1. ТЕОРЕТИЧНІ ВІДОМОСТІ ДО ВИКОНАННЯ КУРСОВОГО ПРОЕКТУ

Процес розробки системи з використанням об'єктно-орієнтованої технології логічно можна поділити на три етапи: об'єктно-орієнтований аналіз (OOA), об'єктно-орієнтоване проектування (OOD) та об'єктно-орієнтоване програмування (OOP). На рис.1.1 схематично зображено співвідношення цих етапів із загальноприйнятими фазами життєвого циклу будь-якої системи, що створена людиною.

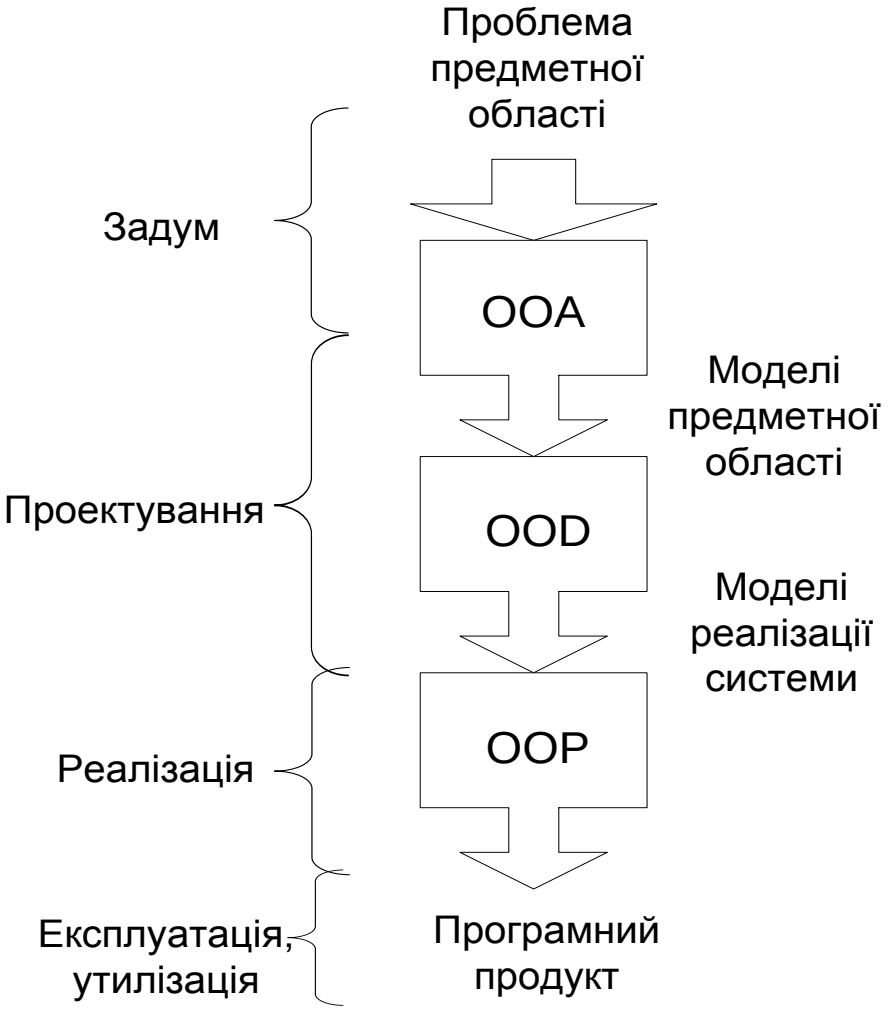

Рис.1.1. Етапи об'єктного підходу й життєвий цикл системи

Як видно з наведеної схеми, уточнення цілей та шляхів вирішення поставленої задачі виконується поступово від побудови моделей самої проблеми до моделей програмної системи, яка буде вирішувати цю проблему. На основі результатів OOA формуються моделі, які складають базу для об'єктно-орієнтованого проектування. В процесі OOD, в свою чергу, створюється фундамент для остаточної реалізації системи з використанням методології об'єктно-орієнтованого програмування. Зазначимо, що схема відображає лише логічне поділення процесу, й насправді зв'язки між етапами у часі значно складніші й містять численні зворотні зв'язки.

Далі докладніше розглянемо кожний з означених етапів.

1.1. Об'єктно-орієнтований аналіз

Об'єктно-орієнтований аналіз (object-oriented analysis) – це методологія, за якої вимоги до системи сприймаються з точки зору класів та об'єктів, виявлених у предметній області.

Об'єктно-орієнтований аналіз спрямовано на створення моделей реальної дійсності. Метою аналізу є описання поставленої задачі таким чином, щоб визначити необхідну поведінку майбутньої програмної системи, тобто відобразити, *що* має робити система. Необхідно, щоб описання було повним, послідовним і дозволяло проводити порівняння створеної моделі з дійсними умовами. В процесі об'єктно-орієнтованого аналізу навколишній світ моделюють, ідентифікуючи в ньому об'єкти і класи, які називають *ключовими абстракціями*, що формують словник предметної області.

Наведемо основні визначення ключових понять об'єктного підходу.

Об'єкт – це абстрактна сутність, що має деякий стан, добре означену поведінку і унікальність.

Стан об'єкта – це перелік всіх властивостей цього об'єкта й поточних значень кожної з цих властивостей.

Властивості об'єкта – це притаманні йому або надбані ним характеристики, риси, якості або можливості, що відрізняють цей об'єкт від інших.

Таким чином, при аналізі предметної області необхідно виділити об'єкти зі своїми характерними властивостями так, щоб усі об'єкти мали властивості предметної області й повністю її охоплювали. Однак об'єкти не знаходяться в якомусь статичному стані, вони, як правило, впливають на інші об'єкти або сприймають вплив. Діяльність об'єкта, яка спостерігається та перевіряється зовні, називається його поведінкою.

Поведінка об'єкта – це сукупність дій, процедур, операцій, які виконує сам об'єкт або ініціюють інші об'єкти.

В термінах об'єктного підходу такі дії називаються «передача повідомлення» або «виклик методу».

Метод – це елементарна операція (процедура або функція), що оперує властивостями конкретного об'єкта або іншими об'єктами.

З одного боку, поведінка об'єкта визначається операціями, що виконуються з ним, та його станом. З іншого боку, стан об'єкта є сумарним результатом його поведінки.

Унікальність – це особливість об'єкта, яка відрізняє його від усіх інших об'єктів.

Унікальність не означає відмінність за іменем або за якоюсь окремою властивістю або набором властивостей (хоч кожен об'єкт і мусить мати такі ключові властивості), вона зберігається, навіть коли

змінився стан об’єкта. Натомість поняття «клас» визначає не окрему сутність, позиціоновану у часі та просторі, а лише абстракцію істотного в об’єкті або групі об’єктів.

Клас – це деяка множина об’єктів, які мають загальну структуру властивостей й однаковий набір методів.

Таким чином, аналізуючи предметну область і поставлену в ній задачу, необхідно перш за все описати унікальні абстракції, що називаються об’єктами, а потім виділити групи «схожих» об’єктів, які формують класи. При цьому природною є ситуація, коли клас містить один-єдиний об’єкт.

Для ідентифікації об’єктів і класів можна використовувати так звані підходи, що сформувались у природознавчих науках (наприклад, хімії, біології), а саме: класичну категоризацію, концептуальну кластеризацію або теорію прототипів. Практика об’єктно-орієнтованого програмування дала можливість одержати ще ряд підходів, серед яких – аналіз поведінки, аналіз предметної області, аналіз варіантів, CRC-картки, а також неформальний опис і структурний аналіз.

1.1.1. Класичні підходи

Класична категоризація. Згідно з цим класичним підходом усі речі, яким характерні певні властивості або сукупності властивостей, формують деяку категорію. При цьому наявність саме цих властивостей є необхідною й достатньою умовою, за якою визначається категорія.

Наприклад, з одного боку, «блондин», «брюнет», «шатен» – це категорії людей, бо кожна людина має певний колір волосся й ця ознака достатня для вирішення питання, до якої категорії належить той або інший індивідуум. З іншого боку, «молоді люди» не визначають категорію, якщо спеціально не уточнено критерій, який дозволить чітко відрізняти молодих людей від немолодих.

Класична категоризація прийшла до нас від Платона і Арістотеля. Останній у своїй класифікації рослин і тварин користувався технікою міркувань, що нагадує сучасну дитячу гру «20 запитань», коли необхідно вгадати якесь слово, ставлячи запитання, на які ведучий може відповісти лише «так» або «ні» («Це живе?» «Це вкрито пір’ям?» «Це може літати?» «Це червоне?» та ін.). Такий підхід знайшов послідовників, найбільш відомими серед яких були Фома Аквінський, Декарт, Локк.

> *"Ми можемо іменувати речі згідно з нашими знаннями про їх природу, які ми отримуємо через пізнання їх властивостей та дій "* *Фома Аквінський*

Принципи класичної категоризації відображено в сучасній теорії розвитку дитини. Є думка, що після першого року життя дитина усвідомлює існування об’єктів і потім починає набувати навичок їх класифікації, спочатку користуючись базовими категоріями, такими, як «собаки», «кішки» або «іграшки». Пізніше дитина усвідомлює, що є більш загальні поняття – «тварини», а є більш конкретні – «колі», «доги», «овчарки».

Таким чином, як основа зазначеного класичного підходу використовується спорідненість властивостей об'єктів у ролі критерію їх схожості. Зокрема, об'єкти можна розбивати на множини, що не перетинаються, залежно від наявності або відсутності деякої ознаки. Як показала практика, кращими є такі набори властивостей, елементи яких мало взаємодіють. Цим пояснюється прагнення усіх до таких критеріїв, як розмір, колір, матеріал і форма. Оскільки ці критерії не перетинаються, щодо якого-небудь предмета можна стверджувати, що він сірий, круглий та залізний. Власне кажучи, властивості не обов'язково мають бути вимірними, замість них можна використовувати поведінку, яку ми спостерігаємо.

Труднощі при використанні такого підходу полягають у неможливості іноді виділити ту властивість або властивості, що дадуть змогу одержати чітку та зручну класифікацію. Згадаємо, що сучасній класифікації хімічних елементів слід завдячувати генію Д.І. Менделєєва, який винайшов необхідні ознаки, що дозволили розмістити елементи у формі таблиці та виділити їх у певні групи. Звична для нас класифікація істот у біології теж є результатом відкриття шведського вченого К. Лінея, завдяки якому розрізняти класи, роди та види істот можуть навіть школярі. Багато вчених, що працювали віками над цими проблемами до них, не змогли запропонувати певні властивості для виділення категорій у хімії та біології, що були б досить універсальними й відповідали потребам людства.

Концептуальна кластеризація – більш сучасний варіант класичного підходу. Він виник зі спроб формального подання знань. При такому підході спочатку формують концептуальні описи класів (кластерів об'єктів), а потім сутності предметної області розподіляють відповідно до цих описів. Наприклад, поняття «художня література» – це саме категорія, а не ознака або властивість, оскільки ступінь «художності» не можна виміряти. Але твори, в яких художньо прикрашено дійсність або які відтворюють фантазії автора, звичайно відносять до цієї категорії.

Концептуальна кластеризація пов'язана з теорією нечітких множин, за якою об'єкт може належати кільком категоріям одночасно з різним ступенем ймовірності. Цей факт можна віднести до недоліків підходу, адже іноді неможливо визначити межі кластерів досить однозначно для ідентифікації об'єктів.

Теорія прототипів. Класична категоризація й концептуальна кластеризація – достатньо виразні методи, досить придатні для проектування складних програмних систем. Але, як було зазначено, ці методи не вільні від недоліків й у деяких ситуаціях не мають сенсу. Більш сучасний метод класифікації – теорію прототипів – започатковано у психології при моделюванні поведінки людини у формі ігор. У категорії «гра» немає чітких меж – їх можна розширити й внести нові види ігор за умови, що вони нагадують уже відомі ігри.

Цей підхід називають теорією прототипів, тому що кожен клас визначається одним об'єктом-прототипом, а нові об'єкти можна віднести

до того чи іншого класу за умови, що він має істотну схожість з прототипом. Такий прийом корисний для абстракцій, які не мають ані чітких властивостей, ані чіткого визначення.

Характерним прикладом є так звана «проблема стільців»: як правило перукарське крісло й складний стілець вважають стільцями не тому, що вони відповідають деякому фіксованому набору ознак прототипу, а тому, що мають достатню «сімейну» схожість з прототипом. Незважаючи на те, що немає ніякого загального набору властивостей прототипу, яке підходило б і для перукарського крісла, й для складного стільця, вони обидва є стільцями, оскільки кожен з них окремо схожий на прототипний стілець, хоч і кожен по-своєму. При цьому головними є властивості, що визначаються при взаємодії з об’єктом.

1.1.2. Підходи до класифікації в ООП

Різні вчені знаходять різноманітні джерела класів та об’єктів, що узгоджені з вимогами предметної області. Такі підходи називають класичними, оскільки вони ґрунтуються на класичній категоризації.

Наприклад, для багатьох предметних областей корисними будуть ознаки та кандидати у класи і об’єкти, що наведено в табл.1.1.

Таблиця 1.1

Категорії	Класи
Відчутні предмети	Автомобілі, телеметричні дані, датчики тиску
Ролі	Мати, вчитель, політик
Події	Посадка, переривання, запит
Взаємодія	Борг, зустріч, перетинання

При проектуванні баз даних можна використовувати категорії, що наведено в табл.1.2. При розробці програмних систем можливі категорії, що подано в табл.1.3.

Таблиця 1.2

Категорії	Ознаки
Люди	Людські істоти, що виконують деякі функції
Місця	Області, пов’язані з розташуванням людей або предметів
Предмети	Відчутний матеріальний об’єкт або група об’єктів
Установи	Формально організована сукупність людей, ресурсів, обладнання, яка має визначену мету і існування якої в цілому не залежить від індивідуумів
Концепції	Принципи і ідеї, які самі по собі не відчутні, але призначені для організації діяльності та/або спілкування, або для нагляду за ними
Події	Дещо, що відбувається з чимось у заданий час або послідовно

В об’єктно-орієнтованій технології розробки програм використовують не тільки класичні, але й нові підходи, що розвинулися в процесі еволюції методологій програмування.

Аналіз поведінки. Якщо в класичних підходах увага зосереджена

на відчутних елементах предметної області, то в цьому підході об'єктно-орієнтованого аналізу – на динамічній поведінці як на джерелах об'єктів і класів. Це нагадує концептуальну кластеризацію, згадану вище, тобто формуються класи, базуючись на групах об'єктів, що демонструють схожу поведінку.

Таблиця 1.3

Категорії	Ознаки
Структури	Відношення "ціле-частина" та "загальне-окреме"
Інші системи	Зовнішні системи, з якими взаємодіє програма
Пристрої	Пристрої, з якими взаємодіє програма
Події	Випадки, які необхідно запам'ятати
Ролі, що відіграються	Ролі, які виконують користувачі, що працюють з додатком
Місця	Будівлі, офіси і інші місця, що є значущими для роботи програми
Установчі одиниці	Групи, до яких входять користувачі

Відправною точкою класифікації при цьому може бути поняття відповідальності, що визначає мету об'єкта та його місце в системі.

Відповідальність об'єкта – це сукупність усіх послуг, які він може надавати іншим об'єктам.

При аналізі поведінки вивчають функціонування вихідної системи, виділяючи різні форми поведінки всієї системи або її частин, і намагаються зрозуміти, які частини ініціюють певні дії, а які беруть участь у них. Тоді ініціатори і учасники, що відіграють певні ролі, розпізнаються як об'єкти й стають відповідальними за ці ролі. В окремі класи об'єднують ті об'єкти, які мають схожі відповідальності, й будують ієрархію класів, де кожен підклас, виконуючи обов'язки надкласу, надає свої додаткові послуги.

Аналіз предметної області. Цей підхід засновано на виділенні об'єктів, операцій і зв'язків, які експерти цієї області вважають найбільш суттєвими. При аналізі області можна зазначити такі етапи:

- побудова каркасної моделі предметної області під час консультації з експертами в цій області;
- вивчення існуючих в предметній області систем і подання результатів у певному стандартному вигляді;
- визначення за участю експертів схожих і відмінних рис систем;
- уточнення загальної моделі щодо пристосування її до потреб конкретної системи.

Аналіз області можна вести вертикально – відносно аналогічних програм або горизонтально – відносно аналогічних частин однієї програми. Наприклад, починаючи проектувати систему обліку, треба розглянути вже існуючі подібні системи, щоб зрозуміти, які ключові абстракції та механізми, що використовуються в них, будуть корисні, а які – ні. Аналогічно система бухгалтерського обліку має формувати різні

види звітів. Якщо вважати звіти деякою предметною областю, її аналіз може допомогти розробникові виявити ключові абстракції та механізми, які обслуговують усі види звітів. Отримані таким чином класи і об'єкти являють собою множину ключових абстракцій і механізмів, відібраних з урахуванням мети початкової задачі – створення системи звітів, тому кінцевий проект буде простішим.

Аналіз варіантів. Класичний підхід, поведінковий підхід або вивчення предметної області, які розглянуто вище, залежать від індивідуальних можливостей і досвіду аналітика. Для більшості реальних проектів одночасне використання всіх трьох підходів неможливе, тому що процес аналізу стає недетермінованим і непередбачуваним. Аналіз варіантів – це підхід, який можна успішно поєднувати з першими трьома, роблячи їх більш упорядкованими. Ключовим поняттям при цьому є варіант.

> ***Варіант*** – це окремий приклад або зразок використання системи, сценарій, який починається з того, що користувач системи ініціює операцію або послідовність взаємозв'язаних подій.

Цей вид аналізу можна починати одночасно з аналізом вимог. У цей момент користувачі, експерти та розробники переглядають сценарії, найбільш значущі для роботи системи, поки що не заглиблюючись у деталі реалізації. Потім вони ретельно опрацьовують сценарії, розкладаючи їх за кадрами, як це роблять робітники телебачення або кінематографісти. Вони встановлюють, які об'єкти беруть участь у сценарії, обов'язки кожного об'єкта і як вони взаємодіють у термінах виконання операцій. Таким чином, група розробників має чітко розподілити області впливу абстракцій. Далі набір сценаріїв розширюється, щоб урахувати виключні ситуації та вторинну поведінку – так звані периферійні аспекти. В результаті з'являються нові або уточнюються існуючі абстракції.

CRC-картки. Абревіатура CRC означає Class-Responsibilities-Collaborators, тобто «клас-відповідальність-учасники». Це простий і ефективний спосіб аналізу сценаріїв. Картки CRC вперше було запропоновано для навчання об'єктно-орієнтованому програмуванню студентів університетів, але такі картки виявились гарним інструментом для мозкових атак і спілкування розробників між собою.

CRC-картки – це звичайні паперові картки розміром 8x13 або 13x18 см, бажано розлініяні й різнокольорові. Кожну картку поділяють на три графи (рис. 1.2.): у верхній пишеться назва класу, нижче в

Рис. 1.2. CRC-картка

лівій – за що відповідає, у правій – з ким взаємодіє. Аналізуючи предметну область, заповнюють карточку на кожний помічений клас і дописують до неї нові пункти. При цьому ознакою отримання коректного результату аналізу є рівномірне заповнення нижніх граф карток всіх класів. Якщо в певному класі якась графа надмірно переповнена, то необхідно або виділити надлишок відповідальності у новий клас, або розподілити відповідальність одного великого класу на декілька більш детальних, або, можливо, передати частину обов'язків іншому, вже існуючому класу.

Зручність такого аналізу полягає в тому, що картки можна розкладати так, щоб уявити форми співробітництва об'єктів. З точки зору динаміки сценарію їх розміщення може показати потік повідомлень між об'єктами, з точки зору статики вони являють собою ієрархію класів.

Неформальний опис. Радикальна альтернатива класичному аналізу була запропонована у дуже простому методі Аббота. Згідно з цим методом потрібно описати задачу або її частину природною мовою, а потім підкреслити іменники та дієслова. Іменники – це кандидати на роль класів, дієслова – на імена операцій. Такий метод завдяки однозначному алгоритму легко автоматизувати, що й було зроблено у Токійському технологічному інституті. Підхід Аббота є корисним, тому що він досить простий і змушує розробника займатися словником предметної області. Проте він досить відносний і не підходить для вирішення складних проблем. До того ж людська мова – дуже неточний спосіб відображення, тому список об'єктів та операцій залежить від уміння розробника висловлювати свої думки. Крім того, для багатьох іменників можна знайти дієслівну форму й навпаки.

Структурний аналіз. У другій альтернативі класичній техніці об'єктно-орієнтованого аналізу використовується структурний аналіз як основа для об'єктно-орієнтованого проектування. Такий підхід привабливий тим, що багато аналітиків застосовують його, і існує велика кількість програмних CASE-засобів, що підтримують автоматизацію цих методів. Після проведення структурного аналізу складається модель системи, що описана діаграмами потоків даних та іншими продуктами структурного аналізу. Ці діаграми відображають формальну модель проблеми, виходячи з якої можна трьома різними способами перейти до визначення обміркованих класів та об'єктів.

Перші два способи базуються на окремих діаграмах потоків даних. Відповідно до них кандидатами в об'єкти є:

- зовнішні сутності;
- сховища даних;
- сховища керуючих сутностей;
- керуючі перетворення,

а кандидатами в класи –

- потоки даних;
- потоки управління.

Перетворення даних можна розглядати як операції з уже існуючими

об’єктами або як поведінку деякого об’єкта, який створено для виконання потрібного перетворення.

Існує ще один метод – аналіз абстракцій, що базується на ідентифікації основних сутностей, які за своєю природою аналогічні основним перетворенням у структурному проектуванні.

> *“У структурному аналізі вхідні й вихідні дані вивчаються доти, поки не досягнуть вищого рівня абстракції. Процес перетворення вхідних даних у вихідні є основним перетворенням”.* *Е.Зайдевіц і М.Старк*

При виконанні абстрактного аналізу вивчають основне перетворення для того, щоб визначити, які процеси й стани являють собою найкращу абстрактну модель системи. Після визначення основної сутності в діаграмі потоків даних переходять до вивчення всієї інфраструктури, вишукуючи вхідні й вихідні потоки даних із центру, групуючи зустрічні процеси й стани. Слід зазначити, що для практичного використання аналіз абстракцій досить складний.

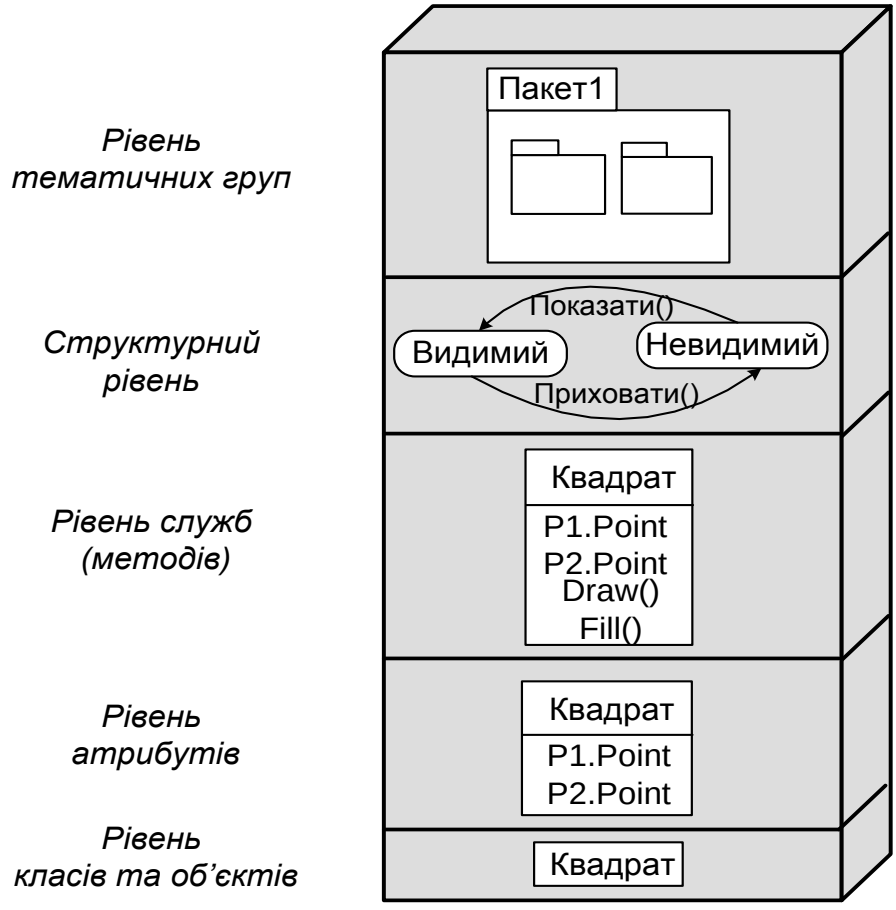

Рис.1.3. Рівні подання моделі в об’єктно-орієнтованому аналізі

Незалежно від того, який із зазначених вище методів об’єктно-орієнтованого аналізу використовується, отримана модель предметної області мусить мати п’ять рівнів відображення, а саме:

- рівень класів та об’єктів – безпосередньо опис об’єктів і класів з

визначенням їх меж;

- рівень атрибутів – опис властивостей екземплярів класу та вимог до них, визначених предметною областю;
- рівень служб – опис методів і зв'язків об'єктів шляхом обміну повідомленнями;
- структурний рівень – опис відношення класів, бажано, щоб це було наслідування і агрегація;
- рівень тематичних груп – опис підсистем, тобто відносно відокремлених частин предметної області.

Зазначимо, що верхній рівень доцільний лише для великих складних систем або предметних областей, які природно складаються з підсистем, блоків або інших структурних одиниць.

1.2. Об'єктно-орієнтоване проектування

> ***Об'єктно-орієнтоване проектування (object-oriented design)*** – це методологія проектування, що поєднує процес об'єктної декомпозиції й засоби подання логічної, фізичної, статичної та динамічної моделей системи, що проектується.

Таким чином, на цьому етапі з використанням об'єктно-орієнтованої декомпозиції логічну структуру системи відображають абстракціями у вигляді класів та об'єктів. Незважаючи на те, що етапи аналізу й проектування можуть частково перетинатись і проводитись паралельно в межах типового проекту, їх слід розглядати як різні дії. Аналіз, як було зазначено вище, визначає, *що* система має робити, а проектування – *як* реалізувати ці вимоги. OOA проводиться згідно з гіпотезою, що існує «ідеальна» технологія, а OOD – з усвідомленням того, що систему буде реалізовано на конкретному апаратному забезпеченні, операційній системі й мові програмування. Якщо модель OOA потребує здебільшого творчого підходу, то модель OOD обмежена вимогами до нотацій, стратегії і якості.

1.2.1. Структура моделі OOD

Модель майбутньої системи можна отримати як розширення моделі предметної області (рис.1.4). Вона містить такі ж п'ять рівнів відображення, але доповнюється чотирма компонентами:

компонент області використання – об'єкти й класи, що виконують істотні функції предметної області. Вихідною версією можуть бути об'єкти й класи OOA, які уточнюються з урахуванням особливостей реалізації;

компонент взаємодії з людиною – технологія інтерфейсу, яку буде використано в програмній системі. Деталі інтерфейсу завжди відокремлюються від основних функцій системи;

компонент управління задачами – елементи операційної системи, які забезпечують функціонування системи в цілому;

компонент управління даними – об'єкти й класи, що необхідні для взаємодії з базою даних.

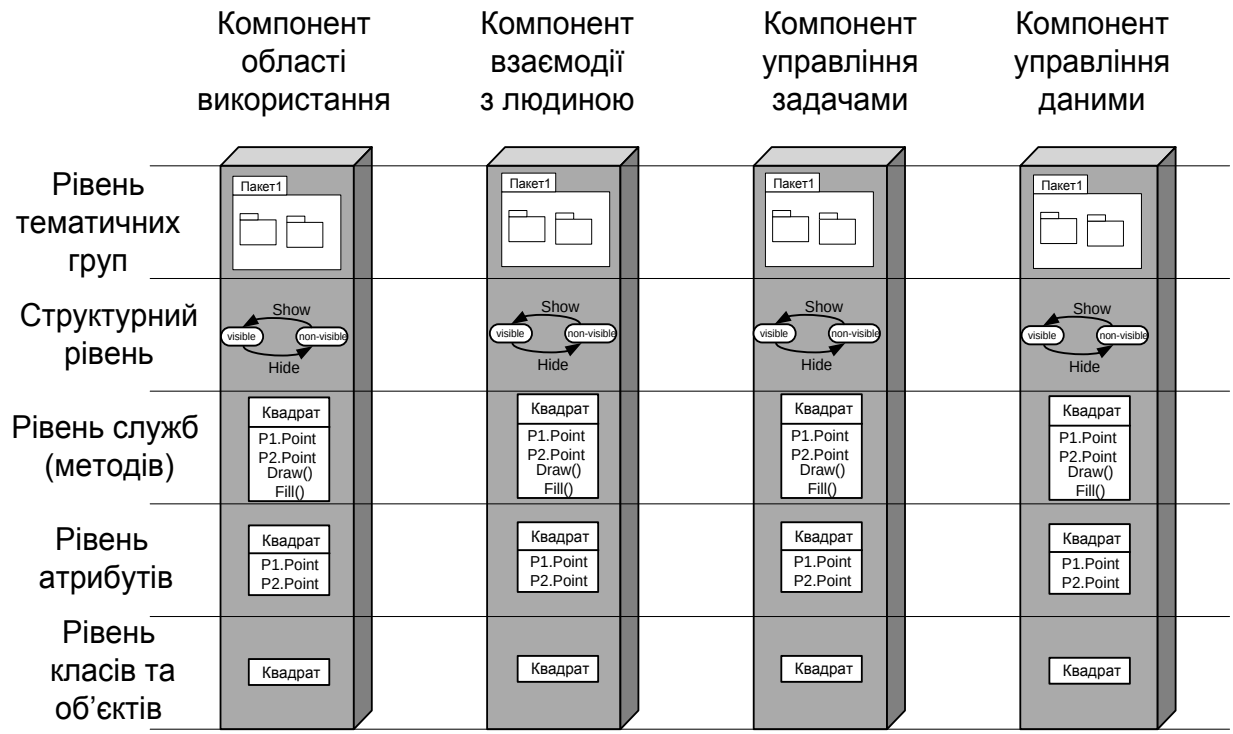

Рис.1.4. Рівні подання моделі в OOD

Як випливає з визначення OOD, при проектуванні корисно розглядати взаємодію класів та об'єктів в двох вимірах, що необхідні для визначення структури й поведінки об'єктної системи:

- логічному й фізичному,
- статичному й динамічному.

Інструментальними засобами подання моделі OOD є діаграми.

Діаграма – це графічне зображення деякого ракурсу (зрізу) моделі системи.

За кожним виміром будують декілька діаграм, які мають вигляд моделей системи в різних ракурсах (рис.1.5). Таким чином, модель системи – це інформаційне відображення класів, зв'язків та ін., а діаграма – це подання однієї з проекцій моделі.

Логічна модель – це опис переліку й змісту ключових абстракцій і механізмів, що формують предметну область або визначають архітектуру системи.

Фізична модель – це опис апаратно-програмної платформи для реалізації системи.

Основні питання, що підлягають опрацюванню в процесі OOD, і відповідні типи діаграм показано на рис.1.6. Ці типи діаграм здебільшого відображають статичну модель системи, проте практично у всіх системах відбуваються події: об'єкти породжуються та руйнуються, пересилають у певному порядку один одному повідомлення, зовнішні події ініціюють операції об'єктів. Тому в OOD існують додаткові діаграми, що

відображають динамічну семантику, а саме: діаграми станів, активності, взаємодії та співробітництва.

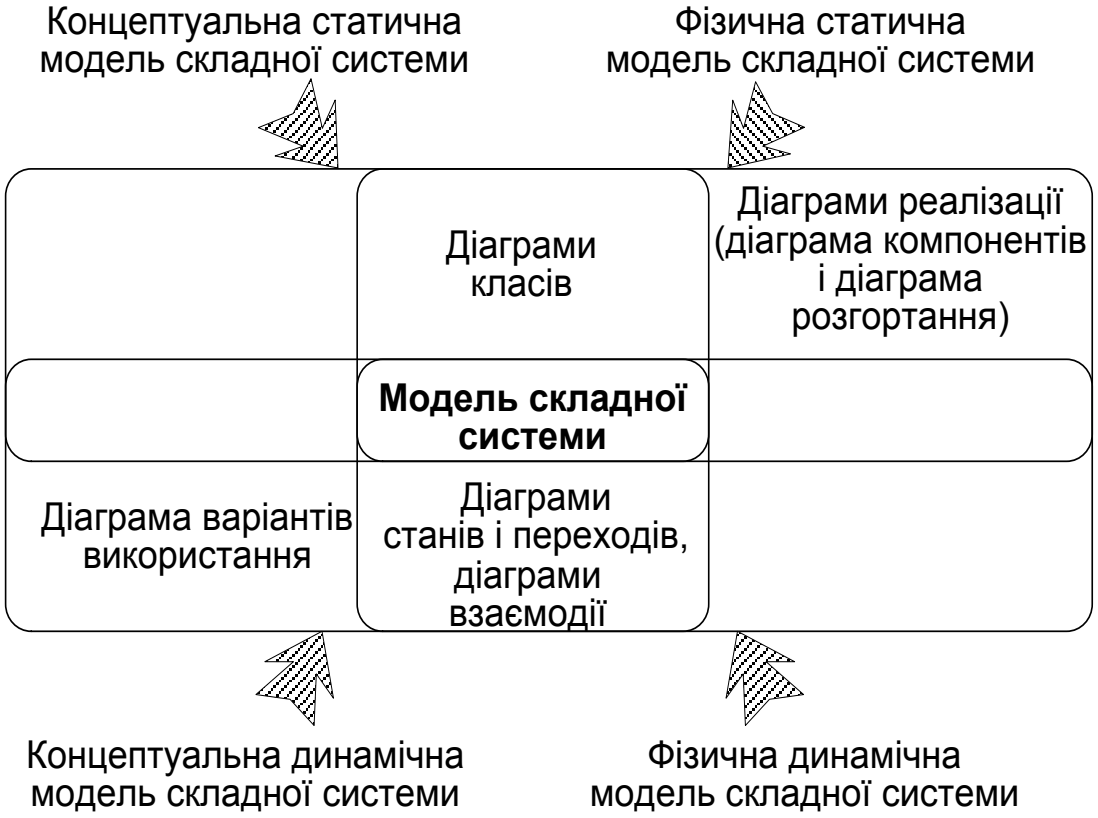

Рис. 1.5. Ракурси моделі OOD

Побудова діаграм – це ще не проектування (проект майбутньої системи існує тільки в голові розробника), проте графічні образи дозволяють описати поведінку системи та показати деталі архітектури. По-перше, стандартна система позначень дозволяє розробникові описати сценарій її роботи або подати структуру таким чином, щоб вона була зрозуміла його колегам. По-друге, продумана система позначень дає розробникові простір для творчості.

> "Звільнивши мозок від зайвої роботи, добра система позначень дозволяє йому зосередитись на задачах більш високого порядку".
>
> *А. Уайтхед*

По-третє, чітка система позначень дозволяє автоматизувати більшу частину стомлюючої перевірки повноти та коректності проекту. З цих причин у 1995 р. Г. Бучем і Дж. Рамбегом було запропоновано універсальну мову моделювання, або універсальний метод, як його тоді назвали.

Уніфікована мова моделювання (скорочено – UML (Unified Modeling Language)) – це засіб специфікації, візуалізації, конструювання й документування програмних систем, а також бізнес-моделей та інших непрограмних систем.

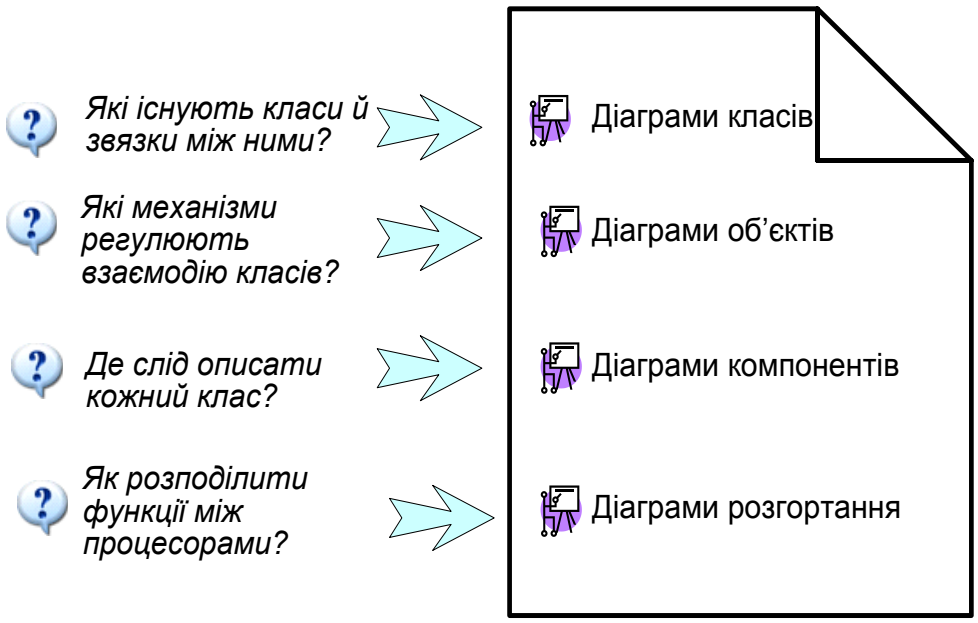

Рис. 1.6. Статичні діаграми

Специфікація – це неграфічна форма, що використовується для повного опису елемента позначення або діаграми в цілому.

1.2.2. Діаграми і їх елементи в нотації UML

Діаграма варіантів використання (use case diagram) – це граф, який відображає діючих осіб (акторів), варіанти використання, а також зв'язки асоціації між діючими особами та випадками використання (або випадків використання між собою).

Діаграми варіантів (випадків) використання показують функціональні можливості системи або класу, як вони виявляються при зовнішній взаємодії з системою. Зміст такої діаграми в тому, що система, яка проектується, подається у вигляді множини зовнішніх сутностей та осіб, що взаємодіють з системою за допомогою так званих варіантів використання. Приклад діаграми зображено на рис.1.7.

Кожен ***варіант використання*** визначає деякий набір дій, який виконує система при діалозі з актором. При цьому не йдеться про те, як буде реалізована взаємодія акторів з системою. Окремий варіант використання зображують на діаграмі еліпсом, всередині якого знаходиться його коротка назва або ім'я у формі дієслова з пояснювальними словами, як показано на рис.1.8.

Актор (діюча особа) – це будь-яка зовнішня відносно системи сутність, що взаємодіє з системою й використовує її функціональні можливості для досягнення певної мети або вирішення конкретних задач.

Актором може бути людина, технічний пристрій, програма або будь-яка інша система, яка може служити джерелом впливу на систему, що моделюється. Це залежить від самого розробника. При цьому актори

служать для позначення узгодженої множини ролей, які можуть виконувати користувачі у процесі взаємодії з системою, що проектується. Кожен актор може розглядатись як окрема роль відносно конкретного варіанта використання. Стандартним графічним позначенням актора на діаграмах є фігурка «чоловічка», під якою записується конкретне ім'я актора (рис. 1.9).

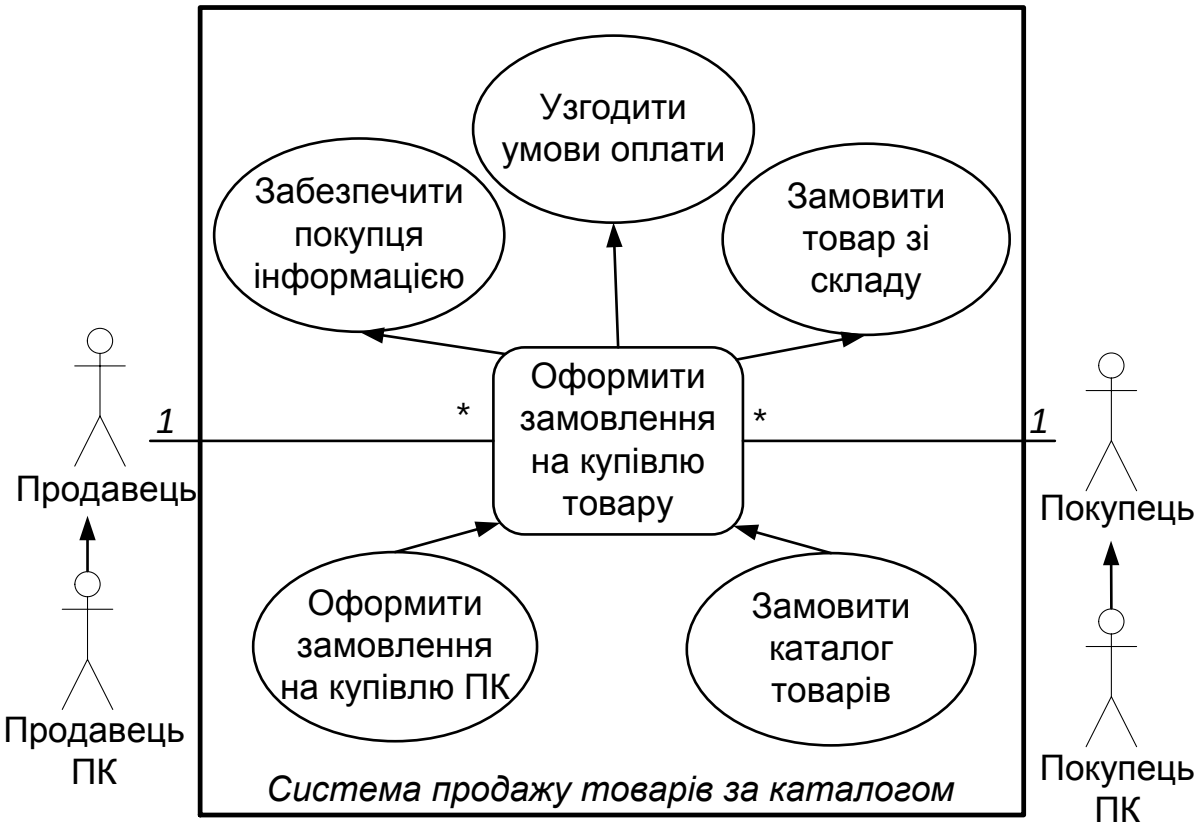

Рис. 1.7. Приклад діаграми варіантів використання

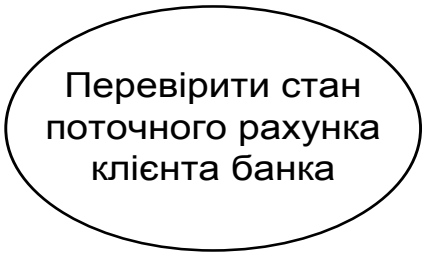

Рис. 1.8. Графічне позначення варіанта використання

Рис.1.9. Графічне позначення актора

Діаграми класів (class diagrams) показують структуру предметної області й відношення з іншими сутностями. Діаграми класів не показують динамічну інформацію, хоч вони можуть мати «матеріалізовані» події або сутності, які описують поведінку у часі. Виділяють діаграму класів і діаграму об'єктів.

Діаграма класів (class diagram) – це набір декларативних (статичних) елементів моделі, таких, як класи, інтерфейси і їх

відношення, поєднані один з одним у вигляді графа.

Позначення класів і відношень між ними в нотації UML показано на рис.1.10.

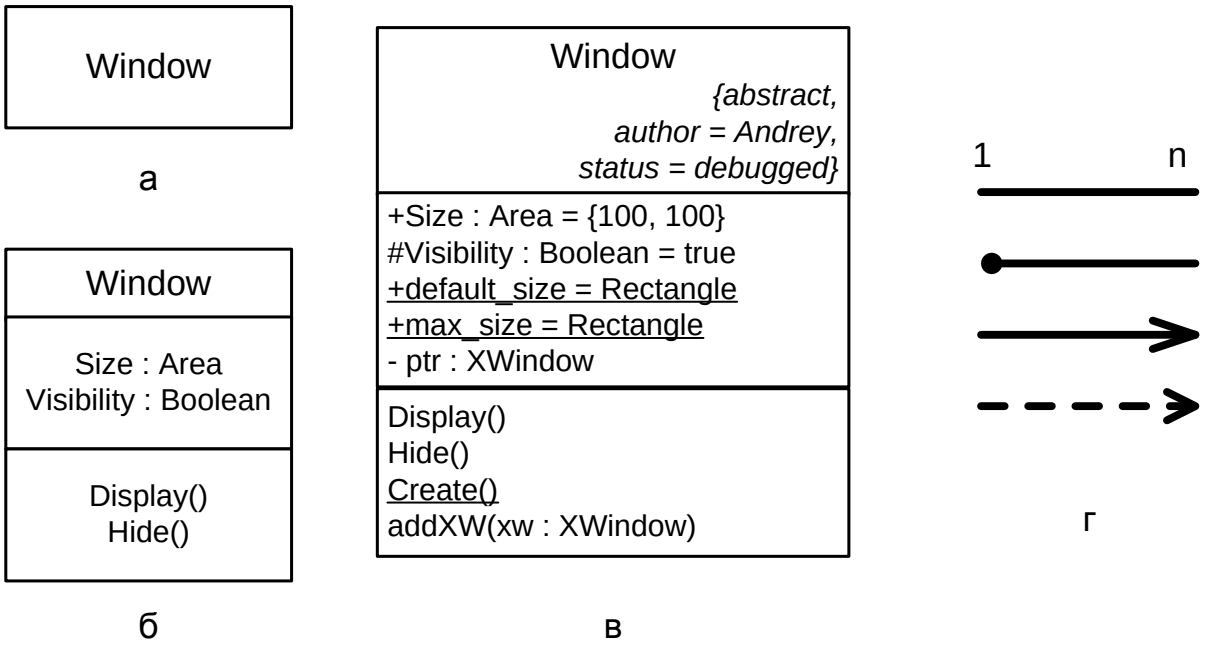

Рис.1.8. Позначення на діаграмі класів: а – деталі сховані; б – деталі рівня аналізу; в – деталі рівня реалізації; г – відношення між класами (*асоціація; агрегація, або «ціле-частина»; наслідування, або «окреме-загальне»; залежність, або використання*)

Діаграми класів можуть бути скомпоновані з пакетів, кожен з утворюючою моделлю, або як окремі пакети, які базуються, в свою чергу, на пакетах, утворюючих модель (рис.1.11).

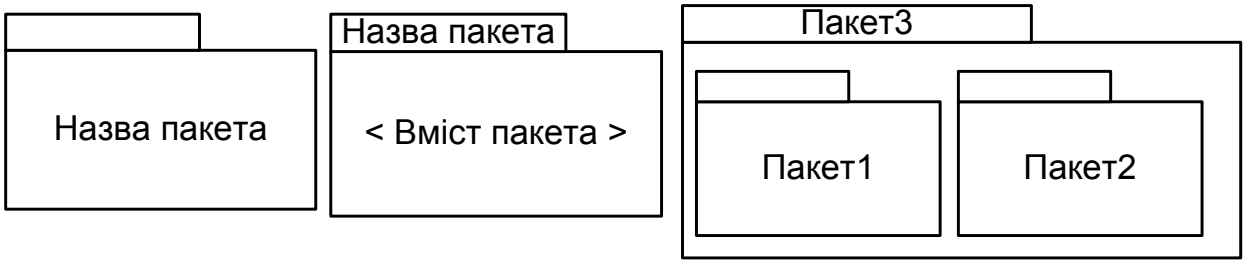

Рис.1.9. Приклади відображення пакетів

Пакет – це «контейнер» для групування елементів моделі.

Діаграма об’єктів (object diagram) – це граф екземплярів, який містить об’єкти й значення даних.

Статична діаграма об'єктів – це екземпляр діаграми класів у вигляді моментального знімку детального стану системи в певний момент часу. Використання діаграми об'єктів досить обмежене, в основному вона показує приклади структури даних. Для програмного забезпечення не потрібна підтримка спеціального формату для діаграм об'єктів. Приклади позначень об’єктів зображено на рис.1.12. Оскільки діаграма класів може містити об'єкти, то діаграма класів з об'єктами й

без класів – це «діаграма об'єктів».

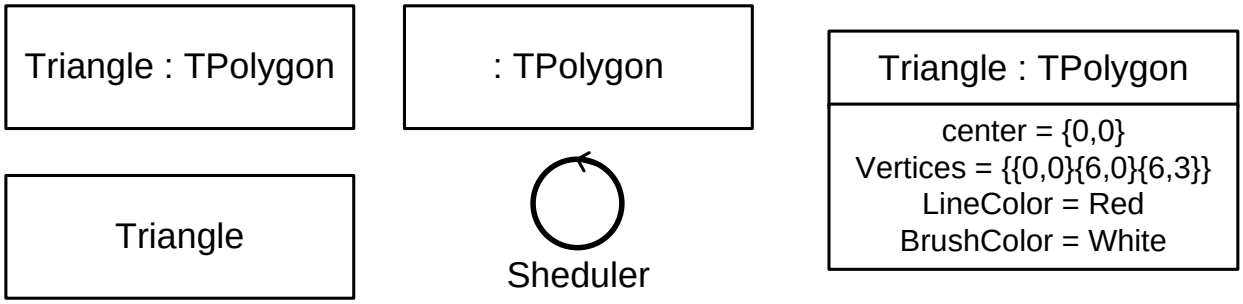

Рис.1.10. Позначення об’єктів

Кожна діаграма об’єктів показує взаємодію або структурні зв’язки, які можуть виникнути у конкретного екземпляра класів, безвідносно до того, які саме екземпляри беруть участь у взаємодії.

На етапі OOD діаграми класів використовують для передачі структури класів, що формують архітектуру системи. Вони також можуть будуватися на етапі OOA для виділення загальних ролей і сутностей, які реалізують необхідну поведінку системи.

Діаграми реалізації показують аспекти реалізації, такі, як структура вихідного тексту й структура реалізації процесу виконання. Відповідно, вони мають дві форми: компонентні діаграми та діаграми розгортання.

Діаграма компонентів (component diagram) – це граф, що відображає залежність між програмними компонентами разом з компонентами вихідного тексту, двійкового коду та виконуваними компонентами.

Програмний модуль може бути поданий як компонентний тип. Деякі компоненти існують під час компіляції, зв’язування або виконання, а деякі існують довше, ніж один проміжок часу. Компонентна діаграма містить тільки подання типу, а не екземпляра. Щоб показати екземпляри компонентів, використовується діаграма розгортання (можливо, вироджена, без вузлів). Компонентна діаграма зображується як граф компонентів, поєднаних відношеннями залежності, як показано на рис.1.13.

Діаграма, що містить типи компонентів і типи вузлів, може використовуватись для показу залежностей компілятора, які зображуються пунктирними стрілками від компонента-клієнта до компонента-сервера, від якого він залежить. Види залежностей специфічні для мови й можуть зображуватись як стереотипи залежностей. Діаграма може також використовуватись для показу інтерфейсів і залежностей виклику між компонентами за допомогою пунктирних стрілок від компонентів до інтерфейсів інших компонентів.

Діаграми розгортання (deployment diagram) – це графічне зображення процесорів, пристроїв, процесів і зв’язків між ними.

На відміну від діаграм логічного подання діаграма розгортання єдина для системи в цілому, оскільки мусить суцільно відображати

особливості її реалізації. Ця діаграма, по суті, завершує процес проектування конкретної програмної системи і її розробка, як правило, є останнім етапом специфікації моделі.

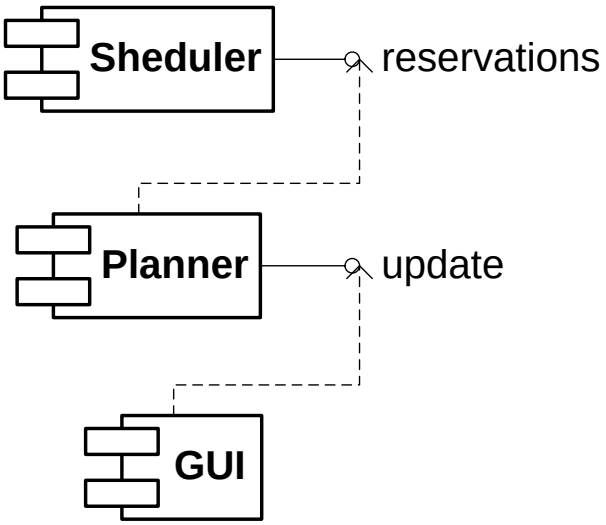

Рис.1.11. Діаграма компонентів

Компонентів, які не є сутностями часу виконання (бо вони компілюються), немає на цих діаграмах, їх слід показувати на компонентних діаграмах. Діаграма являє собою граф вузлів, поєднаних зв’язками асоціації (рис.1.14).

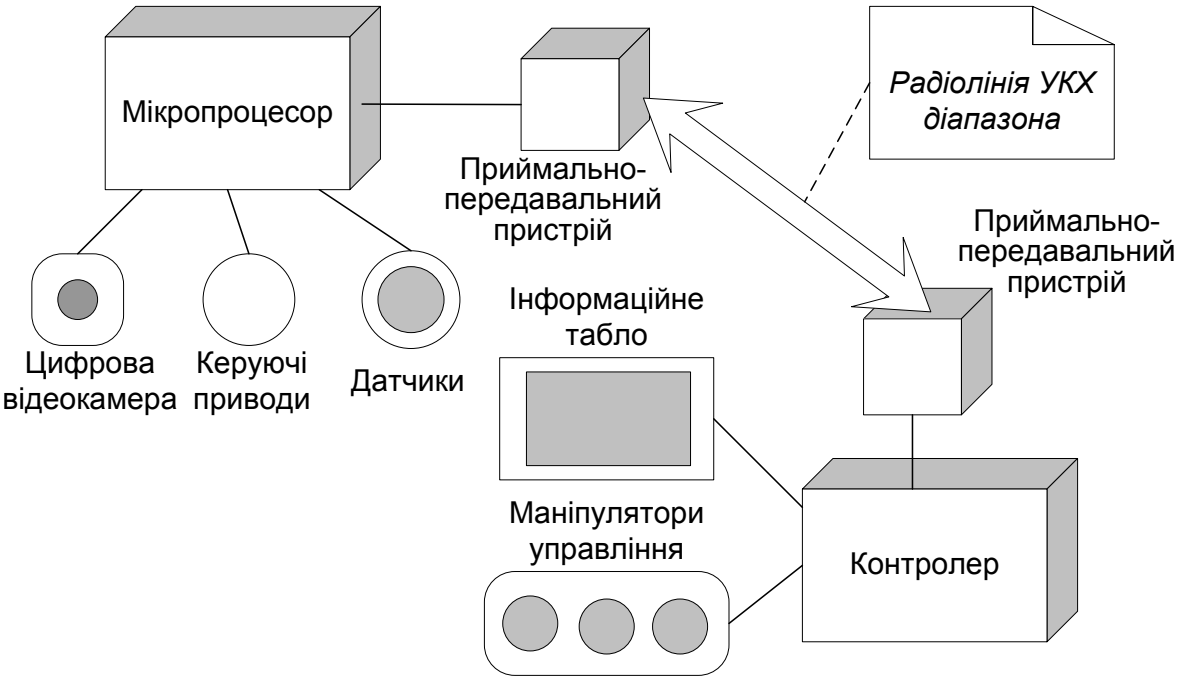

Рис. 1.12. Діаграма розгортання для моделі системи управління транспортною платформою

Вузли можуть містити екземпляри компонентів, які свідчать, що компоненти «живуть» або працюють у вузлі. Компоненти можуть містити об’єкти, що є частинами компонента. Компоненти з’єднуються з іншими компонентами пунктирними стрілками, що вказують на їх зв'язок

(можливо, через інтерфейс). Це означає, що один компонент використовує сервіси іншого. За необхідності для показу точного вигляду залежності може використовуватись стереотип.

Вузол (node) – це матеріальний об’єкт процесу виконання, який надає ресурс обробки й має пам'ять і можливості обробки.

Вузли містять обчислювальні пристрої, а також людські ресурси або ресурси механічної обробки. Вузли можуть являти собою як тип, так і екземпляр, до них можуть належати такі обчислювальні екземпляри часу виконання, як об’єкти і екземпляри компонентів.

Вузол зображують у вигляді символу, схожого на тривимірну проекцію куба, як показано на рис.1.15.

Діаграми станів і переходів визначають всі можливі стани, де може знаходитись об’єкт, а також процеси змінювання станів об’єкта під впливом деяких подій. Діаграма станів і переходів показує простір станів конкретного класу, події, які ведуть до переходу з одного стану в інший і дії, що відбуваються при змінюванні стану.

Розрізняють діаграму станів і діаграму активності (діяльності).

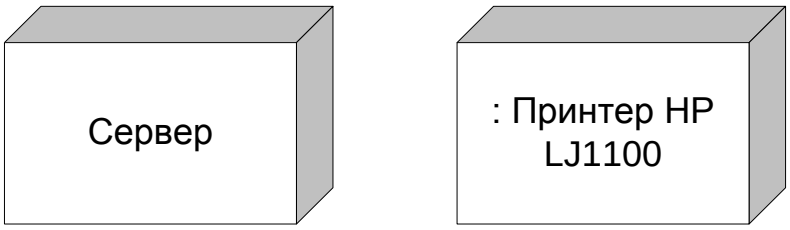

Рис. 1.13. Графічне зображення вузла на діаграмі розгортання

Діаграма станів **(statechart diagram)** – це кінцевий автомат, що відображає послідовність станів, через які проходить об’єкт, і його реакції за період життя у відповідь на отримані стимули.

Діаграма станів – це граф кінцевого автомата. Стани показано символами, переходи – з’єднаними з ними стрілками (рис.1.16).

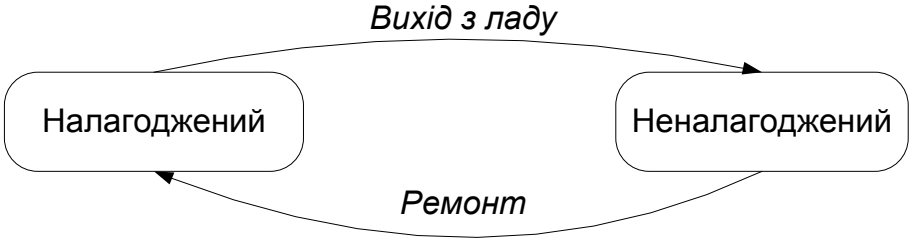

Рис. 1.14. Найпростіший приклад діаграми станів для технічного пристрою

Стан (state) – це ситуація, за якої об'єкт задовольняє деякі умови, виконує деякі дії або чекає якоїсь події.

Стани можуть також містити піддіаграми, створені шляхом фізичного поєднання й розділення на частки (рис.1.17). Об'єкт знаходиться в певному стані обмежений (не нульовий) час.

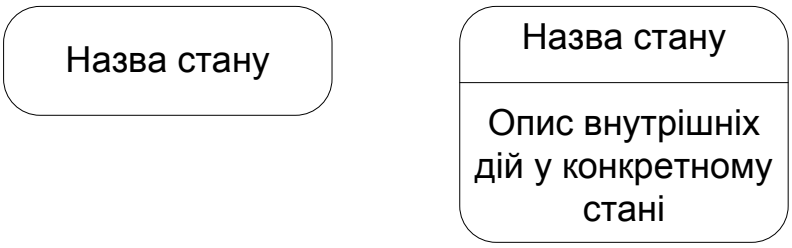

Рис. 1.15. Графічне зображення станів на діаграмі станів

Початковий і кінцевий стани позначаються спеціальними символами (рис.1.18).

а б

Рис. 1.16. Графічне зображення початкового (а) і кінцевого (б) станів

Простий перехід (simple transition) – це відношення між двома послідовними станами, яке вказує на факт заміни одного стану іншим.

Термін «подія» потребує окремого пояснення, оскільки є самостійним елементом мови UML.

Подія (event) – це специфікація деякого факту, що існує в просторі й часі.

Про події говорять, що вони «відбуваються», при цьому окремі події мають бути упорядковані в часі. Після настання деякої події не можна повернутися до попередніх подій, якщо така можливість не передбачена в моделі явно. Семантика поняття події фіксує увагу на зовнішніх проявах якісних змін, що відбуваються під час переходу об'єкта, що моделюється, із стану в стан. Наприклад, при вмиканні електричного перемикача відбувається деяка подія, в результаті якої кімната стає освітленою. Після успішного ремонту комп'ютера також відбувається важлива подія – відновлення його працездатності. Якщо підняти трубку звичайного телефону, то, якщо він працює, буде чути тоновий сигнал, і цей факт теж є подією.

Окрема діаграма станів являє собою конкретний ракурс динамічної моделі окремого класу, істотного з точки зору розробника на етапі OOD. Ці діаграми також можуть використовуватися на етапі OOA для відображення динаміки поведінки системи в цілому.

Діаграма активності **(activity diagram)** – це окремий випадок діаграми станів, що відображає переходи, викликані внутрішніми процесами, в межах вирішення потрібної задачі.

Модель діяльності (активності) є варіантом кінцевого автомата, в якому стани – це діяльність, що відображає виконання операцій, а переходи при цьому ініційовані закінченням операцій (рис.1.19). Вона являє собою кінцевий автомат з процедур, а процедура є реалізацією

операції, яка належить класу.

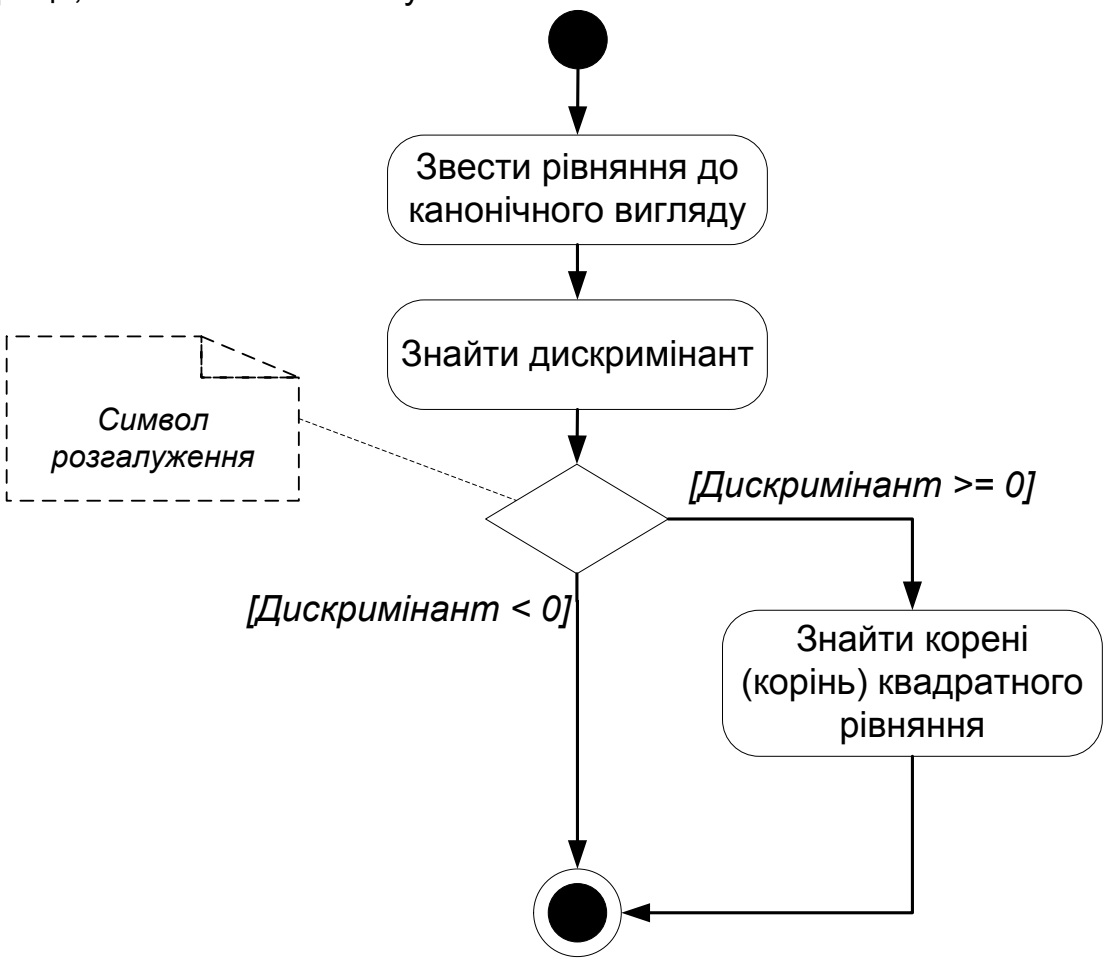

Рис.1.17. Фрагмент діаграми діяльності для знаходження коренів квадратного рівняння

Діаграми взаємодії (interaction diagram) використовують для слідкування за процесом виконання сценарію у тому ж контексті, що й діаграми об'єктів, на відміну від яких діаграма взаємодії більш наочно відображає *послідовність передачі повідомлень*. Разом з тим, за необхідності відображення об'єктів зі складними викликами, атрибутів, ролей і видимості краще користуватися діаграмою об'єктів. На діаграмах взаємодії не використовують нових позначень, ними служать істотні елементи діаграми об'єктів, що подані у вигляді, схожому на таблицю. Назви об'єктів (синтаксис див. вище) на діаграмі взаємодії записують у верхньому рядку, під кожним з них креслять вертикальну пунктирну лінію. Повідомлення, що відправляють, позначають горизонтальною стрілкою від вертикальної лінії клієнта до лінії сервера. Перше відіслане повідомлення показують зверху, а наступне – нижче і т.д. У складних сценаріях, в яких використовують умови і ітерації, зліва від діаграми на рівні необхідного повідомлення можуть бути дописані пояснення. Це може бути текст або конструкція, що подана мовою реалізації.

Діаграми взаємодії краще, ніж діаграми об'єктів відображають семантику сценаріїв на ранніх етапах розробки системи, коли ще не визначено атрибути окремих класів, а необхідно знайти межі системи та взаємозв'язок її елементів. На пізніших стадіях життєвого циклу

програмної системи звичайно користуються діаграмами об’єктів, які докладніше описують семантику об’єктів.

Діаграми взаємодії мають дві форми, що ґрунтуються на загальній інформації, але кожна характеризує її окремий аспект: діаграма послідовності й діаграма співробітництва.

Діаграми послідовності показують взаємодію в часі, зокрема об’єктів, які беруть участь у взаємодії за допомогою «ліній життя», а також повідомлень, якими вони упорядковано обмінюються теж у часі. Вони не відображають асоціації між об’єктами, а показують послідовність передачі повідомлень у формі, що більше підходить для специфікацій реального часу й складних сценаріїв. *Діаграми співробітництва* демонструють стосунки між об’єктами й більше підходять для розуміння всіх ефектів у межах конкретного проекту й для процедурного проектування.

***Діаграма послідовності* (sequence diagram)** – це відображення сукупності повідомлень, які об’єкти передають один одному для реалізації бажаної операції або мети.

На рис. 1.20 зображено приклад діаграми послідовності, що описує динаміку чотирьох об’єктів: «пошукач», «інспектор відділу кадрів», «начальник відділу кадрів» і «ректор», які беруть участь у бізнес-процесі створення наказу про прийняття пошукача на роботу співробітником. Цей бізнес-процес розділено на передачу певних повідомлень: написання заяви, формування параграфа наказу, передавання наказу, редагування наказу, підписання наказу, успішне закінчення.

Діаграма послідовності має два вимірювання: вертикальне відображає час (збільшення – вниз сторінки), горизонтальне – різні об'єкти. Хоч важливою є тільки послідовність подій, в програмах реального часу вісь часу могла б мати реальну метрику. Горизонтальне упорядкування об'єктів не має значення, крім тих випадків, коли деякі об'єкти згруповано в «доріжки процесів».

Слід зауважити, що горизонтальне упорядкування ліній життя довільне. Часто стрілки виклику розміщують в одному напрямку впоперек сторінки, але це не завжди можливо й тоді упорядкування не передає інформацію. Осі можуть бути повернені так, щоб час зростав горизонтально вправо, а різні об'єкти показувалися горизонтальними лініями.

Різноманітні ярлики (часові мітки, описи дій у момент активації та ін.) можуть знаходитися на полях, недалеко від переходів або активацій, які вони маркірують.

Діаграма співробітництва показує взаємодію об’єктів та їх зв’язки один з одним. На відміну від діаграми послідовності вона зображує взаємовідношення ролей об’єктів. Крім того, вона не показує час як окрему розмірність, тому послідовність повідомлень і паралельних шляхів визначається порядковими номерами.

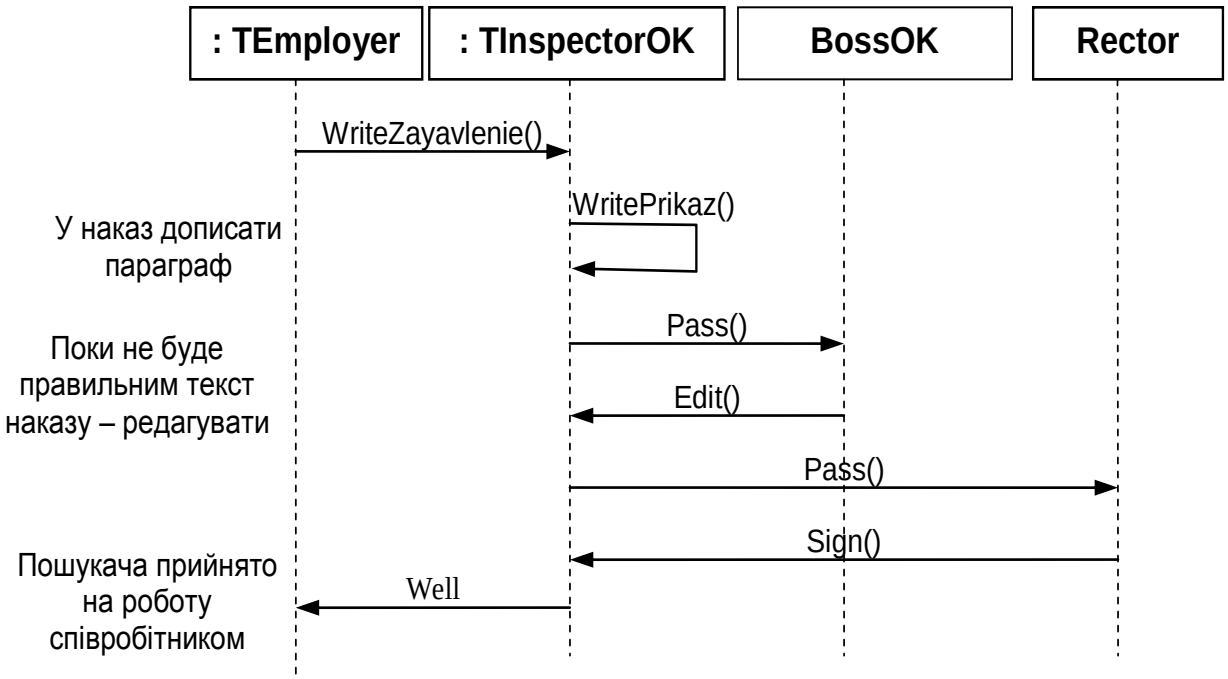

Рис. 1.18. Діаграма послідовності процесу прийняття пошукача на роботу співробітником

Діаграма співробітництва – це граф посилань на об’єкти та зв’язків із зазначенням потоків повідомлень, приєднаних до цих зв’язків.

Діаграма показує об’єкти, які важливі для виконання операції, включаючи конкретно діючі об’єкти або побічно доступні під час операції (рис.1.21). Об’єкти, що створюються під час виконання, можуть бути позначені {new}; об’єкти, що знищуються в процесі виконання, – {destroyed}; об’єкти, що створюються під час виконання, а потім знищуються, – {transient}. Ці зміни в стані існування виводяться з подробиць повідомлень, що передаються між об’єктами й передбачені для зручності подання.

Діаграма також демонструє зв'язки між об'єктами, включаючи часові зв'язки, що відображають аргументи процедури, локальні змінні й зв'язки з самим собою. Оскільки діаграми співробітництва часто використовують для допомоги в проектуванні процедур, вони звичайно зображують навігацію із застосуванням стрілок як зв'язків. Стрілка між прямокутниками об'єктів показує зв'язок з односторонньою навігацією. Стрілкою біля лінії позначають повідомлення, що надходять через зв'язок у заданому напрямку. Очевидно, стрілка повідомлення не може бути спрямованою назад через однонаправлений зв'язок.

Діаграма співробітництва без повідомлень показує оточення, в якому можуть виникнути взаємодії, без відображення яких-небудь окремих відношень. Це може бути використано для відтворення однієї операції або навіть усіх операцій класу або групи класів.

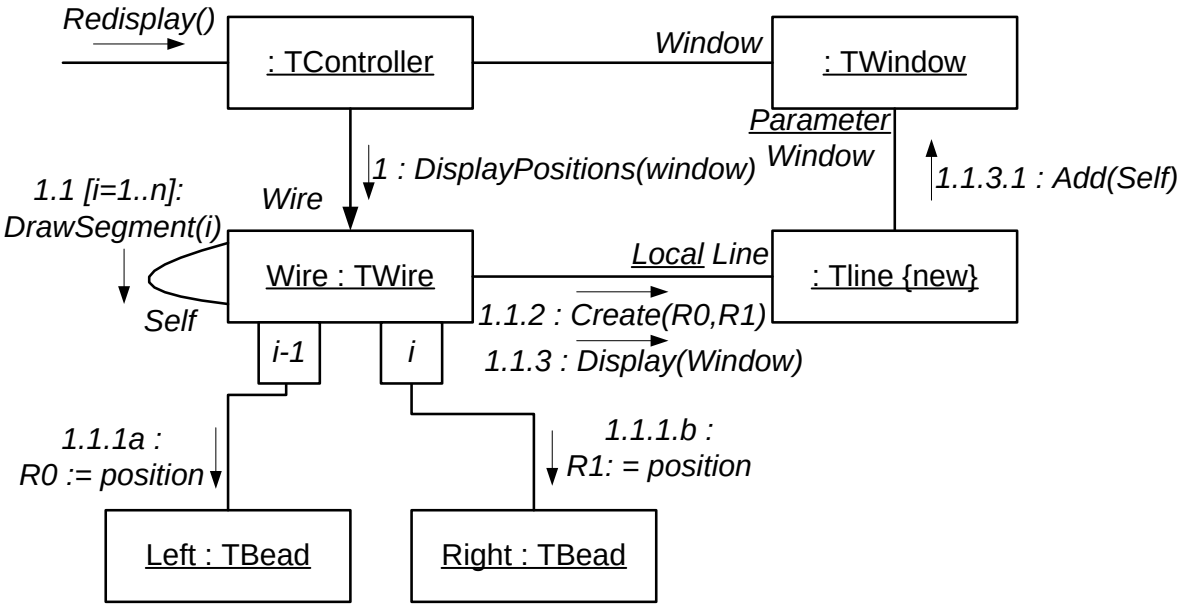

Рис.1.19. Діаграма співробітництва

Кінцевим результатом проектування є набір діаграм перелічених вище видів та їх специфікації. Між цими діаграмами має існувати наскрізний зв'язок. Тоді, починаючи з діаграми послідовності, можна знайти головну програму, позначену у діаграмі компонентів, яка, в свою чергу, містить набори об’єктів і класів. З їх описами можна докладніше ознайомитись на діаграмах класів і діаграмах об’єктів, а з динамічними аспектами – на діаграмах взаємодії та діаграмах станів і переходів.

Наведена система позначень підходить для створення як невеликих систем, так і великих проектів. Крім того, вона враховує можливості використання автоматизованих засобів, які називаються CASE-засобами, для побудови, модифікації та реалізації систем і проектів конкретною мовою програмування. Багато засобів наочного програмування (такі, як Delphi) також дають можливість графічно відображати структуру проекту, модулів і класів розроблюваного програмного продукту.

1.3. Об’єктно-орієнтоване програмування

Об’єктно-орієнтоване програмування – це методологія програмування, що ґрунтується на поданні програми у вигляді сукупності об’єктів, кожен з яких є екземпляром окремого класу, а класи утворюють ієрархію наслідування.

Для написання програм у стилі ООР використовують об’єктно-орієнтовані мови, до яких ставляться певні вимоги:

- слід підтримувати об’єкти, тобто абстракції даних, які мають інтерфейс у вигляді іменованих операцій і власні дані з обмеженням доступу до них;
- об’єкти мають належати конкретним класам;
- класам необхідно наслідувати атрибути суперкласів.

Згідно з такими вимогами до об’єктно-орієнтованих мов відносять Smalltalk, Object Pascal (Delphi), C++, CLOS.

1.3.1. Інтегровані середовища розробки програмного забезпечення

Завдяки візуальному об’єктно-орієнтованому програмуванню була створена технологія, яку стали називати швидкою розробкою додатків (RAD – Rapid Application Development). Візуальне середовище розробки у загальному випадку складається з трьох взаємозв’язаних компонентів: редактора, наладчика та конструктора форм. До RAD-інструментів відносять такі середовища розробки, як Delphi, Visual Basic, C++Builder і Power Builder. Найбільш поширеними є Delphi та C++Builder.

Інтегроване середовище розробки Delphi – продукт компанії Borland International – високопродуктивний інструмент побудови додатків до баз даних в архітектурі «Клієнт – Сервер», а також до локальних машин і файл-серверної архітектури. Це інструментарій, який містить справжній компілятор коду й має засоби візуального програмування трохи схожі на ті, що є в Microsoft Visual Basic 3.0 або в інших інструментах візуального проектування. Мова Object Pascal – основа Delphi і є розвитком об’єктно-орієнтованої мови Pascal (Turbo/Borland Pascal, починаючи з версії 5.5). До Delphi також входить локальний SQL-сервер InterBase 4.0, генератор звітів ReportSmith 2.5, бібліотеки візуальних компонентів та інший інструментарій, необхідний для того, щоб відчувати себе досить впевненим під час професійної розробки інформаційних систем або просто програм Windows-середовища, а SQL-сервер в операційній системі UNIX, Delphi Client-Server може служити зручним інструментом для швидкісної розробки додатків.

Система об’єктно-орієнтованого програмування C++Builder виробництва корпорації Borland призначена для операційних систем Windows 95 і NT. Інтегроване середовище C++ Builder забезпечує швидкість розробки, продуктивність компонентів, що повторно використовуються, а також можливості мовних засобів C++, які удосконалено інструментами та різномасштабними засобами доступу до баз даних.

C++Builder може бути використано всюди, де необхідно доповнити існуючі додатки розширеним стандартом мови C++, збільшити швидкодію та надати користувацькому інтерфейсу професіонального рівня.

Усі компоненти, форми та модулі даних, які накопичили програмісти, що працюють в Delphi, можуть бути без будь-яких змін повторно застосовані в додатках C++Builder для Windows. Delphi поки що продовжує залишатись найпростішою у використанні та найпродуктивнішою системою RAD. Тому C++Builder ідеально підходить тим розробникам, які віддають перевагу виразній можливості мови C++, проте хочуть зберегти продуктивність Delphi. Унікальний зв'язок цих систем програмування дозволяє при створенні додатків легко перейти з одного середовища розробки до іншого.

1.3.2. Деякі аспекти реалізації об’єктів і класів у середовищі Delphi

Мова Object Pascal є базовою для роботи в інтегрованому середовищі Delphi. Характеристиками Delphi є візуалізація програмування, швидка розробка та компіляція додатків, повторне використання коду. До того ж додатки Delphi можуть використовувати розробки іншими мовами, включаючи C++ та асемблер.

Клас – це тип даних, що визначається користувачем.

Те, що в Delphi є безліч вже існуючих класів не суперечить цьому визначенню, адже розробники Delphi теж є користувачами Object Pascal.

Клас слід визначити до того, як буде описано хоча б одну його змінну, тобто клас не можна визначити всередині описання змінної. Клас необхідно описати у певній області видимості всієї програми або окремого модуля. Синтаксис описання класу:

```
TYPE
<ім’я класу> = class (<ім’я класу-родителя >)
        public { доступно всім }
                <поля, методи, властивості, події >
        published {видно в «Інспекторі Объектів» і можуть
                        змінюватися}
                <поля, властивості >
        protected {доступ тільки нащадкам}
                <поля, методи, властивості, події >
        private {доступ тільки в модулі}
                <поля, методи, властивості, події >
 end;
```

Ім'я класу може бути будь-яким допустимим ідентифікатором. Але прийнято ідентифікатори більшості класів починати з символу "T". Ім'я батьківського класу можна не вказувати. Тоді передбачається, що цей клас є безпосереднім нащадком TObject – найбільш загального із уже існуючих класів, тобто еквівалентні такі оголошення:

```
TYPE
TMyClass = class
 ...
end;
і
TYPE
TMyClass = class(TObject)
        ...
end;
```

Клас TObject (модуль System) інкапсулює основні функції, властиві всім об'єктам Delphi. Інтерфейс TObject забезпечує:

- можливість створення, управління й знищення екземплярів об'єктів, включаючи виділення для них пам'яті, ініціалізацію та звільнення пам'яті після їх знищення;

- управління інформацією про об'єкти й типи (run-time type information - RTTI);
- підтримку обробки повідомлень.

Всі класи в Delphi є прямими або непрямими нащадками TObject. Пряме наслідування використовується тільки при оголошенні простих класів, об'єкти яких не є компонентами, не можуть передаватися один одному й не беруть участь в операціях обміну з потоками. Переважна більшість класів є непрямими нащадками TObject і породжуються від проміжних класів. Якщо при оголошенні нового типу об'єктів не вказується клас-предок, то Delphi вважає TObject предком нового класу. Більшість методів TObject не використовується безпосередньо в компонентах, з якими має справу користувач. Вихідні методи TObject звичайно перевантажені в класах-нащадках або замінені іншими, побудованими на їх основі. Хоч формально TObject не є абстрактним класом, об'єкти цього класу створювати не можна.

Клас успадковує поля, методи, властивості, події від своїх предків і може відміняти якісь з цих елементів класу або вводити нові. Доступ до елементів класу, що були оголошені, визначається тим, в якому розділі їх описано.

Розділ ***public*** (відкритий) призначено для опису елементів, що доступні для зовнішнього використання. Це відкритий інтерфейс класу. Розділ ***published*** (що публікується) містить відкриті властивості класу, які з'являються в процесі проектування на сторінці властивостей «Інспектор об'єктів» і які користувач може встановлювати в процесі проектування. Розділ ***private*** (закритий) містить опис полів, процедур і функцій, що використовуються тільки всередині цього класу. Розділ ***protected*** (захищений) містить опис елементів, доступних тільки для нащадків класу, що визначається. Як і в разі закритих елементів, можна приховати деталі реалізації захищених елементів від кінцевого користувача. Проте, на відміну від закритих, захищені елементи залишаються доступними для програмістів, які захочуть створювати від цього класу похідні об'єкти, причому їх не потрібно описувати в цьому ж модулі.

Опис полів виглядає так само, як опис змінних або опис полів в записах: *<ім'я поля >: <тип>; .*

Опис методів в простому випадку також не відрізняється від звичайного опису процедур і функцій. Терміни «операція», «повідомлення», «метод» – синоніми.

Методи бувають чотирьох видів.

Модифікатор – операція, яка змінює стан об'єкта.

Селектор – операція, що зчитує стан об'єкта, але не міняє його.

Конструктор – операція створення об'єкта і/або його ініціалізація.

Деструктор – операція, що звільняє стан об'єкта і/або знищує сам об'єкт.

Наприклад:

```
TYPE
THSchool = class          //описання класу
{public}                  //відкриті властивості класу
Name : string [100];
Place : string [80];
FacultyCount : integer;
SpecList : TList;
…
constructor Create;                          //конструктор
destructor Destroy;                          //деструктор
procedure AddEmploy (Count: integer;); //модифікатор
function GetEmploy Count: integer;   //селектор
…
{private}                 //закриті властивості
EmplCount : integer;
…
End;
VAR
OurUniv : THSchool; // описання об'єкта
…
{ініціалізація, створення об'єкта}
OurUniv:=THSchool.Create;
OurUniv.Name:='Національний аерокосмічний університет ім. М.Є. Жуковського «ХАІ»';
…
{робота з об'єктом}
OurUniv.AddEmploy(2);
…
{знищення об'єкта}
OurUniv.Destroy;
…
```

Агрегація класів – відношення цілого и частини (рис.1.22).

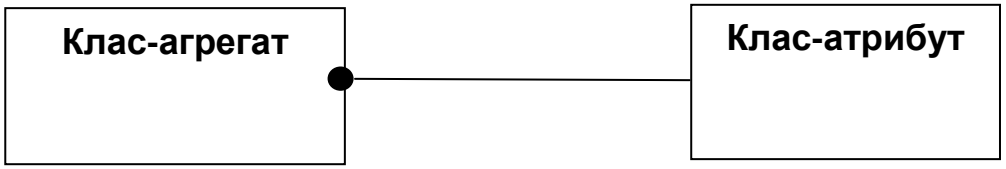

Рис. 1.20. Позначення зв'язку агрегації

Агрегація може бути непрямою, тому її не можна плутати з множинним наслідуванням. Агрегація може бути явною й неявною, при цьому один агрегат може містити один або декілька атрибутів. Таким чином, можна виділити:

1. Явна агрегація одного або декількох атрибутів різних типів (рис.1.23).

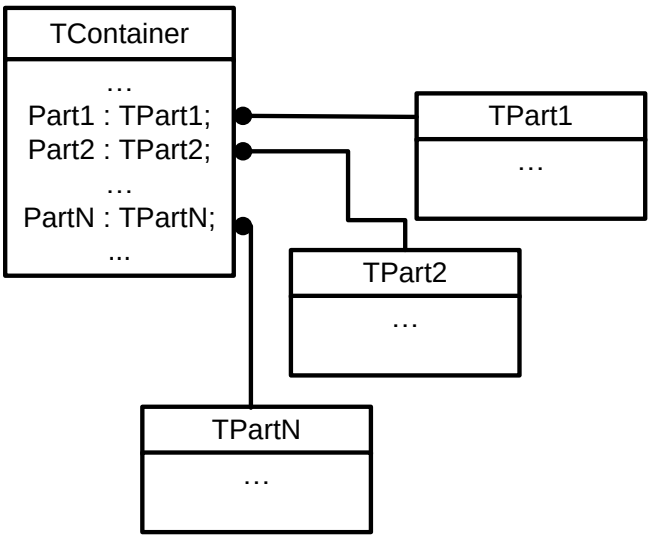

Рис. 1.21. Діаграма класів

```
TYPE
// перший клас-атрибут
TPart1 = class
{}
Property11:integer;
{}
end;
// другий клас-атрибут
TPart2 = class
{}
Property12:string;
{}
end;
{…}
// N-й клас-атрибут
TPartN= class
{}
Property1N:integer;
{}
end;
// клас-агрегат
TContainer= class
 Part1 : TPart1;
 Part2 : TPart2;
{…}
 PartN : TPartN;
{…}
 constructor Create;
end;

implementation
{ реалізація  класу-агрегату}
Constructor TContainer.Create;
  begin
```

```
    inherited;
   Part1 := TPart1.Create;
   Part2 := TPart2.Create;
   {…}
   PartN := TPartN.Create;
 end;
…
VAR
// описання екземпляра класу-агрегату
Container:TContainer;
…
// ініціалізація та робота з об'єктом-агрегатом
Container:=TContainer.Create;
Container.Part1.Property11:=10;
Container.Part1.Property12:='років';
…
```

2. Явна агрегація декількох атрибутів одного типу (рис.1.24).

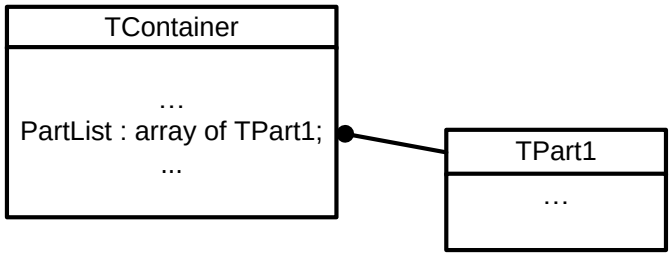

Рис. 1.22. Діаграма класів

```
TYPE
// клас-атрибут
TPart1 = class
{}
Property11:integer;
{}
end;
// клас-агрегат
TContainer= class
 PartList : array [1..10] of TPart1;
 {…}
end;
…
VAR
// описання екземпляра класу-агрегату
Container:TContainer;
…
// ініціалізація та робота з об'єктом-агрегатом
Container:=TContainer.Create;
Container.PartList[1]:=TPart.Create;
Container.PartList[1].Property11:=10;
Container.PartList[2]:=TPart.Create;
Container.PartList[2].Property11:=20;
```

…
3. Неявна агрегація декількох атрибутів одного типу (рис.1.25).

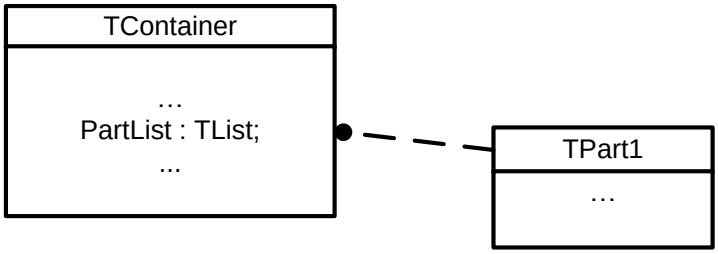

Рис. 1.23. Діаграма класів

```
Type
// клас-атрибут
TPart1 = class
{}
Property11:integer;
{}
end;
// клас-агрегат
TContainer= class
 PartList : TList;
{…}
 constructor Create;
end;
implementation
{ реалізація  класу-агрегату}
Constructor TContainer.Create;
  begin
     inherited;
    PartList := TList.Create;
  end;
…
VAR
// описання екземпляра класу-агрегату
Container:TContainer;
// робочий вказівник на екземпляр класу-атрибуту
TempPart:TPart1;
// робоча змінна цілого типу
…
// ініціалізація та робота з об’єктом-агрегатом
Container:=TContainer.Create;
Container.PartList.Clear;  // очистити список
{ додавання першого елемента }
TempPart:=TPart1.Create;
TempPart.Property11:=10;
Container.PartList.Items.Add(TempPart);  // додати елемент
{додавання другого елемента}
TempPart:=TPart1.Create;
```

```
TempPart.Property11:=20;
Container.PartList.Items.Add(TempPart);  // додати елемент
…
{ звернення до поля атрибута }
a:=TPart1(Container.PartList.Items[0]).Property11;
…
```

Наслідування (англ. – inheritance, subclassing) – це можливість визначення нового об'єкта як розширення вже існуючого за допомогою збереження полів і методів предка і опису лише нових полів і методів.

Концепція наслідування проста й постійно застосовується для того, щоб описати щось нове через вже відомі поняття. Наприклад, вводячи у фізиці поняття «*матеріальна точка»*, визначають його як геометричну точку, що має масу. Поняття «*матеріальна точка»* можна вважати нащадком поняття «*геометрична точка»*, що успадковує всі властивості та характеристики геометричної точки й, крім того, має новий атрибут – масу.

Наслідування об'єктів аналогічне загальноприйнятому поняттю спадкоємства. Коли визначається новий об'єкт як нащадок *(англ. subclass, descendent, child class)* вже існуючого об'єкта, то стверджується, що нащадок здебільшого такий же, як і предок *(англ. ancestor, parent class)*. До об'єкта-нащадка у разі потреби можуть бути добавлені нові поля даних (властивості), не змінюючи існуючі. При описанні нащадка можуть бути добавлені нові методи й перекриті вже існуючі методи предка. Ці додавання й змінювання визначають відмінність нащадка від предка.

Для того, щоб в Delphi наслідувати існуючий клас, необхідно просто вказати його в дужках при описі нового класу. Наприклад, це робиться автоматично кожного разу при створенні форми:

```
TYPE
TForm1 = class(TForm)
…
end;
```

Цей опис показує, що клас TForm1 наслідує всі методи, поля, властивості й події класу TForm, який, у свою чергу, наслідує деякі методи свого предка, й так далі до базового класу TObject. *Object Pascal* допускає у одного предка наявність безлічі безпосередніх нащадків. Проте у кожного нащадка має бути тільки один предок (не допускається *множинне наслідування!*).

Об'єкт-нащадок є сумісним зі своїм предком, тобто екземпляру об'єкта-предка можна присвоїти значення екземпляра об'єкта-нащадка. Але протилежне недопустимо, тобто не можна використовувати об'єкт-предок там, де очікується об'єкт-нащадок. Наприклад,

```
TYPE
TPoint = class  { ТОЧКА - предок ЛІНІЇ }
protected
    x,y : integer;
```

```
    Color : TColor;
public
    constructor Create(xc,yc: integer; cc: TColor);
    function GetX: integer;
    function GetY: integer;
    function GetColor: TColor;
    procedure Show;
end;
TLine = class (TPoint) { ЛІНІЯ - нащадок ТОЧКИ }
private
   x1,y1 : integer;
public
   constructor Create(xc,yc,x1c,y1c: integer; cc: TColor);
   functionGetX1: integer;
   functionGetY1: integer;
   procedure Show;
end;
VAR
  P, P1 : TPoint;
  L : TLine;
BEGIN
 L:= TLine.Create(100,100,200,200,Red);
 P:= TPoint.Create(400,400,Blue);
 P1:= TLine.Create(100,100,200,200,Green); {допускається!}
 P:=L;
END.
```

За умовчанням методи, що описуються в класах, – статичні (*static*). Викликаний статичний метод реалізується з поточного класу. Якщо в ньому процедури або функції з таким ім'ям немає, то шукається метод предка й т.д., поки не знайдеться відповідний за описом метод. Наприклад, у попередньому випадку Show – це статична процедура, і якщо описано об'єкт типу TPoint, то викликається метод виведення точки, якщо об'єкт – TLine – то виведення лінії.

Іноді виникає необхідність усередині методу конкретного об'єкта викликати метод одного з предків цього об'єкта. Для цього перед ім'ям методу, що викликається, необхідно вказати директиву **inherited**. Наприклад,

```
...
constructor TLine.Create(xc,yc,x1c,y1c: integer; cc: word);
begin
  inherited Create(xc, yc, cc); {виклик конструктора класу TPoint}
  x1:=x1c; y1:=y1c;
end;
...
```

Якщо ім'я метода, що викликається, й список параметрів нащадка й предка співпадають, то після зазначення директиви **inherited** можна нічого не писати.

Поліморфізм (англ. polymorphism) – це можливість об'єктів різних класів, пов'язаних через наслідування, по-різному реагувати при зверненні до одного й того ж методу.

У простих випадках поліморфну поведінку об'єктів можна реалізувати оператором CASE … OF..., який залежно від значення деякого поля, що зберігає тип об'єкта, викликатиме потрібний метод. Але в цьому випадку при зміні існуючих або додаванні нових класів доведеться змінювати всі оператори CASE. Крім того, не завжди можна врахувати особливості методів усіх класів-нащадків.

Зручніше поліморфізм реалізується з використанням віртуальних (*virtual*) методів. У об'єктно-орієнтованих мовах програмування для цього існує поняття *пізнього (динамічного) зв'язування*, причому рішення, який саме метод буде викликаний, приймається не при компіляції програми (як у випадку зі статичними методами), а при виклику процедури або функції. Наприклад,

```
…
TYPE
{об'явлення класів T1, T2, T3}
T1 = class
…
procedure print; virtual;
…
end;
T2 = class(T1)
…
procedure print; override;
…
end;
T3 = class(T1)
…
procedure print; override;
…
end;
…
VAR
O1 : T1;  //описання екземпляра базового класу
…
begin
  …
  O1.print;       //виклик процедури відповідного класу
  …
end;
…
```

Як вже було відзначено, за умовчанням методи, що описуються в класах, – статичні (*static*). Для визначення процедури або функції класу-предка як віртуальної після її заголовка в інтерфейсі класу додається директива **virtual** (віртуальний) або **dynamic** (динамічний)**.** Семантично

вони еквівалентні й відрізняються тільки оптимізацією виклику: віртуальні методи оптимізуються за швидкістю, а динамічні – за розміром коду. Проте віртуальні методи ефективніші при реалізації поліморфної поведінки.

У класах-нащадках віртуальні методи можуть бути *перевизначені (overridden)* за допомогою повторного опису цієї процедури або функції з вказівкою директиви **override** (див. приклад вище). Опис перевизначеної процедури в класі-нащадку має повністю співпадати з її описом в класі-предку (порядок і типи параметрів, тип результату, якщо є).

Тільки віртуальні і динамічні методи можуть бути перевизначені, але всі методи (в тому числі й статичні) можуть бути *перезавантажені (overloaded)*. У класі-нащадку після опису методу вказують директиву **overload**, якщо типи і/або кількість параметрів даної процедури відрізняються від успадкованих. У цьому випадку успадкований метод не перекривається, а при виклику активізується той метод, формальні параметри якого відповідають фактичним параметрам виклику.

Якщо в класі-нащадку необхідно *перезавантажувати віртуальний метод*, щоб він не перекривав відповідний метод класу-предка, то після директиви ***overload*** слід використовувати директиву **reintroduce**, яка виключає появу повідомлень компілятора про перекриття попереднього опису процедури. Наприклад,

```
…
T1 = class(TObject)
  procedure Test(I: Integer); overload; virtual;
end;

T2 = class(T1)
  procedure Test(S: string); reintroduce; overload;
end;
 …
SomeObject := T2.Create;
SomeObject.Test('Hello!');  // вызов T2.Test
SomeObject.Test(7);         // вызов T1.Test
…
```

У розділі **published** не може бути описано два перезавантажених методи з однаковими іменами. Реалізація перезавантаженого методу має починатися із заголовка, який повністю співпадає із заголовком в секції інтерфейсу.

Абстрактний клас – це клас, екземпляри якого не можна ініціалізувати внаслідок того, що в ньому описано принаймні один абстрактний метод.

Абстрактний метод – це динамічний або віртуальний метод, реалізація якого не описується в цьому класі.

Абстрактний метод описується директивою **abstract** після директиви ***virtual*** або ***dynamic*** і має бути перевизначений в конкретних

класах-нащадках. Наприклад,

```
…
T1 = class  {абстрактний клас}
…
   procedure DoSomething; virtual; abstract;
…
end;
T2 = class(T1) {конкретний клас}
   procedure DoSomething; override;
end;
…
```

Абстрактні методи відіграють важливу роль у реалізації поліморфізму. Не можна викликати метод класу-нащадка через покажчик класу-предка, якщо цей метод не описано в інтерфейсі класу-предка. Наприклад,

```
…
TAnimal = class      {клас «Тварина»}
public
constructor Create;
function GetKind: string;
function Voice: string; virtual; abstract;
private
Kind: string;
end;
TDog = class (TAnimal)   {клас «Собака»}
public
constructor Create;
function Voice: string; override;
function Eat: string; virtual;
end;
TCat = class (TAnimal)   {клас «Кіт»}
public
constructor Create;
function Voice: string; override;
function Eat: string; virtual;
end;
…
MyAnimal :TAnimal;
…
LabelVoice.Caption := MyAnimal.Voice;  {правильний виклик}
LabelVoice.Caption := MyAnimal.Eat; {генерує помилку “Field identifier expected.”}
…
```

1.4. CASE-системи

У наш час поширені автоматизовані CASE-засоби для побудови, супроводження, модифікації та реалізації інформаційних систем конкретною мовою програмування. Термін CASE (Computer Aided Software Engineering) зараз використовується в досить широкому

розумінні. Початкове значення терміну CASE, обмежене питаннями автоматизації розробки тільки програмного забезпечення (ПЗ), сьогодні набуло нового змісту, що охоплює процес розробки інформаційних систем (ІС) в цілому. Тепер під терміном CASE-засоби розуміють програмні засоби, що підтримують процеси створення та супроводження ІС, включаючи аналіз і формулювання вимог, формування прикладного ПЗ (додатків) і баз даних, генерацію коду, тестування, документування, забезпечення якості, конфігураційне управління та управління проектом, а також інші процеси. CASE-засоби разом із системним ПЗ і технічними засобами утворюють повне середовище розробки ІС.

CASE-технології являють собою методологію проектування ІС, а також набір інструментальних засобів, які дозволяють у наочній формі моделювати предметну область, аналізувати цю модель на всіх етапах розробки та супроводження ІС і розробляти додатки згідно з інформаційними потребами користувачів. Більшість існуючих CASE-засобів базується на методології структурного (в основному) або об'єктно-орієнтованого аналізу та проектування з використанням специфікацій у вигляді діаграм і текстів для опису зовнішніх вимог, зв'язків між моделями системи, динаміки поведінки системи і архітектури програмних засобів.

Інтегрований CASE-засіб (або комплекс засобів, які підтримують повний життєвий цикл (ЖЦ) ПЗ) містить такі компоненти:

- репозиторій, що є основою CASE-засобу. Він має забезпечувати зберігання версій проекту та його окремих компонентів, синхронізацію надходження інформації від різних розробників під час групової розробки, контроль її повноти й несуперечності;
- графічні засоби аналізу й проектування, які забезпечують створення й редагування ієрархічно пов'язаних діаграм (DFD, ERD та ін.), що утворюють моделі ІС;
- засоби розробки додатків, включаючи мови 4GL і генератори кодів, конфігураційного управління, документування, тестування, управління проектом, реінжинірінга.

Усі сучасні CASE-засоби можуть бути класифіковані в основному за типами та категоріями. Класифікація за типами відображає функціональну орієнтацію CASE-засобів на ті або інші процеси ЖЦ. Класифікація за категоріями визначає ступінь інтегрованості за виконуваними функціями й містить окремі локальні засоби, які дозволяють вирішувати невеликі автономні задачі (tools), набір частково інтегрованих засобів, що охоплюють більшість етапів життєвого циклу ІС (toolkit), і повністю інтегровані засоби, що підтримують весь ЖЦ ІС і пов'язані спільним репозиторієм. Крім того, CASE-засоби можна класифікувати так: за методологіями, що застосовуються, та моделями систем і БД; ступенем інтегрованості з СУБЗ; доступними платформами.

Класифікація за типами в основному збігається з компонентним складом CASE-засобів і містить такі основні їх види:

- засоби аналізу (Upper CASE), що призначені для побудови і аналізу

моделей предметної області (Design/IDEF(MetaSofware), BPwin(Logic Works));

- засоби аналізу та проектування (Middle CASE), які підтримують найбільш поширені методології проектування й ті, що використовуються для створення проектних специфікацій (Vantage Team Builder (Cayenne), Designer/2000 (ORACLE), Silverrun (CSA), PRO-IV (McDonnell Douglas), CASE-Аналітик (MakroProject)). Результатом застосування таких засобів є специфікації компонентів та інтерфейсів системи, її архітектури, алгоритмів і структур даних;
- засоби проектування баз даних, які забезпечують моделювання даних і генерацію схем баз даних (як правило, мовою SQL) в найбільш поширених СУБД. До них відносять Erwin (Logic Works), S-Designer (SDP), Data Base Designer (ORACLE) . Засоби проектування баз даних є також у складі CASE-засобів Vantage Team Builder, Designer/2000, Silverrun, PRO-IV;
- засоби розробки додатків. До них відносять засоби 4GL (Uniface(Compuware)), JAM(JYACC), PowerBuilder(Sybase), Developer/2000(ORACLE), New Era(Informix), SQL Windows(Gupta), Delphi(Borland) та ін.) і генератори кодів, що входять до складу Vantage Team Builder, PRO-IV і частково – Silverrun;
- засоби реінжинірінга, що забезпечують аналіз програмних кодів і схем баз даних і формування на їх основі різноманітних моделей і проектних специфікацій. Засоби аналізу і формування схем БД входять до складу Vantage Team Builder, PRO-IV, Silverrun, Designer/2000, Erwin, S-Designer. В області аналізу програмних кодів найбільш поширеними є об’єктно-орієнтовані CASE-засоби, які забезпечують реінжиніринг мовою C++ (Rational Rose (Rational Software), Object Team (Cayenne).

Допоміжні типи містять засоби:

- планування та управління проектом (SE Companion, Microsoft Project та ін.);
- конфігураційного управління (PVCS(Intersolv));
- тестування (Quality Works(Segure Software));
- документування (SoDA(Rational Software)).

На сьогоднішній день російський ринок програмного забезпечення має такі найбільш розвинені CASE-засоби: Vantage Team Builder(Westmount I-CACE), Designer/2000,Silverrun, ERwin, BPwin, S-Designor, CASE_Аналітик.

2. ВАРІАНТИ ПРЕДМЕТНОЇ ОБЛАСТІ

1. М’яч, що стрибає [6] (передбачити можливість створення двох або трьох м’ячів).
2. Маятник [6] (передбачити створення двох або трьох маятників) .
3. Підсилювач з насиченням [6] (необхідно кольорове зображення виходів).
4. Обрив нитки маятника, що коливається [6].
5. Розвезення товарів по складах [6].
6. Система-вентилятор [6].
7. Система терморегулювання будинку [6].
8. Десятинний лічильник [6].
9. Вантаж на пружині [6].
10. Фільтруючий повторювач [6].
11. Об’єкт, що коливається під дією зовнішньої сили (передбачити можливість створення двох або трьох об’єктів) [6].
12. Катапульта літака [6].
13. СУ комбінованим генератором [6].
14. СУ сидінням водія [6].
15. Задача № 1 [6].
16. Задача № 2 [6].
17. Задача № 3 [6].
18. Задача № 4 [6].
19. Задача № 5 [6].
20. Задача № 6 [6].
21. Задача № 7 [6].
22. Задача № 8 [6].
23. Задача № 9 [6].
24. Задача № 10 [6].
25. Задача № 11 [6].
26. Задача № 12 [6].
27. СУ ліфтом [7].
28. СУ теплицями [5].
29. «Пружинна машина» [8].
30. «Інерційна машина» [9].
31. «Водяний пістолет» [10].
32. «Вертоліт» [11].
33. СУ кодовим замком.
34. СУ конвеєром (два – три типи предметів).
35. СУ лебідкою(два – три типи предметів).
36. СУ підйомним краном (два – три типи предметів).
37. Гра “Волейбол”.
38. М’яч, кинутий у стіну (передбачити створення двох або трьох м’ячів).
39. Сонячна система.

Приклад основної частини пояснювальної записки курсового проекту для варіанта об’єкта «Система двох сполучених баків» наведено в дод. 1. У дод. 2 описано загальну структуру документа.

БІБЛІОГРАФІЧНИЙ СПИСОК

1. Сван Т. Основи програмування в Delphi для Windows 95. – К.: Діалектика, 1996. – 479 с.

2. Cantu M. Mastering Delphi 6. – Alameda: SYBEX Inc., 2001. – 1071 с.

3. Енго Ф. Delphi 3. Самоучитель. – К.: DiaSoft, 1998. – 320 с.

4. Буч Г., Рамбо Дж., Айвар Дж. Язык UML. – М.: ДМК Прес, 1998. – 429 с.

5. Буч Г. Об'єктно-орієнтований аналіз і проектування з прикладами додатків на С++: Пер. з англ. – М.: Біном, СПб: Невський Діалект, 1998. – 560 с.

6. Бенькович Е.С., Колесов Ю.Б., Сениченков Ю.Б. Практичне моделювання динамічних систем. – СПб.: БХВ-Петербург, 2002. – 464 с.

7. Йордон Э., Аргилла К. Структурные модели в объектно-ориентированном анализе и проектировании. – М.: Лори, 1999. – 268 с.

8. Кулик А.С., Пасичник С.Н. Лабораторный практикум по курсу «Введение в моделирование систем». Объект моделирования: "Пружинная машина": Учеб. пособие. – Х.: Нац. аэрокосм. ун-т «ХАИ», 2003. – 51 с.

9. Кулик А.С., Пасичник С.Н. Лабораторный практикум по курсу «Введение в моделирование систем». Объект моделирования: "Инерционная машина": Учеб. пособие. – Х.: Нац. аэрокосм. ун-т «ХАИ», 2003. – 47 с.

10. Кулик А.С., Пасичник С.Н., Мирная Е.В. Лабораторный практикум по курсу «Введение в моделирование систем». Объект моделирования: "Водяной пистолет": Учеб. пособие. – Х.: Нац. аэрокосм. ун-т «ХАИ», 2003. – 53 с.

11. Кулик А.С., Пасичник С.Н. Лабораторный практикум по курсу «Введение в моделирование систем». Объект моделирования: "Вертолет": Учеб. пособие. – Х.: Нац. аэрокосм. ун-т «ХАИ», 2003. – 49 с.

Додаток 1

ПРИКЛАД ВИКОНАННЯ КУРСОВОГО ПРОЕКТУ «КОМП'ЮТЕРНЕ МОДЕЛЮВАННЯ СИСТЕМИ УПРАВЛІННЯ ДВОМА БАКАМИ»

Д 1.1. Опис предметної області

Поставлено задачу управління потоком води через систему, що складається з двох баків (рис. Д.1.1). Система містить два циліндричних баки, які розміщені так, що дно першого бака знаходиться на відстані H=0,39 м від дна другого бака. Висота баків h=1м, діаметр першого бака d1=0,12 м, другого – d2=0,05 м. Вхідна труба розміщена на відстані h=1м від дна першого бака. Баки з'єднані трубою, яка розміщена біля самого дна першого бака і на відстані H=0,39 м від дна другого бака. Система забезпечена краном Vinput (вхідний кран першого бака), краном V1 (вихідний кран першого бака – вхідний кран другого бака), й краном V2 (вихідний кран другого бака). Кран Vinput відкривається миттєво. На закриття або відкриття кранів V1 та V2 відведено 80 секунд, за цей час засувки кранів змінюють значення свого положення від P=0 до P=80, швидкість переміщення засувок постійна. Управління станом кранів здійснює контролер.

Швидкість вхідного потоку води, л/год

$$\frac{dVinput}{dt} = \begin{cases} 0, \textit{ якщо } Vinput \textit{ закритий}, \\ 400, \textit{ якщо } Vinput \textit{ відкритий}. \end{cases}, \qquad \text{(Д.1.1)}$$

Якщо позначити площі основних баків як A1 та A2, то систему рівнянь для визначення рівнів води в баках можна записати так:

$$\frac{dh_1}{dt} = \frac{1}{A_1}\left(\frac{dVinput}{dt} - \frac{dV_{12}}{dt}\right); \qquad \text{(Д.1.2)}$$

$$\frac{dh_2}{dt} = \frac{1}{A_2}\left(\frac{dV_{12}}{dt} - \frac{dV_{out}}{dt}\right), \qquad \text{(Д.1.3)}$$

де dV12/dt – швидкість протікання води по трубі між баками, а dVout/dt – швидкість витікання води із системи.

Швидкість протікання води між баками залежить від рівнів води h1 та h2, значення H та розміщення засувки P1 у крані V1 :

$$\frac{dV_{12}}{dt} = \begin{cases} K_1(P_1)\cdot\sqrt{h_1 - (h_2 - H)}, h_2 > H; \\ K_1(P_1)\cdot\sqrt{h_1}, h_2 \le H. \end{cases} \qquad \text{(Д.1.4)}$$

Швидкість витікання води із системи залежить від рівня води у другому баці та розміщення засувки P2:

$$\frac{dV_{out}}{dt} = K_2(P_2)\cdot\sqrt{h_2}\ . \qquad \text{(Д.1.5)}$$

Індивідуальні властивості кранів визначаються функціями:

$$K_1(P_1) = \begin{cases} 1.85\cdot10^{-4}\cdot e^{-6\cdot10^{-6}\cdot P_1^3}, 0 \le P_1 < 80 \\ 0, P_1 = 80 \end{cases}; \qquad \text{(Д.1.6)}$$

$$K_2(P_2)=\begin{cases}2.26\cdot10^{-4}\cdot e^{-5.7\cdot10^{-6}\cdot P_2^3}, 0\le P_2<80\\ 0, P_2=80\end{cases}. \qquad (Д.1.7)$$

Загальний вигляд системи зображено на рис. Д.1.1.

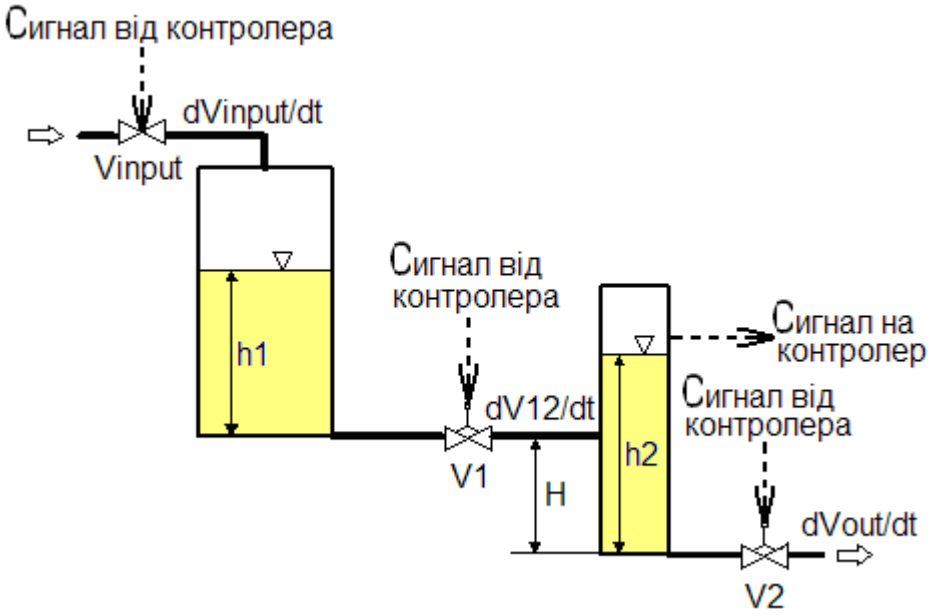

Рис. Д.1.1. Система двох баків

Робота системи. У початковий момент часу всі крани закриті, баки пусті. Контролер посилає сигнал відкрити вхідний кран Vinput, який відкривається миттєво, й протягом часу Time1 перший бак наповнюється водою. Коли цей час минає, контролер посилає сигнал відкрити кран V1 (відкриття відбувається протягом 80 секунд), вода починає надходити до другого баку. Коли мине час Time2, починається контроль стану крана V2. Якщо вода у баці опускається нижче рівня Lmin, кран V2 закривається, якщо піднімається вище рівня Lplus, то кран V2 відкривається. Аварійними вважаються ситуації, коли переповнюється один з баків (у такому випадку система блокується) або відбувається періодичне відкриття й закриття вихідного крана. Нормальним режимом системи вважають стан, коли всі крани відкриті й вода протікає через систему з постійною швидкістю.

Слід розглянути стан системи, коли періодично відкривається та закривається вихідний кран. При цьому змінні мають значення, що наведені в табл. Д.1.1.

Таблиця Д.1.1 – Значення параметрів стану системи при періодичному режимі роботи кранів

Time1, с	Time2, с	Lplus, м	Lmin, м
60	25	0,9	0,30

Д.1.2. Об’єктно-орієнтований аналіз предметної області

Вибір підходу. У процесі аналізу необхідно скласти словник предметної області, тобто виділити класи і об’єкти, виконати їх ідентифікацію. Для цього використовують класичну категоризацію,

концептуальну кластеризацію, теорію прототипів, аналіз поведінки, аналіз предметної області, аналіз варіантів, CRC-картки, неформальний опис, структурний аналіз.

У класичному підході всі предмети, які володіють якоюсь певною властивістю або сукупністю властивостей, формують деяку категорію. Таким чином, в класичній категоризації в ролі критерію схожості об'єктів використана спорідненість їх властивостей.

За допомогою концептуальної кластеризації формують опис класів (кластерів об'єктів), далі класифікують сутності відповідно до цих класів. Концептуальну кластеризацію можна пов'язати з теорією нечітких множин, тобто об'єкт може належати кільком категоріям одночасно з різним ступенем точності.

Теорію прототипів застосовують для класифікації абстракцій, які не мають чітких властивостей і чіткого визначення. Тут клас визначається одним об'єктом, який є прототипом. Новий об'єкт належить до цього класу, якщо його наділено істотною схожістю з прототипом.

Використовуючи аналіз поведінки, слід зосередитись на динамічній поведінці сутностей, тобто сформувати класи, базуючись на групах об'єктів, які виявляють схожу поведінку. Використовуючи аналіз предметної області, необхідно вивчити вже існуючі додатки в межах цієї області, ключові абстракції, що є корисними в схожих системах. Аналіз можна виконувати відносно декількох додатків (вертикально) або відносно частин одного додатка (горизонтально). Також бажано проконсультуватися з експертом щодо цієї предметної області, найчастіше це – користувач системи.

Після аналізу варіантів перераховують та опрацьовують сценарії, найбільш придатні для роботи системи. Визначають, які об'єкти беруть участь у сценарії, їх обов'язки, як вони взаємодіють. Таким чином визначають область впливу кожної абстракції. У результаті з'являються нові абстракції або уточнюються вже існуючі.

Використовуючи CRC-картки, кожному позначеному класу виділяють карточку. На картці зверху записують назву класу, знизу в лівій половині – за що цей клас відповідає, знизу в правій половині – з ким співробітничає. Якщо клас має занадто велику відповідальність, то необхідно виділити новий клас, або перенести частину відповідальності на кілька більш детальних класів.

Згідно з методом неформального опису задачу слід описати природною мовою, а потім підкреслити іменники та дієслова. Іменники стануть кандидатами на роль класів, дієслова – кандидатами на операції.

Інколи використовують структурний аналіз як основу для об'єктно-орієнтованого аналізу та проектування. Після проведення структурного аналізу формують модель системи, що описується діаграмами потоків даних. Виходячи з формальної моделі системи, можна визначити осмислені класи об'єктів. Кандидати в об'єкти виходять з навколишнього середовища, з важливих вхідних і вихідних даних, а також з продуктів,

послуг та інших ресурсів, якими система управляє.

У постановці задачі перелічено елементарні складові системи (два баки і три крани) і описано, як мусить працювати система, тобто її поведінку. Виходячи з цього, визначають більш точно, які сутності мають бути в наявності. Тому, найдоцільніше у цьому випадку для аналізу системи скористатися одним з найбільш зручних методів – аналізом поведінки.

Опис підходу. Виконуючи аналіз поведінки, увагу слід концентрувати не на найбільш значущих елементах предметної області, а на динамічній поведінці як на основі класів та об'єктів. Класи утворюють, спираючись на групи об'єктів, які демонструють схожу поведінку. Можна зробити висновок, що об'єднуються об'єкти, які мають схожі відповідальності й будується ієрархія класів, у якій підклас, що виконує обов'язки суперкласу, привносить свої додаткові послуги. Під відповідальністю розуміють сукупність послуг, які об'єкт може надати, це спосіб визначити мету цього об'єкта, його місце в системі. Розглядаючи поведінку всієї системи, намагаються зрозуміти, яка її частина ініціює поведінку і які частини в ній беруть участь. Ініціатори і учасники стають кандидатами на роль об'єктів, а ті функції, в яких вони безпосередньо беруть участь, відносять до їх відповідальності.

Опис об'єктів і класів. На основі роботи системи у вербальній моделі можна визначити основні дії, що відбуваються в системі: її управляюча частина посилає сигнал на відкриття або закриття кранів. Баки наповнюються водою або спорожняються, здійснюється контроль рівня води в них і у визначений момент посилається сигнал контролеру. Таким чином, можна виділити основні частини системи, які беруть участь у її поведінці і ініціюють її. Ініціатором роботи за сигналом користувача є управляюча частина системи – контролер, а учасниками – крани та баки.

Поведінку контролера складають такі функції: послати сигнал, прийняти сигнал, при переповненні баків блокувати роботу системи; а стан контролера визначають властивості: значення часу Time1 і Time2, значення максимального та мінімального рівнів води Lplus і Lmin. Замість крана доцільніше розглядати пристрій, який приймає сигнали від контролера і управляє механізмом відкриття та закриття кранів. В системі є три крани, їх властивості різні, оскільки вхідний кран повністю відкривається миттєво, а два інші – через 80 секунд після надходження сигналу. Тому пристрій, що управляє вхідним краном та двома іншими необхідно розділити. Пристрій, який управляє вхідним краном, можна назвати VinputControl. До його функцій відносять розрахунок швидкості вхідного потоку води Vinput. Пристрій, що управляє двома іншими кранами – KControl. Він розраховує швидкості потоку води між баками V12 і вихідного потоку Vout; його властивості характеризують атрибут P, який визначає стан крана. Функціями баків є розрахунок рівнів води h1 та h2, а властивостями – діаметри баків D1 і D2 і їх площі A1, A2.

Між класами TController, TVinputControl, TKControl і TTank існують відношення використання. Клас TController залучає класи TVinputControl і

TKControl для управління потоками води, що тече в баки, посилаючи їм повідомлення про відкриття або закриття кранів. TController використовує клас TTank для отримання інформації щодо рівнів води в баках і відповідної подальшої координації роботи системи. Клас TTank одержує інформацію про швидкості вхідних та вихідних потоків води від класів TVinputControl і TKControl.

Відношення наслідування і агрегації як підсистеми на цьому етапі в системі не виділяють.

Узагальнена діаграма об'єктів і діаграма класів. На етапі об'єктно-орієнтованого аналізу діаграму класів будують для визначення спільних ролей і сутностей, які реалізують необхідну поведінку системи, а діаграму об'єктів – для показу семантики основних і другорядних сценаріїв, що визначають поведінку системи. Діаграми класів та об'єктів цієї системи показано на рис. Д.1.2 та Д.1.3.

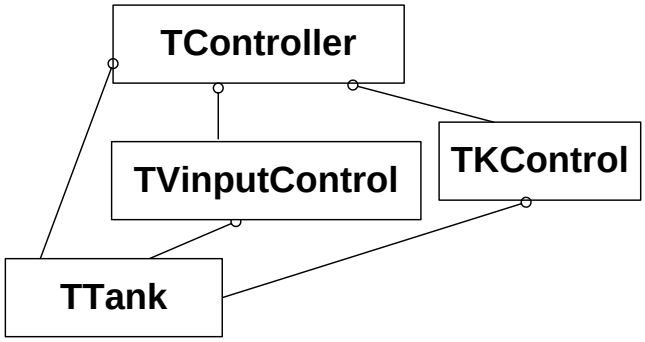

Рис. Д.1.2. Діаграма класів

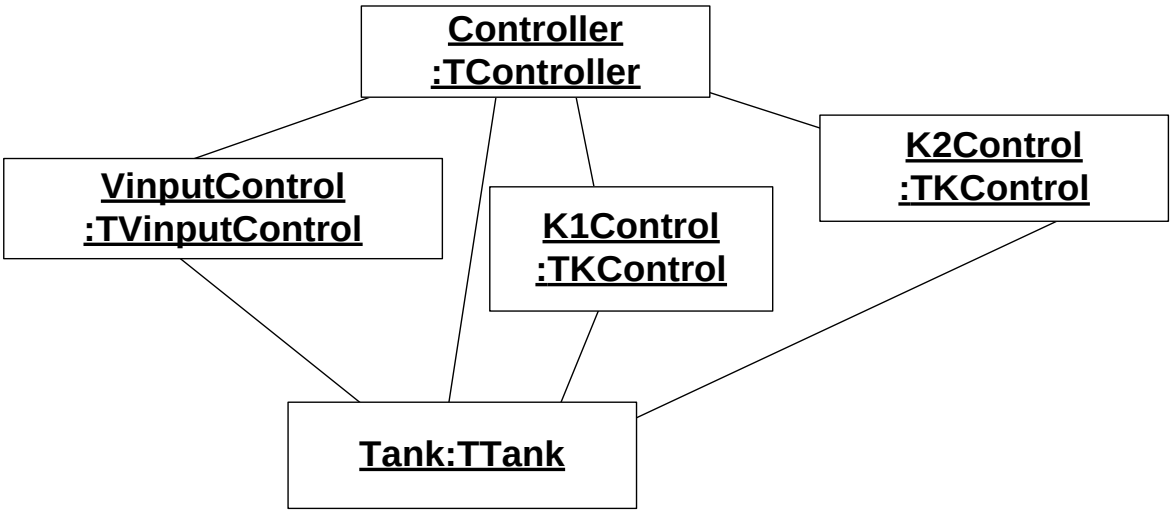

Рис. Д.1.3. Діаграма об'єктів

Висновки: у процесі об'єктно-орієнтованого аналізу складено словник предметної області, визначено поведінку системи, виділено класи і об'єкти та визначено їх відповідальність.

Д.1.3. Об'єктно-орієнтоване проектування майбутньої програмної системи

Вибір технічних і програмних засобів. Під час проектування системи необхідно враховувати особливості її реалізації з використанням конкретного апаратного забезпечення, певної

операційної системи й конкретною мовою програмування.

Апаратне забезпечення містить такі складові:

- процесор Celeron 1.00A GHz FCPGA BOX ;
- материнську плату Socket370 KOBIAN-815E TFSX, Int815E, SB, ATX;
- пам'ять DIMM 256Мб;
- відомості щодо операційної системи: Microsoft Windows XP Professional версія 2002.

Програмна система реалізується мовою Object Pascal як найбільш вивченою в інтегрованому середовищі розробки Delphi 5.

Delphi – загальновизнаний лідер інструментів для створення додатків і систем, що функціонують на платформі Windows. Версія Delphi 5 містить засоби підтримки Web та інтеграції з існуючими Windows-додатками. В основі підтримки розподілених розрахунків у системі Delphi лежить два основних принципи: орієнтація на стандарти та високорівнева розробка клієнтської й серверної логік додатків на основі компонентної моделі. Комплекс технологій, компонентів, інструментів і засобів підтримки об’єктних інфраструктур (middleware) – COM, CORBA, XML – MIDAS – є невід’ємною частиною Delphi 5.

Переваги Delphi перед аналогічними програмними продуктами:

- швидкість розробки додатків;
- висока продуктивність розробленого додатка;
- низькі вимоги розробленого додатка до ресурсів комп’ютера;
- розширюваність системи за рахунок добудови нових компонентів та інструментів у середовищі Delphi;
- можливість розробки нових компонентів та інструментів власними засобами Delphi (існуючі компоненти і інструменти, що доступні у вихідному коді);
- опробувана ієрархія об’єктів.

У системі Delphi чудово відображено візуальне програмування: під час проектування форми та розміщення компонентів Delphi автоматично формує коди програми, добавляючи відповідні фрагменти, які описують конкретний компонент. Відбувається повторне використання коду, розробка інтерфейсу значно спрощується. Це дозволяє програмісту за хвилини або години зробити те, на що раніше витрачалися місяці роботи.

Діаграми мовою UML. Уніфікована мова моделювання (UML) являє собою універсальну мову, яка дозволяє одночасно з аналізом створювати документацію для проектування складних ієрархічних систем, щоб потім ввести її в працездатний код будь-якою мовою програмування.

Розроблений у термінах UML проект можна легко реалізувати будь-якою існуючою мовою, яка підтримує об’єктно-орієнтовану технологію.

Діаграми дозволяють описати поведінку системи (для аналізу) й показати деталі архітектури (для проектування). Під час проектування ставляться такі запитання: які існують класи та зв’язки між ними; як розподілити функції між процесорами? При пошуку відповідей на ці

запитання використовують діаграми класів, об'єктів, модулів і процесів. Для опису динаміки системи використовують діаграми переходів і станів, а також діаграми взаємодії.

Будуючи діаграми в OOD, необхідно враховувати компоненти області додатків, взаємодії з людиною, управління задачами та даними.

Компонент області додатка – це об'єкти, що виконують важливі функції цієї області. Компонент взаємодії з людиною – технологія інтерфейсу, яка застосовується в програмній системі. Компонент управління задачами – це елементи операційної системи, які забезпечують функціонування програмної системи в цілому. Компонент управління даними – це об'єкти, необхідні для взаємодії з базою даних.

Діаграма варіантів використання. Далі слід побудувати діаграму варіантів використання, яка описує функціональне призначення системи або, іншими словами, те, що система буде виконувати при функціонуванні. Мета варіанту використання – визначити закінчений аспект або фрагмент поведінки деякої сутності без розкриття внутрішньої структури цієї сутності. Діаграму варіантів використання для цієї системи зображено на рис. Д.1.4.

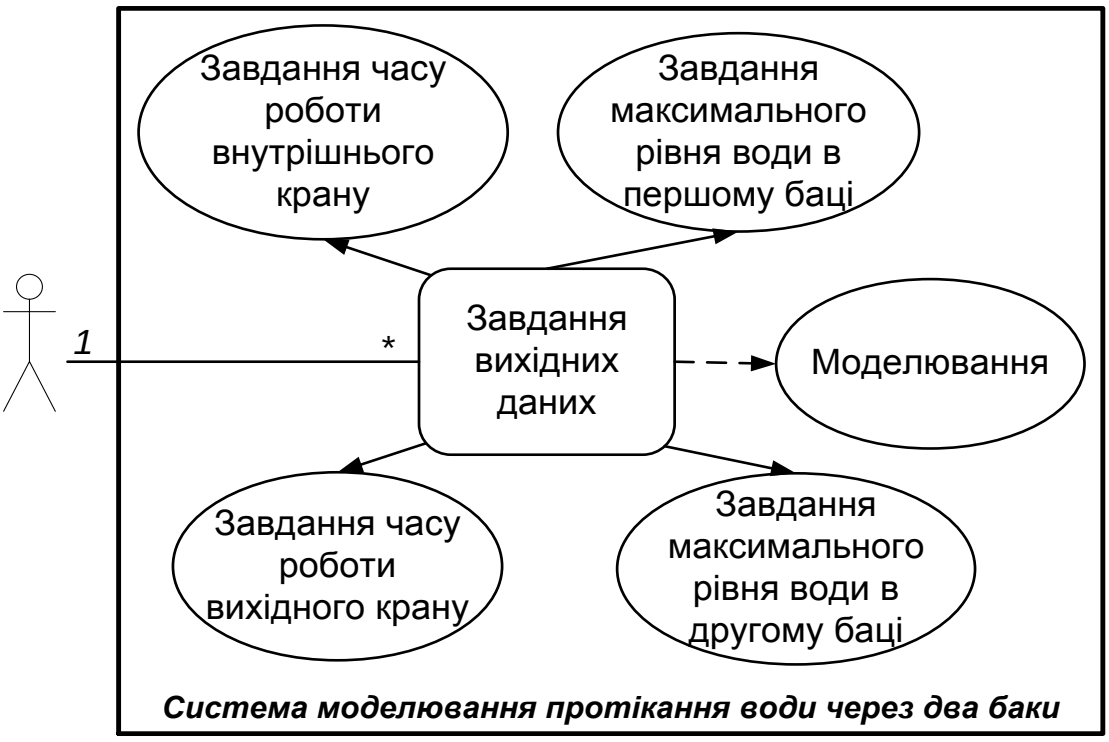

Рис. Д.1.4. Діаграма варіантів взаємодії

Діаграма класів. На стадії проектування користуються діаграмою класів (Д. 1.5), щоб показати структуру класів та їх взаємовідношення, що формують архітектуру системи.

Специфікація до діаграми класів:

Ім'я: **TController.**

Визначення: пристрій, що виконує контроль роботи системи.

Обов'язки: передача сигналу, що управляє механізмом відкриття та закриття кранів системи, а також контроль рівня води в баках.

Атрибути: VinOn, V1On, V2On.

Операції: Env_VinOn(), Env_V1On(), Env_V1Off(), Env_V2On(), Env_V2Off(), Env_VinOff().

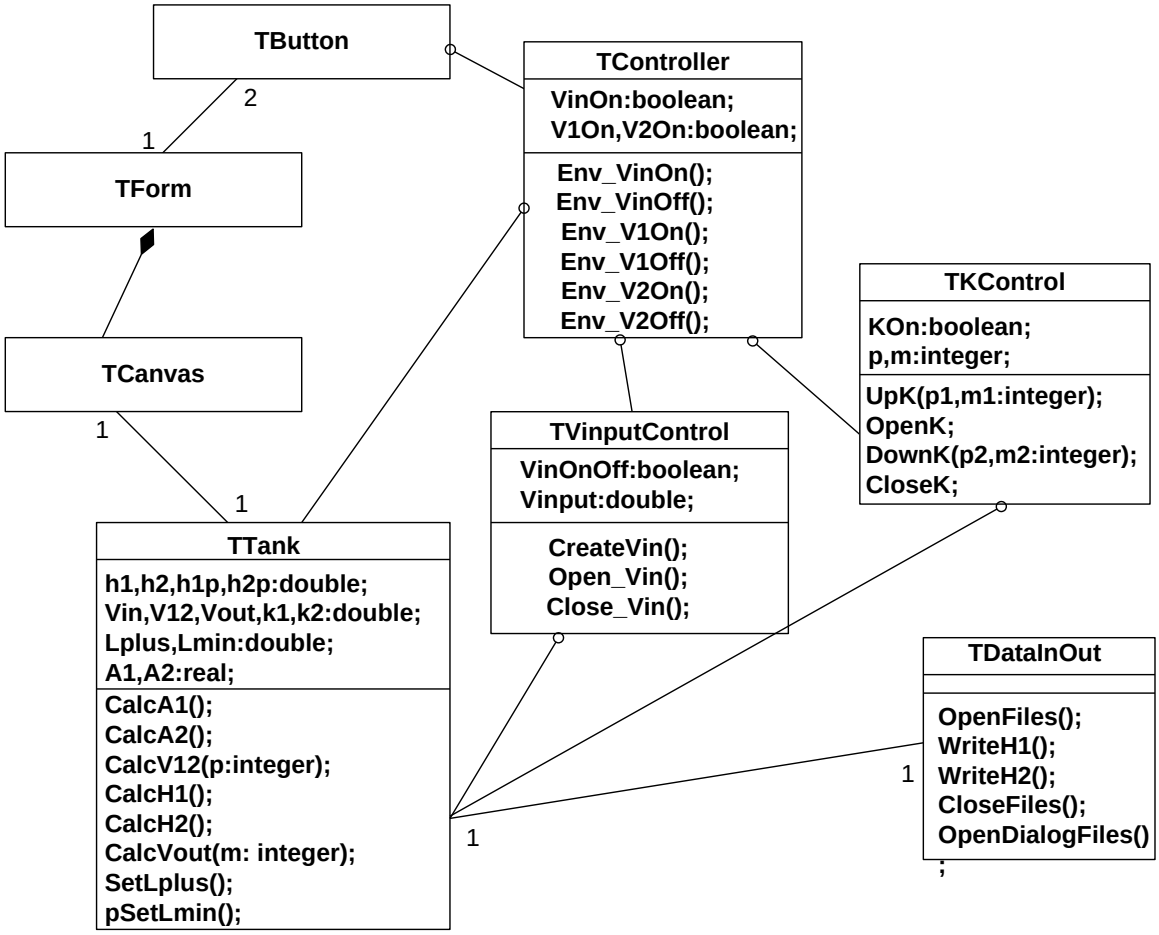

Рис. Д.1.5. Діаграма класів

Ім’я: **TVinputControl**.

Визначення: пристрій, який управляє вхідним краном системи.

Обов’язки: відкриття та закриття вхідного крана системи.

Атрибути: Vinput, VinOnOff.

Операції: Open_Vin(), Close_Vin().

Ім’я: **TKControl**.

Визначення: пристрій, який відповідає за відкриття та закриття кранів V1та V2;

Обов’язки: відкриття та закриття протягом 80 секунд кранів V1 и V2.

Атрибути: KOn.

Операції: UpK(), OpenK(), DownK(), CloseK().

Ім’я: **TTank**.

Визначення: підсистема, яка складається з двох баків.

Обов’язки: розрахунок значень рівнів h1, h2 води в баках.

Атрибути: h1, h2, h1p, h2p, Vin, V12, Vout, k1, k2, Lplus, Lmin, A1, A2.

Операції: CalcH1(), CalcH2(), CalcA1(), CalcA2(), CalcV12(p:integer), CalcVout(m: integer), SetLplus(), SetLmin().

Обмеження: h1<=1, h2<=1.

На **діаграмах станів і переходів** (рис. Д.1.6 – Д.1.9) зображено: простір станів певного класу; події, які обумовлюють перехід з одного стану в інший; дії, які відбуваються під час змінювання стану. Окрема діаграма станів і переходів являє собою визначений ракурс динамічної моделі окремого класу або цілої системи. Слід будувати діаграми станів і переходів тільки для класів, поведінка яких істотна. Можна також побудувати діаграми переходів і станів, що використовуються при аналізі для того, щоб показати динаміку поведінки системи в цілому, а при проектуванні – для відображення поведінки окремих класів або їх взаємодій.

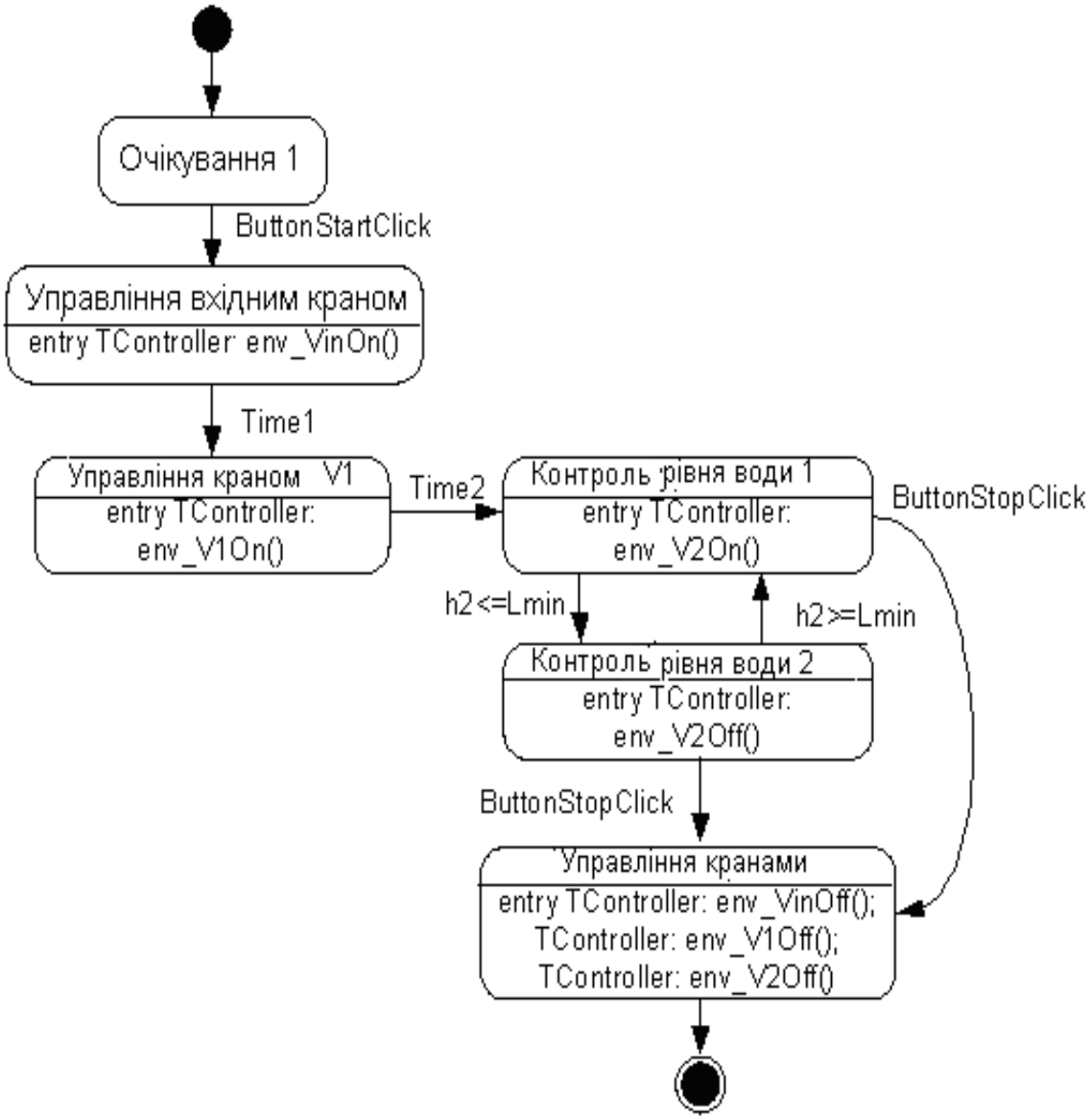

Рис. Д.1.6. Діаграма станів і переходів для класу TController

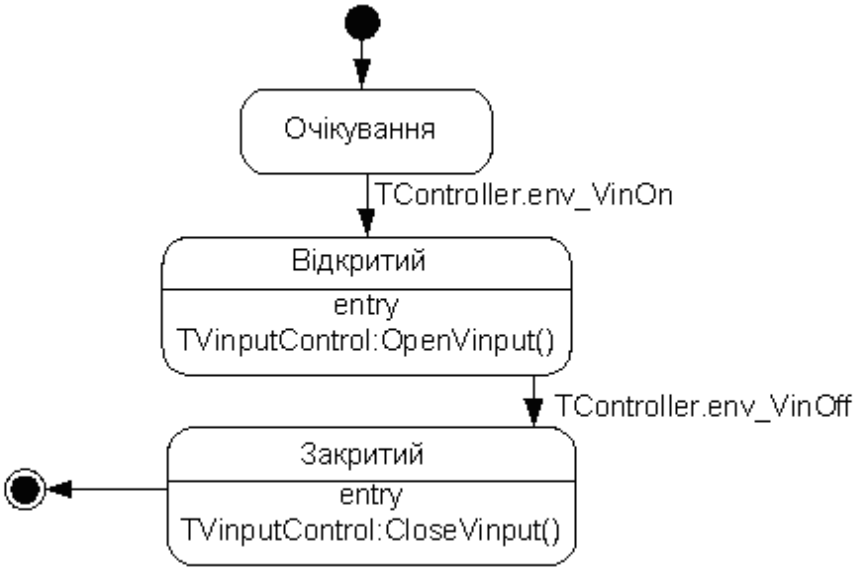

Рис. Д.1.7. Діаграма станів і переходів для класу TVinputControl

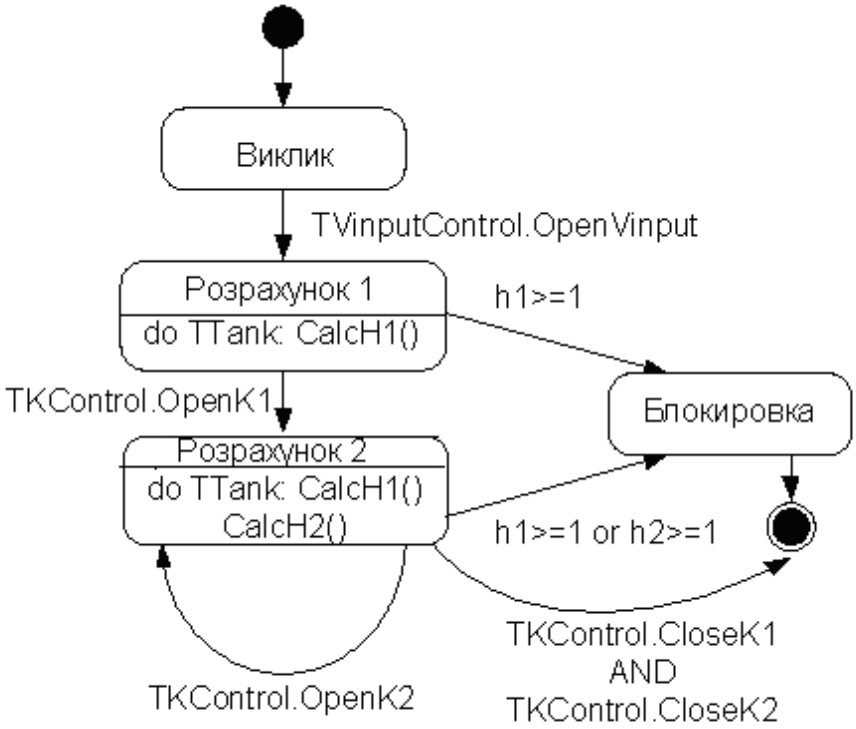

Рис. Д.1.8. Діаграма станів і переходів для класу TTank

Діаграма послідовності (рис. Д.1.10) використовується для того, щоб прослідкувати виконання сценарію в тому ж контексті, в якому подано й діаграму об’єктів. Перевага діаграми послідовності полягає в тому, що на ній легше читається порядок передачі повідомлень, а перевага діаграми об’єктів – в тому, що вона більше підходить для

багатьох об’єктів із складними викликами й дозволяє використовувати іншу інформацію, а саме: зв’язки, значення атрибутів, ролі.

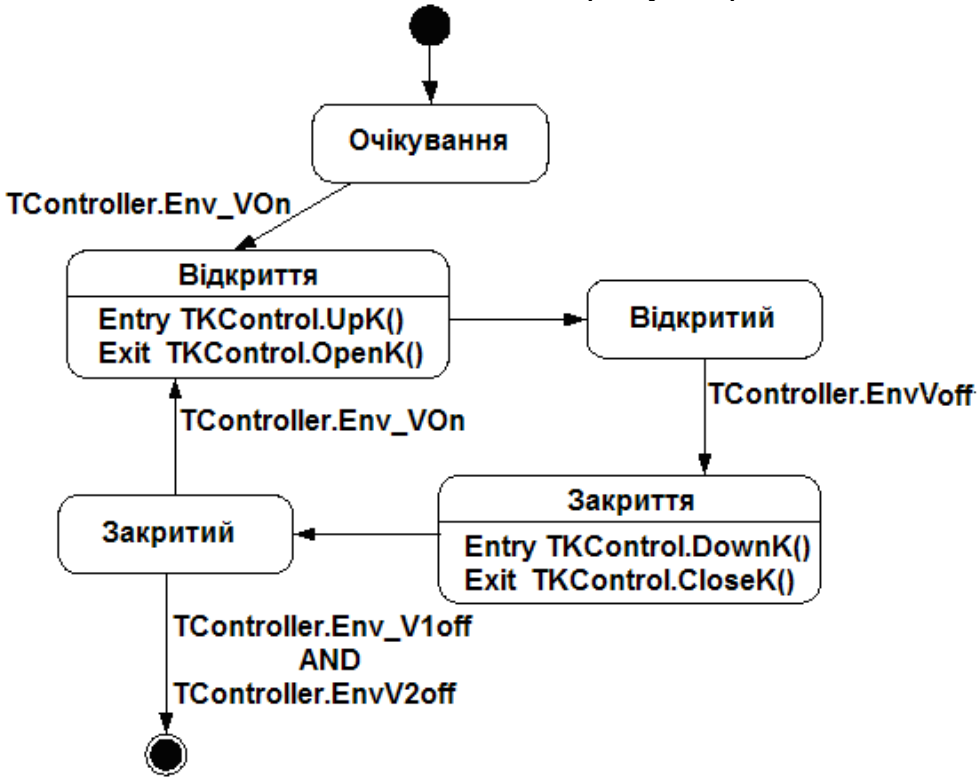

Рис. Д.1.9. Діаграма станів і переходів для класу TKControl

Діаграма компонентів (рис. Д.1.11) дозволяє визначити архітектуру розроблюваної системи, коли знайдено залежність між програмними компонентами, якими можуть бути вихідний, бінарний, виконуваний коди. У багатьох середовищах розробки модуль або компонент відповідає файлу. Пунктирні стрілки, які поєднують модулі, показують відношення взаємозалежностей, що аналогічні тим, які існують під час компіляції текстів програми.

Ім’я: **Pr_Tank.exe**.

Визначення: виконуваний файл, який реалізує роботу системи.

Ім’я: **UnitSystem**.

Визначення: модуль, який описує форму й реалізує функції взаємодії з користувачем.

Ім’я: **ClassUnit**.

Визначення: модуль, що містить опис класів, які реалізують систему.

Ім’я: **GraphUnit**.

Визначення: модуль, що містить процедури й функції, що забезпечують графічне зображення системи.

Ім’я: **DataUnit.**

Визначення: модуль, що містить процедури й функції, які забезпечують запис даних про систему до файлу.

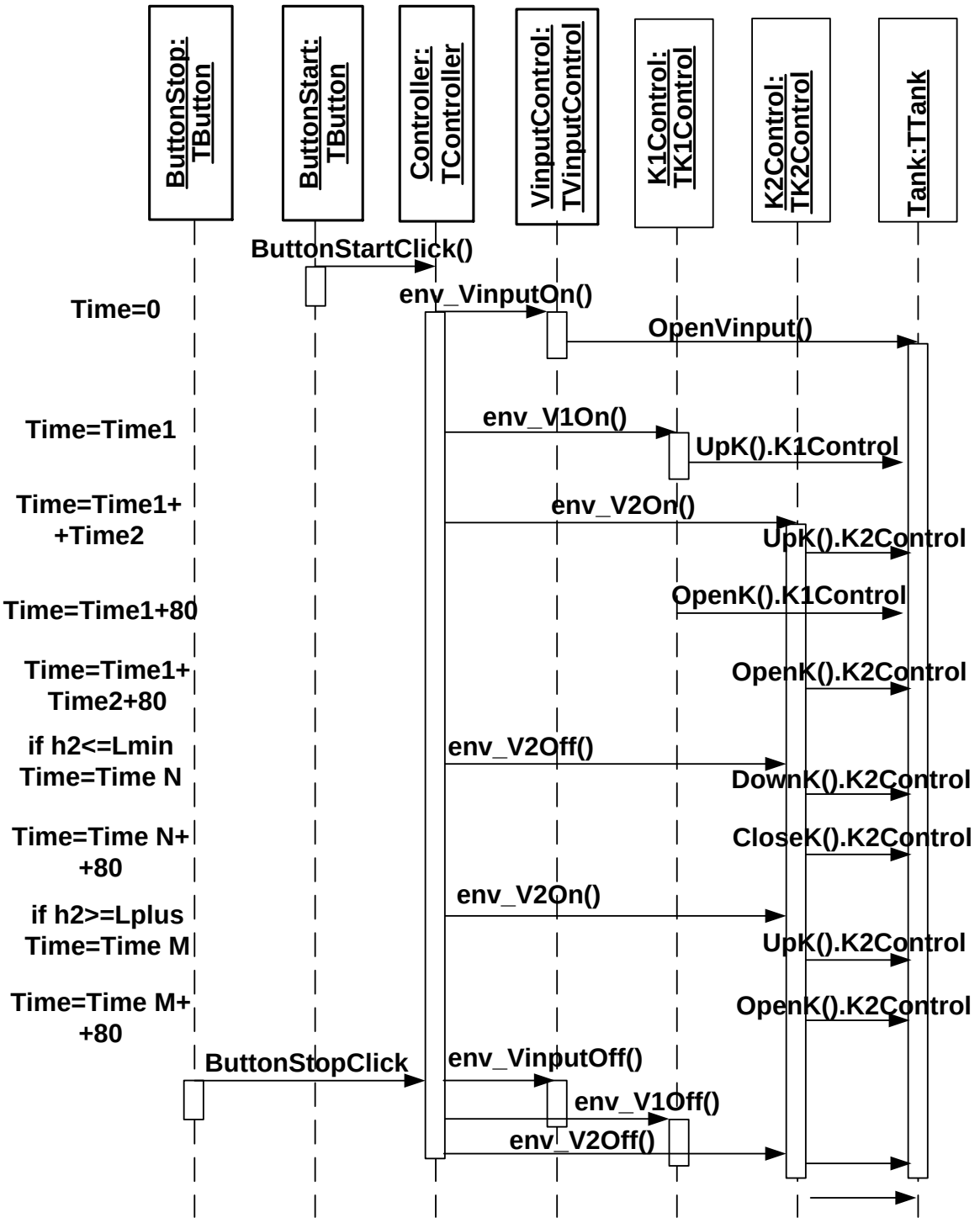

Рис. Д.1.10. Діаграма послідовності

Висновок: під час проектування використано діаграми класів, варіантів застосування, компонентів, станів і переходів, а також діаграму послідовності. Таким чином, розглянуто систему в різних ракурсах – з точки зору логічної та фізичної структур і з точки зору статичної та динамічної семантики.

Д.1.4. Опис програмного продукту

Загальні відомості. Темою розробки є задача управління потоком води через систему, яка складається з двох циліндричних баків.

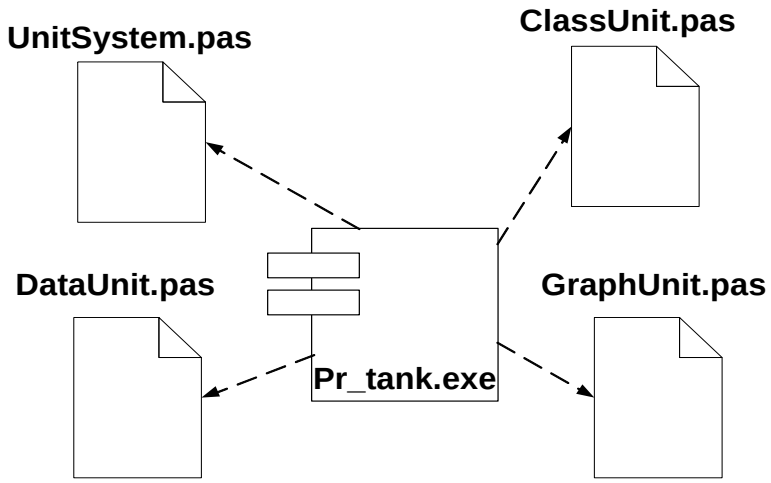

Рис. Д.1.11. Діаграма компонентів

Програма написана об'єктно-орієнтованою мовою Object Pascal в інтегрованому середовищі розробки Delphi версії 6.0.

Функціональне призначення. Програма призначена для управління функціонуванням процесів, які відбуваються в системі двох баків, та їх візуального відображення. Програма виводить на екран графіки рівнів води в баках, зображення системи в динаміці й записує дані про рівні води в баках у текстові файли (за умовчанням h1.txt и h2.txt).

Опис логічної структури. Алгоритми програми щодо роботи системи, яка складається з двох баків, найбільш зручно й наглядно подати у вигляді блок-схеми (рис. Д.1.12).

Об'єкт класу *Контролер* (TController) посилає сигнал об'єкту класу *КонтролерВхідногоКрана* (TVinputControl) відкрити вхідний кран Vinput – процедура Env_VinOn. *КонтролерВхідногоКрана* відкриває кран – процедура Open_Vin, кран відкривається миттєво й перший бак наповнюється водою. За кожну секунду відбувається розрахунок значень швидкості води у внутрішньому та вихідному кранах і значень рівнів води в двох баках(процедури класу *Бак* (TTank) – CalculV12, CalculVout, CalculH1, CalculH2).

Після закінчення часу Time1 *Контролер* посилає сигнал відкрити внутрішній кран V1 – процедура Env_V1On, *КонтролерКранів (*TKControl*)* дає команду відкрити кран – процедура UpK. Відкриття крану відбувається протягом 80 секунд – процедура OpenK, вода починає надходити до другого баку.

Після закінчення часу Time1+Time2 (від моменту нульового відліку) починається управління станом крана V2 – процедура *Контролер* Env_V2On, *КонтролерКранів* UpK, OpenK.

Якщо рівень води в другому баці стає нижчим за рівень Lmin , то кран V2 закривається – процедури *Контролер* Env_V2Off, *КонтролерКранів* DownK, CloseK, якщо рівень води стає вищим за рівень Lplus, то кран V2 відкривається – процедура H2Control. Аварійними вважаються ситуації, коли переповнюється один з баків (у цьому випадку система блокується).

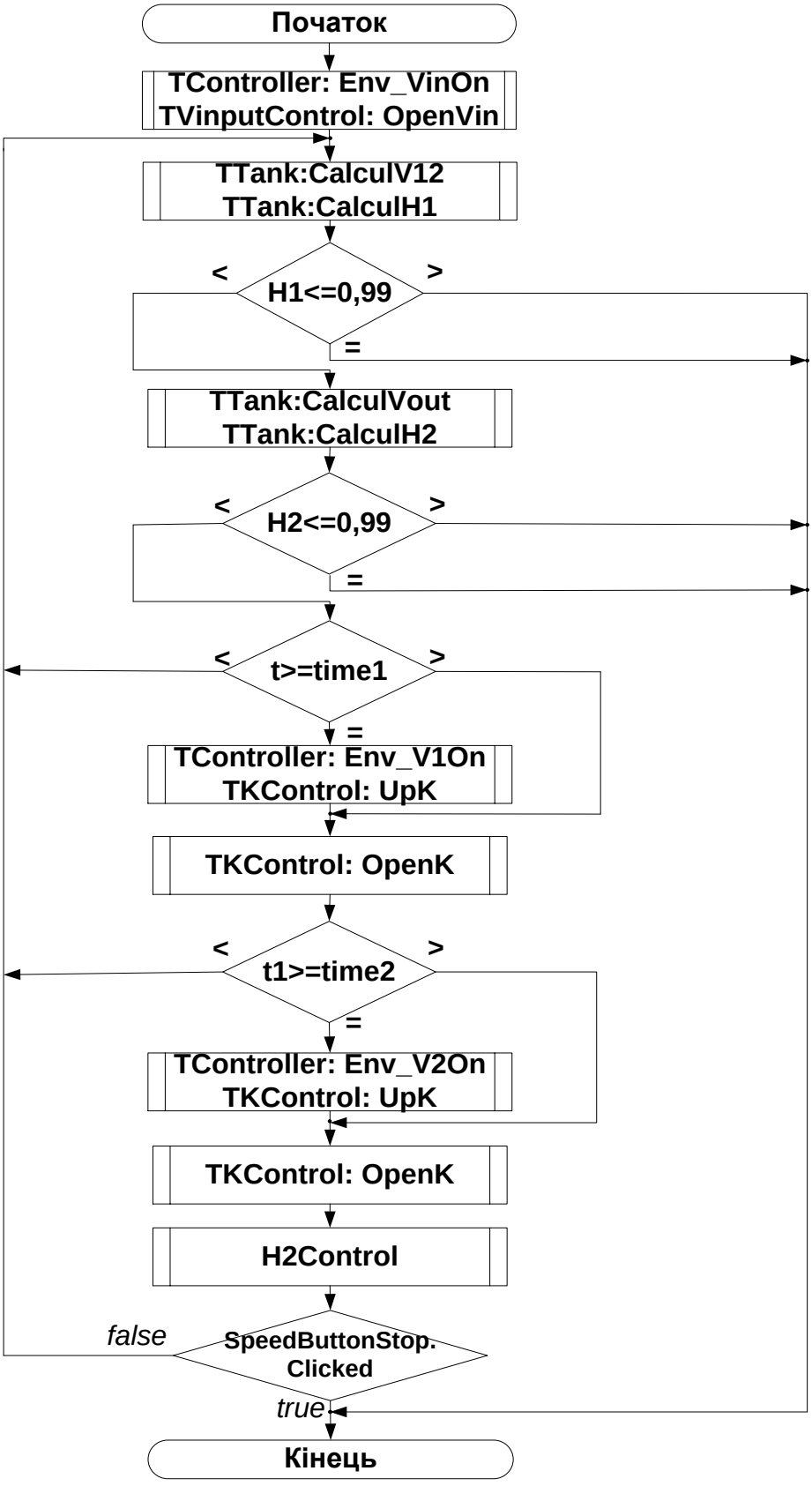

Рис. Д.1.12. Блок-схема програми

На рис. Д.1.13 показано блок-схему процедури H2Control, яка контролює рівень води в другому баці. Якщо він перевищує максимально можливе значення, то вихідний кран починає відкриватись, при рівні води нижче мінімального значення – закриватись.

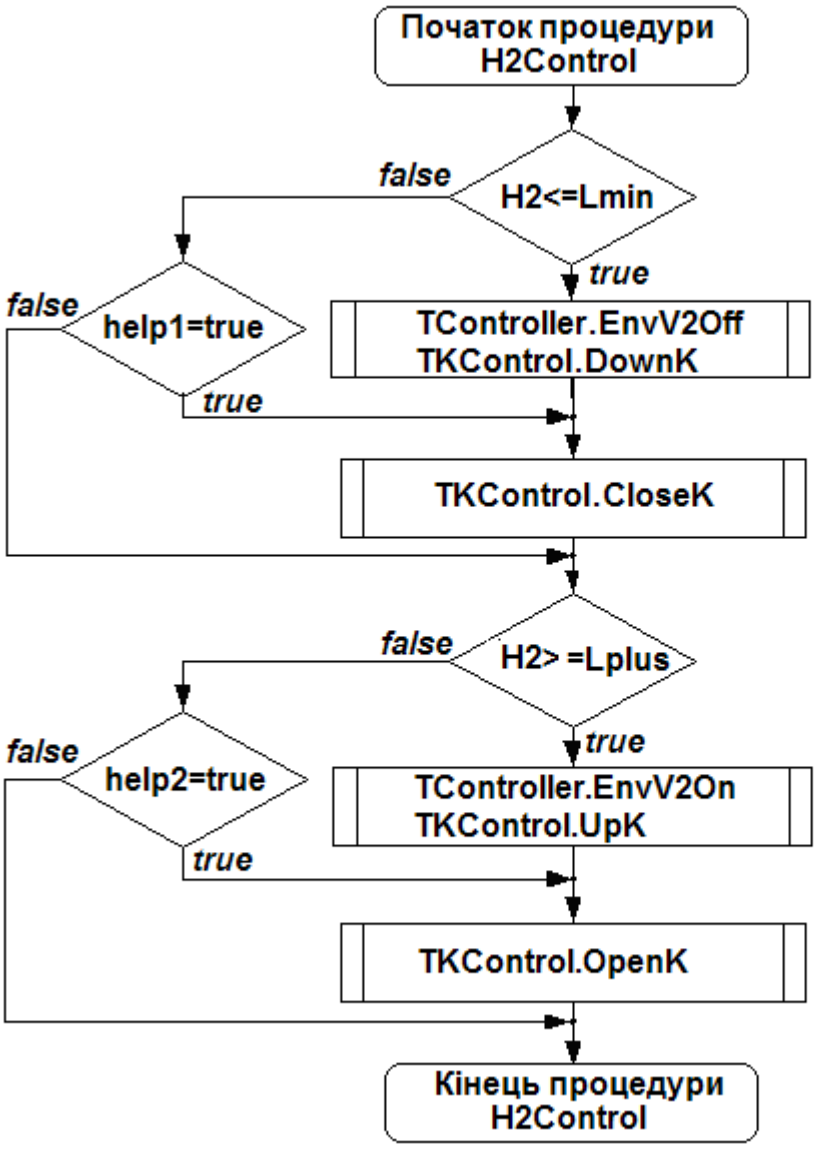

Рис. Д.1.13. Блок-схема процедури H2Control

Структура програми. Програма являє собою сукупність модулів, генерованих у середовищі *Borland Delphi* мовою *Object Pascal.* Структуру програми зображено на рис. Д.1.14.

Модуль *Unit1* містить опис основної форми комп'ютерної системи (вигляд екранної форми наведено в додатку 2) й здійснює управління викликом інших модулів і підпрограм.

Модуль *Unit_Classes_1* містить опис класів системи двох баків та їх реалізацію.

Модуль *GraphUnit* містить опис класу, який відповідає за виведення на екран графіків і зображення системи.

Модуль *DataUnit* містить процедури, які здійснюють запис значень рівнів води в баках у текстові файли, що знаходяться в каталозі \Data\ поточного каталогу.

Модуль *UnitHelp* містить опис компонентів модальної форми, в яких є довідкові дані для користувача.

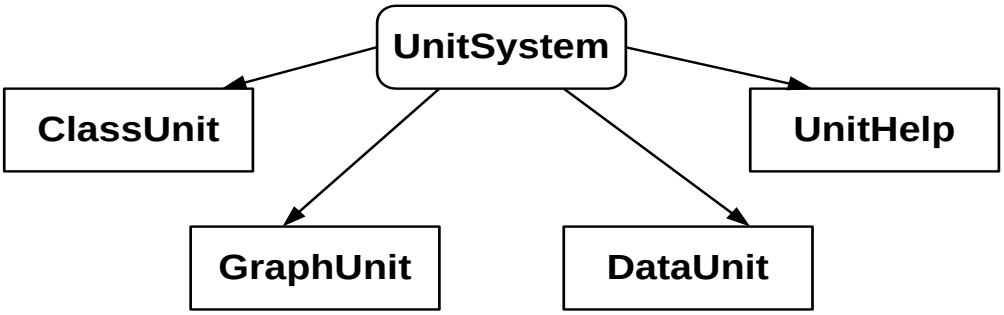

Рис. Д.1.14. Структура програми

Перелічені модулі ініціалізують такі процедури:

Модуль *Unit1*:

– процедура FormCreate (Sender: TObject) – створює форму, її ініціалізує, створює каталог \Data\ у поточному каталозі й пов'язує його з текстовими файлами, імена яких введені за умовчанням у діалогові вікна на формі;

– процедура SpeedButton1Click (Sender: TObject) – здійснює перевірку вхідних даних, дає дозвіл на початок моделювання.

Виклик та завантаження програми здійснюються шляхом запуску файла Pr_tank.exe, який знаходиться на дисковому накопичувачі або компакт-диску.

Вхідні та вихідні дані.

Вхідні дані програми:

- значення часу, із закінченням якого мусить відкритись внутрішній кран; від початку відліку; формат – integer;
- значення часу, із закінченням якого мусить відкритись вихідний кран; від моменту відкриття внутрішнього крана; формат – integer;
- значення максимально можливого рівня води в другому баці; формат – double;
- значення мінімально можливого значення рівня води в другому баці; формат – double.

Вихідні дані:

- значення рівнів води у першому баці, які записуються в текстовий файл(за умовчанням h1.txt); формат – double;
- значення рівнів води у другому баці, які записуються в текстовий файл(за умовчанням h2.txt); формат – double.

Повідомлення. При наявності критичних помилок програмою передбачені повідомлення про:

- переповнення баків (робота системи припиняється);
- спустошення баків (робота системи також припиняється).

Екранні форми повідомлень наведено в додатку 2.

Опис тестової задачі. Для тестової задачі слід прийняти:

- значення часу, із закінченням якого відкриється внутрішній кран – 10 с;
- значення часу, із закінченням якого відкриється вихідний кран після відкриття внутрішнього крана – 5 с;

- значення максимально можливого рівня води в другому баці – 0,9 м;
- значення мінімально можливого рівня в другому баці – 0,3 м.

Після того, як минуть 1138 секунд необхідно отримати такі значення характеристик системи:

- рівень води в першому баці – 0,40409 м;
- в другому баці – 0,4903 м;
- ступінь положення засувки вихідного крана – 43.

Екранна форма результатів тестової задачі наведена в додатку 2.

Висновки. Під час виконання курсового проекту проведено комп'ютерне моделювання динамічних процесів протікання води через систему, що складається з двох баків і трьох кранів, з використанням підходів об'єктно-орієнтованого аналізу та проектування.

На основі отриманих діаграм класів, об'єктів, станів і переходів, діаграми взаємодії, а також діаграми модулів і з використанням методів об'єктно-орієнтованого програмування написано програму, яка моделює роботу системи в реальному часі. За її допомогою в моменти часу, кратні секунді, можна отримати значення рівнів води в кожному з баків, значення швидкостей потоку води у внутрішньому та вихідному кранах.

Недоліком програми є те, що користувач не може самостійно відкривати або закривати крани й тим самим імітувати ситуацію вимушеної подачі або перекриття потоку води через систему, оскільки внутрішній кран відкривається, коли мине заданий проміжок часу, а стан вихідного крана регулюється автоматично.

Додаток 2

ЕКРАННІ ФОРМИ РЕЗУЛЬТАТІВ РЕАЛІЗАЦІЇ ПРОЕКТУ

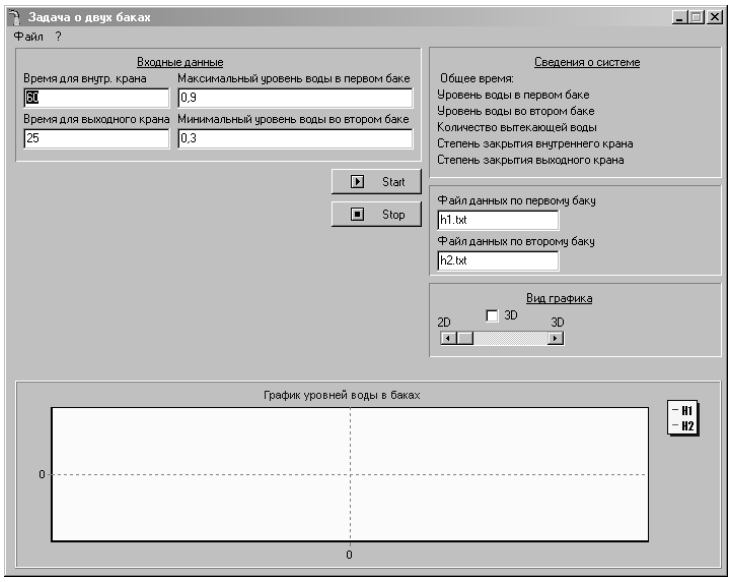

Рис. Д.2.1. Головне вікно програми

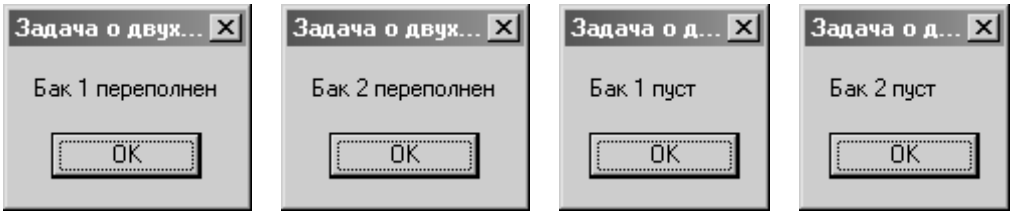

Рис. Д.2.2. Повідомлення про помилки

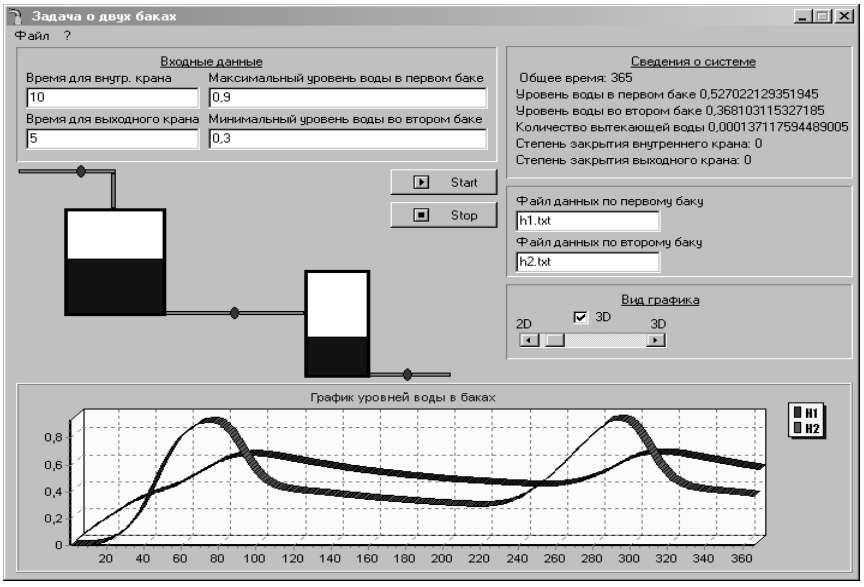

Рис. Д.2.3. Форма при виконанні тестової задачі

Додаток 3

СТРУКТУРА ЗМІСТОВНОЇ ЧАСТИНИ ПОЯСНЮВАЛЬНОЇ ЗАПИСКИ

Розділ	**Обсяг, с**
Вступ	*3*
1 Вербальна модель предметної області	*3*
2 ООА предметної області	*5*
2.1 Вибір підходу	1
2.2 Опис підходу	2
2.3 Узагальнені діаграми класів та об’єктів	2
3 OOD майбутньої програмної системи	14
3.1 Вибір технічних і програмних засобів	2
3.2 Діаграми мовою UML	2
3.2.1 Діаграма варіантів використання	2
3.2.2 Діаграма класів	3
3.2.3 Діаграма станів і переходів	3
3.2.4 Діаграма взаємодії	2
3.2.5 Діаграма реалізації	2
4 Опис програмного продукту	9
4.1 Загальні відомості	1
4.2 Функціональне призначення	1
4.3 Опис логічної структури	3
4.4 Виклик і завантаження	1
4.5 Вхідні та вихідні дані	1
4.6 Повідомлення	1
4.7 Опис тестової задачі	1
Висновки	1
ВСЬОГО	*35*

ЗМІСТ