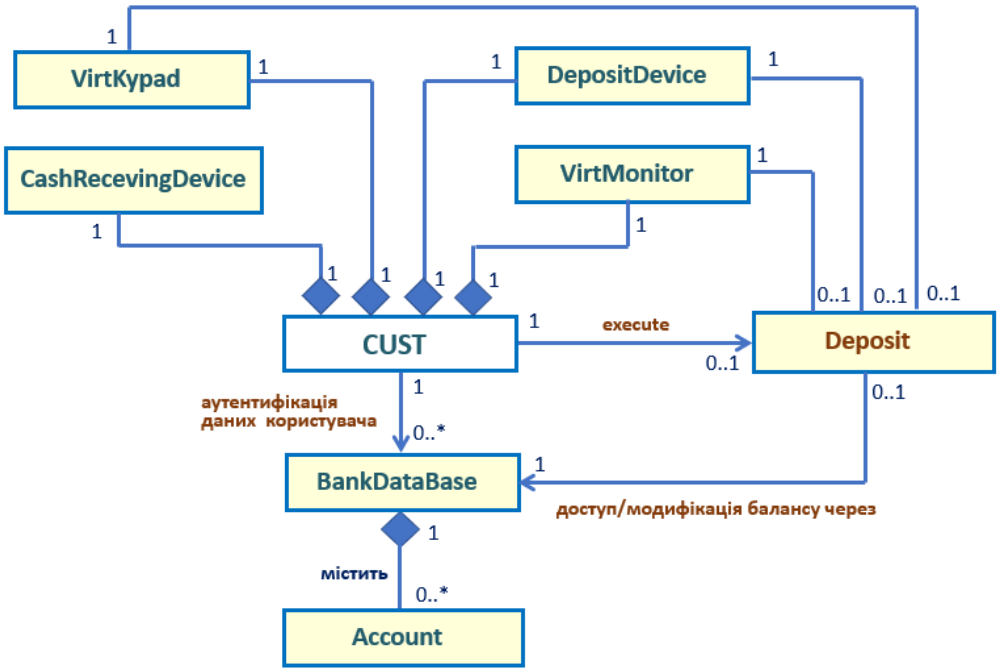


Рис. 2.7. Діаграма класів для системи CUST App для ВВ «Внести депозит» (включає клас **Deposit**)

Об'єкт класу **BalanceRequest** асоціюється також з об'єктом класу **VirtMonitor**, щоб показувати користувачеві баланс рахунку (рис.2.8).

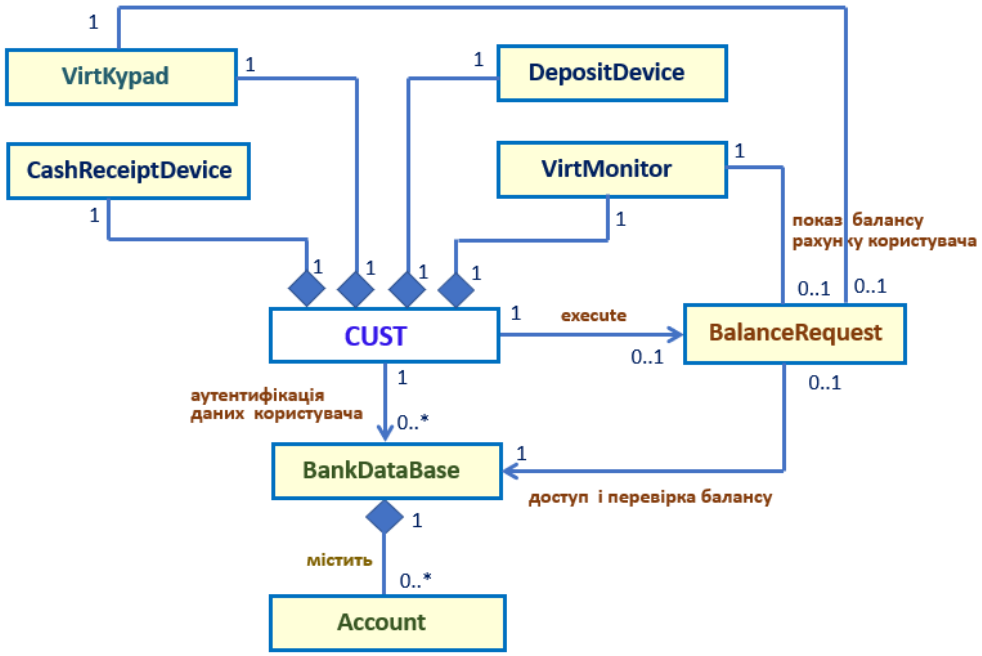


Рис. 2.8 Діаграма класів для системи CUST App для ВВ «Показати Баланс» (включає клас **BalanceRequest**)

Користуючись діалоговим вікном Class Diagrammes (рис.2.5.1) IBM RSA з використанням спливаючої контекстної панелі інструментів, підменю Proprties для параметризації атрибутів та операцій, вкладки Filters з контекстними підменю для

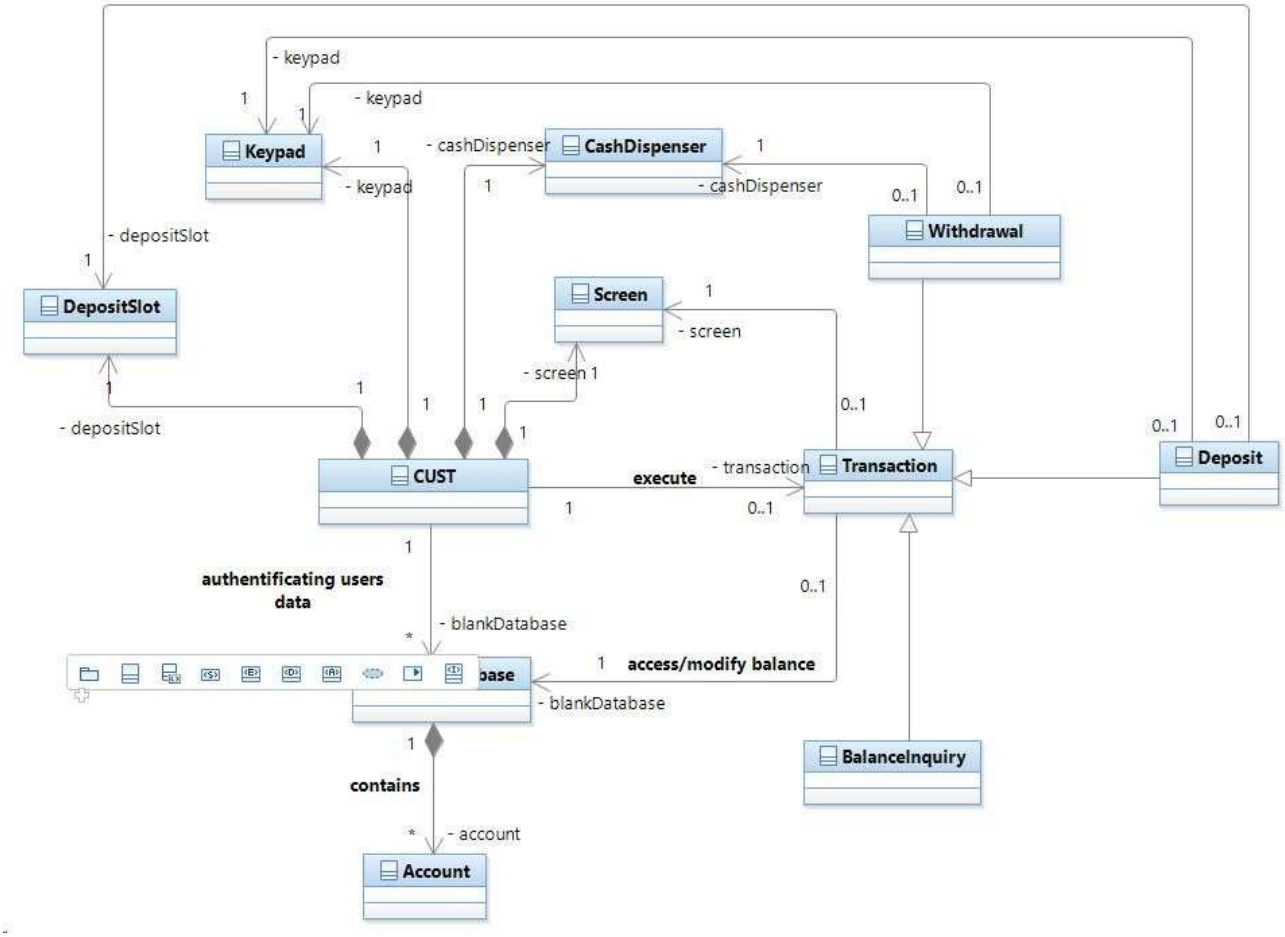


Рис. 8. 1. Вигляд повної діаграма ієрархії класів UML в для системи CUST (в проєкті папки Model/CI_Diagram/Sample_Task/Associations/ClassDiagram) зі спливаючою контекстною палітрою інструментів в IBM RSA

3. Конструювання ПЗ. Ідентифікація атрибутів класів в CUST App

На початковому етапі OOD-проектування для: системи CUST App - **аналіз специфікації вимог** та **ідентифікації класів**, необхідних для реалізації системи перераховано іменники та іменні конструкції у специфікації вимог та ідентифіковано окремі класи для тих, що відіграють істотну роль в CUST App. Потім моделювались класи та їх взаємовідношення на діаграмі класів UML (рис. 2.6). Класи мають атрибути (дані) та операції (дії, поведінка). Атрибути класу реалізуються на C++ як елементи даних, а операції класу - методи. Визначимо багато з атрибутів, необхідних у системі CUST App та досліджуємо, як ці атрибути представляють стан об'єкта та визначимо операції класів.

Ідентифікація атрибутів

Атрибути деяких об'єктів реального світу. Атрибути людини включають ріст, вагу, колір очей, ґрупу крові, вік. Атрибути персонального комп'ютера - марка (ASUS, Apple, IBM),

тип монітора, обсяг основної пам'яті, кількість ядер, … Можна ідентифікувати багато атрибутів класів в системі CUST App, пошукавши пов’язані з ними описові слова і словосполучення в Специфікації вимог.

На рис. 3.1 поданий список слів і словосполучень зі специфікації вимог, які описують кожен із класів, ідентифікуючи будь-які слова чи словосполучення, що відносяться до характеристик класів у системі. Напр., специфікація вимог описує кроки, необхідні для отримання «суми, що знімається», тому приписуємо до класу **ReceivingCash** слово "сума" та ін.

Клас	Описові слова та словосполучення
CUST	користувач авторизований
VirtMonitor	Текстові повідомлення для користувачів
VirtKeypad	Інформація, яку вводить користувач (номер оції команд, числові знач. сум для зняття/внесення на депозит)
CashReceiptDevice	щодня завантажується 500 20$ банкнот
DepositDevice	Суми, готівки що знімається
BankDataBase	дані (посилання) про рахунки користувачів
Account	номер рахунку, PIN, баланс
BalanceRequest	номер рахунку
ReceivingCash	номер рахунку, сума
Deposit	номер рахунку, сума

Рис. 3.1. Класи та пов’язані з ними описові слова і словосполучення із вимог CUST App

Керуючись списком (рис. 3.1), створюємо один атрибут класу **CUST**. Клас **CUST** зберігає інформацію про стан CUST app. Словосполучення "Користувач авторизований" описує стан CUST-об’єкта (стани вводимо у р.4), тому вводимо в клас булевий атрибут userAuthenticated (true/false). Стандартний тип UML Boolean відповідає типу bool C++. Атрибут вказує, чи вдалося **CUST** підтвердити особистість поточного користувача, щоб система дозволила користувачеві робити транзакції та звертатися до інформації рахунку. Значення атрибуту userAuthenticated=true, якщо користувач є авторизований. Класи **BalanceRequest**, **ReceivingCash**, **Deposit** мають один і той же атрибут accountNumber. Кожна транзакція пов'язана з певним номером рахунку користувача, що здійснює транзакцію. Кожному класу транзакції присвоюємо цілий атрибут accountNumber, щоб ідентифікувати рахунок, до якого належить об'єкт класу.

Описові слова та словосполучення у специфікації вимог говорять також про деякі відмінності в атрибутах, необхідні кожному з класів транзакцій. Специфікація показує, що при знятті/внесенні коштів користувач повинен ввести конкретну суму грошей, яка повинна бути відповідно знята/внесена. Тому, присвоюємо класам **ReceivingCash** і **Deposit** атрибут amount, який зберігає вказане користувачем значення.

Клас **Account** має декілька атрибутів. Специфікація Вимог свідчить, що кожен банківський рахунок має «номер рахунку» та «PIN», які система використовує для ідентифікації рахунків та авторизації користувачів. Надаємо класу **Account** два атрибути типу Integer: accountNumber і PIN.

Клас **Account** все одно повинен зареєструвати суму, внесену користувачем. Тому вирішуємо, що клас **Account** мав би представляти баланс двома атрибутами UML-типу

Double: availableBalance та totalBalance. Однак, оскільки для проєктування нами використовується інструментальний засіб розробки **IBM Rational SoftwareArchterct**, перелік стандартних типів якого обмежений множиною стандартних примітивних типів Boolean, Integer, String і UnlimitedNatural (рис. 3.2), тип Double при описі цих атрибутів замінимо на тип UnlimitedNatural, який при ґенеруванні коду цими інструментальним засобом перетвориться у відповідний дійсний тип, визначений у конкретній ООП-мові, дозволеній у відповідній версії IBM RSA (напр., в C++ це буде тип duble чи float). Атрибут availaBLeBalance відслідковує суму грошей, яку користувач може зняти з рахунку. Атрибут totalBalance відноситься до загальної суми грошей, яку користувач має на депозиті (сума доступних грошей плюс сума, яка очікує на закінчення перевірки або очищення). Напр., припустивши, що користувач CUST App вносить $50.00 на порожній рахунок, атрибут totalBalance збільшиться до $50.00, але availaBLeBalance залишився б нульовим. Передбачається, що система оновлює атрибут availaBLeBalance об'єкта :**Account** після того, як CUST-об’єкт виконає транзакцію у відповідь на підтвердження, що в пакеті знаходяться 50$ готівкою/чеком. Припускається також, що це оновлення виконується за допомогою транзакції, здійсненої банківським службовцем за допомогою якогось ПЗ, що не відноситься до системи CUST App. Тому не розглядаємо цю транзакцію.

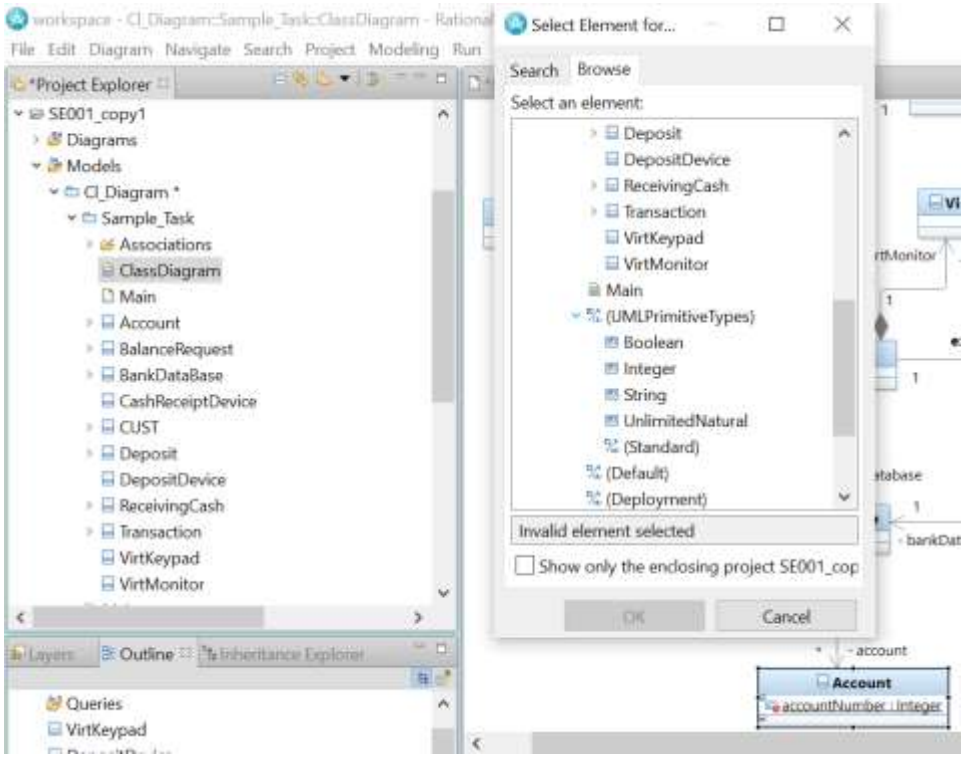


Рис. 3.2. Спливаюче меню IBM RSA для вибору типів атрибутів, арґументів методів та та значень, що повертаються у вигляді класів користувача та стандартних примітивних типів

Клас **ReceivingCash** має один атрибут. Специфікація вимог говорить, що у ДОК кожного дня завантажується 500 20$-банкнот. ДОК повинен відслідковувати число банкнот, що містяться в ДОК, щоб визначати чи достатньо готівки для задоволення запитів на зняття грошей. Тому, у клас **ReceivingCash** включаєм атрибут count типу Integer, що початково встановлюється рівним 500. У реальних задачах програмної індустрії немає гарантії, що специфікація вимог буде достатньо повною, щоб проєктувальник ООП-систем міг визначити

всі атрибути чи навіть класи. Необхідність запровадження додаткових класів, атрибутів та поведінки може з'ясовуватися під час процесу проектування.

Моделювання атрибутів

Діаграма класів на рис. 3. 3 перераховує деякі з атрибутів для класів у системи CUST App, ідентифіковані описові слова та словосполучення з рис. 3.1. Для простоти рис. 3.3 не показує асоціації між класами (вони добре показані на рис. 2.6). Атрибути класу розміщуються в середньому відділенні прямокутника класу на діаграмі. Записуємо ім'я та тип кожного атрибуту, розділяючи їх двокрапкою (**:**). В деяких випадках слід ставити знак рівності (=) та початкове значення.

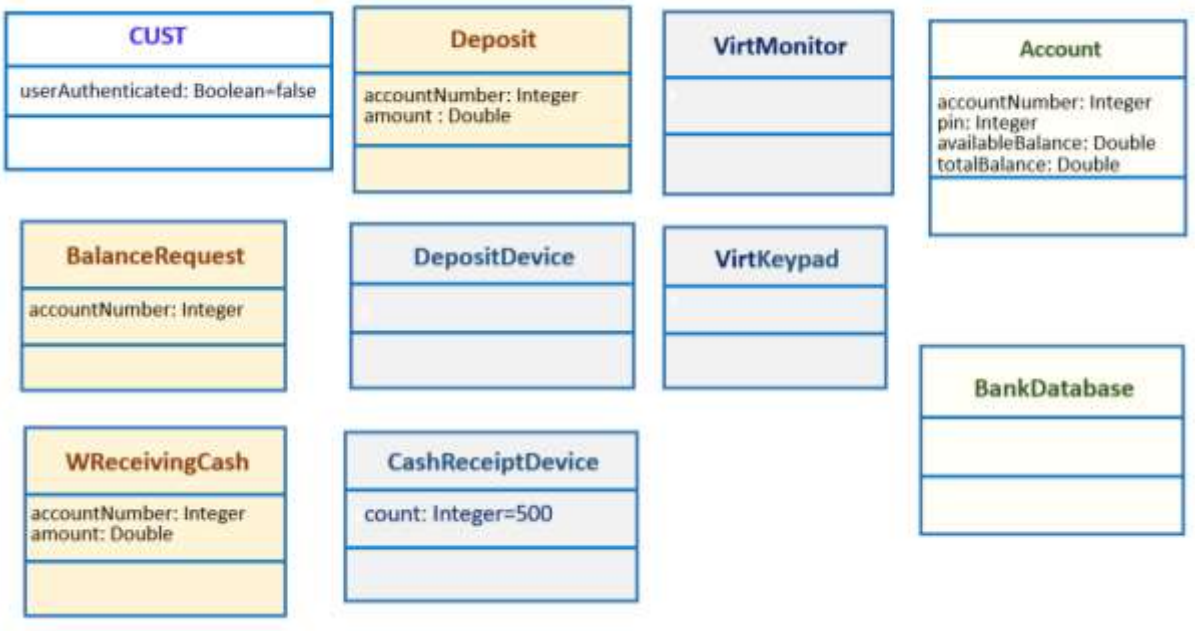


Рис. 3.3. Класи системи CUST App з атрибутами

4. Конструювання ПЗ. Ідентифікація станів об'єктів та діяльності у системі CUST App

Діаграми станів

Діаграми станів (masine state, **MS**) моделюють ключові стани об'єкта та показують, за яких умов він змінює свій стан. На відміну від діаграм класів, орієнтовані на структуру, діаграми станів моделюють аспекти поведінки системи. Діаграма станів (рис. 4.1) моделює деякі із станів об'єкта класу **CUST**. Кожен стан на діаграмі представляється в UML округленим прямокутником з ім'ям стану всередині.

Рис. 4.1. **MS**-діаграма (станів) для об'єкту **CUST**

Діаграми діяльності

Подібно до діаграми станів, діаграма діяльності (activity diagramme, **AD**) моделює аспекти поведінки системи. Але, на відміну від першої, **AD** моделює **робочий потік** (послідовність подій) об'єкта під час виконання програми. Це послідовність дій, в якому порядку буде змінюватись поведінка об'єкт. **AD**-діаграми в ООП успішно замінили колишні «рудиментні» блок-схеми для ілюстрації програмного потоку в керуючих операторах, операторах циклів та ін. обчислювальних процедур (надзвичайно зручно).

AD на рис. 4.2 моделює дії, пов'язані з виконанням транзакції **BalanceRequest**. Припускаємо, що об'єкт :**BalanceRequest** вже ініціалізований і йому надано дійсний номер рахунку(відповідний поточному користувачеві), так що об'єкт знає, який баланс потрібно витягти. У **AD** входять дії, що виконуються після того, як користувач вибере в головному меню довідку про баланс, як CUST App поверне користувача в головне меню; об'єкт :**BalanceRequest** не продукує і не ініціює ці дії, тому їх тут не моделюємо. **AD** починається з «вилучення з БД готівкового балансу на рахунку користувача». Потім транзакція **BalanceRequest** «отримати загальний баланс рахунку» виводить баланс рахунку на Віртуальний Монітор». Цією дією виконання транзакції завершується.

Рис. 4.2. Діаграма діяльності для транзакції **BalanceRequest**

Дія. На діаграмі діяльності UML представляється як стан дії у вході прямокутника, бічні сторони якого замінені дугами опуклістю назовні. Кожен стан дії містить вираз дії – напр., «отримати з БД готівковий баланс рахунку користувача» - формулює конкретну дію, яка має бути зроблена. Лінія зі стрілкою, що сполучає 2 стани дії, вказує порядок, у якому повинні виконуватися дії станами дії. Суцільний круг (рис. 4.2) візуалізує початковий стан діяльності - початок робочого потоку, перед тим, як об'єкт виконуватиме моделювання дії. В даному випадку транзакція виконує перший вираз дії «одержати з БД готівковий баланс рахунку користувача». Потім транзакція повертає загальний баланс рахунку. Далі, транзакція виводить обидва баланси на Віртуальний монітор. Суцільний круг, вкладений всередині порожнього круга (рис. 4.2, знизу) описує кінцевий стан - кінець робочого потоку після того, як об'єкт виконає моделювання дій.

На рис. 4.3 показана **AD** для транзакції **ReceivingCash** «отримання готівки». Припускаємо, що об'єктом :**ReceivingCash** був отриманий дійсний номер рахунку.

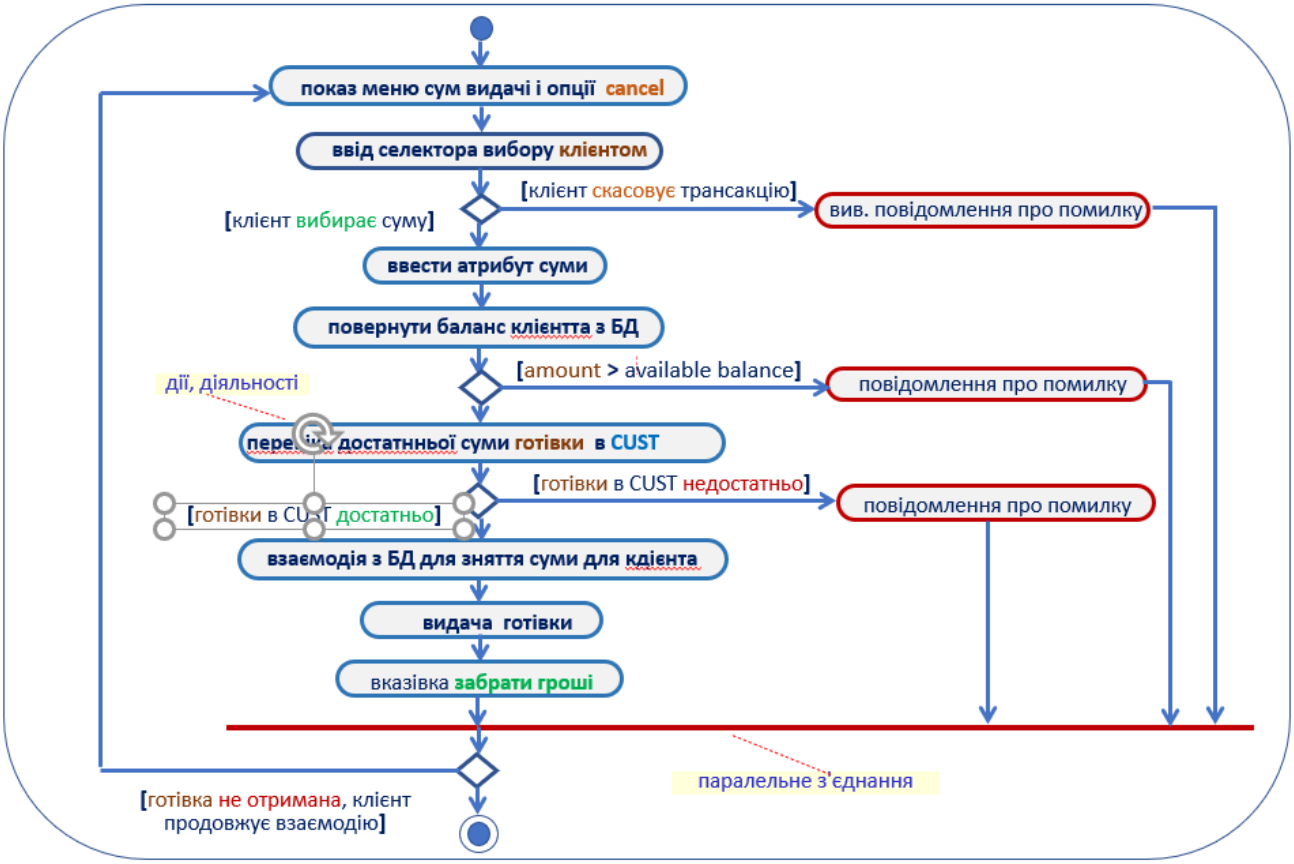


Рис. 4.3 Діаграма діяльності для транзакції **ReceivingCash**

Транзакція **ReceivingCash** спочатку показує меню стандартних сум, що знімаються (рис. 1.3) з опцією для скасування транзакції. Потім транзакція вводить вибір користувача. Робочий потік досягає тепер блоку **рішення**. У цій точці наступна дія вибирається в залежності від асоційованих з нею контрольних умов. Якщо користувач скасовує транзакцію, система виводить відповідне повідомлення. Потім потік скасування досягає блоку **злиття**, де цей потік з'єднується з іншими можливими у транзакції потоками діяльності. Злиття може мати скільки завгодно вхідних стрілок переходу, але тільки одну вихідну. Рішення у нижній частині **AD** визначає, чи має транзакція повторитись від початку.

Якщо користувач скасував транзакцію **ReceivingCash**, контрольна умова «готівка видана або користувач скасував транзакцію» = true, управління переходить до кінцевого стану діяльності. Якщо користувач вибрав у меню суму зняття значення транзакції встановлює в amount (атрибут класу **ReceivingCash**, початково змодельований на рис. 3.3). Потім транзакція отримує з БД **готівковий баланс** рахунку користувача (атрибут availaBLeBalance об'єкта :**Account** користувача).

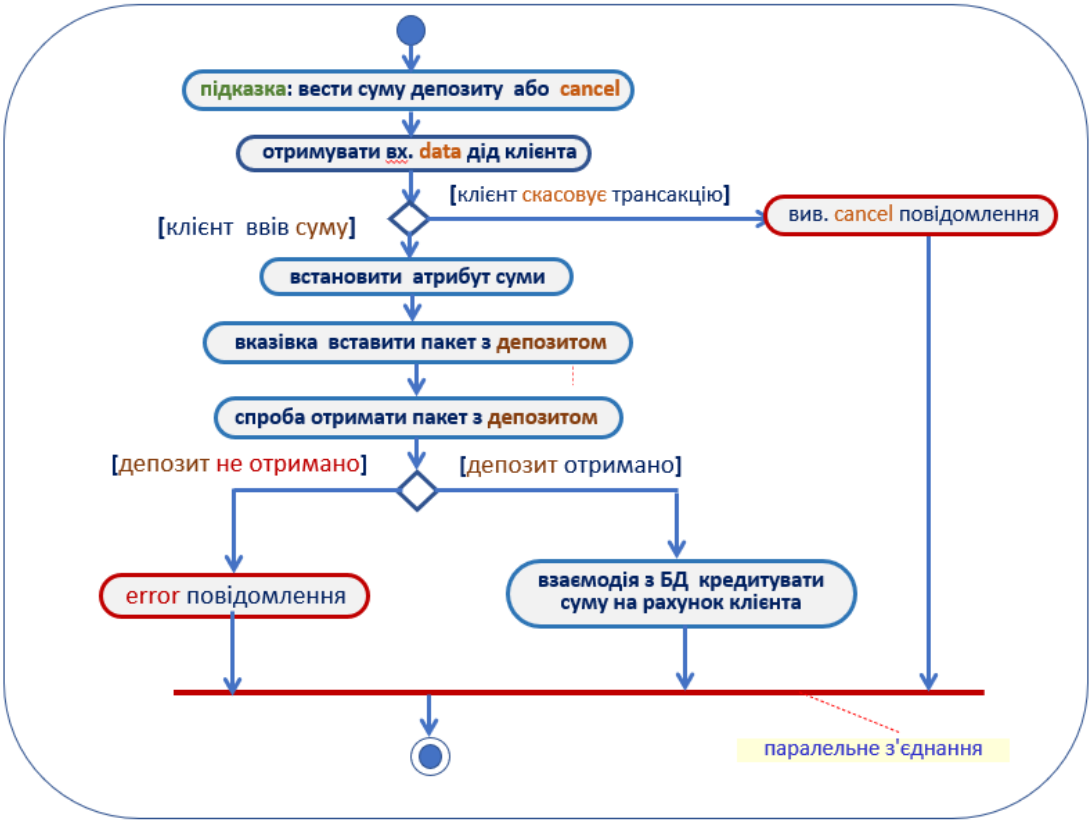


Рис. 4.4. **AD** діаграма для транзакції **Deposit**

Це перші кроки у моделюванні поведінки системи CUST App, де показано як атрибути об'єкта беруть участь у діяльності об'єкта. Далі досліджуємо операції класів, щоб створити повнішу і цілеснішу модель поведінки системи.

5. Конструювання ПЗ. Ідентифікація операцій класів у системі CUST App

На цей момент виконано такі етапи об'єктно-орієнтованого проектування CUST App:

- ідентифіковано класи, необхідні для реалізації ;
- створено першу діаграму класів; описано деякі атрибути класів;
- досліджувались стани об'єктів та моделювались переходи станів та діяльність об'єктів.

Визначимо деякі операції (дії або поведінку) класів, необхідних реалізації системи CUST App.

Ідентифікація операцій

Операція - послуга, яку об'єкти класу надають клієнтам класу. Розглянемо операції деяких об'єктів реального світу.

Багато операцій кожного класу можна вивести за допомогою **дослідження ключових дієслів та дієслівних конструкцій** у специфікації вимог. Потім співвіднесемо їх із конкретними класами проєктованої системи (рис. 5.1). Дієслові конструкції (рис. 5.1) допоможуть визначити операції кожного з класів.

Клас	Дієслова і дієслівні конструкції
CUST	керує виконанням фінансових трансакцій
VirtMonitor	показує користувачеві повідомлення
VirtKeypad	приймає від користувача числовий ввод
CashReceiptDevice	видає кошти, повідомляє, чи достатньо готівки для задоволення запиту про зняття та грошей
DepositDevice	приймає суму з депозитом
BankDataBase	авторизує користувача, отримує баланс рахунку, кредитує суму, що вноситься на рахунок, дебетує суму, що знімається з рахунку
Account	отримує баланс рахунку, кредитує суму, що вноситься на рахунок, дебетує суму, що знімається з рахунку
BalanceRequest	Виконує перевірку загального балансу рахунку користувача
ReceivingCash	виконує трансакцію отримання готівки користувачем
Deposit	виконує трансакцію депозиту коштів користувача

Рис. 5.1. Дієслова і дієслівні конструкції для кожного із класів проектування системи CUST App

Моделювання операцій

Для ідентифікації операцій досліджуємо дієслівні конструкції. для кожного класу (рис. 5.1). Конструкція «виконує фінансові транзакції», асоційована з класом **CUST**, визначає, що клас **CUST** дає вказівки виконуватися транзакціям. Отже, до кожного з класів **BalanceRequest**, **ReceivingCash** та **Deposit** потрібна спільна операція, що надає **CUST** таку послугу. Цю операцію (execute) розміщуємо у третій розділ цих трьох класів транзакцій у модифікованій діаграмі класів (рис. 5.2). Під час сеансу **CUST** об'єкт **: CUST** активуватиме операцію execute() кожного об'єкта транзакції, вимагаючи, щоб вони виконувались. У UML операції (реалізовані як методи C++) надаються у вигляді перерахування імені операції, списку розділених комами параметрів у дужках, двокрапки та типу, що повертається: імяОперації(арґумент1, арґумент2, ... , арґументN) : значення, що повертається. Кожен арґ у розділеному комами списку складається з імені арґумента, за яким слідують двокрапка і тип арґумента: ім'яАрґумета: типАрґумента. Поки що не перераховуємо арґ операцій; згодом ідентифікуємо і змоделюємо арґументи деяких із них. Для деяких операцій тип, що повертається, може бути ще невідомим, тому на діаграмі опускаємо його. Під час подальшого проектування та реалізації по мірі необхідності будемо вводити інші типи, що повертаються, у Таблицю властивостей (Propeties) класів у нижній частині робочої області меню «Створення діаграм класів» діаграми класів IBM RSA.

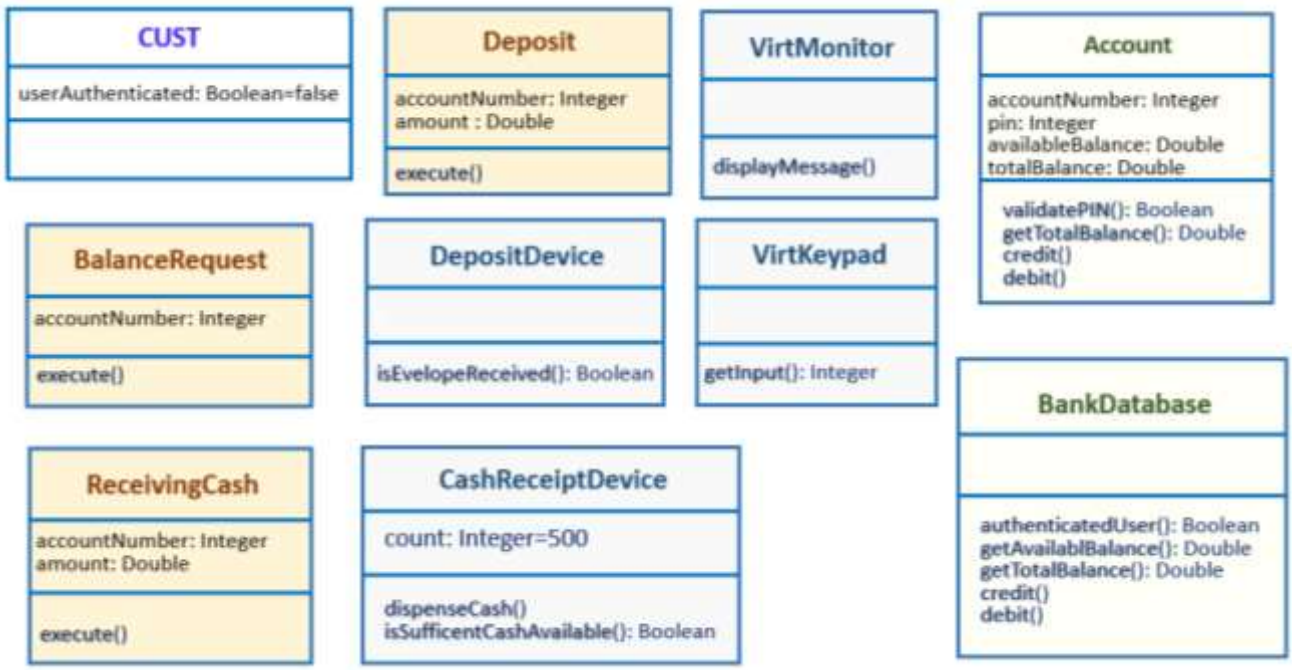


Рис. 5.2. Класи в CUST App з проектованими атрибутами і операціями

Кредитування рахунку (як при внесенні грошей) додає грошову суму лише до атрибуту totalBalance. З іншого боку, дебетування рахунку (як при знятті) віднімає грошову суму з обох атрибутів балансу. Ці деталі реалізації операцій приховуємо реалізації всередині класу **Account**, що є показовим прикладом інкапсуляції інформації.

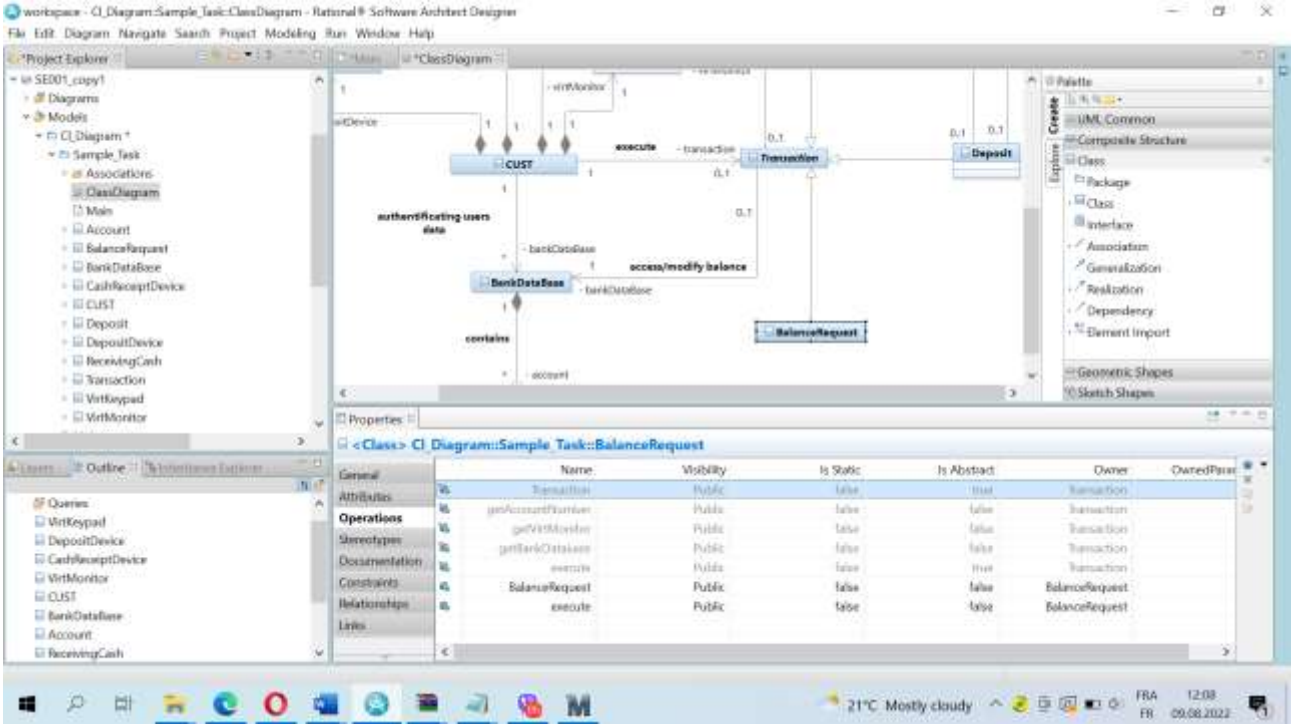


Рис. 5.2.1. Демонстрація у підменю **Propeties** інкапсульованих операцій виділеного на діаграмі класу **BalanceRequest**, включаючи успадковані

Ідентифікація та моделювання параметрів операцій

До цих пір ми тільки обмежувались загальним розумінням операцій кожного класу. Ідентифікуємо параметри операції, вивчаючи, які дані потрібні операції для виконання свого завдання. Розглянемо це на прикладі операції authenticateUser() класу **BankDatabase**. Щоб авторизувати користувача, ця операція повинна знати номер рахунку та PIN, надані користувачем. Тому, специфікуємо, що операція authenticateUser() приймає арґументи userAccountNumber і userPIN типу Integer, які вона має порівняти їх з відповідними номером рахунку та PIN об'єкта **Account** у БД. Даємо іменам цих арґументів префікс user щоб не плутати імена параметрів операції з іменами атрибутів, що належать до класу **Account**. Записуємо ці арґументи у діаграмі класів конкретно для класу **BankDatabase.**

Звичайно, що надмірна деталізація діаграми ієрархії класів шляхом детальної параметризації усіх атрибутів і операції є поганою практикою програмної інженерії. Хоча багато розробників, не говорячи вже про студентів, цим нехтують, намагаючись показати ніби все і зразу. Однак це говорить про низький рівень чи про відсутність архітектурного, естетичного та художнього смаку розробників. Такі діаграми є дуже громіздкими, практично нечитабельними, у яких важко розбиратись. Саме застосування сучасних версій IBM RSA дає можливість уникнути такого убогого стилю розробки шляхом використання вбудованих засобів фільтрації та інкапсуляції на діаграмі класів надмірних і мало важливих в певному контексті зв’язків, атрибутів та методів (див. рис.5.3, де на спливаючому меню показна вкладка **Filters** з додатковими вікнами розгалужених можливостей інкапсуляції введених в БД Властивостей діаграми класів різних груп чи окремо взятих зв’язків, атрибутів та методів та ін.). Детальніший опис з цими інструментами продемонстровано у розділі опису роботи IBM RSA по створенню діаграм класів.

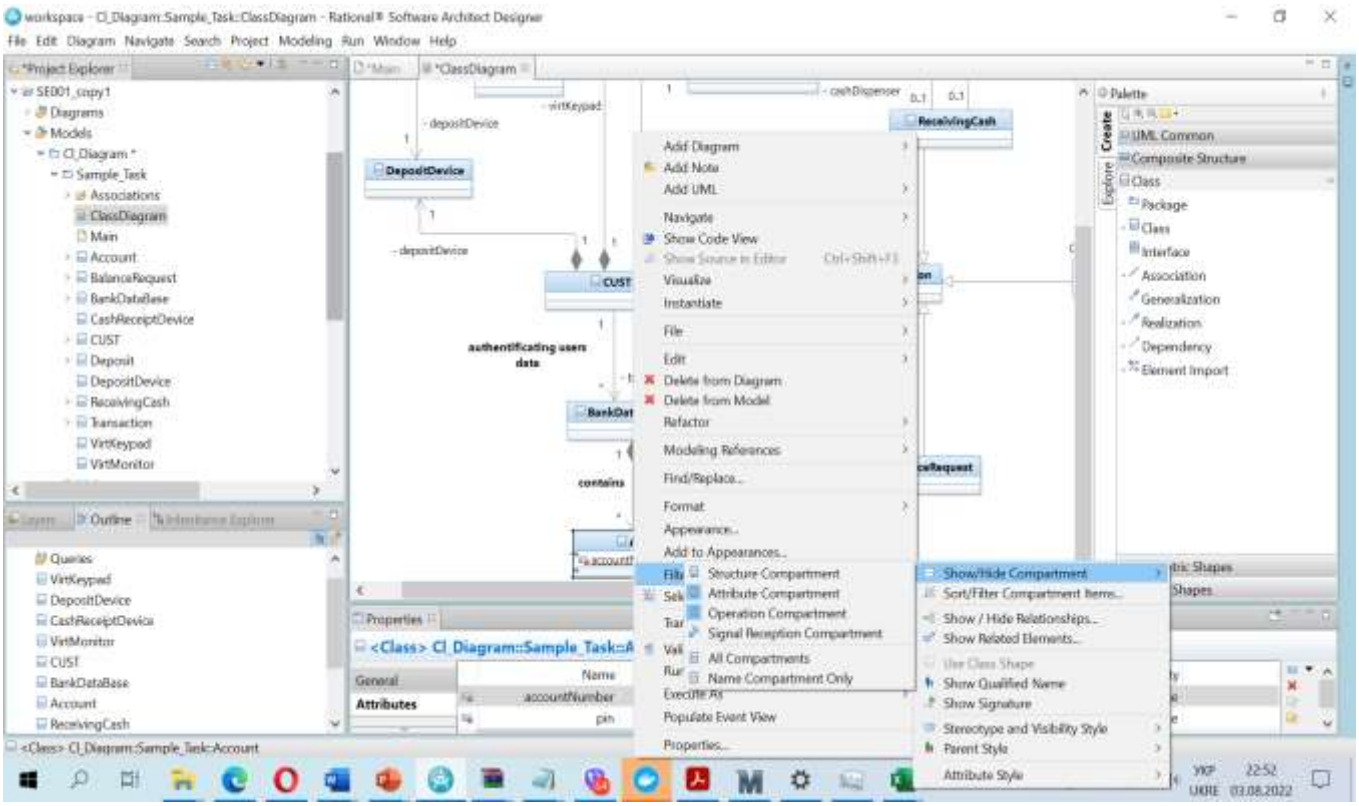


Рис. 5.3. Спливаюче меню на робочій області IBM RSA з вкладкою **Filters** та додатковими вікнами інкапсуляції властивостей та структури діаграми класів

Це суттєво також підвищує ефективність проектування ПЗ за рахунок уникнення багатократної побудови зайвих діаграм ієрархії класів та діаграм із детальним зображенням окремих класів. Засоби фільтрації та інкапсуляції IBM RSA дають можливість багатофункціонального і багаторазового використання однієї і тієї ж максимально повно параметризованої діаграми класів шляхом побудови різних її виглядів, що акцентують увагу на специфіці окремих сценаріїв реалізації, окремих елементів структури чи окремих класів, інкапсулюючи мало значимі зв'язки, класи, атрибути, методи. Можна окремо повністю візуалізувати структуру окремих класів без повторного створення їх діаграм.

На рис. рис. 5.4 подана окрема діаграма, яка моделює та деталізує властивості і поведінку тільки одного класу **BankDataBase**. В даному випадку хочемо насамперед досліджувати параметри цього окремого класу, тому опускаємо решту класів і зав'язків між ними. Пізніше, коли параметри (арґументи) операцій вже не будуть перебувати в центрі уваги: для економії місця їх також можна опускати (інкапсульовувати) в діаграмах класів.

Кожен арґумент у розділеному комами списку арґументів операції моделюється в UML іменем арґумента, за яким слідує двокрапка і тип (в нотації UML). Рис. 5.4 специфікує, що операція authenticateUser() має 2 арґументи - userAccountNumber і userPIN, обидва типу Integer. При реалізації системи на C++ значення цих аґументів матимуть тип int.

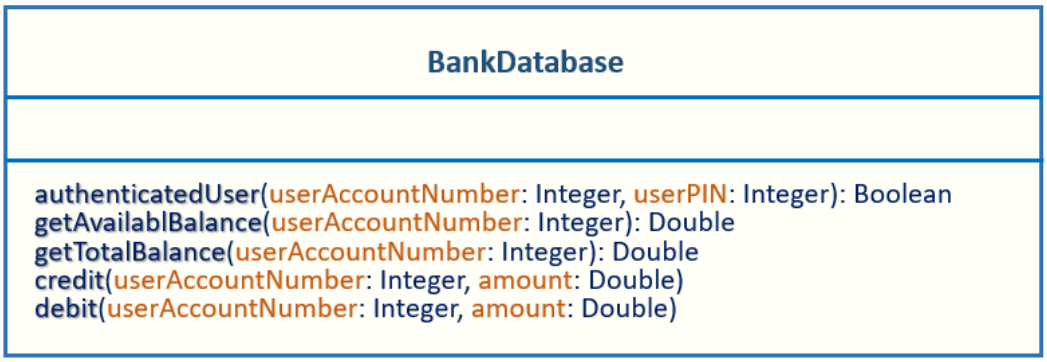


Рис. 5.3. Клас **BankDataBase** з операціями, їх арґументами, значеннями, що повертаються

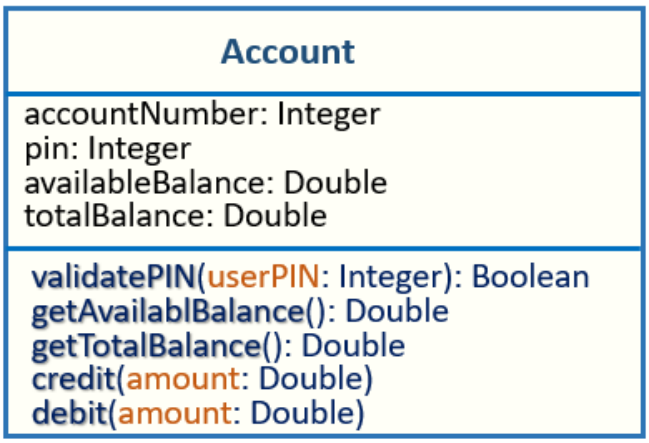


Рис. 5.4. Клас **Account** з операціями

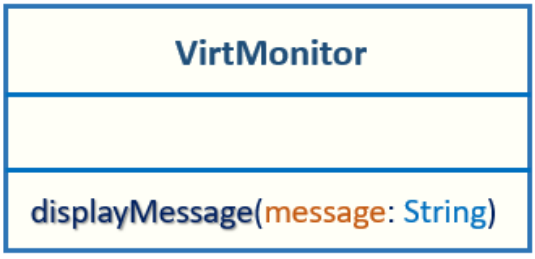


Рис. 5.5. Клас **VirtMonitor** з операціями

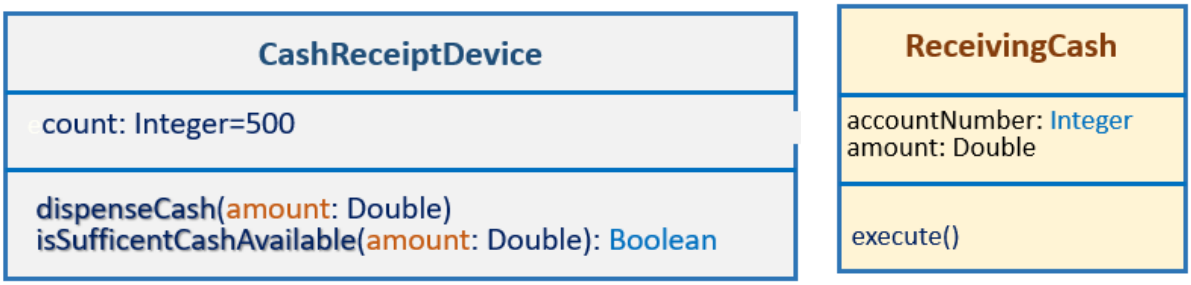


Рис. 5.6. Класи **CashReceiptDevice** і **ReceivingCash** з атрибутами та операціями

6. Конструювання ПЗ. Кооперація об'єктів у системі CUST App

Обговоримо кооперації (взаємодії) об'єктів в проєктованій системі CUST. Коли 2 об'єкти вступають між собою у комунікацію, вони кооперуються, активуючи операції один одного. Кооперація полягає в тому, що об'єкт одного класу посилає повідомлення об'єкту іншого класу. У C++ повідомлення надсилаються через виклики методів (у р.5 визначено багато операцій класів системи). Зосередимо увагу на **повідомленнях**, якими ці операції активуються. Щоб ідентифікувати у системі кооперації, повернемось до **специфікації вимог** (р. 1), що описує різноманітну діяльність, що відбувається у час сеансу CUST App (напр., ідентифікація користувача, виконання транзакцій).

Покрокові описи того, як система повинна виконувати кожне із цих завдань, є першою вказівкою на кооперацію в системі. Тут можемо розкрити додаткові кооперації.

Ідентифікація кооперацій у системі. Ідентифікуємо кооперації у системі, вивчаючи розділи специфікації вимог, у яких описується, що повинена робити система CUST App, щоб авторизувати користувача та здійснювати транзакції кожного типу. Для кожної дії або кроку, описаного в специфікації, вирішуємо, які об'єкти системи повинні взаємодіяти, щоб досягти бажаного результату.

Ідентифікуємо один об'єкт як **об'єкт-відправник** (об'єкт, що надсилає повідомлення), а інший - як **об'єкт-одержувач** (що пропонує клієнтам класу цю операцію). Потім вибираємо одну з операцій об'єкта-одержувача (ідентифікованих у р. 5), яка має бути активована об'єктом-відправником для здійснення потрібної поведінки. Напр., у стані очікування об’єкт **:CUST** показує вітальне повідомлення. Об'єкт класу **VirtMonitor** виводить користувачеві повідомлення за допомогою своєї операції displayMessage(). Отже, вирішуємо, що система може показати вітальне з повідомлення, здійснюючи кооперацію між **:CUST** та :**VirtMonitor**, в якій **:CUST** посилає об’єкту :**VirtMonitor** повідомлення displayMessage() шляхом активації операції displayMessage() класу **VirtMonitor**. Для уникнення повторень слів «об'єкт класу або роль, яку він виконує при реалізації ВВ», використовуємо ім'я класу з двокрапкою попереду, - напр., **:CUST**. На рис. 6.1 перераховані кооперації, які можна вивести зі специфікації вимог. Для кожного об'єкта-відправника перераховуємо кооперації у тому порядку, як вони

обговорюються у специфікації вимог. Записуємо кожну кооперацію, що включає унікальні повідомлення відправника до одержувача, лише один раз, хоча кооперація може відбуватися неодноразово протягом сеансу :**CUST**. Напр., перший рядок на рис. 6.1 означає, що :**CUST** кооперується зі :**VirtMonitor** у всіх випадках, коли CUST App має показати користувачеві повідомлення.

Об'єкт класу …	Відправляє повідомлення	Об'єкту класу …
CUST	displayMessage()	**VirtMonitor**
	getInput()	**VirtKeypad**
	authenticateUser()	**BankDataBase**
	execute()	**BalanceRequest**
	execute()	**ReceivingCash**
	execute()	**Deposit**
BalanceRequest	getAvailaЬleBalance()	**BankDataBase**
	getTotalBalance()	**BankDataBase**
	displayMessage()	**VirtMonitor**
ReceivingCash	displayMessage()	**VirtMonitor**
	getInput()	**VirtKeypad**
	getAvailaЬleBalance()	**BankDataBase**
	isSufficientCashAvailaЬle()	**ReceivingCash**
	deЬit()	**BankDataBase**
	dispenseCash()	**ReceivingCash**
Deposit	displayMessage()	**VirtMonitor**
	getInput()	**VirtKeypad**
	isEnvelopeReceived()	**DepositDevice**
	credit()	**BankDataBase**
BankDataBase	validatePIN()	**Account**
	getAvailaЬleBalance()	**Account**
	getTotalBalance()	**Account**
	deЬit()	**Account**
	credit()	**Account**

Рис. 6.1. Кооперація в системі CUST App

Аналіз кооперацій (рис. 6.1). Перед тим як дозволити користувачеві робити будь-які транзакції, **:CUST** повинен запропонувати користувачеві ввести номер свого рахунку, а потім ввести PIN. Об'єкт **:CUST** вирішує кожне з цих завдань, посилаючи :**VirtMonitor** повідомлення displayMessage(). Обидві ці дії відносяться до однієї і тієї ж кооперації між **:CUST** та **:VirtMonitor**, яка вже врахована (рис. 6.1). Об'єкт **:CUST** отримує від користувача введення у відповідь на запрошення, посилаючи об'єкту **:VirtKeypad** повідомлення getInput(). Далі :**CUST** має визначити, чи відповідають номер рахунку та PIN, зазначені користувачем, номеру та PIN рахунку, що зберігається у БД. Він робить це, посилаючи :**BankDatabase** повідомлення authenticateUser(). Об'єкт **:BankDatabase** не може авторизувати користувача безпосередньо - тільки **Account** користувача (:**Account**, який містить вказаний користувачем номер рахунку) має доступ до PIN користувача та може його авторизувати.

Відповідно до рис. 6.1 враховано кооперацію, в якій об'єкт **:BankDatabase** посилає об'єкту **:Account** повідомлення validatePIN(). Як тільки користувач авторизований, об'єкт **:CUST** показує головне меню, посилаючи ряд повідомлень displayMessage(), і отримує введення - вибір в меню, - посилаючи об'єкту **:VirtKeypad** повідомлення getInput().

Після того, як користувач вибере потрібний тип транзакції, об’єкт **:CUST** виконує цю транзакцію, надсилаючи повідомлення execute() об'єкту відповідного класу транзакції (**:BalanceRequest**, **:ReceivingCash** або **:Deposit**).

Якщо користувач вибрав трансакцію «Перевірка балансу», об’єкт **:CUST** надсилає повідомлення об'єкту **:BalanceRequest** виконати його метод execute(). Подальше вивчення специфікації вимог розкриває кооперації, щодо виконання транзакцій кожного типу. Об’єкт **:BalanceRequest** отримує значення суми готівкових коштів на рахунку користувача (суму, яку користувач може забрати з рахунку), надсилаючи повідомлення об'єкту **:BankDataBase**, виконати його метод getAvailaBLeBalance(). Об'єкт **:BankDataBase** у відповідь на це посилає повідомлення об'єкту **:Account** виконати перевантажений його метод getAvailaBLeBalance().

Аналогічно, об’єкт **:BalanceRequest** отримує значення суми коштів, що є на депозиті, посилаючи об’єкту **:BankDataBase** повідомлення виконати його метод getTotalBalance() – повернути загальний баланс (суму депозиту) користувача, а **:BankDataBase** посилає те саме повідомлення об'єкту **:Account** користувача. Для одночасного виведення обох сум балансу (готівкових і депозитних коштів користувача), об’єкт **:BalanceRequest** посилає об'єкту **:VirtMonitor** повідомлення виконати його метод displayMessage().

Якщо користувач вибрав трансакцію «Зняти готівку», то об’єкт **:ReceivingCash** посилає об'єкту**: VirtMonitor** ряд повідомлень виконати його метод displayMessage(), щоб вивести меню стандартних сум, що знімаються ($20, $40, $60, $100 і $200). Щоб отримати вибір користувача в меню, об’єкт **:ReceivingCash** посилає повідомлення об’єкту :**VirtKeypad** виконати його метод getInput(). Потім об’єкт **:ReceivingCash** перевіряє чи запитана користувачем сума менша або рівна доступному балансу готівкових коштів на рахунку користувача. Об’єкт **:ReceivingCash** може отримати суму готівки з рахунку користувача, надіславши повідомлення об’єкту **:BankDataBase** виконати його метод getAvailaBLeBalance() – повернути значення доступної суми готівкових коштів. Об’єкт **:ReceivingCash** потім перевіряє, чи достатньо готівки у ДОК, посилаючи об’єкту **:ReceivingCash** повідомлення isSufficientCashAvailaBLe(). Об’єкт **:ReceivingCash** посилає повідомлення об'єкту **:BankDatabase** виконати його метод debit(), щоб зняти запитувану користувачем суму з його балансу готівкових коштів в БД. Об’єкт **:BankDatabase** у відповідь на це посилає повідомлення getAvailaBLeBalance() об'єкту **:Account**.

Операція дебітування коштів з рахунку зменшує значення атрибутів totalBalance, так і availaBLeBalance. Для видачі запитуваної суми готівки об’єкт **:BankDatabase** посилає об'єкту **:ReceivingCash** повідомлення dispenseCash(). Відповідно, об’єкт **:ReceivingCash** посилає об'єкту **:VirtMonitor** повідомлення displayMessage(), пропонуючи користувачеві забрати гроші.

Якщо користувач вибрав трансакцію «Покласти кошти на депозит», то об’єкт **:Deposit,** отримавши повідомлення від об’єкта **CUST** виконати свій метод execute(), посилає спочатку повідомлення об'єкту **:VirtMonitor** виконати його метод displayMessage(), щоб запропонувати користувачеві ввести депоновану суму.

Щоб отримати введення користувача, об’єкт **:Deposit** посилає **:VirtKeypad** повідомлення getInput(). Потім об'єкт **:Deposit** посилає об’єкту **:VirtMonitor** повідомлення displayMessage(), щоб запропонувати користувачеві вставити пакет із депозитом у ДВД. Щоб визначити, чи отримав ДВД депозит, об’єкт **:Deposit** посилає об’єкту повідомлення **:DepositDevice** виконати його метод isEnvelopeReceived(). Якщо ОК, об’єкт **:Deposit** оновлює рахунок, посилаючи повідомлення об'єкту **:BankDatabase**, виконати його метод credit(). Об'єкт **:BankDatabase** у свою чергу посилає повідомлення credit() об’єкту **:Account** користувача. Кредитуввання коштів на рахунок збільшує тільки значення атрибута totalBalance, але не атрибута availableBalance.

Діаграми взаємодій

Коли вже ідентифіковано набір можливих кооперацій між об'єктами в системі, необхідно графічно змоделювати ці взаємодії за допомогою UML. У UML передбачено декілька типів діаграм взаємодії, що моделюють поведінку системи шляхом моделювання того, як об'єкти взаємодіють один з одним. Діаграма комунікації наголошує те, які об'єкти беруть участь у кооперацiях (у ранніх версіях UML діаграми комунікації називалися діаграмами кооперації). Діаграма послідовності (як і діаграма комунікації) показує кооперації між об'єктами, але акцент у ній робиться на тому, як повідомлення, що посилаються об'єктами співвідносяться у часі.

Діаграми комунікації

Рис. 6.2 показує діаграму комунікації (comunication diagramme, **CD**), яка моделює взаємодію об’єкта :**CUST**з об’єктом :**BalanceRequest** щодо посилання першим повідомлення другому виконати його метод execute(), що здійснює реалізацію транзакції **BalanceRequest** (запит/перевірка балансу). Об'єкти моделюються в UML прямокутниками з іменами об'єктів у формі ім'я об'єкта: ім'я класу. Тут ім'я об'єкта опущено (використовується безіменний об’єкт з метою обмеження великої кількості непотрібних назв), вказано тільки ім'я класу, якому передує двокрапка. Якщо на **CD** моделюється декілька об'єктів одного класу, то тоді рекомендується специфікувати ім'я кожного об'єкта.

Об'єкти, що є у комунікації, з'єднуються суцільними лініями, а повідомлення між ними передаються вздовж цих ліній у напрямі, вказаному стрілками. Ім'я повідомлення, яке вказується поруч зі стрілкою, є ім'ям операції (методу) , що належить об'єкту-одержувачу. Інакше кажучи, ім'я операції – це послуга, яку об'єкт-одержувач пропонує об'єктам-відправникам (своїм «клієнтам»).

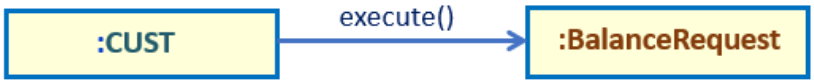


Рис. 6.2 **CD**-діаграма для :**CUST**-об’єкта, що виконує перевірку балансу

Суцільна лінія зі стрілкою (рис. 6.2) моделює повідомлення - або синхронний виклик - в UML і виклик методу C++. Вона вказує, що потік управління йде від об'єкта-відправника (**:CUST**) до об'єкта-одержувача (:**BalanceRequest**). Оскільки це синхронний виклик, об'єкт-відправник не може надсилати інші повідомлення або робити що-небудь ще, поки об'єкт-одержувач не обробить повідомлення і не поверне управління відправнику. Напр., об’єкт **:CUST** (рис. 6.2) викликає метод **execute**() об'єкта :**BalanceRequest** і не може надіслати інше повідомлення, поки **execute**() не закінчиться і не поверне керування **:CUST**. Якби це був **асинхронний виклик** (нежирна стрілка), об'єкту-відправнику не потрібно було чекати поки об'єкт-одержувач не поверне управління, - він міг би відразу після асинхронного виклику продовжити надсилання додаткових повідомлень.

Діаграми послідовності

CD - діаграми акцентовані на учасників кооперацій, але дещо незграбно моделюють їх часовий аспект. Діаграми послідовності (sequence Diagrames, **SD**) допомагають моделювати часові відношення кооперацій. Рис. 6.3 показує **SD**, що моделює послідовність

взаємодій при виконанні **ReceivingCash**. Пунктирна лінія, що йде від об'єкта (нотація прямокутника) вниз - лінія життя (ЛЖ) даного об'єкта протягом часу. Дії відбуваються вздовж ЛЖ об'єкта в хронологічному порядку - дія, розташована ближче до верхнього кінця ЛЖ, відбувається раніше дії, що розташованої ближче до її нижнього кінця.

Надсилання повідомлень на **SD** зображується аналогічно надсилання повідомлень на **CD**. Повідомлення показується суцільною стрілкою, що йде від об'єкта-відправника до об'єкта-одержувача. Стрілка вказує на активацію на ЛЖ об'єкта-одержувача. **Активація** (фокус активації об’єкта, ФА), зображений вузьким вертикальним прямокутником, що показує, що об'єкт виконується. Коли об'єкт повертає управління, то від його ФА до ФА об'єкта, що послав вихідне повідомлення, проходить зворотне повідомлення (пунктирна лінія з незафарбованою стрілкою). Щоб не захаращувати **SD**, стрілки зворотних повідомлень опускаєм (UML це допускає для чіткішої візуалізації **SD**). **SD** і **CD** можуть вказувати імена параметрів повідомлення у дужках після імені повідомлення.

Послідовність повідомлень (рис.6.4) починається, коли об'єкт :**ReceivingCash** пропонує користувачеві вибрати суму, що знімається, надсилаючи повідомлення displayMessage() об'єкту **VirtMonitor**. Потім :**ReceivingCash** надсилає повідомлення getInput() об'єкту **VirtKeypad**, який отримує дані введення користувача. Логіка управління при виконанні :**ReceivingCash** нами вже моделювалась у **AD** на (рис. 4.3), тому не показуємо її на **SD** на рис. 6.3. Натомість моделюємо тут найкращий сценарій ВВ, коли баланс користувача рахунку >= сумі, що знімається, а у ДОК достатньо готівки для задоволення запиту. Після отримання суми, що знімається, :**ReceivingCash** посилає повідомлення об'єкту :**BankDatabase**, який у свою чергу надсилає повідомлення getAvailableBalance() об'єкту :**Account**. У припущенні, що на рахунку користувача достатньо грошей для дозволу транзакції, об'єкт :**ReceivingCash** потім надсилає повідомлення isSufficientCashAvailaBLe() об'єкту :**ReceivingCash**. У припущенні, що у ДОК достатньо готівки, :**ReceivingCash** зменшує баланс рахунку користувача (тобто атрибути і totalBalance, і availaЬleBalance), надсилаючи повідомлення debit() об'єкту **BankDatabase**. Останній реагує на це, надсилаючи повідомлення debit() об'єкту :**Account** користувача. Нарешті, :**ReceivingCash** посилає повідомлення dispenseCash() об'єкту :**ReceivingCash** та повідомлення displayMessage() об'єкту :**VirtMonitor**, інформуючи користувача, що він може забрати з **CUST**-об’єкта гроші.

Отже, з допомогою **SD** і **CD** змодельовано ідентифіковано кооперації між об’єктами в системі CUST App.

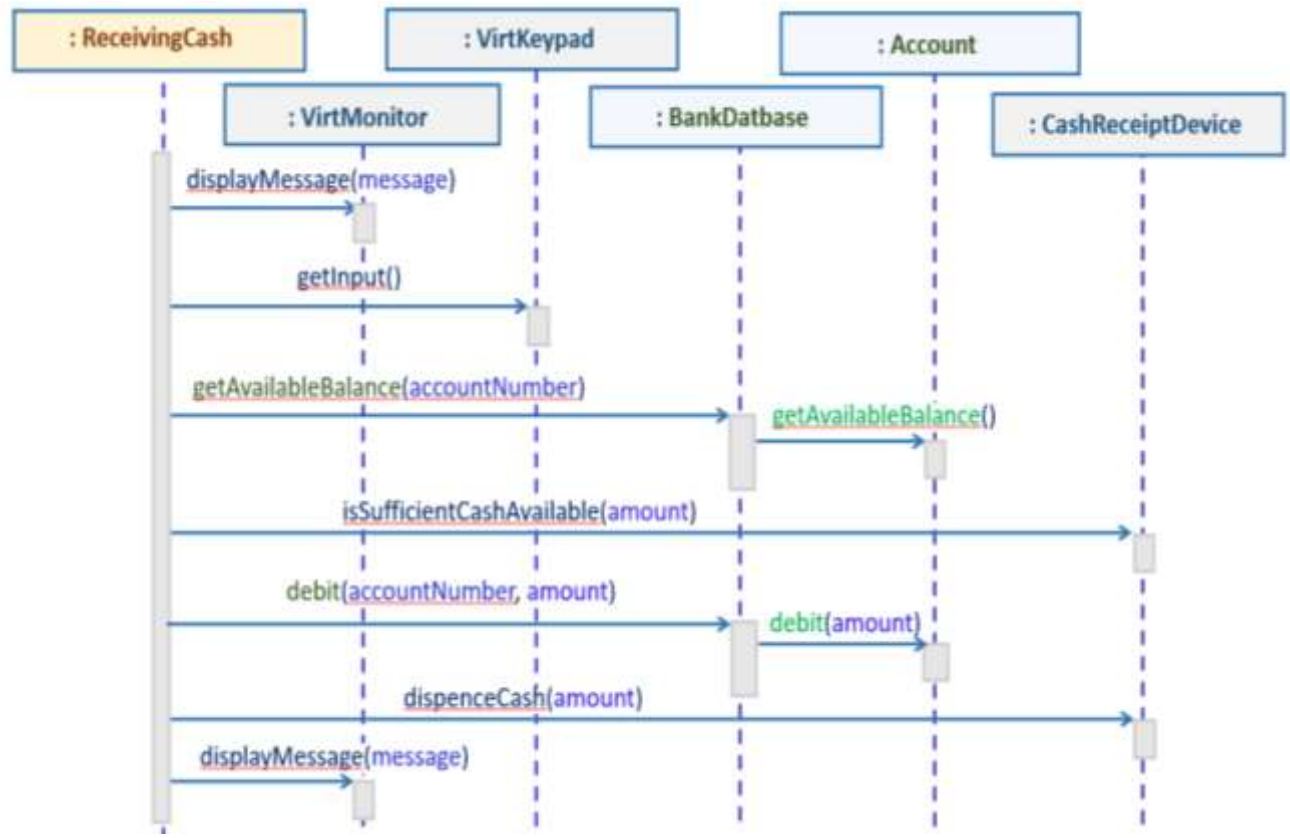


Рис. 6.3 **SD**-діаграма, що моделює реалізацію сценарію «Зняти готівку з рахунку» (**ReceivingCash**)

На рис. 6.4 представлена **SD**, що моделює взаємодії об'єктів у системі CUST, що відбуваються у разі, коли :**Deposit** завершується успішно. **SD** показує, що спочатку :**Deposit** посилає повідомлення displayMessage() об'єкту :**VirtMonitor**, щоб попросити користувача ввести суму депозиту. Потім :**Deposit** посилає повідомлення getInput() об'єкту :**VirtKeypad**, щоб отримати введену користувачем суму. Після цього :**Deposit** просить користувача внести пакет депозите в ДВД, надсилаючи повідомлення displayMessage() об'єкту :**VirtMonitor**. :**Deposit** потім посилає повідомлення isEnvelopeReceived() об'єкту :**DepositDevice**, щоб підтвердити, що пакет отримано. Нарешті, :**Deposit** збільшує атрибут totalBalance (але не availaBLeBalance) об'єкта :**Account** користувача, надсилаючи повідомлення credit() об'єкту :**BankDatabase**. Останній реагує на це, посилаючи те саме повідомлення об'єкту :**Account**.

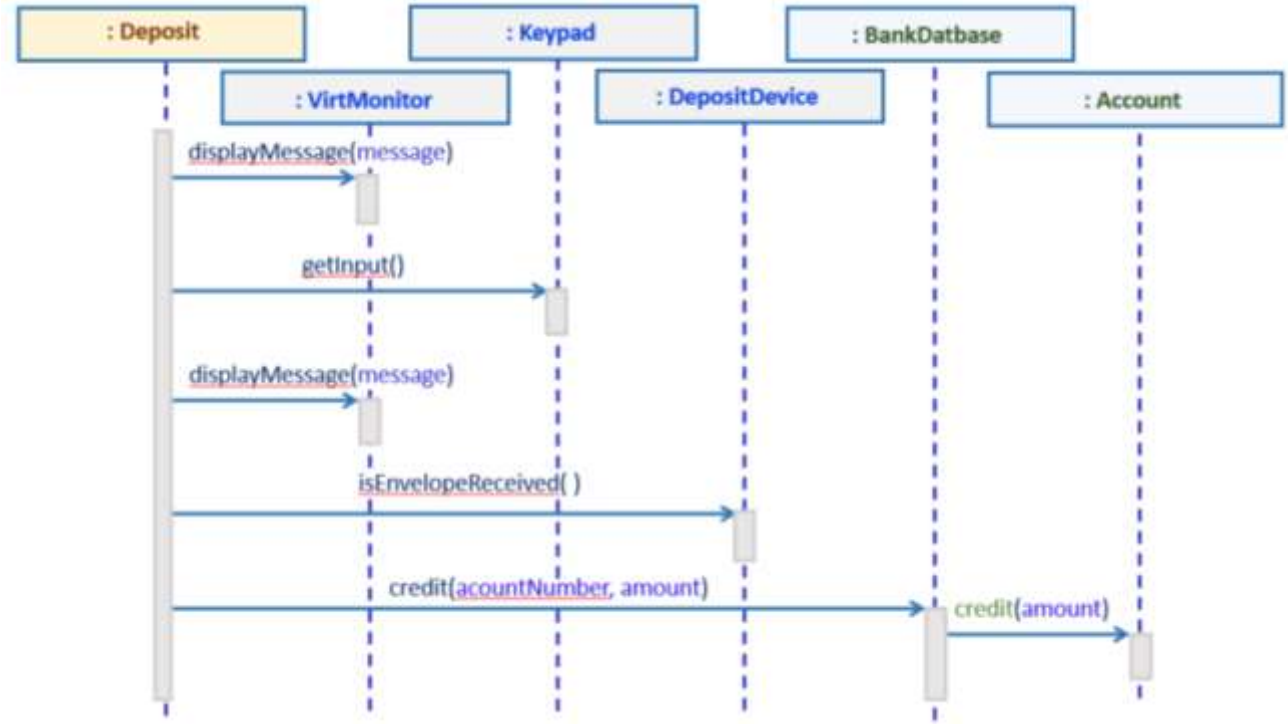


Рис. 6.4. **SD**- діаграма, що моделює реалізацію сценарію трансакції «Покласти депозит на рахунок» (**Deposit**)

7. **Конструювання ПЗ з деталізацією атрибутів і методів класів системи** CUST App

Вдосконалимо структуру моделі ПЗ, завершивши цим попереднє об'єктно-орієнтоване проектування ПЗ та почнемо реалізацію ООП-проєкту системи CUST App на C++, показавши як перетворити побудовані діаграми класів на заголовні файли C++. При конструюванні реального ПЗ і отримання повного набору коду реалізації, модифікуємо ці заголовні файли відповідно з об'єктно-орієнтованою концепцією успадкування.

Видимість. Застосовуємо до елементів наших класів специфікатори доступу. Специфікатори доступу (puЬlic, private) визначають видимість чи доступність, атрибутів та операцій об'єкта для інших об'єктів. Перш ніж почати реалізацію нашого ООП-проекту, необхідно розглянути, які атрибути та операції класів системи повинні бути відкритими (puЬlic), а які закритими (private). На етапі Аналізу нами зазначалось що елементи даних зазвичай мають бути закритими, а методи, що викликаються «клієнтами» класу – відкритими. У той же час, методи, що викликаються лише іншими методами класу як «сервісні функції» , зазвичай робляться закритими.

Для моделювання видимості атрибутів та операцій в UML використовуються маркери видимості: відкрита видимість позначається знаком плюс (+) перед операцією або атрибутом; знак мінус (-) позначає закриту видимість. Рис. 7.1 показує модифіковану діаграму класів CUST App, доповнену маркерами видимості. Тут не включено ніяких параметрів операцій, що абсолютно нормально. Додавання маркерів видимості не впливає на параметри, нами змодельовані раніше діаграмах класів (рис. 5.3-5.6).

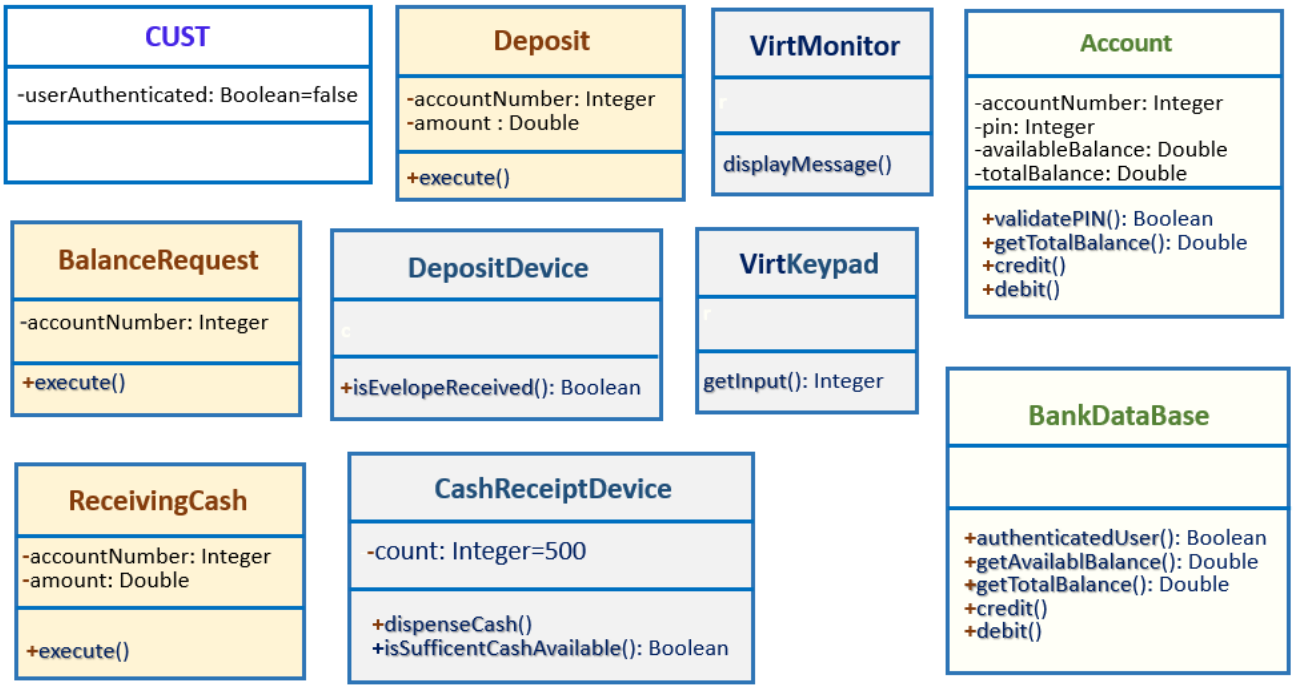


Рис. 7.1. Діаграма класів предметної області CUST App маркерами видимості атрибутів та методів

Навігація. Перш ніж розпочати реалізацію нашого проєкту CUST App на C++, введемо ще одне умовне позначення UML. Діаграма класів на рис. 7.2 детальніше уточнює взаємовідносини між класами у системі CUST App, додаючи до ліній асоціацій стрілки навігації, що показують в якому напрямку може проходити асоціація. Навігація ґрунтуються на кооперaціях, що моделюються в **CD** та **SD**. Реалізуючи систему, спроектовану за допомогою UML, програмні інженери використовують навігацію, щоб швидко визначити, яким об'єктам потрібні посилання або вказівники на інші об'єкти. Напр., навігація, спрямована від класу **CUST** до класу **BankDataBase** показує, що можна пройти від першого до другого, що дозволяє класу **CUST** активувати операції класу **BankDatabase**. Однак, оскільки на рис. 7.2 немає навігаціїі від класу **BankDatabase** до класу **CUST**, клас **BankDataBase** не може звертатися до операцій класу **CUST**. Асоціації з 2 стрілками/без стрілок на діаграмі класів є двонапрямними.

На діаграмах класів, рис. 2.6, 7.2 опущені (інкапсульовані) класи **BalanceRequest** і **Deposit**, щоб не ускладнювати діаграму. Навігації в цих упущених класах подібні і до навігацій класу **ReceivingCash**. Клас **Balancelnqniry** має асоціацію з класом **VirtMonitor** (р.2). Можна пройти по цій асоціації від до класу **BalanceRequest** до класу **VirtMonitor**, але не можна пройти від класу **VirtMonitor** до классу **BalanceRequest**. Отже, якщо б на рис. 7.2 добавити клас **BalanceRequest**, то слід ввести навігацію, направлену від **BalanceRequest** до **VirtMonitor**.

Так само другий відсутній тут клас **Deposit** асоціюється з класами **VirtMonitor**, **VirtKeypad** і **DepositDevice**. Можна пройти від класу **Deposit** до кожного з цих класів, але не навпаки. Необхідно зробити навігації, направлені від **Deposit** з до класів **VirtMonitor**, **VirtKeypad** і **DepositDevice**. Ці додаткові класи змоделюємо пізніше на остаточній діаграмі класів у, після того, як спростимо структуру розроблюваної ООП-системи, скориставшись ООП-концепцією успадкування, ввівши новий абстрактний базовий клас ***Transaction*** (р. 8).

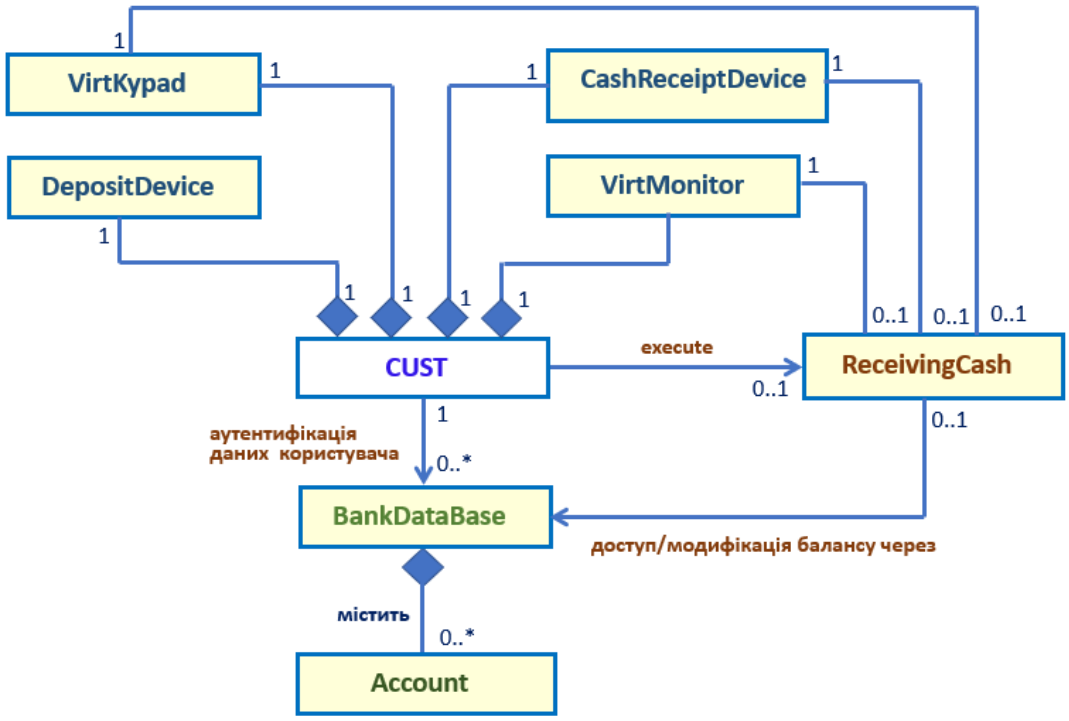


Рис. 7.2. Діаграма ієрархії класів предметної області CUST App зі стрілками навігації для сценарію реалізації трансакції **ReceivingCash** «Отримати кошти»

Р.8. Удосконалення діаграми класів предметної області за рахунок введення успадкування та делеґування (поліморфізму). **Формування стійкої до змін архітетури системи**

Створені нами діаграми ієрархії класів на рис. 2.6-2.8 описують структурні відношення між класами відповідно для кожної окремо взятої з розглядуваних трансакцій «Отримати кошти», «Покласти кошти на депозит», «Перевірити баланс». Вони є цінними для візуалізації розуміння зав'язків між класами при виконанні кожної цих трансакцій. Однак для розуміння цілісності системи, виникає необхідність об'єднати ці три моделі в одну з перспективою її можливої еволюції шляхом внесення змін згідно новим вимогам користувача. Чисто механічне таке об'єднання цих діаграм призведе до значного накопичення і загромадження зв'язків, їх надлишковості, що призведе до значного ускладнення самої діаграми, поганої читабельності, убогого вигляду та відсутності своєрідного художнього смаку та естетичності, якими мав би володіти сучасний програмний інженер (на жаль на це практично мало звертають увагу студенти) та, головне, розуміння її суті (рис.8.1).

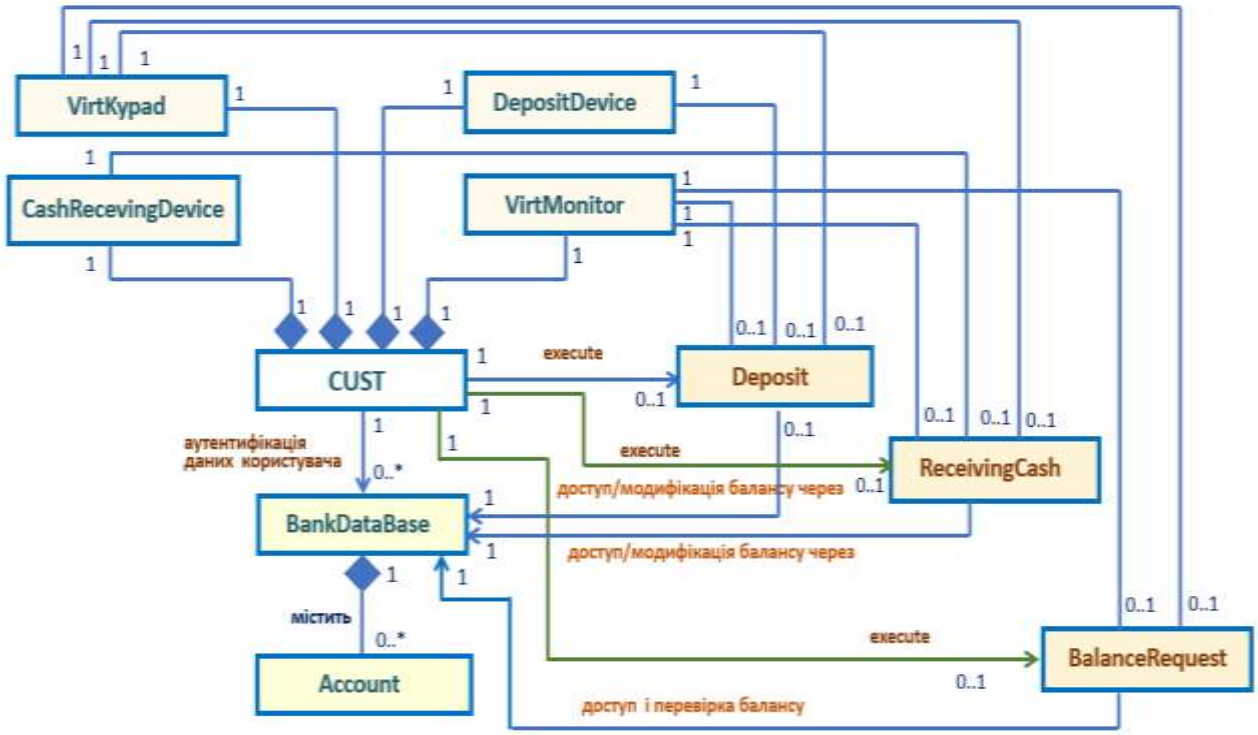


Рис. 8.1. Неправильно побудована діаграма класів предметної області, шляхом механічного об'єднання моделей сценаріїв ВВ, що призводить до перевантаження діаграми, нечитабельності, значного загромадження зв'язків, та їх надлишковості.

В результаті програмний код, побудований за такою моделлю перетвориться на суцільне «спаґетті». Це ще один приклад «тяжкої спадщини», що в кінцевому рахунку може привести до неуспішного проєкту.

Якщо спробувати застосувати ООП-концепцією успадкування, ввівши новий абстрактний базовий клас ***Transaction***, який поліморфно будуть успадковувати класи **ReceivingCash**, **Deposit**, **BalanceInquire**, то матимемо динамічну схему успадкування класів

фінансових трансакцій, до якої можна добавляти нові класи, що описують нові ВВ, що описують нові трансакції, забезпечуючи стійкість структури та архітектури системи (рис. 8.1).

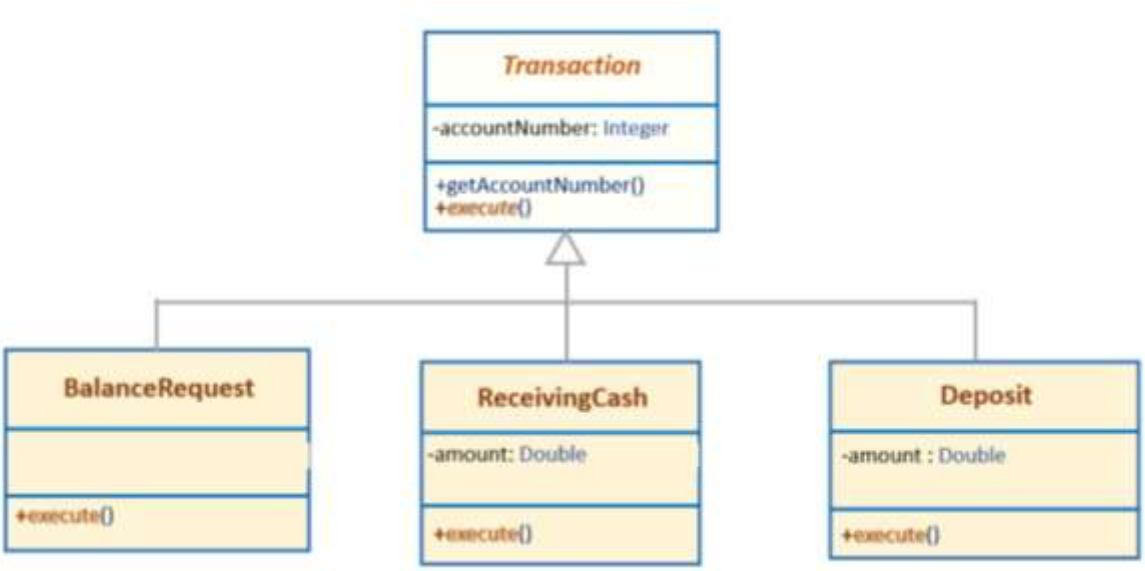


Рис. 8.2. Нове ефективне архітектурне рішення шляхом реалізації делегування повноважень через успадкування фінансових трансакцій в CUS App на основі абстрактного базового класу ***Transaction***

В результаті загальна діаграма ієрархії класів, що цілісно описуватиме всю систему з усіма ВВ, динамічну архітектуру системи, забезпечуючи її стійкість і цілісність. З іншого боку така діаграма класів матиме значно оптимізований і читабельніший та стильний і естетичний вигляд (рис. 8.3).

Архітектура системи є головним артефактом проєктування ПЗ, визначає як структурну так і поведінкову характеристики системи. Вона визначає подальший хід детального проєктування класів, їх об'єднання у підсистеми та компоненти і написання програмного коду. Існують різні вигляди арітектури ПЗ, що описують систему з різних точок зору: загальний вигляд на рівні взаємодіючих підсистем (вигляд з «пташинного польоту»), варіантів використання, класів та інтенрфейсів, компонетів з кодом, включаючи різні бібліотеки, фреймворки, БД та ін. Все це робить систему доступнішою і простішою в розробці, в подальшому розвитку і управлінні, естетичнішою, зручною в експлуатації.

Вибір правильної архітектури за рахунок правильної структурної композиції та реалізації різних механізмів делегування повноважень похідним класам, що реалізують окремі сервіси, у тому числі додавання нових, визначає динамічність і гнучкість системи, стійкість до внесення нових змін і розвитку, що не можуть забезпечити рудиментні статичні схеми, тим більше, процедурний підхід, реалізований на функціях і процедурах, а не на класах і об'єктах.

Реалізована на розширеній діаграмі класів (рис.8.3) архітектурна схема делегування повноважень (рис.8.2) є основним елементом цілої низки стандартних архітектурних моделей (патернів проєктування), що застосовуться до розробки ПЗ різних застосувань (Facade, Abstract Factory, Strategy, Composite, Flyweight, Visitor, Observer, Chaine Responsibility, Command, Decorator та ін., що детально вивчаються в рамках дисципліни «Архітектура та проєктування програмного забезпечення»). В нашому випадку розроблена архітектура орієнтована на типову систему, що надає різні послуги клієнтам (реалізовує різні фінансові сервіси, трансакції) є близькою до патерна Facade. Для систем прийняття рішень можна використовувати архітектурну модель Strategy. Для складніших систем з врахуванням їх

предметних областей та ідей та фантазій розробників можна розробляти і набагато складніші гібридні архітектури, напр., багаторівневу Abstract Factory, Idea Strategy Factory, Flyweight разом з Composite та ін.

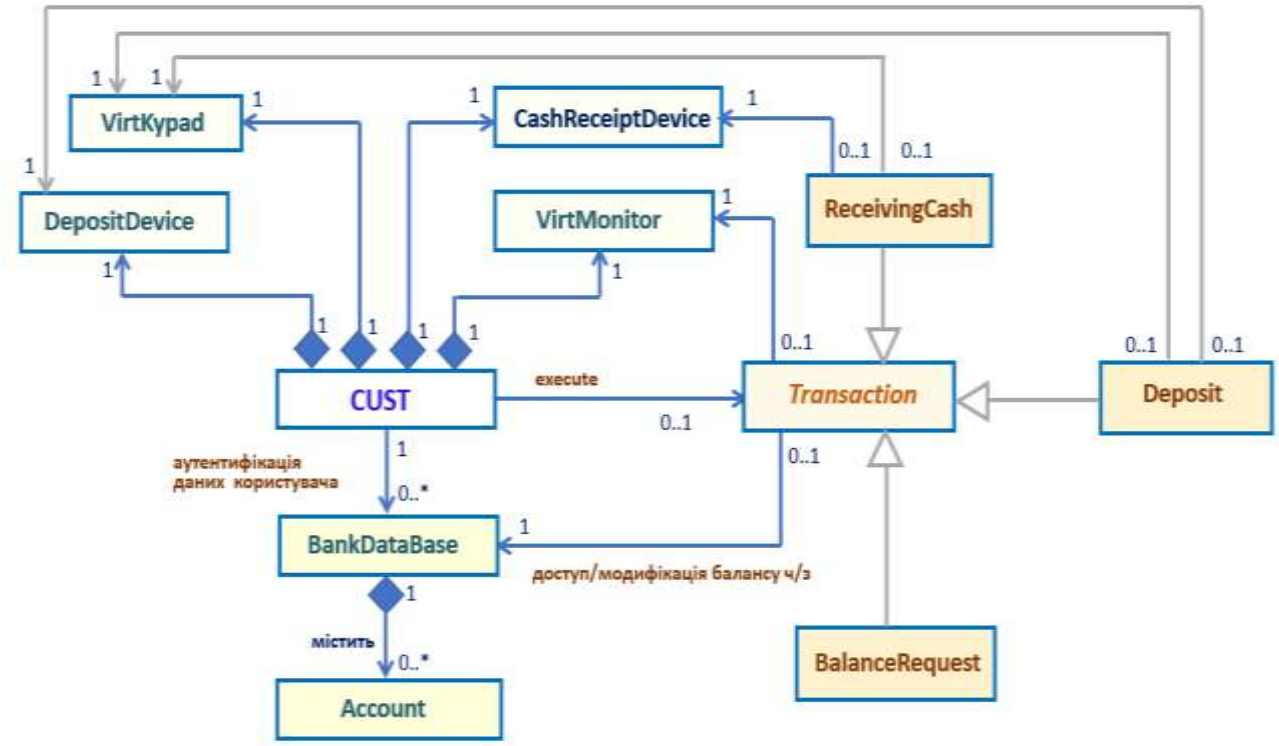


Рис. 8.3. Оптимізована та естетично оформлена діаграма ієрархії класів цілісної системи з усіма ВВ на основі успадкування та делеґування повноважень

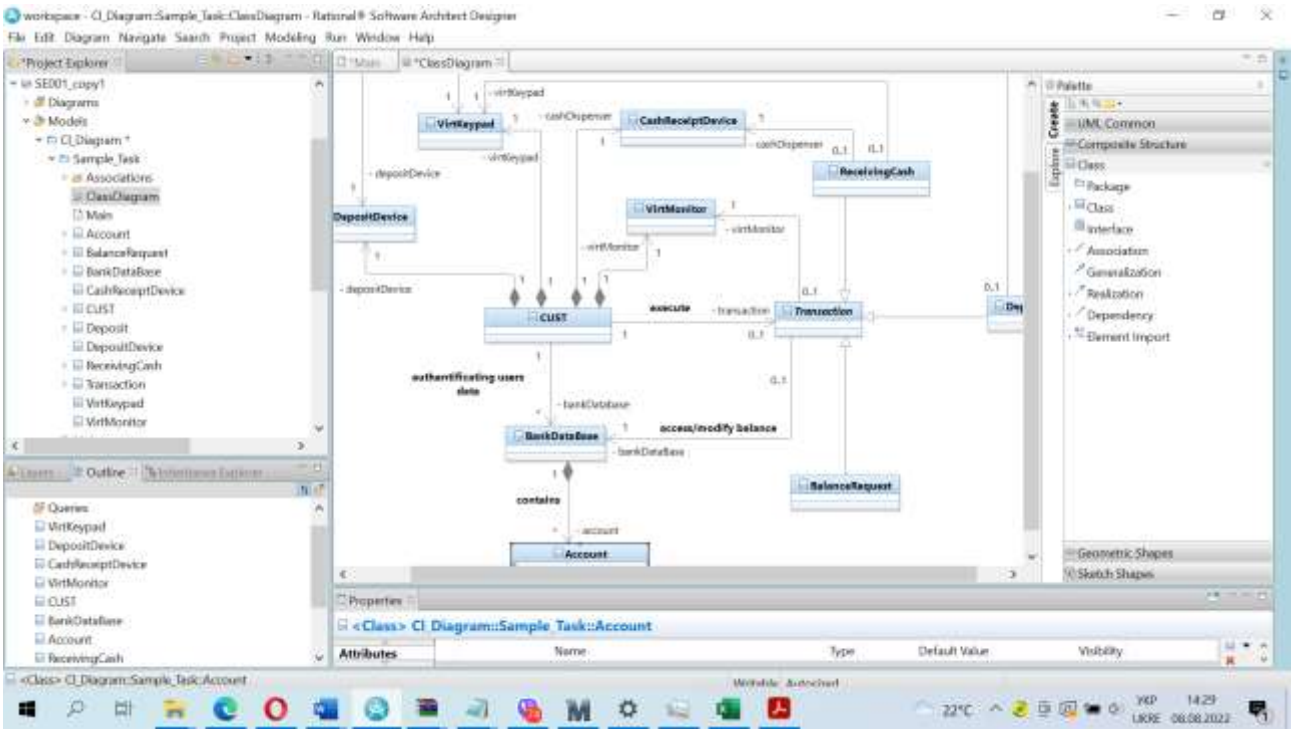


Рис. 8.3.1 Аналогічна (рис.8.3) діаграма ієрархії класів цілісної системи CUST App з усіма ВВ на основі успадкування та делеґування, що є динамічною архітектурною моделлю (АМ) системи (АМ типу **Facade**), побудована з використанням інструментальних засобів розробки **IBM Rational Software Architect**

Не слід також перевантажувати навіть добре збалансовану та оптимізовану діаграму класів деталями, які є малосуттєвими на даному етапі проєктування. Поєктування архітектури не вимагає деального представлення на діаграмі класів атрибутів та операцій, як напр. на рис.8.3.2.

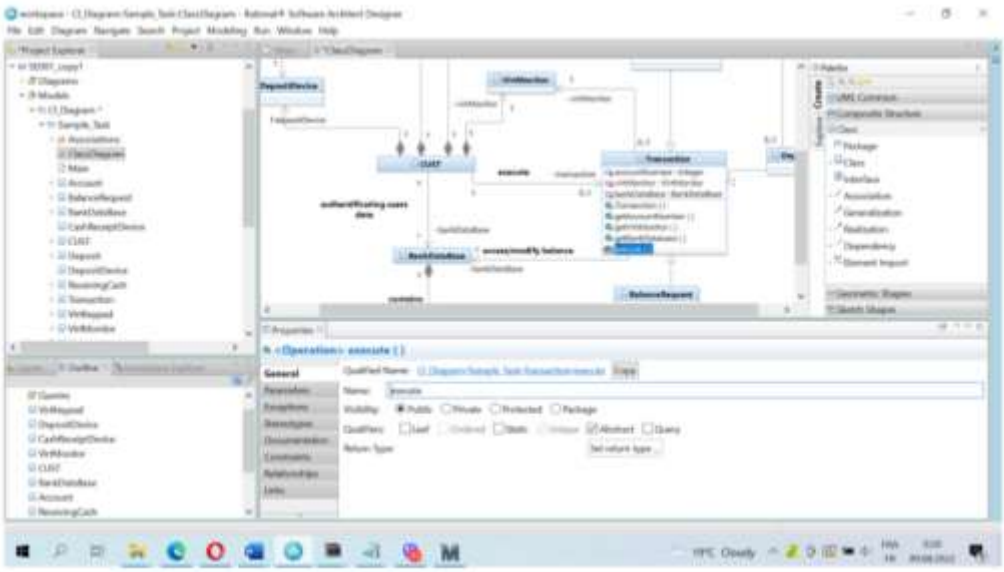


Рис.8.3.2. Фраґмент побудованої в IBM RSA діаграми класів з надлишково деталізованими атрибутами і операціями (клас ***Tranaction***)

Звичайно, атрибути та операції класів необхідно детально параметризовувати, але на діаграмі ієрархії класів важливішими є зв’язки, а не атрибути та операції, які слід інкапсусльовувати, користуючись механізмами фільтрації IBM RSA. При необхідності їх завжди можна побачити в таблиці **Propeties** у нижній частині робочого області меню Class Diagrame (рис. 8.3.3.)

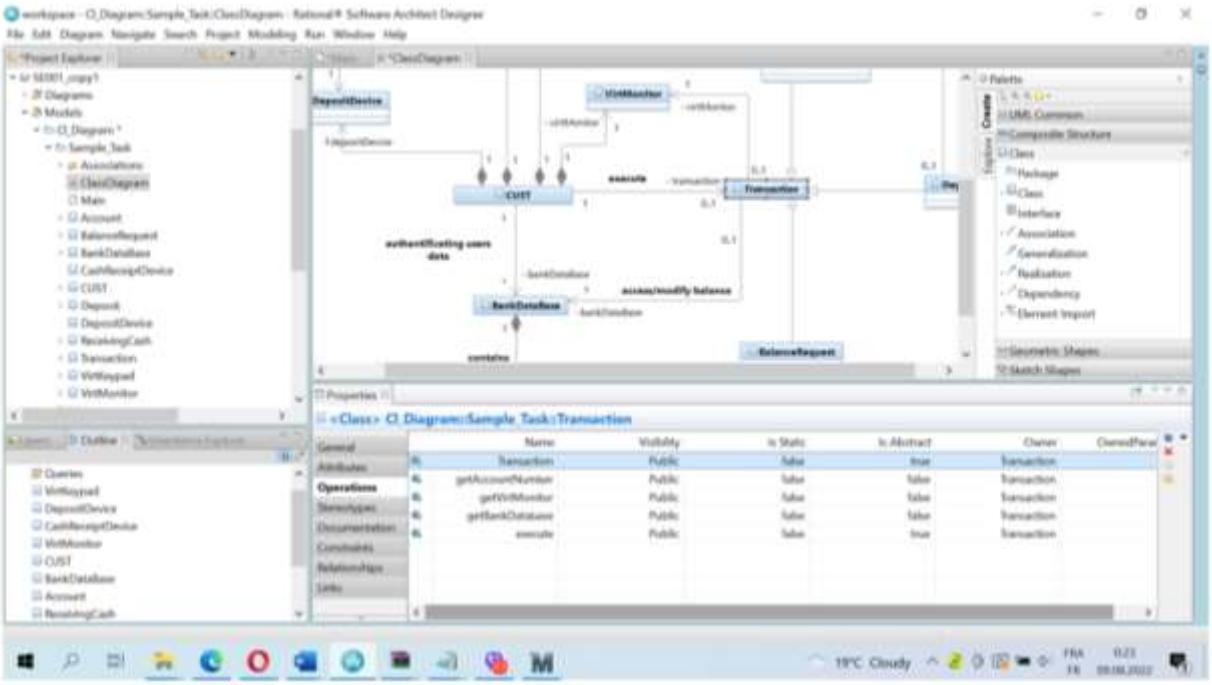


Рис. 8.3.3. Цей же фраґмент діаграми класів в IBM RSA з інкапсульованими атрибутами і операціями. Для виділеного класу (***Tranaction***) операції чи атрибути можна побачити у таблиці **Propeties**

Не використання вказаних підходів проектування ПЗ із застосуванням таких сучасних інструментальних засобів як IBM Rational Software Architect, приводить до громіздких і значно перевантажених моделей, а відтак до складного заплутаного коду, що тяжіє примітивізмом, в якому важко розібратись, особливо новому члену команди, що «розгрібає важку спадщину, залишену команді працівником, що звільнився» та ін.

Р.9. Детальне проєктування класів та реалізація програмного коду з допомогою інструментальних засобів IBM Rational Software Architect

Для ґенерації програмного коду згідно розробленої архітектури системи та параметризації усіх класів згідно діаграми класів (згідно рис. 8.3, 8.3.1) використовуємо робочі інструменти Меню IBM RSA для виконання трансформацій (рис. 9.1). Дана версія IBM RSA (10.07) включає такі види трансформації:

UMLмодель – C++ код,
UMLмодель – C# код,
UMLмодель – Java код ;
зворотні трансформації.

Вибираємо із списку Меню трансформацію UMLмодель – C++код і виконуємо послідовність супровідних команд що вибору проекту-джерела і проекту-цілі. Фіналізувавши усі операції, у проекті-цілі отримуємо список h-файлів зі зґенерованим C++ кодом для описаних в моделі класів, що визначають архітектуру системи.

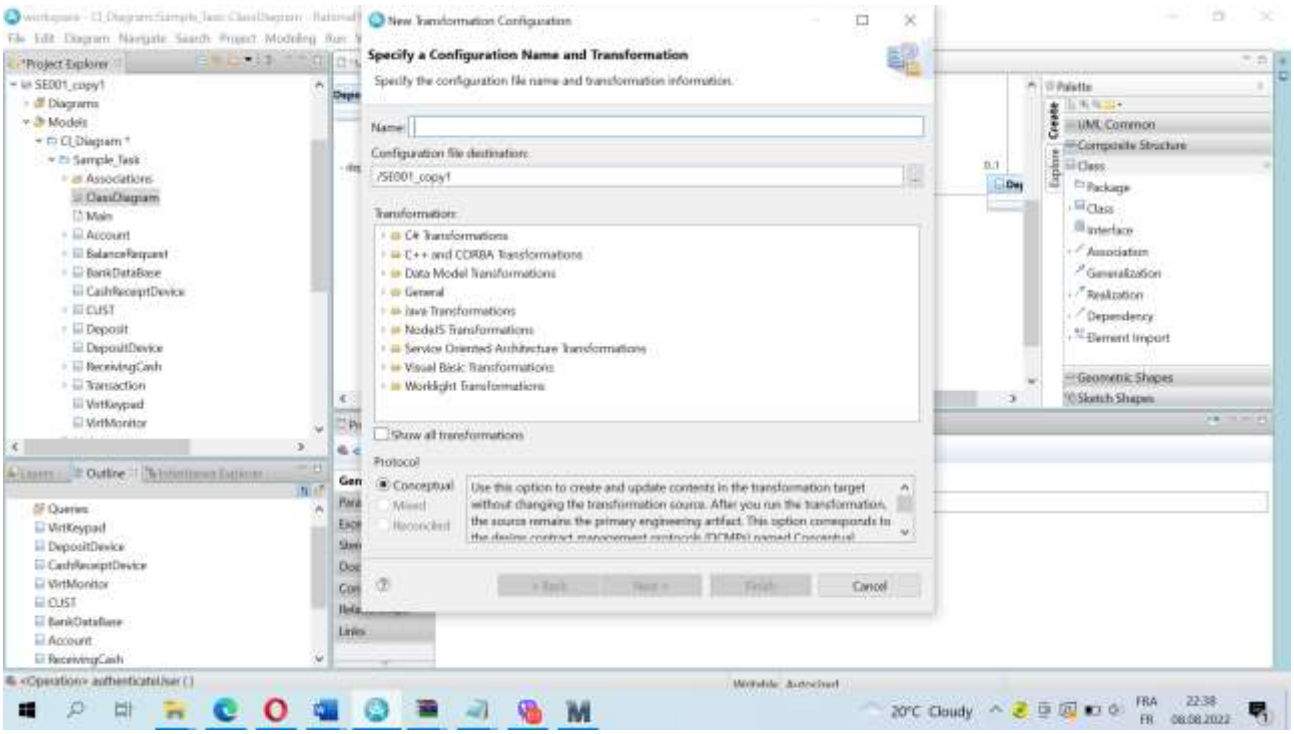


Рис. 9.1. Меню IBM RSA для виконання транформацій: UMLмодель – C++ код, UMLмодель – C# код, UMLмодель – Java код ; зворотні трансформації

Вказаний список .h-файлів зі зґенерованим C++ кодом для усіх описаних в моделі класів можна побачити у лівому верхньому вікні Меню , що появилось після успішного завершення процесу трансформації і ґенерації коду (рис. 9.2). Виділивши окремо взятий класі зі списку h-файлів, наприклад, клас **ReceipDevice**, у лівому вікні Меню можна переглянути і проаналізувати програмний код цього класу. Усі h-файли класів (зі зґенерованим C++ кодом) находяться у директорії створеного при ґенерації проєкту-цілі.

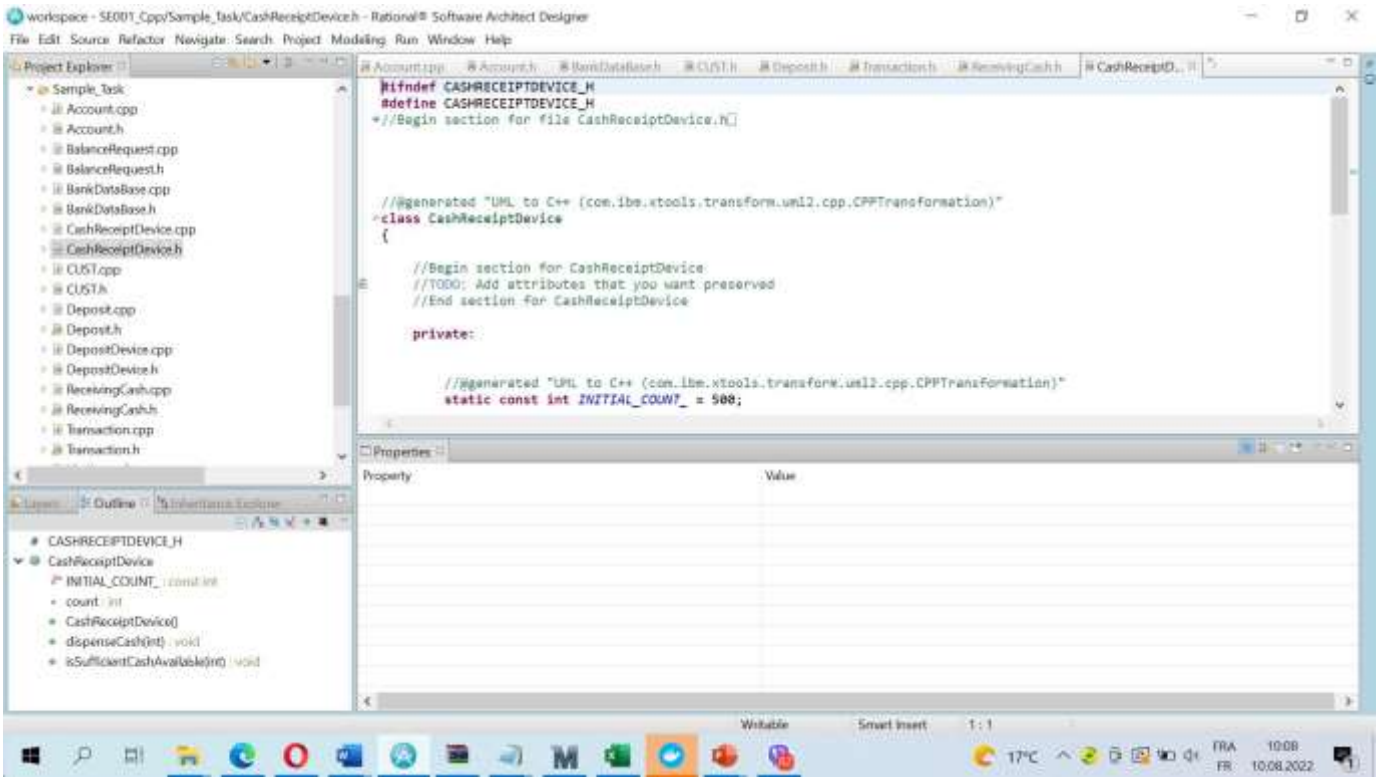


Рис.9.2. Меню IBM RSA зі зґенерованим C++ кодом згідно розробленої архітектурної моделі (рис.8.3): верхнє підменю зліва: список h-файлів класів із зґенерованим кодом, верхнє підменю справа – C++ код для виділеного зі списку класу (клас **ReceipDevice**)

Нижче у Додатку 1 подані тексти програмного C++ коду усіх зґенерованих класів системи.

Додаток 1

Тексти програмного C++ коду усіх зґенерованих класів системи
(з розширеними коментарями)

Fig. G.1. **CUST** class definition

```
// CUST.h
// CUST class definition. Represents an automated teller machine
#ifndef CUST_H
#define CUST_H

#include "VirtMonitor.h"              // VirtMonitor class definition
#include "VirtKeypad.h"               // VirtKeypad class definition
#include "ReceivingCash.h"            // ReceivingCash class definition
#include "DepositDevice.h"            // DepositDevice class definition
#include "BankDatabase.h"             // BankDatabase class definition


class Transaction;          // forward declaration of class Transaction

class CUST
 {
   private:
      bool userAuthenticated;  // whether user is authenticated чи пройшов автентифікацію користувач
      int currentAccountNumber;             // current user's account number
      VirtMonitor  virtMonitor;                    // CUST's VirtMonitor
      VirtKeypad  virtKeypad;                   // CUST's VirtKeypad
      ReceivingCash receivingCash;        // CUST's cash dispenser
      DepositDevice  depositDevice;          // CUST's deposit slot
      BankDatabase  bankDataBase;       // account information database

      // private utility mehods
      void authenticateUser();            // attempts to authenticate user
      void performTransactions();         // performs transactions
      int    displayMainMenu() const;   // displays main menu

     // return object of specified Transaction derived class
     Transaction* createTransaction( int);

   public:
      CUST();                  // constructor initializes atributs
      void run();              // start the CUST
 }; // end class CUST

#endif // CUST_H
```

Fig. G.2. **VirtMonitor** class definition

```
1  // VirtMonitor.h
2  // VirtMonitor class definition. Represents the VirtMonitor of the CUST
3  #ifndef VIRTMONITOR_H
4  #define VIRTMONITOR_H
5
6  #include <string>
7  using std::string;
8
9  class VirtMonitor
10  {

11     public:
12        void displayMessage( string ) const;            // output a message
13        void displayMessageLine( string ) const;      // output message with newline
14        void displayDollarAmount( double ) const;  // output a dollar amount
15  }; // end class VirtMonitor
16
17 #endif // VIRTMONITOR_H
```

Fig. E.5. **VirtKeypad** class definition

```
1 // VirtKeypad .h
2 // Визначення  класу VirtKeypad (кнопкова панель CUST)
3 # ifndef КЕУPAD_H
4 #define КЕУPAD_H
5
6 class VirtKeypad
7    {
8       puBLic :
9          int getinput() const ; // повернути data введене користувачем
10   }; // end  class  VirtKeypad
11
12 #endif // Keypad.h
```

Fig. G.13. **BankDatabase** class definition

```
1 // BankDatabase.h
2 // BankDatabase class definition. Represents the bank's database
3 #ifndef BANK_DATABASE_H
4 #define BANK_DATABASE_H
5
6 #include <vector> // class uses vector STL to store Account objects
7 using std::vector;
8
9 #include "Account.h"   // Account class definition
10

11 class BankDatabase
12  {
13     private:
14        vector < Account > accounts;         // vector STL of the bank's Accounts
15
16        // private utility method
77        Account* getAccount(int );          // get pointer to Account object
```

```
18    public:
19       BankDatabase(); // constructor initializes accounts
20        //determine whether account number and
21       // PIN match those of  Account:чи свівпад. PIN I number з обліковим
22        bool authenticateUser(int, int);     // returns true if Account authentic
23
24        double getAvailableBalance(int); // get an available balance of Cache
25        double getTotalBalance(int);       // get an Account's total balance of Deposite
26        void credit(int, double);             // add amount to Account balance
27        void debit(int, double);             // subtract amount from Account balance
28    }; // end class BankDatabase
29
30 #endif  // BANK_DATABASE_H
```

Fig. E.11. **Account** class definition

```
1 // Account.h
2 // Визначення класу Account. (банківський  рахунок)
3 #ifndef ACCOUNT_H
4 #define ACCOUNT_H
5
6 class Account
7   {
8     private :
9         int accountNumber ;          // номер рахунку
10        int pin ;                          // PIN для авторизації
11        double availableBalance;    // сума готівкових коштів, доступна для зняття
12       double totalBalance ;          // загальна сума  депозиту

13    puBLic:
14        Account(int, int, douBLe, double) ;  // встановл. атрибути (№ рах, PIN, availBal, totBal)
15       bool validatePIN ( int ) const;              // чи введений PIN правильний?
16       douBLe  getAvailableBalance( ) const;   // повертає наявний баланс коштів
17        douBLe getTotalBalance( ) const;         // повертає загальний баланс депозиту
18       void credit( double );                          // додає  суму до балансу рахунку
19       void debit( double) ;                          // віднімає суму з балансу рахунку
20       int getAccountNumber( ) const;          // повертає номер  рахунку
21   } ; // end  class  Account
22
23 #endif // ACCOUNT_H
```

Fig. E.15. **Transaction** abstract class (АБК) definition

```
1 // Transaction.h
2 // Визначення абстрактного базового класу Transaction
3 #ifndef TRANSACTION_H
4 #define TRANSACTION_H

5  class BankDatabase; //випереджаюче оголошення класу BankDatabase
6
7  class Transaction
8   {
9     private :
10         int accountNumber ; // номер  банківського рахунку користувача
11         VirtMonitor& virtMonitor ;   / посилан.  на об'єкт   Монітор к, що є скл. част.  CUST
12         BankDatabase &bankDataBase; // посилання на об'єкт класу БД  рахунків користув
```

```
13    puBLic :
14       // к-ор ініціалізує  спільні (загальні) атрибути усіх транзакцій
15       Transaction (int, VirtMonitor &, BankDataBase &) ;
16       virtual ~Transaction( )   // віртуальнblй  деструктор
17                   { }
18       int getAccountNumber( ) const ; // повернути номер рахунку
19       VirtMonitor& getVirtMonitor( ) const;     // повернути посил. на об'єкт Монітор
20       BankDataBase & getBankDataBase() const; //поверн посил. на об'єкт БД рахунків
21
22        // чисто віртуальний метод для  виконання  транзакціа
23        virtual void execute( ) = 0 ; // замінюється в похідних класах
24
25    } ; // end  class Transaction
26#endif    // TRANSACTION_H
```

Fig. G.19. **ReceivingCash** class definition

```
1 // ReceivingCash.h
2 // ReceivingCash class definition. Represents a ReceivingCash transaction
3 #ifndef RECEIVINGCASH_H
4 #define RECEIVINGCASH_H
5  #include "Transaction.h"     // Transaction class definition
6
7 class VirtKeypad;            // forward declaration of class VirtKeypad
8 class ReceivingCash; // forward declaration of class ReceivingCash
9
10 class ReceivingCash : public Transaction
11   {
12     private:
13       int amount;               // amount to withdraw
14       VirtKeypad &VirtKeypad;       // reference to CUST's VirtKeypad
15       ReceivingCash &ReceivingCash;       // reference to CUST's cash dispenser
16       int displayMenuOfAmounts() const;  // display the ReceivingCash menu

17     public:
18       ReceivingCash( int, VirtMonitor &, BankDatabase &, VirtKeypad &, ReceivingCash &);
19        virtual void execute();  // perform the transaction
20   }; // end class ReceivingCash
21
22 #endif // RECEIVINGCASH_H
```

Fig. E.7. **ReceivingCash** class definition

```
1 // ReceivingCash.h
2 //визначення класу ReceivingCash (ДОК)
3 #ifndef RECEIVINGCASH_H
4 #define RECEIVINGCASH_H
5
6 class ReceivingCash
7   {
8     private:
9        const static int INITIAL_COUNT = 500 ;
10         int count;   //число 20 $-купюр, що залишилось

11     puBLic:
```

```
12          ReceivingCash ( ) ; // конструктор ініціалізує  лічильник купюр
13
14         // імітує видачу вказаної суми готівки
15         void dispenseCash(int);
16
17          // повідомляє, чи можна  видати необх. суму
18          bool isSufficientCashAvailable (int) const;
19       }; // end  class ReceivingCash
20
21 #endif // RECEIVINGCASH_H
```

Fig. E.9. **DepositDevice** class definition

```
1 // DepositDevice.h
2 // Визначення класу DepositDevice (ДВД)
3 #ifndef DEPOSIT_DEVICE_H
4 #define DEPOSIT_DEVICE_H
5
6 class DepositDevice
7    {
8      puBLic:
9          bool isEnvelopeReceived() const ; // повідомляє чи отримано депозит
10  }; // end class DepositDevice
11
12 #endif // DEPOSIT_DEVICE_H
```

Fig. E.17. **BalanceRequest** class definition

```
1 // BalanceInquiry.h
2 // Визначення  класу Balanceinquiry  (довідка про баланс)
3 #ifndef BALANCE INQUIRY H
4 #define BALANCE=INQUIRY=H
5
6 #include "Transaction .h" // визначення класу Transaction
7
8 class BalanceRequest: puBLic Transaction
9    {

10     puBLic:
11          BalanceRequest (int, VirtMonitor &, BankDataBase &) ; // конструктор 2 арґ
12          virtual void execute( ); // виконати транзакцію
13   } ; // end class BalanceRequest
14
15 #endif  // BALANCE_REQUEST_H
```

Fig. E.21. **Deposit** class definition

```
1 // Depisit.h
2 // Визначення класу Deposit (внесение коштів на депозит)
3 # ifndef DEPOSIT_H
4 #define DEPOSIT_H
5
6 # include "Transaction.h" // визначення класу Transaction
7 class VirtKeypad;          // випереджаюче оголошення класу VirtKeypad
8 class DepositDevice;       // випереджаюче оголош. класу DepositDevice
9
10 class Deposit : puBLic Transaction
11  {
12    private:
13       douBLe amount ;                    // сума, що вноситься на депозит
14       VirtKeypad& VirtKeypad ;           // посилання  на  Вірт.кнопк. панель
15       DepositDevice& DepositDevice ;  // посилання на ДВД
16       douBLe promptForDepositAmount() const;  //здіснити внесення суми

17    public:
18       Deposit(int , VirtMonitor&, BankDataBase&, VirtKeypad&,
                 DepositDevice&) ;    // к-р 5 арґ
19       virtual void execute( ) ;          // виконати транзакцію
20  }; // end  class Deposit
21
22 #endif // DEPOSIT_H
```