

Міністерство освіти і науки України
Херсонський державний педагогічний університет

М.С. Львов
О.В. Співаковський

Вступ
до об'єктно-орієнтованого
програмування

Навчальний посібник

Херсон 2001

УДК 519.682.1

Львов М.С., Співаковський О.В.

Вступ до об'єктно-орієнтоване програмування. Навчальний посібник. -
Херсон: ХГПУ, 2000.- 238 с.:іл.

В систематичному вигляді викладаються основи об'єктно-орієнтованого програмування, объектно-ориентировані засоби мови системи програмування Borland Pascal і техніка їх використання, розглядаються основи методології об'єктно-орієнтованого проектування. В книзі наведена велика кількість учбових прикладів і вправ. В заключній главі пропонується орієнтована схема лабораторного практикуму з об'єктно-орієнтованого програмування.

Посібник призначений для студентів педагогічних ВУЗів, що вивчають відповідну дисципліну, учителів інформатики середніх і середніх спеціальних учбових закладів, а також для інших категорій читачів, що вивчають програмування.

1.

ВСТУП

1.1. Мета і задачі книги

Цей навчальний посібник призначений для студентів педагогічних вищих навчальних закладів – майбутніх вчителів інформатики.

Як відомо, однією з базових дисциплін, що формують фахівця – вчителя інформатики, є дисципліна «Основи алгоритмізації і програмування» (ОАП).

Саме там студенти здобувають знання, уміння і навички складання алгоритмів, їхнього опису структурною алгоритмічною мовою і реалізації в системі програмування у вигляді комп'ютерної програми. Основна увага при цьому приділяється проблемам правильності й ефективності алгоритмів, організації структур даних і управління.

Як правило, навчальними мовами програмування обираються мови Паскаль, Бейсик і відповідні системи програмування (Turbo Pascal, Qbasic і т.д.) В результаті студенти опановують елементи структурного програмування. Вони оволодівають технікою процедурного програмування «у малому», тобто здобувають той комплекс знань, умінь і навичок, що дозволяє створювати невеликі за розміром програми, що вирішують конкретні задачі опрацювання інформації. Структура таких програм, як правило є досить простою «введення – обчислення – виведення».

Разом з тим стає очевидним, що цих знань явно недостатньо для написання «великих» (чи «реальних») прикладних програм, тобто програм, які можна порівнювати з комерційними додатками. Виявляється, що існує великий якісний розрив між тими комп'ютерними програмами, що студенти пишуть самі, і тими, якими вони користуються на практиці, працюючи за сучасним комп'ютером. Ще кілька років назад цей розрив можна було усунути, використовуючи методи модульного програмування і стандартні модулі (такі, як Turbo Vision).

Зараз, у результаті переходу до сучасних об'єктно-орієнтованих операційних систем типу Windows і до відповідних об'єктно-орієнтованих технологій, цей розрив уже не можна ліквідувати, залишаючись у рамках старих методів і технологій програмування.

Сучасна комп'ютерна програма під Windows у принципі не може бути «малою», тобто мати структуру типу «введення – обчислення – виведення». Вона завжди використовує, як мінімум, стандартні об'єкти Windows, і, саме головне, має об'єктно-орієнтовану архітектуру. Для створення такої програми володіння структурним стилем програмування і знання процедурних технологій у принципі недостатньо.

Для того, щоб одержати представлення про методи проектування сучасних програмних систем, майбутній вчитель інформатики повинен вивчити і теоретичні основи сучасної методології об'єктно-орієнтованого програмування (ООП), і оволодіти практичними навичками програмування в об'єктах. При цьому він повинен побачити, що об'єктно-орієнтоване програмування – логічне продовження і розвиток процедурного програмування, наступний ступінь в його освіті, пов'язаний з переходом до програмування «у великому».

Розподіл обов'язків між курсами ОАП і ООП виявляється наступним: в ОАП вивчаються методи реалізації поведінки окремих об'єктів, а в ООП – методи реалізації їхньої взаємодії в процесі функціонування програмної системи.

Таким чином, навчальна дисципліна «Основи об'єктно-орієнтованого програмування» є необхідним і логічним продовженням дисципліни «Основи алгоритмізації і програмування». Вона продовжує спеціальну теоретичну підготовку майбутнього вчителя інформатики і програмування, закладаючи фундамент для вивчення конкретних технологій програмування під Windows.

1.2. Про історію розвитку методів проектування програм

Як створювати великі комп'ютерні програми? Фахівець початку 50-х років на це питання відповів би так: «Потрібно використовувати бібліотеки стандартних підпрограм». Пізніше такий підхід стали називати машинно-орієнтованим. Машинно-орієнтоване програмування використовує уявлення про комп'ютерну програму як про послідовність машинних команд, що обробляє дані, розташовані в чарунках пам'яті комп'ютера

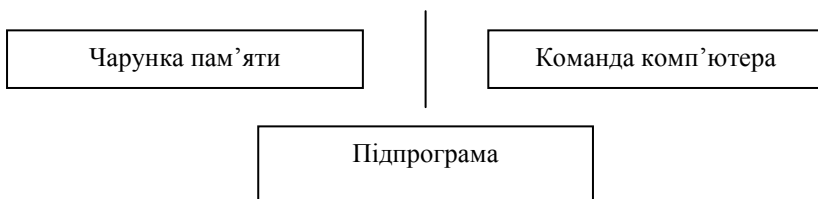


Рис. 1.1 Концепція машинно-орієнтованого програмування.

Машинно-орієнтовані мови програмування – мови асемблера – підтримують розподіл пам'яті під змінні – дані, позначені в програмі іменами, і використання підпрограм у якості основних програмних компонентів.

Програміст початку 60-х років указав би на необхідність використання мов програмування (Мови програмування високого рівня Фортран, Кобол, Алгол-60 з'явилися саме в цей час). Виявилося, що для створення великих програмних систем доцільно використовувати системи спеціальних понять, правил і рекомендацій, що формулюються як результат наукової точки зору на комп'ютерну програму. Дотримання таких понять і правил називають стилем програмування (підходом, методологією програмування).

Розробка і наукове обґрунтування різних стилів програмування – одна з основних задач теорії програмування. Виявилося, що мова програмування повинна відповідати стилю програмування. Найбільш розповсюдженим підходом став так називаний структурний підхід, підтримуваний процедурними мовами програмування (Паскаль, Модула-

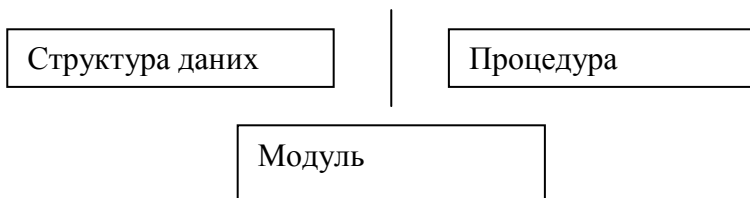


Рис. 1.2. Концепція процедурного програмування.

2, Сі й ін.).

Ключові поняття процедурного програмування – поняття структури даних, оператора і процедури (функції) як основної семантичної одиниці комп'ютерної програми, що описує алгоритм. Оскільки дані, об'єднані в структури, обробляються різними процедурами, сукупності семантично зв'язаних між собою процедур поєднуються в програмні модулі.

Спеціальні класи систем програмування створюються для застосування й інших стилів програмування, підтримуваних відповідними мовами програмування – логічного програмування (Prolog), функціонального програмування (Lisp, FP), алгебраїчного

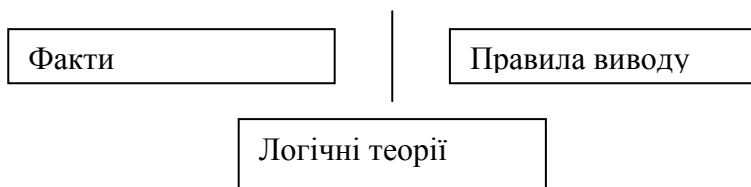
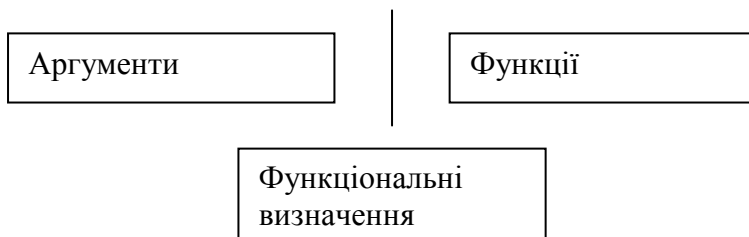


Рис 1.3. Концепція логічного програмування.

програмування (Reduce, Obj, Aplan).

Логічне програмування використовує точку зору на алгоритми, вироблену в математичній логіці. Дані логічних програм представлені фактами, що грають роль спеціальних аксіом відповідної предметної області, а алгоритми описуються у виді сукупності так званих правил логічного висновку. Таким чином, факти і правила висновку утворюють спеціальну логічну теорію. Реалізацію висновку в цій логічній теорії



називають машиною (логічного) висновку.

Рис 1.4. Концепція функціонального програмування.

Функціональне програмування являє собою підхід, при якому алгоритм описується як алгоритмічно обчислювана функція. Описи програм-функцій здійснюються в термінах функціональних визначень, а основною операцією, що реалізує обчислення функціональної програми, є операція аплікації - застосування функції до аргументу.

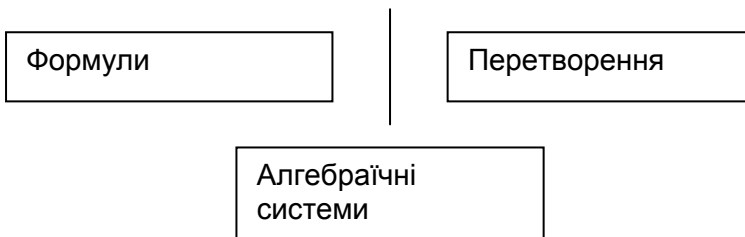


Рис 1.5. Концепція алгебраїчного програмування.

Алгебраїчне програмування використовує представлення про алгоритми, вироблене абстрактною алгеброю. Як і при алгебраїчних обчисленнях, дані алгебраїчних програм представляються формулами (алгебраїчними виразами), а їхня обробка – перетвореннями цих формул (перепикуваннями). Системи припустимих перетворень формул – даних відповідної предметної області є основними семантичними одиницями алгебраїчної програми. Сукупність правил побудови формул і правил перетворення формул визначає предметну область, що називають алгебраїчною системою.

Принципово важливим методологічним розходженням між процедурним стилем і логічним стилем опису алгоритмів є те, що процедурний стиль припускає опис алгоритму у виді послідовності кроків (дій), а логічний стиль – опис алгоритму як сукупності визначальних властивостей спеціальної предметної області.

Процедурний алгоритм *імперативний* – він *наказує* виконувати строго визначеної послідовності дій.

Логічний алгоритм *декларативний* – він *описує* логічні властивості спеціальної предметної області.

Тому процедурний стиль програмування називають *імперативним програмуванням*, а логічний стиль – *декларативним програмуванням*.

Відзначимо, що і функціональне програмування, і алгебраїчне програмування є уточненнями декларативного програмування, оскільки вони надають програмісту спеціальні засоби опису предметних областей. Функціональне й алгебраїчне програмування можна також розглядати як різні варіанти реалізації логічного підходу в програмуванні, оскільки і функціональні визначення, і алгебраїчні перетворення є спеціальними формами логічного висновку.

Дослідження різних підходів до опису алгоритмів сформували загальні концепції, про які ми говорили вище, вже в 70-х роках. Тому освічений програміст 70-х років рекомендував би дотримувати не тільки (і не стільки) визначеної мови програмування, але визначеного стилю, адекватного предметній області і структурі проектованої програмної системи.

1.2. Об'єктно-орієнтовані мови програмування

Перша об'єктно-орієнтована мова програмування Simula 67 була розроблена наприкінці 60-х років у Норвегії. Автори цієї мови дуже точно угадали перспективи розвитку програмування: їхня мова набагато випередила свій час. Найважливіші риси мови Simula 67 були помічені програмістами, і в 70-і роки було розроблено велику кількість експериментальних об'єктно-орієнтованих мов програмування: наприклад, мови CLU, Alphard, Concurrent Pascal і ін. Ці мови так і залишилися експериментальними, але в результаті їхнього дослідження були розроблені сучасні об'єктно-орієнтовані мови програмування: C++, Smalltalk, Eiffel і ін.

Найбільш розповсюдженою об'єктно-орієнтованою мовою програмування безумовно є C++. Вільно розповсюджені комерційні системи програмування C++ існують практично на будь-якій платформі. Розробка нових об'єктно-орієнтованих мов програмування продовжується. З 1995 року стала широко поширюватися нова об'єктно-орієнтована мова програмування Java, орієнтована на мережі комп'ютерів і, насамперед, на Internet. Синтаксис цієї мови нагадує синтаксис мови C++, однак ці мови мають мало загального. Java є інтерпретуємою мовою: для неї визначені внутрішні представлення й інтерпретатор цього представ-

лення, що уже зараз реалізовані на більшості платформ. Інтерпретатор спрощує налагодження програм, написаних мовою Java, забезпечує їх переносимість на нові платформи й адаптованість до нових оточень. Він дозволяє виключити вплив програм, написаних мовою Java, на інші програми і файли, що мають на новій платформі, і тим самим забезпечити безпеку при виконанні цих програм. Ці властивості мови Java дозволяють використовувати її як основну мову програмування для програм, розповсюджуваних по мережах (зокрема, по мережі Internet).

Популярним серед розроблювачів прикладних систем стало середовище візуального програмування Delphi, мовою якого є об'єктно-орієнтоване розширення Паскаля.

Об'єктно-орієнтована методологія програмування почала активно розроблятися порівняно недавно – з початку 80-х років. Затребуваним об'єктно-орієнтований підхід виявився ще пізніше – з появою ОС типу Windows. Принципи цього стилю, викладені нижче, вплинули і на методологію проектування програмних систем, і на розвиток мов програмування.

Сьогодні багато програмістів, відповідаючи на питання про те, як писати великі програми, після довгих стогонів, охв і подихів, відповідають: “використовуйте об'єктно-орієнтований підхід, і допоможе Вам Бог”. Тим самим вони повідомляють Вам про те, що писати великі програми, як і раніше, важко, але ООП – це краще, ніж те, що було раніш і, безумовно, краще, ніж нічого.

Насправді методологія об'єктно-орієнтованого проектування, безумовно, є передовою і ефективною для проектування широкого спектру великих програмних систем, таких, як інтерактивні системи, системи реального часу. Концепції ООП добре поєднуються з іншими підходами до написання комп'ютерних систем.

Методи ООП займають гідне місце й у технологіях програмування майбутнього, органічно сполучаючись з іншими (як старими, добре відомими, так і новими, ще не усвідомленими і не сформульованими) підходами до проектування великих програмних систем.

2. КОНЦЕПЦІЯ ОБ'ЄКТНО-ОРІЄНТОВАНОГО ПРОГРАМУВАННЯ

Вивчення будь-якої методології передбачає вивчення як фундаментальних, так і прикладних її аспектів. У програмуванні використовуються методології і технології, що спираються як на високонаукові ідеї, так і на розвинені прикладні системи. Для інженера важливо знати, насамперед, технологію програмування. Іншими словами, йому необхідно навчитися тим методам програмування, які він може застосовувати на практиці. Викладачу або теоретику програмування важливі, у першу чергу, ті основні ідеї і принципи, що лежать в основі технології. Однак і практикуючі програмісти, і теоретики, для того, щоб добре засвоїти один з аспектів методології, повинні знати й інший її аспект. Тому в посібнику приділяється увага як фундаментальним, так і прикладним аспектам об'єктно-орієнтованого програмування (ООП).

Методологія ООП є реалізацією змін у довгому ланцюзі рішень, які пропонувалися для подолання принципових труднощів, що виникають перед програмістами при ускладненні розроблювального ними програмного забезпечення ЕОМ. Термін "криза програмного забезпечення", який виник у середині 60-х років, означав, що потреби розробників програмного забезпечення перевершили можливості існуючих технологій.

У цьому розділі ми опишемо мовою метафор основні ідеї, принципи і позначимо методи, що утворюють у сукупності те, що прийнято називати парадигмою об'єктно-орієнтованого програмування.

Відзначимо, що ООП - це еволюційний етап у розвитку методології програмування, що спирається на ті ідеї структурного і модульного програмування, які виявилися плідними і перспективними для розвитку.

Разом з тим ООП - це революційний, тобто якісно новий етап у методології програмування, що використовує принципово нові ідеї і методи, які не існували раніше.

Нарешті, методологія ООП дійсно дозволяє краще, ніж старі методології, справлятися зі зростаючими потребами розроблювачів програмного забезпечення й апетитами замовників і споживачів

програмного забезпечення, хоча програмування і залишається однією з найбільш важких галузей людської діяльності. Не слід тому вважати, що подальший прогрес в області розвитку методологій програмування неможливий.

2.1. Основні поняття об'єктно-орієнтованої методології програмування

Першою важливою перевагою об'єктно-орієнтованої методології програмування, котру, як відзначає Т. Бадд [1], часто не помічають, є сила метафор. Справді, ми існуємо і діємо в термінах тих понять, що склалися в силу історичних, соціальних, юридичних і інших причин. Тому, наприклад, термін "моделювання" ми розуміємо не так, як термін "обчислення". Об'єктно-орієнтована методологія програмування пропонує описувати програмну систему як інформаційну модель реальної чи абстрактної системи. Необхідність будувати програмну систему як інформаційну модель об'єкта, модель системи об'єктів, модель взаємодії об'єктів, модель функціонування системи вже орієнтує на використання ідей, понять і методів моделювання. З'являється погляд на програмування як на побудову моделі деякого світу, населеного взаємодіючими об'єктами. Таким чином, процес програмування (аналіз, проектування і реалізація) системи метафорично виглядає як побудова об'єктів, визначення методів їхньої взаємодії і функціонування. Після того, як усе це зроблено, залишається тільки натиснути клавішу Enter.

Об'єктно-орієнтована методологія програмування виробила свою систему понять-метафор, саме використання якої і є принципово важливим нововведенням у програмуванні. Ця система інтуїтивно зрозуміла і знайома. Програмуючи, ми спираємося на свій досвід і інтуїцію, використовуємо своє розуміння термінів-метафор об'єктно-орієнтованої методології, і виявляється, що це розуміння добре узгоджується з формалізмом опису програмного коду.

2.2. Об'єкти. Атрибути, методи, властивості

Поняття об'єкта - одна з основних категорій об'єктно-орієнтованого програмування. Об'єктами називають інформаційні моделі реальних чи

абстрактних об'єктів, для яких розробляється відповідне програмне забезпечення.

Якщо, наприклад, система, що проектується, повинна підтримувати взаємодію клієнта і банку, основними об'єктами цієї системи будуть Клієнт і Банк. Ці об'єкти являють собою інформаційні моделі відповідно банку і (юридичної чи фізичної) особи, що користується його послугами.

Як інформаційна модель, об'єкт зберігає в собі ту інформацію, що необхідна для реалізації системи, що проектується.

Дані об'єкта називають атрибутами.

Методи структурування інформації (представлення інформації в таких структурах даних, як записи, списки, масиви, файли і т.і.) були розвинені і використовувалися іншими методологіями програмування, насамперед, структурним програмуванням.

Принципово новим і важливим підходом методології ООП виявилось включення в структуру об'єкта так званих методів (операцій обробки даних).

Інформаційна модель геометричної фігури – об'єкт Фігура, наприклад, повинна містити як атрибути (дані, що цілком характеризують цю фігуру), так і операції обробки цих даних, які дозволяють вирішувати задачі, подані в технічному завданні. Якщо мова йде про систему, що здійснює геометричні перетворення, то до операцій можна віднести паралельний перенос, поворот, гомотетію, і т.і. Якщо ж Ваша система призначена для розв'язання обчислювальних задач, то операціями будуть, зокрема, операції обчислення площі, периметра й інших чисельних характеристик геометричної фігури.

Реалізацію (інтерпретацію) операції над об'єктом називають методом.

Тим самим у термінології підкреслюється різниця між поняттям операції над даними (що зробити) і поняттям методу (як зробити). Семантичну одиницю програми, оформлену відповідно до синтаксису об'єктно-орієнтованої системи програмування як метод, включають в опис об'єктного типу.

Таким чином, операції обробки даних повинні бути віднесені

Об'єкт

Атрибути (данні)

Методи (операції)

(приписані) об'єктам. Відзначимо, що методологія процедурного програмування не орієнтує програміста на таке структурування системи.

Рис 2.1. Об'єкт як об'єднання даних і операцій

Зауважимо, що структурний підхід передбачає тільки опис безадресних процедур та функцій, не пов'язує їх до структур даних. Процедури, таким чином, лише передають одна одній потоки даних. Методологія ООП ж, навпаки, основана на строгому застосуванні принципу інкапсуляції, тобто об'єднання в єдине ціле даних та методів їх опрацювання.

Під інкапсуляцією розуміють об'єднання атрибутів (даних) і методів їх опрацювання в об'єкті.

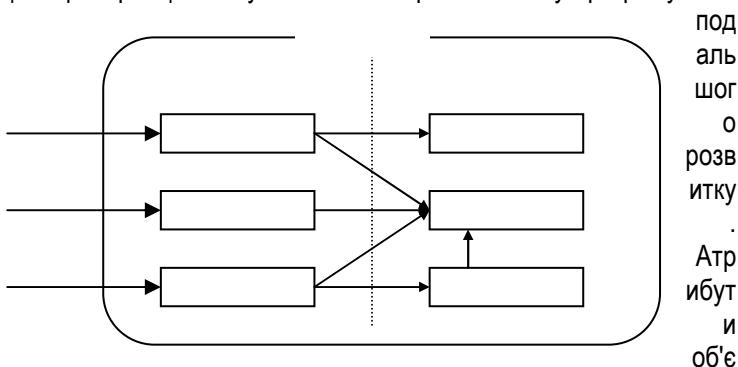
Методи об'єкта дозволяють визначати так звані *властивості* цього об'єкта.

Властивостями називають ті дані про об'єкт, які можна отримати в результаті застосування методів.

Поняття властивості об'єкта, таким чином, не співпадає з поняттям атрибута. Так, наприклад, площа геометричної фігури може не входити до списку атрибутів об'єкта Фігура, але її можна обчислити, застосувавши відповідний метод. З іншого боку, деякі атрибути об'єкта можна приховати, якщо не визначити метод їх одержання.

Повний список доступних методів і властивостей об'єкта називають його протоколом.

Принцип приховання інформації, розвинений у рамках структурного підходу, полягає в тому, щоб розділити програмний модуль на інтерфейсну і реалізаційну частину, надати в розпорядження користувача цього модуля інтерфейсний протокол і приховати від нього реалізацію. Цей принцип набув в об'єктно-орієнтованому програмуванні



кта і деякі методи, оголошені як приватні, невідомі поза об'єктом. Користувач об'єкту може взаємодіяти з ним тільки за допомогою загальнодоступних методів, одержуючи від об'єкта властивості і змінюючи їх, якщо це дозволено (тобто, якщо існують методи зміни цих властивостей).

Рис 2.2. Принцип інкапсуляції даних і операцій об'єкта.

2.3. Взаємодія об'єктів системи: повідомлення і розподіл обов'язків.

Програмні системи, що моделюють реальні процеси, як відзначалося вище, утворюють світ, населений об'єктами. Об'єкти зобов'язані взаємодіяти. У цьому полягає сенс їх існування (життєве призначення). Взаємодія об'єктів полягає в обміні повідомленнями.

Повідомленням називають запит об'єкта-відправника, спрямований об'єкту-одержувачу (виконавцю) на виконання деякої операції.

Таким чином, в акті взаємодії завжди є Відправник (клієнт) і Одержувач повідомлення (сервер). Кожне повідомлення містить доручення виконати конкретну дію, і об'єкт-одержувач зобов'язаний виконати цю дію одним зі своїх методів і, якщо це необхідно, повернути Відправнику повідомлення - результат. Повідомлення може містити параметри – дані, необхідні Одержувачу для виконання доручення.

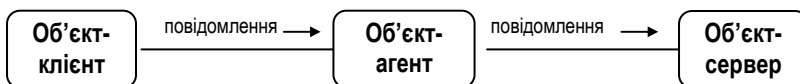


Рис. 2.3. Принцип взаємодії об'єктів – обмін повідомленнями.

Так, об'єкт *Покупець*, може надіслати об'єкту *Банк* повідомлення з Дорученням перевести на рахунок об'єкта *Магазин* суму N. Параметрами повідомлення будуть число N і юридична адреса об'єкта *Магазин*.

Коректність у взаємодії об'єктів полягає в тому, щоб Відправник повідомлення доручав Одержувачу виконувати ті дії, що він у змозі виконати, а Одержувач – обов'язково виконував ці дії. Реалізація методів Одержувача прихована від Відправника повідомлення. Зокрема, Одержувач може, в свою чергу, виступити в ролі відправника повідомлення третьому об'єкту, передоручивши йому, таким чином, виконання частини своїх обов'язків. Зрозуміло, наприкінці такого ланцюжка розподілу обов'язків повинний знаходитися об'єкт, що виконує всю доручену йому роботу самостійно.

2.4. Функціонування об'єктів системи: стани і поведінка

Однією з основних метафор об'єктно-орієнтованого програмування є погляд на об'єкти як на "живі", тобто активно діючі істоти. Наприклад, об'єкт *Куля* (для гри в більярд) може самостійно пересуватися, виявляти перед собою іншу кулю, лузу, повертати, відбиваючи від стінки, і т.і. Поведінка об'єкта залежить від його стану. Це означає, що деякі методи об'єкта можуть призводити до різних дій у залежності від того, у якому стані знаходиться об'єкт.

Стан об'єкта цілком визначається сукупністю значень його атрибутів. Серед атрибутів, як правило, є такі, значення котрих не впливають на поведінку об'єкта, на його реакції на повідомлення і такі, значення яких керують поведінкою об'єкта. Так, наприклад, реакція системи – графічного редактора на повідомлення *Нарисувати фігуру* не залежить від того, які фігури уже нарисовані. Реакція банку на одержання платіжного доручення від клієнта А не залежить від фінансового стану іншого клієнта. Однак, банк повинен контролювати рахунок клієнта, що відправив платіжне доручення (цей рахунок може бути просто заблокований). Тому об'єкт *Рахунок Клієнта*, що є частиною об'єкта *Банка*, крім змінної *Сума*, повинний містити логічну змінну *Відкритий*, значення якої керує поведінкою *Банку*.

Дуже часто метод об'єкта виконується за декілька кроків, на кожному з яких об'єкт приймає рішення про зміну свого стану, і, отже, зміну поведінки. З погляду теорії, об'єкт є автоматом з виходом. Таким чином, задачі опису поведінки взаємодіючих об'єктів розв'язуються методами теорії автоматів.

2.5. Класи об'єктів. Спадкування і перевизначення методів

Використання типів даних, визначених програмістом - одна з характерних рис структурного стилю програмування. Саме типізація відрізняє сучасні процедурні мови програмування (Модула, Паскаль, Сі, і т.д.) від своїх попередників. Тому визначення об'єкта як змінної деякого об'єктного типу є розвитком методології структурного стилю програмування.

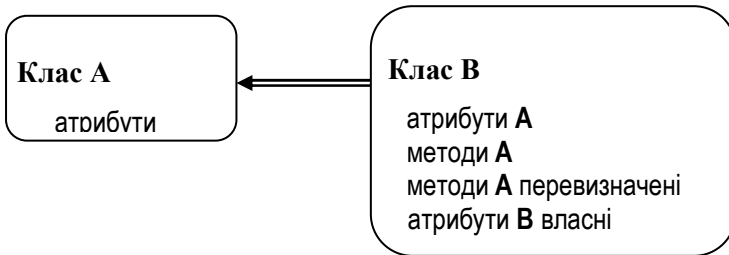
В об'єктно-орієнтованому програмуванні об'єктні типи звичайно називають класами, а змінні цих типів – екземплярами класів.

Револьюційною ідеєю об'єктно-орієнтованої методології програмування є використання поняття спадкування як основного методу опису класів.

Під спадкуванням розуміють таке відношення двох класів (об'єктних типів), при якому одному з них (дочірньому) приписуються атрибути, властивості і методи іншого (батьківського).

У найбільш повному вигляді - у вигляді відкритого спадкування, якщо А і В – класи (об'єктні типи) і В успадковує А, це означає, що всі атрибути А, за означенням, є атрибутами В, і всі методи А, за означенням, є методами В.

При закритому спадкуванні клас В одержує від класу А в спадщину



тільки виділену частину атрибутів і методів.

Клас А можна назвати класом-предком, В – нащадком. Наприклад, клас (об'єктний тип) Людина може породити об'єктні типи *Студент* і *Викладач*. У цьому випадку говорять, що клас *Студент* (відповідно, *Викладач*) успадковує клас *Людина*.

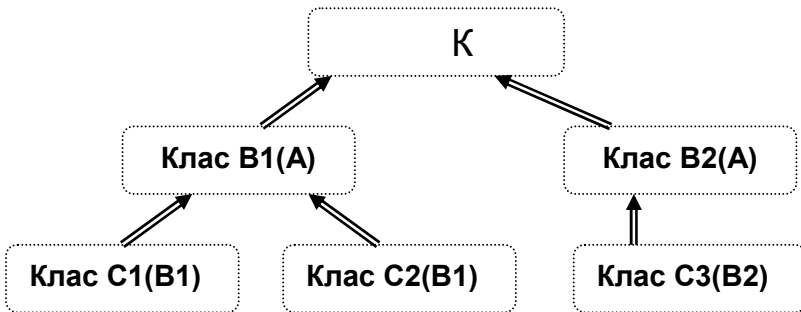


Рис 2.4. Принцип спадкування класів.

Методологія породження нових об'єктів за допомогою механізму спадкування використовується для створення ієрархій класів, які по суті є генеалогічними деревами.

Рис 2.5. Фрагмент ієрархії класів.

Відмітимо, що в цьому означенні класи є одностатевими. Насправді, багато з об'єктно-орієнтованих мов програмування підтримують і так зване множинне спадкування, що означає можливість мати декількох батьків.

Верхні рівні ієрархічного дерева часто представлені так званими абстрактними класами, тобто такими об'єктними типами, які не можна (і не має сенсу) використовувати для створення і використання змінних (екземплярів класів). Так, якщо Ваша система підтримує задачі АСУ ВНЗ, для неї не потрібні об'єкти типу Людина, оскільки всі її об'єкти – або студенти, або викладачі, або інші категорії осіб, що працюють чи вчаться у даному навчальному закладі. Всі вони – об'єкти спеціалізованих класів, дочірніх для класу *Людина*. Таким чином, абстрактні класи використовуються тільки для визначення спеціалізованих класів.

Механізм спадкування забезпечує спадкоємність властивостей і методів. Отже, спадкування, забезпечує подібність структури і поведінки предка і нащадка. Однак, разом зі спадкоємністю, необхідно підтримувати і мінливість. Як відомо, саме мінливість видів забезпечує прогрес популяції. Тому об'єктно-орієнтована методологія підтримує і цей механізм, допускаючи перевизначення методів, а також їхнє пригноблення. Метод дочірнього класу, що має те ж ім'я, що і метод батьківського класу, може виконувати над об'єктом зовсім іншу за змістом операцію.

Той факт, що для різних об'єктів можуть бути визначені однойменні методи, означає, що інтерпретація цих методів (операції, що реалізуються методами), залежать від об'єкта, якому адресоване повідомлення. Це звільняє програміста від необхідності вводити в програму нові імена для схожих за змістом операцій, тому програма стає більш прозорою, точною, яка легко читається і змінюється.

Механізм спадкування дозволяє використовувати спільний код, реалізований у методах батьківського класу, а механізм пере-

визначення методів (так званий поліморфізм), дає змогу модифікувати цей код, задовольняючи потреби програміста в реалізації індивідуальних особливостей поведінки об'єктів.

2.6. Класи – шаблони. Проблема розробки універсального програмного забезпечення

Однією з основних перешкод на шляху подолання кризи програмного забезпечення є розробка такої методології і технології програмування, що дозволяла б створювати багаторазово використовувані програмні модулі і компоненти. Та специфіка, якою повинен відзначатися конкретний додаток, займає відносно мало місця в проекті програмної системи в порівнянні зі стандартними, неодноразово реалізованими в минулому загальними місцями. Іншими словами, праця програміста дотепер є сізифовою: він неодноразово робить те, що колись уже було зроблено або ним, або його колегами.

Серед декількох причин, що обумовили таке положення справ, відзначимо одну істотну: традиційні технології програмування, що спираються на типізацію даних, не дозволяють просто реалізувати універсальний програмний модуль, тобто такий модуль, що однаково придатний обробляти подібні, але різнотипні дані. В об'єктно-орієнтованому програмуванні проблему універсалізації алгоритмів вирішують за допомогою поняття класу-шаблону (параметризованого класу), параметрами якого є типи даних, що опрацьовуються.

В основу реалізації механізму параметризації закладено принцип відокремлення загальних, універсальних методів, визначених у класі-шаблоні, від спеціалізованих, залежних від типів даних методів, які визначаються в дочірніх спеціалізованих класах.

2.7. Принципи об'єктно-орієнтованого проектування

Т. Бадд [1] приводить наступні правила - положення А. Кея, що є фундаментальними характеристиками об'єктно-орієнтованого програмування:

- Усе є об'єктом.

- Обчислення здійснюються шляхом взаємодії (обміну даними) між об'єктами, при якому один об'єкт вимагає, щоб інший об'єкт виконав деяку дію. Об'єкти взаємодіють, посилаючи й одержуючи повідомлення. Повідомлення – це запит на виконання дії, доповнений набором аргументів, що можуть знадобитися при виконанні дії.
- Кожен об'єкт має незалежну пам'ять, що складається з інших об'єктів.
- Кожен об'єкт є представником класу, що виражає загальні властивості об'єктів.
- У класі задається поведінка (функціональність) об'єкта. Тим самим всі об'єкти, що є екземплярами одного класу, здатні виконувати однакові дії.
- Класи організовані в єдину деревоподібну структуру з загальним коренем, що називається ієрархією спадкування. Пам'ять і поведінка, пов'язана з екземплярами визначеного класу, автоматично доступні будь-якому класу, розташованому нижче в ієрархічному дереві.

Об'єктно-орієнтоване програмування – це особливий погляд на світ, ціла філософія. Її основні закони, концепції, категорії так, як їх бачать теоретики об'єктно-орієнтованого програмування, коротко будуть викладені нижче. Більш повний і докладний виклад можна знайти в [1]-[3].

Послідовне введення в методологію ООП, побудоване за принципом “від простого до складного”, ми будемо супроводжувати описом об'єктно-орієнтованих засобів мови системи Borland Pascal 7.0. Читач повинний бути підготовлений до роботи з цією книгою: він повинний знати мову Паскаль і вміти програмувати “у малому”.

3. АБСТРАКТНІ ТИПИ ДАНИХ І МОДУЛІ

3.1. Абстракції даних

Проблема проектування комп'ютерної програми, якщо її розглядати як проблему опису властивостей і методів відповідної предметної області, відіграє центральну роль у теорії і практиці алгоритмізації. Саме їй присвячені практично всі наукові дослідження і навчальні курси з програмування. Концептуально важливими теоретичними поняттями, виробленими в рамках структурного програмування, стали поняття структури даних і абстрактного типу даних (АТД).

Сукупності даних, що обробляються алгоритмами, прийнято називати структурами даних.

Структури даних (способи опису упорядкованих даних) впливають не тільки на ефективність алгоритму, але також і на простоту його розуміння і програмної реалізації. Структури даних є головними будівельними блоками, з яких формуються алгоритми. Алгоритмічні структури даних повинні, з одного боку, адекватно описувати ту предметну область, над якою визначається програмне середовище, з іншого боку – ефективно відображатися в універсальні структури даних (наприклад, в структуру комп'ютерної пам'яті). Ці в якомусь ступені суперечливі вимоги приводять до наступного, більш формального розуміння терміна «структура даних».

Структура даних складається з трьох основних компонентів:

- Набір предметно-орієнтованих операцій для обробки специфічних типів абстрактних об'єктів описуваної предметної області.
- Структура пам'яті, у якій зберігаються дані, що описують абстрактні об'єкти.
- Інтерпретація (реалізація) кожної з операцій у термінах структури пам'яті.

Перший компонент визначення – набір операцій над абстрактними об'єктами – називається абстрактним типом даних. Другий і третій компоненти разом утворюють реалізацію структури даних.

АТД визначає, що робить структура даних - які операції вона підтримує, не розкриваючи, як вони виконуються.

Приклад 3.1. АТД Планіметрія

Абстрактний тип даних цього приклада призначений для програмної системи, що підтримує такий розділ геометрії, як задачі на побудову. Користувач цієї програми має можливість або самостійно вирішувати задачі на побудову, або знайомитися з ходом їх рішення, що демонструється програмою.

Примітивні типи об'єктів: точка, пряма, окружність.

Примітивні операції:

Line: (точка, точка) \rightarrow пряма

$l = \text{Line}(A, B)$ визначає пряму l , що проходить через точки A, B

Circle: (точка, точка) \rightarrow окружність

$o = \text{Circle}(A, B)$ визначає окружність o з центром у т. A , що проходить через т. B .

CircleRad: (точка, точка, точка) \rightarrow окружність

$o = \text{CircleRad}(A, B, C)$ визначає окружність o з центром у т. A , побудовану розчином циркуля з ніжками, встановленими в B, C .

IntersectLines(пряма, пряма) \rightarrow точка

$A = \text{Intersect}(l, m)$ визначає точку перетинання прямих l і m .

IntersectCircles(окружність, окружність) \rightarrow (точка, точка)

$(A, B) = \text{IntersectCircles}(o, p)$ визначає пари точок A, B перетинання окружностей o і p .

IntersectLineCircle(пряма, окружність) \rightarrow (точка, точка)

$(A, B) = \text{IntersectLineCircle}(l, o)$ визначає пари точок A, B перетинання прямої l і окружності o .

З примітивних типів об'єктів АТД Планіметрія, як з базових, будуються так звані похідні складені типи. Наприклад, тип відрізок можна визначити як пари точок, а тип трикутник - як трійку точок (вершин):

Відрізок = (точка, точка)

Тим самим відрізок визначається в системі як пари точок – початок і кінець відрізка.

Трикутник = (точка, точка, точка)

Трикутник у системі визначається точками – своїми вершинами.

Для цих типів, у свою чергу, визначаються операції. Наприклад, операції

GetBeginPoint(Відрізок) → Точка

GetEndPoint(Відрізок) → Точка

виділяють відповідно точки – початок і кінець відрізка.

З примітивних (базових) операцій АТД можна визначати похідні операції. Наприклад, операція

ParrallelLine: (точка, пряма) → пряма,

результатом якої є пряма, що паралельна даній прямій і проходить через дану точку, може бути визначена через примітивні операції і використана потім в алгоритмах розв'язання задач на побудови. Відзначимо, що реалізацію (інтерпретацію) АТД можна здійснювати окремо, використовуючи структури даних, орієнтовані на структуру пам'яті комп'ютера. Одне з можливих рішень – введення і використання декартової системи координат на площині. Це рішення відноситься тільки до реалізації, оскільки в задачах на побудову системи координат не використовуються.

Таким чином, ядро програмної системи, що вирішує поставлені задачі, має три рівні:

- рівень опису додатків АТД Планіметрія;
- рівень опису АТД Планіметрія;

- рівень реалізації АТД Планіметрія.



Рис 3.1. Логічна структура ядра системи Планіметрія.

Середній рівень цієї структури утворює інтерфейс між верхнім і нижчим її рівнями. Додатки тепер можна програмувати в термінах АТД Планіметрія, не вникаючи в проблеми нижнього рівня, пов'язані з реалізацією.

Використання концепції абстрактних типів даних при проектуванні програмних систем у багатьох системах програмування підтримується засобами модульного програмування.

Модульний підхід до алгоритмізації (програмування), полягає в декомпозиції проектованої системи в сукупність окремих модулів (бібліотек) з добре відпрацьованим інтерфейсом.

3.2. Модулі середовища Borland Pascal

Модулі - це засіб підтримки модульного стилю проектування програм, основною концепцією якого є проектування програми у виді ієрархії модулів (бібліотек). Модулі верхніх рівнів ієрархії коректним чином використовують засоби, описані в модулях нижніх рівнів.

Коректність використання, як уже відзначалося, означає приступність інтерфейсних засобів і прихованість засобів реалізації. Ще однією важливою перевагою модуля є його універсальність. Реалізований один раз, модуль стає засобом не тільки тієї прикладної програми, для якого він розроблявся, але і засобом всієї системи

програмування (звичайно, тієї її копії, що містить цей модуль). Тому про модулі говорять як про бібліотеки процедур, функцій і об'єктів.

Опишемо стисло поняття програмного модуля системи програмування Borland Pascal.

Модуль (Unit) складається з заголовка і тіла.

<Модуль> ::= <Заголовок> ; <Тіло> .

<Заголовок> ::= Unit <Ім'я> {(Список параметрів)}

<Тіло> ::= Interface

<опис інтерфейсу>

Implementation

<опис реалізації>

<опис ініціалізації>

<Опис інтерфейсу> ::=

{<Розділ Uses >}

{<Розділ констант>}

{<Розділ типів>}

{<Розділ змінних>}

{<Розділ заголовків процедур і функцій>}

<Опис реалізації> ::=

{<Розділ Uses >}

{<Розділ міток>}

{<Розділ констант>}

{<Розділ типів>}

{<Розділ змінних>}

{<Розділ процедур і функцій>}

<Опис ініціалізації> ::= end | <розділ операторів>

Використання модулів при розробці прикладної програми призводить до того, що Ваша програма виявляється розташованою в декількох програмних файлах. Для того, щоб усе це працювало, взаємодіючи один з одним, необхідно знати технологію модульного програмування, реалізовану в середовищі Borland Pascal. Ми опишемо її для ОС MS DOS.

Для того, щоб використовувати уже створений модуль у прикладній програмі, необхідно “підключити” його до вихідного модуля цієї програми (імпортувати), описавши в розділі `Uses`. Наступний приклад 3.2. ілюструє таке підключення:

Приклад 3.2.

Припустимо, що Ви написали і налагодили модуль `Proba`:

Unit `Proba`;

Interface

Procedure `Print`(`S` : `String`);

Implementation

Procedure `Print`(`S` : `String`);

begin

`WriteLn`(‘ `S` = ’, `S`)

end;

end.

Далі, Ви написали прикладну програму `Prog`, що використовує модуль `Proba`:

Program `Prog`;

Uses `Proba`;

begin

`Print`(‘ Hello World’)

end.

Тепер задача полягає в тому, щоб правильно розташувати ці вихідні програмні файли у файловій системі, їх відкомпілювати і зв'язати, а також зробити модуль `Proba` універсальним. Для цього необхідно:

- Зберегти вихідний програмний файл під тим же ім'ям, яким Ви назвали модуль у спеціально створеному каталозі (наприклад,

OwnUnits). Каталог OwnUnits призначений для збереження Ваших модулів і у вихідному, і в об'єктному коді.

- Встановити в розділі Options\Directories\Objects&Units правильний шлях до цього каталогу (наприклад, C:\BP\OwnUnits).
- Виконати незалежну компіляцію модуля Proba на диск (Compile/Destination Disk/Compile/Compile). У результаті в цьому каталозі (OwnUnits) буде створений файл із розширенням .tpu, що містить об'єктний код Вашого модуля (Proba.tpu).

При підключенні стандартних модулів середовища Borland Pascal досить включити ім'я цього модуля в список імен відповідного розділу Uses. Наприклад, модуль Crt підключається до програми Prog разом з Вашим модулем Proba рядком

Uses Crt, Proba;

Таким чином, розділ Uses визначається так:

<Розділ Uses > ::= **Uses** <Список імпортованих модулів>

Відзначимо, що відношення імпортовності не є транзитивним: якщо модуль Unit1 імпортує модуль Unit2, а модуль Unit2, у свою чергу, імпортує модуль Unit3, це не означає, що в Unit1 можна використовувати засоби Unit3.

Додатково до синтаксичного опису модуля опишемо стисло призначення кожного розділу модуля і методологію використання модулів.

3.2.1. Розділ Interface

У цьому розділі означаються всі програмні об'єкти модуля, якими можуть користуватися інші модулі і програми “зовнішнього світу”, що імпортують його. Крім того, тут вказані ті модулі, що він імпортує. Взаємодія модуля з іншими програмними компонентами системи здійснюється засобами інтерфейсу.

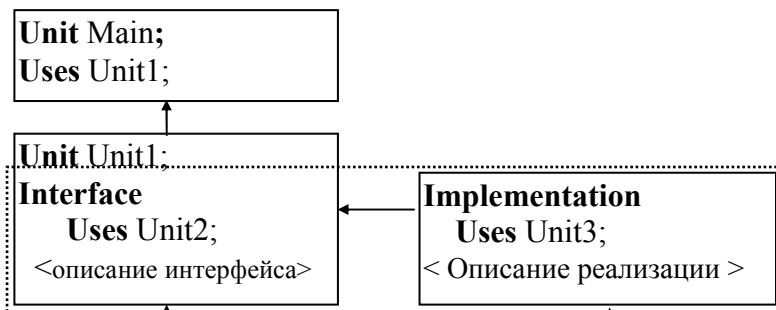


Рис. 3.2. Взаємодія модуля з компонентами системи

Таким чином, у розділі **Uses** описується список імпорту, в інших розділах (констант, типів, змінних, заголовків процедур і функцій) - список експорту модуля. Відзначимо, що в списку імпорту інтерфейсного розділу перелічуються імена тільки тих імпортованих модулів, інформація яких (тобто константи, типи і змінні) використовується в цьому розділі. Список експорту містить описи констант, типів, змінних і заголовків процедур і функцій, експортованих з цього модуля в інші програмні компоненти. Оскільки модулі створюються саме для цієї мети, інтерфейс треба проектувати особливо ретельно - як на виставку-продаж.

3.2.2. Розділ **Implementation**

Розділ **Implementation** реалізації модуля містить описи, що реалізують процедури і функції, заголовки яких винесені в інтерфейсний розділ. Усе, що описано в розділі реалізації, недоступно іншим програмним компонентам. Приховання реалізації, як уже відзначалося, - важливий засіб забезпечення надійності програмної системи.

Розділ реалізації, у свою чергу, може імпортувати інші модулі. Тому перед програмістом виникає задача класифікації імпортованих модулів на внутрішні - використовувані тільки для реалізації, і на інтерфейсні. Про те, які модулі імпортуються для внутрішніх цілей реалізації, може не знати жоден інший програмний компонент системи.

Зв'язок між розділами **Interface** і **Implementation** здійснюється через заголовки інтерфейсних процедур і функцій, що тут відіграють

роль попередніх об'яв. Тому всі інтерфейсні процедури і функції повинні бути реалізовані. Заголовки інтерфейсних реалізованих процедур і функцій можна скорочувати (тобто вони можуть не містити списку формальних параметрів). Однак, практика показує, що це не зовсім зручно. Тому ці заголовки можна описувати цілком. У цьому випадку описи заголовків в обох розділах повинні бути ідентичними. Для цього досить скопіювати список описів заголовків з інтерфейсного розділу в розділ реалізації, а потім дописати необхідні тіла.

Відзначимо, що крім інтерфейсних процедур і функцій, у розділі реалізації можуть описуватися й інші (внутрішні) процедури і функції, а також інші програмні об'єкти. Таким чином, структура реалізаційного розділу модуля практично не відрізняється від структури програми.

Розділ операторів модуля містить оператори, що виконуються при кожному виконанні того програмного компонента, що імпортував даний модуль. Більш точно, виконуються розділи операторів всіх модулів списку імпорту розділу **Uses** у тому порядку, в якому вони там описані. Тому розділ операторів модуля повинен містити оператори його ініціалізації. Зокрема, якщо розділ операторів модуля порожній, замість пари операторних дужок **begin end** можна обійтися тільки закриваючою дужкою **end**.

Важливою особливістю модуля є те, що всі програмні об'єкти (тобто мітки, константи, типи, змінні, процедури і функції), описані в ньому як глобальні, визначені тільки в цьому модулі. У зв'язку з цим з'являється поняття області видимості програмного об'єкта.

Областю видимості програмного об'єкта, описаного в модулі, є цей модуль.

Крім того, на відміну від процедури чи функції, для модуля не існує зовнішнього світу у вигляді програмного компонента "за замовчуванням". Тому, якщо у Вас виникла необхідність описати програмний об'єкт, що Ви збираєтеся потім використовувати в багатьох інших модулях, можна зробити так: визначити модуль з ім'ям типу **Universum**, **Global**, **Main**, і т.п., описати в ньому Ваші глобальні об'єкти як інтерфейсні, а потім імпортувати цей модуль туди, куди це необхідно.

Приклад 3.1. (Продовження)

Розглянемо тепер модульну структуру ядра прикладної програми
Planimetry, опис абстрактного типу якої ми привели вище.

{ модуль містить описи прикладних задач на побудову}

Unit Planimetry;

Interface

Uses PlaneObjects;

{описи заголовків процедур додатків у термінах

геометричних об'єктів}

Implementation

Uses ADTPlanimetry;

{описи процедур додатків у термінах АДТ Планіметрія}

end.

{ модуль містить описи геометричних об'єктів}

Unit PlaneObjects;

Interface

Type

{описи геометричних об'єктів: точка, пряма, окружність, відрізок,
трикутник, і т.д. }

Implementation

{описи операцій над геометричними об'єктами}

end.

{ модуль містить опис АДТ Планіметрія}

Unit ADTPlanimetry;

Interface

Uses PlaneObjects;

{описи заголовків процедур примітивних операцій у

термінах примітивних об'єктів}

Implementation

{описи примітивних операцій}

end.

Фрагмент модульної структури програмної системи приклада показаний на рис. 3.3.

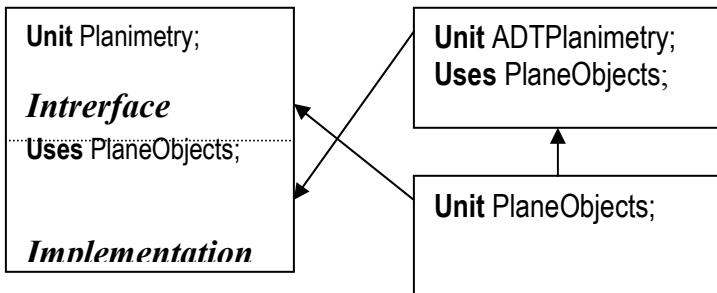


Рис. 3.3. Фрагмент модульної структури системи Планіметрія.

Як видно з нашого прикладу, поняття абстрактного типу даних дозволило здійснити логічний аналіз предметної області проєктованої програмної системи, тобто виділити її основні поняття й визначити зв'язки між ними, одержавши правильну уяву про логічну модель ядра системи.

Разом з тим, очевидно, що проєкт системи вимагає ще відпрацювання дуже багатьох уявлень і рішень: наприклад, про те, яким чином здійснювати візуалізацію операцій, що означені абстрактно, яким повинний бути інтерфейс користувача системи, якою повинна бути фізична модель системи, тобто як реалізувати розподіл обчислювальних ресурсів і керувати ними, і. т.д. Розглянемо більш детально деякі з цих проблем.

Задача візуалізації операцій

Одна з операцій АТД Планіметрія – операція **Line** описана так:

I = Line(A, B) визначає пряму I, що проходить через крапки **A, B**

У педагогічно орієнтованій програмній системі користувач повинен побачити на екрані монітора спеціальне вікно, що зображує аркуш паперу, на якому здійснюються всі побудови, точки **A** і **B**, зображені і позначені на цьому листі папера. Процес виконання операції може бути анімаційним – шкільна лінійка встановлюється на точки **A**, **B**, а потім олівець малює прямую **I**.

Так в системі з'являються моделі фізичних об'єктів – аркуша паперу, лінійки, олівця, циркуля, зображень точок, прямих, окружностей. Ці об'єкти характеризуються своїми властивостями і поведінням.

Задача проектування інтерфейсу користувача

Проектована система є інтерактивною. Отже, її корисність у вирішальній ступені залежить від того, наскільки зручно буде в ній працювати і школяреві, і вчителю. Засоби керування такою системою є окремою і винятково важливою задачею. Тому їх треба ретельно проектувати. Як, наприклад, організувати процес самостійного розв'язання задачі учнем? Які саме дії він може і повинен уміти виконувати, починаючи розв'язання задачі або виконуючи один крок розв'язання? Всі креслярські інструменти, кнопки, меню, шпаргалки і т.п., якими він користується, також є моделями фізичних об'єктів, організованих в інтерактивну систему керування, яку і називають інтерфейсом користувача.

Фізична модель системи

Всі програмні системи використовують обчислювальні ресурси комп'ютера. Наша система не є виключенням. Вона так само буде використовувати монітор, мишу, зовнішні запам'ятовуючі пристрої, принтер, і т.д. Де, наприклад, будуть зберігатися умови і розв'язки задач при виключеному комп'ютері? Яка система вікон знадобиться програмній системі? У якому вигляді геометричні побудови потрібно виводити на принтер? У задачах такого типу програміст має справу з об'єктами - моделями зовнішніх пристроїв комп'ютера, що уже мають характерні властивості і функціонування, визначені операційним середовищем.

Підводячи підсумки розглядів прикладу, відзначимо, що поняття об'єкта, що має характерні властивості і поведінку, як деякої моделі фізичного об'єкта, є найбільше адекватною метафорою для рішення задач проектування комп'ютерних програм.

3.3. Висновки

- Поняття структури даних і абстрактного типу даних є одним з концептуально важливих понять структурного програмування.
- Абстрактні типи даних використовуються для опису логічної структури програмної системи.
- Програмні модулі використовуються для побудови програми у виді структури модулів.
- Структура програмного модулю дозволяє виділити його інтерфейс – сукупність засобів, використовуваних іншими програмними компонентами. Реалізація цих засобів прихована від інших програмних компонентів системи.
- Поняття об'єкта є найбільш адекватною метафорою для рішення задач проектування комп'ютерних програм.

4. ПОНЯТТЯ ОБ'ЄКТА: АТРИБУТИ І МЕТОДИ

4.1. Об'єкти і їх описи засобами мови Borland Pascal

Об'єктно-орієнтоване програмування — це методологія програмування, що заснована на представленні програми у виді сукупності *об'єктів*, кожний з яких є реалізацією певного *класу* (типу особливого виду).

Об'єктами є, наприклад, точки декартової площини, кулі для гри в більярд, вікна, що використовуються для введення-виведення інформації, різного роду пристрої, призначені для опрацювання інформації (калькулятори, перекладачі, так звані броузери), розклад занять факультету ВНЗу, список викладачів кафедри і т.п.

Об'єкт описується своїми *атрибутами* (*інформаційними полями*) і *методами*.

4.1.1. Класи

Класи (об'єктні типи) призначені для опису сімейств об'єктів, що у системі грають однакові ролі. Тому всі об'єкти того самого класу характеризуються однаковими (однотипними) наборами атрибутів (інформаційних полів). Термін “атрибути” підкреслює змістовну сторону цього поняття, а термін “інформаційні поля” - синтаксичну. Однак об'єднання об'єктів у класи визначається не тільки наборами атрибутів, але і поведженням. Так, наприклад, об'єкти “точка площини” і “вектор” можуть мати однакові атрибути - координати. При цьому вони можуть відноситися до одного класу, якщо розглядаються в задачі просто як пари чисел, або до різних класів, якщо для розв'язання задач системи вектори потрібно складати, а точки - переміщати.

Помітимо, що необхідно розрізняти термін “клас” (об'єктний тип) і термін “екземпляр класу” (об'єкт) точно так само, як розрізняються поняття “тип” даних і “змінна” даного типу.

Синтаксис опису об'єктного типу мовою Borland Pascal являє собою узагальнення такого структурного типу, як комбінований. Повне

визначення об'єктного типу мовою НФБ буде дано пізніше. Зараз достатньо користуватися наступним визначенням:

Type {опис об'єктних типів}

<Опис об'єкта> ::=

 <Ім'я Об'єктногоТипу> = **object**

 <Список описів інформаційних полів>

 <Список описів методів>

end;

Список описів інформаційних полів об'єкта синтаксично визначається так само, як і список описів полів запису. Єдина відмінність полягає в тому, що символ “;” треба ставити після кожного опису поля чи метода.

 <Опис полів (1)>;

 <Опис полів (2)>;

 <Опис полів (k)>;

 <Опис Поля> ::= <Ім'я>:<Тип>

або

 <Опис полів> ::= <СписокІмен>:<Тип>

Приклад 4.1.

Type {----- опис об'єкта TVector -----}

TVector = **object**

 {опис полів}

 pr : Real;

 pr : Real;

 {опис методів}

procedure Init;

procedure Print;

procedure Add(U, V : TVector);

procedure Sub(U, V : TVector);

```
procedure Mult(U : TVector; C : Real);  
end;
```

Об'єктно-орієнтований стиль програмування, крім синтаксичних конструкцій мови, орієнтованих на його застосування, передбачає використання спеціальних правил-угоди, які не підтримуються мовою програмування, але додержуються програмістами. Одне з таких правил, яких дотримуються програмісти – користувачі системи Borland Pascal, застосовується в описі об'єктного типу TVector:

Велика буква T, що ставиться перед ім'ям Vector, відповідно до угоди, означає об'єктний тип.

Імена об'єктних типів, описані і використовувані в програмі, повинні починатися з великої букви T.

4.1.2. Імена об'єктів, атрибутів і методів

Імена програмних об'єктів у методології об'єктно-орієнтованого програмування відіграють особливу роль. Ім'я структурного компонента програми, крім синтаксичного і прагматичного навантаження, повинне нести ще і максимальне семантичне навантаження. Це означає, що воно повинно бути не тільки синтаксично правильним і унікальним у своїй області видимості, але і максимально точно позначати той фізичний об'єкт, інформаційна модель якого описується. Варто пам'ятати, що Ви створюєте програмні компоненти, які будуть використовувати Ваші колеги по робочій групі проекту. Тому необхідно дотримувати наступні правила-угоди "доброго стилю" об'єктно-орієнтованого програмування.

Ім'я структурного компонента програми (об'єкта, типу, поля, методу і т.п.) повинне починатися з великої букви, не містити скорочень слів і, якщо це не мотивовано, цифр. Ім'я може містити кілька слів. У цьому випадку кожне слово повинне починатися з великої букви. Ім'я повинне нести змістове навантаження, по можливості доповнюючи собою коментарі.

Приклад 4.2.

Type

{-----Data Types -----}

TDay = 1..31;

TMonth = (Jan, Feb, Mar, Apr, May, Jun, Jul, Aug, Sep, Oct, Nov, Dec);

TYear = 0..9999;

TName = String[20];

{-----}

{-----Object Types-----}

TDate = **object**

{ інформаційні поля - атрибути }

Day : TDay;

Month : TMonth;

Year : TYear;

{ заголовки методів }

Procedure Init(D:TDay; M:TMonth; Y:TYear);

Procedure Print;

end;

TPerson = **object**

{ інформаційні поля - атрибути }

Name: TName;

BirthDay: TDate;

{ заголовки методів }

Procedure Init(UserName: TName; UserBirthDay: TDay);

Procedure Print;

end;

{-----}

4.1.3. Класифікація атрибутів

об'єкта

Інформаційні поля призначені для збереження інформації про об'єкт та її використання як самим об'єктом, так і іншими об'єктами. Як ми вже знаємо, ці дані називаються атрибутами. З

точки зору використання атрибуту об'єкта можна класифікувати в такий спосіб:

- *Атрибути – ідентифікуючі характеристики об'єкта.* Ідентифікуючі атрибути об'єкта - це ті дані, що характеризують власне об'єкт. Розрізнення цих атрибутів означає відмінність об'єктів. Ідентифікуючі атрибути об'єкта, отже, приписуються йому в момент створення (народження). Вони не повинні мінятися протягом життєвого циклу цього об'єкта. Сукупність таких атрибутів називають ідентифікатором об'єкта. Ідентифікатор об'єкта повинен бути унікальним. Якщо серед атрибутів об'єкта можна обрати декілька груп атрибутів, що ідентифікують об'єкт, одна з таких груп повинна бути призначена ідентифікатором “за означенням”. У цьому випадку говорять про привілейований ідентифікатор об'єкта.
- *Атрибути – описові параметри об'єкта.* Описові параметри об'єкта визначають його стан у даний момент часу. Вони можуть змінюватися протягом його життя.

Як приклад (приклад 4.4) розглянемо опис об'єкта TBilliardBall (більярдна куля). Незмінними залишаються такі атрибути, як радіус, номер, колір і маса. Це – ідентифікуючі властивості. Атрибути, що змінюються – координати центра, вектор швидкості. Це – описові атрибути (параметри) більярдної кулі. Очевидно, що ідентифікатором більярдної кулі як об'єкта є його номер. Незважаючи на те, що радіус і маса не змінюються протягом життя кулі, вони не ідентифікують його як об'єкт, хоча б тому, що їхні чисельні значення, швидше за все, однакові для всіх більярдних куль. Деякі кулі можуть бути розфарбовані однаково, тому колір кулі тільки розділяє (факторизує) групу куль на декілька одноколірних груп.

З іншого боку, центр кулі може ідентифікувати його як об'єкт, оскільки дві кулі не можуть знаходитися в одній точці більярдного столу. Однак, оскільки координати кулі змінюються, цей атрибут не є ідентифікатором “за означенням”.

Приклад 4.4. Об'єктний тип Більярдна куля.

Type

```
BilliardBall = object
    Number      : TballNumber; {ідентифікатор}
    Center      : TPoint;       {параметр-опис}
    Radius      : Real;         {ідентифікуюча характеристика}
    Color       : Tcolor;       {ідентифікуюча
характеристика}
    Massa       : Real;         {ідентифікуюча характеристика}
    Velocity    : Tvector       {параметр-опис}
    {опис методів}
end;
```

- *Допоміжні атрибути* (інформаційні поля – посилання). Інформаційні поля посилального типу реалізують атрибути, спеціально призначені для установки і підтримки зв'язків між об'єктами. Ці зв'язки називають залежностями, а відповідні атрибути – допоміжними. (Термін “залежність” запозичений з теорії баз даних. З математичної точки зору залежність – це відношення на множині об'єктів).

Наприклад, відношення “людина X – батько людини Y” визначено на множині людей і встановлює залежність батьківства між деякими парами людей. Відношення “місто X - столиця країни Y” установлює залежність між елементами різних множин – множини міст і множини країн.

Приклад 4.5. Допоміжні атрибути. Залежності.

Type

```
TPerson = object
    Name : Tname;
    Father : ^TPerson; {допоміжний атрибут}
    {опис атрибутів}
    {опис методів}
end;
```

```

TSity = object
    Name : Tname;
    {опис атрибутів}
    {опис методів}
end;
TLand = object
    Name : Tname;
    Capital : ^TSity;           {допоміжний атрибут}
    {опис атрибутів}
    {опис методів}
end;

```

Мова програмування Паскаль строго типізована. Це означає, що кожне дане, описане і використане в програмі, має свій тип. Тому інформаційні поля, що відповідають атрибутам, класифікуються, перш за все, своїми типами. З цього погляду, інформаційні поля можуть бути:

- логічними;
- перелічувальними;
- інтервальними;
- числовими;
- символьними;
- строковими;
- структурами;
- посиланнями;
- об'єктами.

Перелічувальні типи, що визначені програмістом для опису об'єктних типів, використовуються найчастіше для опису атрибутів - якісних характеристик об'єктів. Інтервальні типи використовують для контролю діапазонів відповідних даних (визначення доменів).

Type

```

Sex = (Man, Women);
WeekDay = (Monday, Wednesday, . . .);
Month = 1..12;

```


Логічні поля визначають логічні властивості об'єктів, числові – їхні чисельні параметри і характеристики.

Строкові поля несуть текстову інформацію про об'єкт. Часто зустрічаються ситуації, коли значення строкових атрибутів можна так чи інакше закодувати. Числовий код відповідного атрибута в цьому разі включається до переліку атрибутів, а операції кодування і декодування реалізуються методами цього об'єкта.

Області визначення об'єктів називають доменами.

Домени, отже, визначають множини припустимих значень атрибута.

Визначення домену, взагалі кажучи, більш загальне поняття, ніж його тип, що означається в описі. Домени можуть визначатися ще і так званим *інваріантом* (наприклад, посиланням на так чи інакше реалізований перелік своїх значень або правилом, що визначає допустимість атрибута і реалізований у вигляді логічної функції).

Наприклад, множина значень атрибута Місто_країни може зберігатися в спеціальному документі Список_міст.

Правилом, що визначає домен атрибута Центр об'єкта Більярдна куля, є система нерівностей

$$0 \leq X.Center \leq BilliardTableSize,$$

$$0 \leq Y.Center \leq BilliardTableSize.$$

Зазначимо, що інваріанти можуть установлювати домени не тільки для окремих атрибутів, але і груп атрибутів, а також для всього об'єкта. Так, якщо більярдний стіл має форму кола, інваріантом є правило

“Центр кулі належить колу радіуса `BilliardTable.Radius`.”

Комбіновані й інші складені типи в описах об'єктів відіграють особливу роль. Відзначимо, перед усім, що записи, масиви “у чистому вигляді” в описи об'єктів, як правило, не включають. Справа в тому, що кожне дане, якщо воно включено до опису об'єкта, повинне оброблятися, відповідно з дисципліною об'єктно-орієнтованого програмування, тільки сукупністю специфічних методів. Це означає, що атрибут-запис можна і потрібно замінити атрибутом-об'єктом. Включення в структуру об'єкта іншого об'єкта називають *агрегацією або композицією*. Використання агрегатів поряд з іншими способами комбінування об'єктних типів для опису більш складних типів ми розглянемо пізніше.

Сказане вище в значній мірі відноситься й до інших структурних типів мови. Якщо дане структуроване, завжди існує необхідність визначення операцій його обробки.

4.2. Методи

Як уже відзначалося, найважливішим нововведенням, що характеризує ОО методологію програмування, є включення у визначення об'єкта операцій, що може виконувати цей об'єкт.

Під методами в ООП розуміють алгоритми, спрямовані на виконання об'єктом операцій (дій) як за запитом іншого об'єкта, так і для “задоволення своїх власних потреб” в опрацюванні інформації.

Таким чином, методи об'єктів в ООП слугують тим же цілям, що і процедури (функції) в структурному програмуванні. Однак, на відміну від структурного програмування, у якому для розв'язання підзадачі основна програма (чи процедура) викликає відповідну процедуру, передаючи їй необхідні дані і приймаючи результати опрацювання у вигляді параметрів, ОО підхід припускає, що кожен об'єкт, реагуючи на повідомлення - запит від іншого об'єкта, вирішує поставлену задачу, виконуючи необхідну операцію власним методом. Таким чином, для будь-якої дії (операції), що виконується над атрибутами об'єкта, повинний бути написаний окремий метод.

4.2.1. Реалізація методів засобами мови Borland Pascal

Опис кожного методу об'єкта включає заголовок методу й опис процедури чи функції, що реалізує цей метод. Заголовки методів, як ми вже знаємо, включаються в опис об'єкта. Заголовок методу грає дуже важливу роль – він представляє ту інформацію про операцію, яку повинен знати користувач об'єкта. На етапі проектування і реалізації системи користувачем є той програміст, що використовує описуваний об'єкт, як готовий до вживання, в своїй частині проекрованої системи. В процесі виконання програмної системи об'єкти, отже, взаємодіють, використовуючи повідомлення-запити у формі операторів звернення до відповідних процедур і функцій, які цю операцію реалізують.

Описи процедур і функцій, що реалізують методи, включаються у відповідний розділ процедур і функцій та оформляються як реалізаційна частина окремого модуля.

Описи заголовків методів розташовуються після описів інформаційних полів об'єкта і закінчуються символом “;”.

```
<Список описів методів> ::= { <Заголовок методу> ; }  
<Заголовок методу> ::=  
  Procedure <Ім'я методу> (<описи формальних параметрів>) |  
  Procedure <Ім'я методу> |  
  Function <Ім'я методу> (<описи аргументів>) : <тип> |  
  Function <Ім'я методу> |  
  Constructor <Ім'я методу> (<описи формальних параметрів>) |  
  Destructor <Ім'я методу> (<описи формальних параметрів>)
```

Методи **Constructor** і **Destructor** відіграють особливу роль. Їхні об'яви, реалізації і використання ми розглянемо пізніше.

Синтаксично заголовки методів – процедур і функцій виглядають так само, як і заголовки звичайних процедур і функцій.

Імена методів означають дії, реалізовані цими методами. Тому в якості імен (ідентифікаторів) методів рекомендується використовувати відповідні дієслова.

Наприклад, методи об'єкта TVector: Add, Sub, Mult (скласти, відняти, помножити). Відповідні заголовки методів приведені в описі цього об'єкта (приклад 4.1).

Методи об'єкта TbilliardBall: Setup – установити (на стіл); Shut – ударити (києм); TouchUpon - зштовхнути (з іншою кулею); Mirror - відбити (від борта) і т.д.

```
Setup(X : TTableSizeX; Y : TTableSizeY);  
Shut(V : TVector);  
TouchUpon(B : TBilliardBall);  
Mirror;
```

Опис метода-процедури (функції), у свою чергу, складається з заголовка і тіла. Заголовок опису методу конструюється з імені об'єктного типу і заголовка методу, відділених символом “.”

<Заголовок опису методу> ::= <ім'я об'єктного типу>.<заголовок методу>

Це синтаксичне правило означає, що заголовок опису методу повинний у точності повторити заголовок цього методу, приведенний в описі об'єктного типу, з додаванням перед ним імені відповідного об'єктного типу, що закінчується крапкою. Наприклад:

```
TbilliardBall.SetonTable(X:TtableSizeX;Y:TTableSizeY);  
TbilliardBall.Shut(V: TVector);  
TbilliardBall.TouchUpon(B: TBilliardBall);  
TbilliardBall.Mirror;
```

Власне процедури і функції – описи методів включаються, як і звичайні процедури і функції, в розділ процедур і функцій.

Процедурний стиль програмування пропонує створювати і використовувати процедури і функції для опису алгоритмів опрацювання даних. ОО стиль програмування пропонує створювати і використовувати методи опрацювання даних, збережених (інкапсульованих) як атрибути об'єкта. Використання методів об'єкта принципово відрізняється від викликів процедур або функцій.

Тіло методу з точки зору синтаксису, власне кажучи, ідентично тілу процедури або функції (це блок). Семантична відмінність полягає в тому, що в блоці опису методу спрощений доступ до інформаційних полів відповідного об'єкта.

У тілі опису методу об'єкта інформаційні поля цього об'єкта доступні по їхніх внутрішніх іменах.

Порівняємо техніку програмування деякої процедури обробки полів об'єкта, реалізовану в процедурному стилі з технікою програмування відповідного методу (тобто техніку програмування в ОО стилі). Нехай

ObjectName – ім'я об'єкта – перемінної;

ObjectType – ім'я об'єктного типу;

ObjectDataField_1, ObjectDataField_2, ... – внутрішні імена інформаційних полів об'єкта;

ObjectMethod – ім'я методу того об'єкта, що описується.

Тоді опис процедури обробки може виглядати так:

Опис процедури **ObjectMethod** у процедурному стилі

Procedure ObjectMethod(ObjectName:ObjectType);

begin

With ObjectName **do begin**

 <оператори, що використовують внутрішні імена полів

 ObjectDataField_1, ObjectDataField_2, ... >

end

end;

Так, опис методів **Print** і **Init** об'єкта **TDate** виглядає таким чином:

Procedure TDate.Print;

begin

 WriteLn(Day);

 WriteLn(Ord(Month)+1);

 WriteLn(Year)

end;

Procedure TDate.Init(D:TDay; M:TMonth; Y:TYear);

begin

Day := D;

Month := M;

Year := Y

end;

Опис методу ObjectMethod в ОО стилі

Procedure ObjectName.ObjectMethod;

begin

<оператори, що використовують внутрішні імена полів

ObjectDataField_1, ObjectDataField_2, ... >

end;

Метод TDate.Init залежить від формальних параметрів. Зауважте, що техніка опису і використання формальних параметрів не відрізняється від техніки їхнього опису і використання для процедур і функцій.

У розділі операторів блока, у якому використовується змінна-об'єкт, метод активується відповідним оператором. Синтаксично оператор використання методу виглядає так:

<Ім'я об'єкта>.<Ім'я методу>(<список фактичних параметрів>)

Фактичні параметри можуть бути відсутні:

<Ім'я об'єкта>.<Ім'я методу>

Метафора ОО стилю програмування інтерпретує активацію методу як результат повідомлення - запиту до об'єкта на виконання деякої дії. Цей запит може супроводжуватися параметрами, необхідними для виконання даної дії. Таким чином, у розділі операторів містяться повідомлення-запити до об'єкта MyBirthDay ініціалізувати свої інформаційні поля (метод Init) і вивести на екран їх значення (метод Print).

4.2.2. Використання методів

об'єкта

Методи об'єкта використовуються в розділах операторів програм, процедур, функцій і інших методів. Об'єкт, метод якого використовується, повинен бути описаним і доступним у відповідному розділі операторів (знаходитися в області видимості цього розділу). Наприклад:

Var

MyBirthDay: TDate;

begin

MyBirthDay.Init(14, sep, 1948);

MyBirthDay.Print

end.

Іноді в тілі опису методу зручно вказувати на приналежність інформаційного поля саме тому об'єкту, метод якого описується. Наприклад, у методі Add, що визначає додавання двох векторів, результат – проекції на осі координат можна іменувати, використовуючи стандартне ім'я **Self**:

Опис методу Add {додавання векторів}

Procedure Tvectod.Add(U, V : TVector);

Begin

Self.prX := U.PrX + V.PrX;

Self.prY := U.PrY + V.PrY;

end;

Для іменування об'єкта в тілі опису його методу можна використовувати стандартне ім'я *Self*.

Приклад 4.5. Об'єкт Текстове поле, призначений для висновку в алфавітно-цифровий екран коротких текстів.

Ми приводимо повний лістинг програми, у якій визначений об'єктний тип Текстове поле, описані методи цього типу і, на закінчення, приведені розділи змінних і операторів основної програми, що демонструє використання двох змінних описаного типу. Відзначимо, що в прикладі використані тільки ті засоби, про які йшла мова вище. Тому він далекий від досконалості.

Об'єкт Текстове поле. Опису типів

Uses Crt;

Const N = 40;

Type

TSizeX = 1..80;

TSizeY = 1..25;

T textSize = 0..N;

TStringN = String[N];

TTextField = **object**

Xpos : TSize; { позиція об'єкта на екрані }

Ypos : TSize;

TextLen : T textSize; { довжина текстового рядка }

CaptionColor: Byte; { колір напису }

TxtColor : Byte; { колір текстового рядка }

FieldColor : Byte; { колір тла текстового рядка }

CaptionLen : T textSize; { довжина напису }

Caption : TString; { напис }

Procedure Init(X:TSizeX; Y:TSizeY; CLen,TLen:T textSize;
CCol, TCol, FCol:Byte; Title : TStringN);

Procedure Print(Content : TStringN);

Procedure Clear;

Procedure Done;

end;

Об'єкт Текстове поле. Опису методів

Procedure TTextField.Init


```

    (X: TSizeX; Y: TSizeY; CLen, TLen: TTextSize;
     CCol, TCol, FCol: Byte; Title: TStringN);
begin
    Xpos := X;
    Ypos := Y;
    CaptionLen := CLen;
    TextLen := TLen;
    CaptionColor := CCol;
    TxtColor := TCol;
    FieldColor := FCol;
    Caption := Title;
    GotoXY(Xpos, Ypos);
    TextColor(CaptionColor);
    Write(Caption);
end;

```

```

Procedure TTextField.Print(Content : TStringN);
begin
    TextColor(TxtColor);
    TextBackGround(FieldColor);
    GotoXY(Xpos+CaptionLen, Ypos);
    Write(Content);
end;

```

```

Procedure TTextField.Clear;
Var
    i : Byte;
begin
    GotoXY(Xpos+CaptionLen, Ypos);
    TextBackGround(Black);
    For i := 1 to N do Write(' ');
    GotoXY(Xpos+CaptionLen, Ypos);
end;
Procedure TTextField.Done;
var
    i : Byte;
begin

```

```
GotoXY(Xpos, Ypos);
TextBackGround(Black);
For i := 1 to N+CaptionLen do Write(' ');
GotoXY(0, 0);
end;
```

Демонстрація Об'єкт Текстове поле.

Var

T, S : TTextField;

begin

```
ClrScr;
T.Init(5, 1, 10, 16, Red, White, Blue, 'Timer');
ReadKey;
T.Print('12.14.45');
ReadKey;
T.Clear;
ReadKey;
T.Done;
ReadKey;
ClrScr;
S.Init(35, 15, 10, 16, Blue, Magenta, Blue, 'Date');
ReadKey;
S.Print('01.01.2000');
ReadKey;
S.Clear;
ReadKey;
S.Done;
```

end.

Об'єкт, реалізацію якого ми щойно розглянули, призначений для виведення на екран стислих (не більш ніж 40 символів) текстових повідомлень. Операції Init і Done відповідно виводять на алфавітно-цифровий екран і видаляють з екрана цей об'єкт, а метод Print виводить в зазначену позицію той текст, що він одержує як параметр. Це означає, що метод Print призначений тільки для зовнішнього користувача.

4.3. Модулі й описи об'єктів

Концепція програмного модуля системи Borland Pascal і реалізація модулів (Units) у цьому середовищі відповідає принципу інкапсуляції об'єктно-орієнтованого програмування. Справді, описи об'єктних типів передують їхньому використанню в інших програмних модулях проектованої системи. У цих модулях вони описуються як окремі сутності - змінні відповідного об'єктного типу, що опрацьовуються методами цього типу.

Оскільки використання засобів модуля неможливо здійснити “за замовчуванням” і програмісту модулів верхнього рівня доступні тільки ті методи обробки об'єктних змінних, які в модулі використовуваного об'єкта описані як інтерфейсні, саме модуль є тією капсулою, що значною мірою захищає об'єкти від некоректних дій програміста. Нарешті, об'єктний клас, реалізований як модуль, може бути використаний і повторно (наприклад, для проектування інших програмних систем).

4.3.1. Використання модулів для структурування опису об'єктних типів

Розглянемо приклад використання модуля для описів об'єктних типів і використання об'єктів. Об'єкт TtextField реалізовано в модулі IOTexts. Оскільки реалізація використовує стандартний модуль CRT, цей модуль підключається в розділ Implementation модуля IOTexts.

Основна програма приклада, що демонструє техніку використання методів об'єкта TtextField, повинна тепер використовувати модуль IOTexts:

Об'єкт Текстове поле. Опис типів.

Unit IOTexts;

Interface

{Опис типів об'єкта Текстове поле}

Implementation

Uses Crt;

{Опис методів об'єкта текстове поле}

Демонстрація Об'єктТекстове поле.

Uses CRT, IOTexts;

Var

T, S: TextField;

Begin

{Основна програма з демонстрацією методів}

end.

4.4. Інкапсуляція

М. Буч [2] визначає реальний об'єкт як сутність, що має чітко визначені границі. Об'єкт характеризується станом, поведінкою і ідентичністю; структура і поведінка об'єктів визначають загальний для них клас. Це визначення має яскраво виражений філософський характер, оскільки воно оперує філософськими категоріями “поведінка”, “ідентичність” і т.д.

Вище ми “визначили” програмні об'єкти як інформаційні моделі реальних об'єктів і процесів. Звичайно, це теж ще не означення в математично строгому розумінні цього терміну, а лише пояснення, що розкривають тільки одну прагматичну сторону цього поняття. Саме, як інформаційна модель, об'єкт “зсередини” описується такими даними й операціями їхньої обробки, що за поведінкою об'єкта дозволяють ідентифікувати його з абстракцією реального об'єкта.

Означення об'єкта стає математично точним тільки в рамках формальної системи, такої, наприклад, як конкретна об'єктно-орієнтована система програмування.

Однак, незалежно від характеру означень, усі вони описують об'єкт як єдність даних і операцій, відокремлюючи при цьому зовнішні прояви цієї єдності у вигляді поведінки від внутрішнього устрою – структури, що визначає поведінку. Таким чином, існують чіткі границі між устроєм об'єкта і його функціонуванням (поведінкою). Поведінка об'єкта описує його з абстрактної точки зору, а структура об'єкта визначає механізм

реалізації функціонування. Відокремлення описів поведінки і реалізації його структури називають інкапсуляцією.

Інкапсуляція – це процес відокремлення друг від друга елементів опису об'єкта, що визначають його устрій і поведінку; інкапсуляція необхідна для того, щоб ізолювати контрактні зобов'язання абстракції від їхньої реалізації.

Практично це означає, що опис об'єкта повинен бути розділений на дві частини – інтерфейсну, що визначає поведінку об'єкта, і реалізаційну, що визначає механізми досягнення бажаного функціонування. Перший крок у реалізації інкапсуляції в мові Паскаль ми щойно розглянули. Подальший розвиток цього принципу представляє розподіл описів на так звані загальнодоступні і приватні.

4.4.1. Загальнодоступні і приватні атрибути і методи

Описи об'єктних типів, у свою чергу, можуть бути розділені на так звані сховані (приватні) і відкриті (загальнодоступні) частини (сегменти). Описи відкритого сегмента об'єктного типу випереджаються службовим словом **public**, а описи схованого сегмента – службовим словом **private**.

Таким чином, опис об'єктного типу може мати два розділи: розділ відкритого сегмента об'єкта, що містить описи загальнодоступних атрибутів і методів, і розділ приватного сегмента - з описами приватних атрибутів і методів. Відповідне синтаксичне означення має вид:

Type

Tobject = **object**

public

<відкриті (загальнодоступні) поля>;

<відкриті (загальнодоступні) методи>;

private

<сховані (приватні) поля>;

<сховані (приватні) методи>;

end;

У такому описі можна опустити слово `public`, якщо загальнодоступний розділ розмістити на початку опису об'єктного типу. Ті поля і методи, описи яких розміщені до розділу `private`, будуть інтерпретуватися “за замовчуванням” як загальнодоступні:

Type

`Tobject = object`

<відкриті (загальнодоступні) поля>;

<відкриті (загальнодоступні) методи>;

private

<сховані (приватні) поля>;

<сховані (приватні) методи>;

end;

Явне використання директиви `public` дозволяє змінювати місцями загальнодоступні і приватні сегменти опису:

Type

`Tobject = object`

private

<сховані (приватні) поля>;

<сховані (приватні) методи>;

public

<відкриті (загальнодоступні) поля>;

<відкриті (загальнодоступні) методи>;

end;

Таким чином, концепція приховання деяких деталей опису програмних об'єктів, що для модулів реалізована розділами `Interface` і `Implementation`, реалізована і для об'єктних типів засобами розділів `public` і `private`.

Опис приватного розділу об'єктного типу доступні тільки усередині того модуля, в якому цей об'єкт визначений описом.

Зокрема, можна приховати всі інформаційні поля об'єктного типу, заборонивши тим самим доступ ззовні до цих полів будь-якими

засобами, крім тих, котрі програміст надав для використання як загальнодоступні іншим компонентам програми.

Методологія ОО проектування орієнтована на розробку великої і складно влаштованої комп'ютерної програми. Такі програми проектуються колективом розроблювачів. Тому методологія ОО проектування зобов'язана підтримувати розподіл праці.

Припустимо, що розроблювальна програмна система складається з декількох компонентів, кожен з яких пише свій програміст. Що він повинен знати про інші компоненти? Тільки їхню правильну поведінку - тобто повідомлення-запити на виконання завдань, якими можна користуватися і реакції іншого компонента на ці повідомлення. Протокол програмного компонента-об'єкта, отже, єдине необхідне її зовнішній опису. Все інше повинно бути приховано і надійно захищене.

Т.Бадд [1] приводить наступні правила - принципи Парнаса:

- Розроблювач програмного компонента повинен представляти користувачу цього компонента всю інформацію, що потрібна для ефективного її використання, і нічого крім цього.
- Розроблювач програмного забезпечення, що використовує програмний компонент-додаток, повинний знати тільки необхідне функціонування компонента, і нічого крім цього.

Зауважимо на закінчення, що цей важливий принцип програмування не просто усвідомити програмістам, що працюють поодиноці над розробкою невеликої за розміром програмою (наприклад, студентам). Найбільшу складність у таких програмах можуть представляти проблеми реалізації віртуозних алгоритмів, що вирішують конкретні задачі опрацювання даних. Тому може скластися враження, що зі збільшенням розміру програми збільшується складність алгоритмів, пов'язана з "накручуванням" циклів, рекурсивних викликів і т.п. Насправді основною проблемою тут є проблема узгодження програмних компонентів, що розроблюються незалежно. Рішенню цієї проблеми присвячені всі принципи об'єктно-орієнтованого програмування, у тому числі і принципи відокремлення інтерфейсної і реалізаційної частин, щойно сформульовані.

Програміст компоненти може експериментувати над її реалізацією, поліпшуючи працюючі всередині алгоритми й вдосконалюючи внутрішні структури даних, не модифікуючи при цьому інтерфейсного протоколу. Зовнішній світ, тобто інші програмні компоненти при цьому не змінюють правил своєї поведінки, тому не змінюється і функціонування системи в цілому.

4.5. Прості об'єкти

Будь-яка складна програмна система являє собою сукупність взаємодіючих об'єктів тієї чи іншої структури і складності. Методи визначення більш складних об'єктів з більш простих, методи взаємодії об'єктів ми розглянемо пізніше. Очевидно, однак, що будь-які такі побудови містять в собі описи конкретних об'єктів, що мають досить просту побудову і спеціалізовану поведінку. Об'єкти, про які піде мова, з'являються, як правило, на заключному етапі проекту - етапі, на якому приймаються чисто технічні, остаточні рішення. Деякі приклади таких об'єктів ми зараз розглянемо. Наша мета полягає в тому, щоб одержати більш глибокі представлення про ті конкретні проблеми, що виникають і розв'язуються на цьому етапі. Приведена нижче "класифікація" не є ні стандартною, ні повною, але в достатній мірі відображає мету.

4.5.1. Атрибути об'єктів і об'єкти-атрибути

Об'єкти-атрибути призначені для збереження і спеціалізованої обробки даних. Як правило, кожен такий об'єкт поєднує в собі дані, що описують одну з комплексних характеристик конкретного реального об'єкта. З іншого боку, ця характеристика може використовуватися для визначення багатьох об'єктних типів. Тому вона є самостійною об'єктною одиницею і повинна бути описана і реалізована окремо. Наприклад, об'єкт TDate (дата), визначений у прикладі 4.2, ми використовуємо в описі об'єкту TPerson. Але, зовсім очевидно, що цей об'єктний тип може і повинний бути включений у визначення всіх класів об'єктів, що використовують дати. До об'єктів-атрибутів належать, наприклад, такі об'єкти, як TPerson, TPostAdress, TVector і т.п. Імена об'єктів-атрибутів входять у словник проекту програмної системи як іменники-ідіоми.

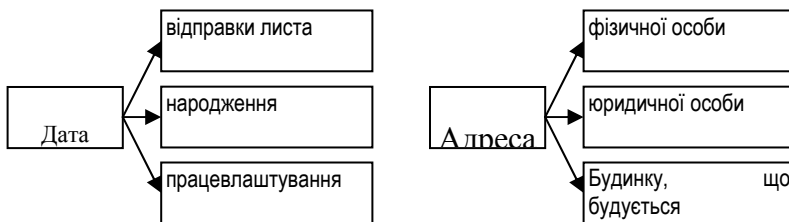
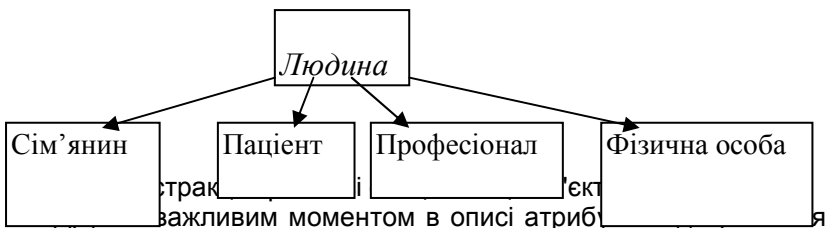


Рис. 4.1. Абстракції і спеціалізації об'єктів.

Властивість універсальності (можливості багаторазового використання) об'єктів-атрибутів, мабуть, є найбільш важливою з практичної точки зору. Її необхідно забезпечувати в першу чергу. Цього можна досягти, насамперед, відповідним рівнем абстракції опису об'єкта.

Атрибути об'єкта повинні описувати тільки ті його характеристики, що формують відповідну даному типу абстракцію, тобто є найбільш загальними і специфічними в цій абстракції.

Так, атрибутами класу TPerson повинні бути такі характеристики, як ім'я (привілейований ідентифікатор), стать, можливо, ще деякі інші атрибути. При подальшому використанні тип TPerson доповнюється іншими характеристиками, що визначають ролі людини.



важливим моментом в описі атрибу

Типи атрибутів повинні забезпечувати відповідний рівень абстракції і максимальні зручності для специфічних операцій опрацювання об'єкта-атрибута. Так, наприклад, опис типу TweekDay у виді

TweekDay = Array[1..7] of String[10]

є неприйнятним, тому що ім'я дня тижня в цьому описі прив'язується до рядка. Очевидно, що значенням поля буде найменування дня тижня, написане якоюсь конкретною мовою (англійською, російською, суахілі, тощо), що обов'язково призведе до ускладнення методів опрацювання цього даного. Тип `TWeekDay`, реалізований таким чином, є зручним лише для виведення на екран або принтер значення дня тижня, а операція виведення не є специфічною для цього класу. Нормальним у цьому описі є використання перелічувального типу `TDayName`.

`TWeekDay = Array[1..7] of TDayName`

4.5.2. Методи доступу

Характерними операціями об'єктів - атрибутів є операції доступу до інформаційних полів цього об'єкта.

Оскільки при реалізації об'єктних типів дотримується принцип приховання даних, власне інформаційні поля об'єкта недоступні користувачу об'єкта. Тому для кожного атрибута, яким можна користуватися, повинні бути визначені операції доступу для читання і запису. Методи цих операцій називають відповідно методами-селекторами і методами-модифікаторами. Найбільш простим селектором є операція читання, а модифікатором – операція запису. Звичайно селекцію (доступ-читання) реалізують методами з іменами типу `Get<Ім'я_атрибута>`. Наприклад, для класу `TDate` повинні бути визначені методи `GetDay`, `GetMonth`, `GetYear`, `GetWeekDay`. Методи-селектори найкраще реалізовувати як функції:

```
Function TDate.GetDay : Tday;  
begin  
    GetDay := Day;  
end;
```

Коли тип інформаційного поля не дозволяє визначити метод-селектор як функцію, використовують процедури:

```
Procedure TMan.GetName(var F : Tname);  
begin  
    F := Name  
end;
```

Модифікації атрибутів об'єкта частіше реалізують за допомогою операції з іменами-дієсловами типу Put<Ім'я атрибута>, Set<Ім'я атрибута>. Наприклад:

```
Procedure TDate.PutDay (DD: Tday);  
begin  
    Day := DD;  
end;
```

Оскільки нове значення атрибута імпортується, методи-модифікатори реалізують як процедури з відповідним параметром.

Правила захисту інформаційних полів об'єкта від некоректних змін вимагають від програміста чіткого розподілу всіх атрибутів об'єкта на дві групи:

- атрибути “тільки для читання” - визначені тільки методи-селектори (типу Get);
- атрибути “для запису і читання” - визначені методи-селектори і методи-модифікатори (типу Put...,Set...)...

4.5.3. Властивості й атрибути

Принципи приховання й інкапсуляції означають, що будь-який об'єкт може бути ідентифікований “у зовнішньому світі”, тобто використаний іншими об'єктами тільки відповідно до протоколу, у якому перелічені і специфіковані його операції (у виді заголовків методів). Операції – селектори виділяють у протоколі об'єкта його властивості.

Властивістю об'єкта називають дане, що представляє інформацію про цей об'єкт і доступне іншим об'єктам.

Існує, отже, принципове розходження між атрибутами і властивостями об'єкта. Властивості є зовнішнім проявом набору атрибутів об'єкта і його зв'язків із зовнішнім світом.

Як приклад розглянемо дві можливі реалізації класу TDate.

1. Об'єктний тип TDate реалізований за допомогою системної функції, що повертає системну дату у форматі <MM-DD-YY-WD>, де MM - номер місяця, DD - день місяця, YY - рік, - WD - назва дня тижня англійською мовою. У цій реалізації тип TDate атрибутів не має.

2. Об'єктний тип TDate реалізований набором атрибутів Month - номер місяця, Day - день місяця, Year - рік.

Нехай і в першій, і в другій реалізації визначені ті самі операції з іменами методів GetDay, GetWeekDay, GetMonth, GetYear. З погляду зовнішнього користувача ці об'єктні типи тотожні. Вони володіють тими самими властивостями. Це дійсно дві різні реалізації одного і того ж типу, тому що вони представлені одним протоколом. Оскільки в конкретній програмній системі об'єктний тип ідентифікується ім'ям, ці реалізації альтернативні. Програміст може реалізувати тільки один варіант типу TDate.

Відмітимо, що в першому варіанті операції-модифікатори можуть бути реалізовані тільки за допомогою іншої системної функції - функції зміни системної дати. Це - дуже небезпечна операція, що повинна бути заборонена для виконання в прикладних програмах. Якщо ж у програмній системі дійсно необхідні і поточна дата, і деякі інші дати, необхідно визначити два різних класи TCurrentDate і TSomeDate, один із яких обробляє системну дату методами-селекторами, а інший довизначений ще і методами-модифікаторами, що змінюють атрибути TDate. Однак, крім операцій доступу, в додатках можуть знадобитися й інші операції над датами, реалізація яких не залежить від того, з яким типом дати вони виконуються. Такий об'єктний тип необхідно створювати окремо.

Сукупність методів об'єктного типу містить методи особливого типу, що визначають властивості об'єктів. Властивості об'єктів ідентифікують об'єкти для зовнішніх користувачів.

4.5.4. Основні і похідні атрибути

Використовуючи математичну термінологію, властивість Property найпростішого об'єкта-атрибута Obj можна визначити як функцію від його атрибутів $Atr_1, Atr_2, \dots, Atr_k$ як від змінних:

$$Obj.Property = F(Obj.Atr_1, Obj.Atr_2, \dots, Obj.Atr_k)$$

Тому сукупність $Atr_1, Atr_2, \dots, Atr_k$ атрибутів об'єкта повинна задовольняти вимозі *функціональної повноти*.

Функціональна повнота набору атрибутів означає, що будь-яка властивість об'єкта може бути виражена у виді функції його атрибутів, що може бути обчислена.

Наприклад, атрибутами вектора на площині можуть бути його декартові координати, оскільки відомо, що будь-які інші характеристики і властивості векторів виражаються через їхні декартові координати.

Вимога функціональної повноти набору атрибутів об'єкта грає важливу методологічну роль. Нею потрібно керуватися при визначенні набору атрибутів. Тоді під час реалізації методів не виникне ніяких проблем принципового характеру.

З властивістю функціональної повноти тісно пов'язана властивість незалежності набору атрибутів.

Набір атрибутів називають функціонально незалежним, якщо жоден з атрибутів цього набору не може бути обчислений через інші атрибути.

Наприклад, набір атрибутів Day, Month, Year, WeekDay є функціонально залежним, оскільки день тижня WeekDay можна обчислити, знаючи значення Day, Month, Year.

Атрибути функціонально незалежного набору називають *основними*, а інші - *похідними*.

Як правило, у наборі всіляких даних, які можна розглядати як кандидатів в атрибути об'єкта, виділяють тільки функціонально незалежний набір. Дані цього набору визначають як атрибути, а інші дані реалізують як властивості, описуючи методи доступу до них у виді обчислень. Іноді, однак, має сенс доповнити функціонально незалежний набір атрибутів декількома похідними атрибутами, якщо це сильно вплине на якість програми (наприклад, дозволить різко збільшити її швидкодію). У цьому випадку реалізація методів-селекторів ускладнюється, оскільки в кожному з них потрібно реалізувати обчислення значень похідних атрибутів. Розглянемо приклад:

Приклад 4.6.

Припустимо, що в деякій програмі потрібно визначити такий об'єкт TPerson, для якого одними з основних операцій були би операції визначення батька і матери цього об'єкта.

```

TPerson = object
  private
    FamilyName,
    OwnName,
    FatherName : Tname;
    Father,
    Mother : ^TPerson;
  public
    Procedure GetFamilyName(var P : Tname);
    Procedure GetOwnName(var P : Tname);
    Procedure GetFatherName(var P : Tname);
    Procedure GetFuther(var P : TPerson);
    Procedure GetMother(var P : TPerson);
end;

```

У цьому описі сукупність атрибутів є функціонально залежною, оскільки значення атрибута FatherName може бути обчислене через посилання-значення поля Father^.OwnName. Отже, інформаційне поле FatherName можна виключити. При цьому, однак, програмісту доведеться реалізовувати далеко не простий алгоритм побудови ім'я по батькові за ім'ям, в основі якого лежать спеціальні правила української мови. Виникає законне питання про те, чи варто це робити, якщо можна просто включити до списку атрибутів ще один похідний атрибут, поклавши рішення мовних проблем на користувача програми.

У більш загальній формі, властивості об'єктів залежать ще і від властивостей інших об'єктів як від параметрів. Наприклад, об'єктний тип TPerson ми описали, використовуючи зв'язок з об'єктами того ж типу. Тим самим методи визначення властивостей TPerson можуть використовувати і дані об'єктів Father, Mother.

4.5.5. Об'єкти-обчислювачі

Основною функцією ще одного типу найпростіших об'єктів - об'єкта-обчислювача є реалізація одного чи декількох обчислювальних алгоритмів. Об'єкти-обчислювачі іноді називають віртуальними машинами.

У найбільш простій ситуації об'єкт-обчислювач являє собою реалізацію одного, але, як правило, досить складного обчислення. Це може бути, наприклад, ефективний алгоритм сортування масивів (швидке сортування), який у системі використовують декілька об'єктних типів. Інший приклад - об'єкт-генератор випадкових чисел із заданою функцією розподілу ймовірностей. Опис об'єктів такого типу містить відповідний обчислювальний метод. Об'єкти-обчислювачі можуть мати і атрибути - такі поля, як *Точність_обчислень*, *Кількість_ітерацій*, і т.д. Наприклад:

Type

```
TSorter = object  
    QuickSort(var A:array of Integer; First, Last: Index);  
end;
```

Тут First і Last - відповідно перший і останній індекси діапазону, що сортується в масиві A.

```
TRandomGenerator = object  
    Genetate(N : Integer );  
end;
```

Тут N - кількість випадкових чисел, що генеруються

У більш загальному випадку об'єкти-обчислювачі можуть містити в собі методи реалізації декількох обчислювальних алгоритмів з однієї предметної області. Розглянемо як приклад такої предметної області розділ аналітичної геометрії на площині, що оперує з точками, відрізками, прямими. Характерним для цієї предметної області є те, що її операції визначені над різнотипними об'єктами. Тому має сенс визначити декілька об'єктних типів-атрибутів, інкапсулюючи в них лише операції ініціалізації, доступу, а потім визначити об'єкт - предметну область, описавши там так називані багатосортні операції, що вирішують основні задачі відповідного розділу аналітичної геометрії.

Приклад 4.7. Аналітична геометрія.

Type

TPoint = **object**

private

X, Y : Real;

public

function Get : Real;

function Get : Real;

{ інші методи }

end;

TSegment = **object**

private

A, B : TPoint;

public

function Get : Real;

function Get : Real;

{ інші методи }

end;

TLine = **object**

private

CoefA, CoefB, CoefC : Real;

public

function GetCoefA : Real;

function GetCoefB : Real;

function GetCoefC : Real;

{ інші методи }

end;

TGeometry = **object**

{Розподіл відрізка в даному відношенні}

Procedure DivideSegment(s : TSegment; lambda : Real; var A:Tpoint);

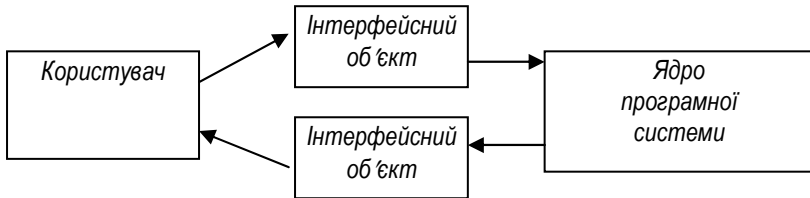
{Довжина відрізка}

Function GetSegmentLenth(s : Tsegment) : Real;

{Формування відрізка AB}

Procedure MakeSegment(A, B : TPoint; var l : TSegment);

{Рівняння прямої, що проходить через A, B}
 Procedure MakeLine(A, B : TPoint; var l : TLine);
 {Точка перетинання двох прямих}
 Procedure IntersectLines(l, p : TLine; var A : TPoint);
 {Кут між двома прямими}



Function GetAngle(l, p : Tline) : Real;
 {інші методи розв'язання задач аналітичної геометрії }

end;

4.5.6. Зовнішні об'єкти

Істотну роль у проектуванні комп'ютерної системи грає її інтерфейс користувача.

Під інтерфейсом користувача програмної системи розуміють сукупність способів взаємодії користувача з програмною системою і засобів програмної системи, за допомогою яких ці способи реалізовані.

Как і інші частини програмної системи, інтерфейсна її частина проектується у виді системи об'єктів, що взаємодіють як один з одним, так і з об'єктами внутрішньої частини (ядра) програмної системи. Ці об'єкти називають зовнішніми або інтерфейсними.

Призначення зовнішнього об'єкта – реалізація операцій введення – виведення, пошуку й інших операцій взаємодії користувача з програмною системою.

Рис 4.3. Схема інтерфейсу користувача.

Проектування навіть найпростіших зовнішніх об'єктів має ряд особливостей. Основна з них полягає в тому, що програміст повинен визначити особливого типу об'єкт “Користувач” і описати протокол взаємодії цього об'єкта з проєктованим інтерфейсним об'єктом. Оскільки реальний користувач працює на комп'ютері, застосовуючи пристрої введення-виведення, операції користувача є деяким обмеженням набору операцій відповідного пристрою. Кожен пристрій має свій ресурс, частину якого віддається об'єкту. Таким чином, при проектуванні введення-виведення програміст вирішує задачу розподілу ресурсів зовнішніх пристроїв і управління ними.

Ще одна особливість проектування інтерфейсних пристроїв полягає в тому, що типи (формати) даних, підтримуваних зовнішніми пристроями, відрізняються в загальному випадку від форматів даних – атрибутів “внутрішнього об'єкта”, з якими взаємодіє інтерфейсний об'єкт. Тому повинні бути реалізовані операції перетворення типів даних з одного представлення в інше.

Таким чином, інтерфейсний об'єкт, як і будь-який пристрій сполучення, має двосторонню, “двошарову” структуру – одна сторона “обслуговує” користувача, а інша – внутрішній об'єкт.

Приклад 4.8.

На рис. 4.4. зображено схему реалізації інтерфейсу виведення на екран монітора тексту, що відповідає даним, що зберігаються внутрішнім об'єктом.

Розглянемо більш детально ті проблеми, що виникають при реалізації цієї схеми.

Розподіл ресурсів зовнішнього пристрою.

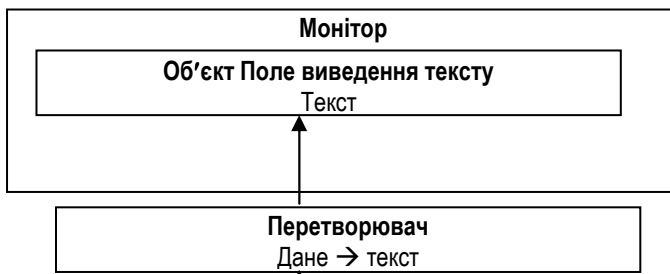


Рис. 4.4. Інтерфейс висновку Дане-текст.

На екрані монітора Об'єкт Поле висновку тексту виглядає так:

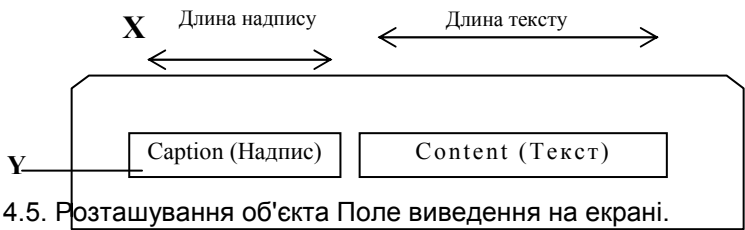


Рис 4.5. Розташування об'єкта Поле виведення на екрані.

Об'єкт Поле виведення тексту представлений наступним описом:

Клас : Поле виведення

Тип : TTextField

Атрибути		
Імена	Типи	Зміст
Xpos	TsizeX	Координата X позиції поля на екрані
Ypos	TsizeY	Координата Y позиції поля на екрані
Caption	TStringN	Надпис до тексту
CaptionLen	T textSize	Довжина надпису до тексту
CaptionColor:	Byte;	Колір напису до тексту
TextLen	T textSize	Довжина тексту, виведеного на екран
TxtColor :	Byte	Колір тексту
FieldColor :	Byte	Колір тла текстового поля

		Методи	
Імена	Параметри	Типи	Зміст
Init	Всі атрибути		Ініціює об'єкт
Done			Видаляє об'єкт з екрана
Print	Content	TString	Виводить текст на екран
Clear			Очищає поле тексту

Метричні атрибути об'єкта розподіляють екран, обмежуючи розміри об'єкта, інші атрибути визначають колірну гаму об'єкта. Атрибут Caption ідентифікує даний екземпляр об'єктного типу. Відзначимо, що аналіз атрибутів цього об'єкта дозволяє чітко виділити три найпростіших об'єкти-атрибути: об'єкти Позиція на екрані, Надпис і Текст – Зміст.

Перетворення форматів даних

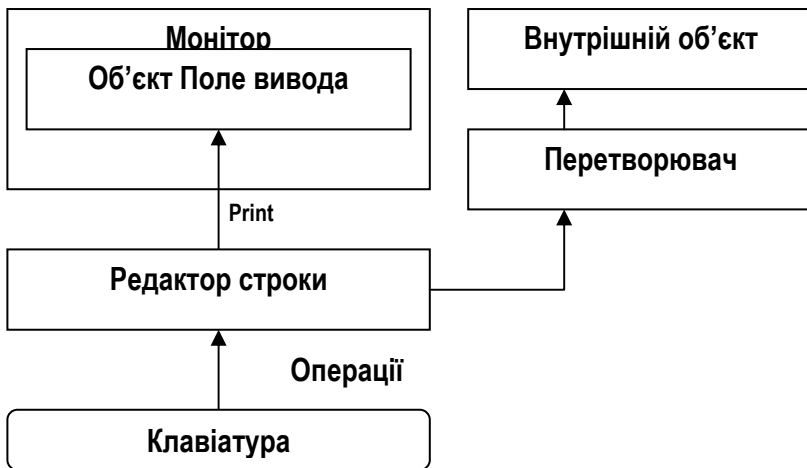
У тих випадках, коли внутрішній об'єкт у системі унікальний, операції перетворювача можна реалізувати методами цього об'єкта. Нехай, наприклад, у системі передбачене спеціальне вікно для виведення різних повідомлень про події, що відбуваються в процесі роботи системи. Кожна з цих подій має свій “внутрішній” код. Проблему дешифрування коду – перетворення його в текстове повідомлення – повинний вирішувати той об'єкт, у якому ця подія відбулася.

В іншому варіанті, коли унікальним є зовнішній об'єкт, операції перетворювача, навпаки, приписують цьому зовнішньому об'єкту. Наприклад, поточну дату на сторінці барвисто оформленого календаря потрібно переводити власними методами календаря, оскільки об'єкт Поточна Дата унікальний і тільки для цих цілей і створений.

Нарешті, у тих випадках, коли власне перетворення є універсальною операцією, що обслуговує декілька різнотипних зовнішніх і внутрішніх об'єктів, доцільно визначити об'єкт-обчислювач, що здійснює ці перетворення.

Наприклад, у всіх фінансових документах підсумкову суму необхідно роздруковувати особливим чином - ціле число основних грошових одиниць – прописом, а “копійок, (центів), ...” - у вигляді двохзначного числа. У цьому випадку доцільно визначити окремий об'єкт-обчислювач, оскільки і документів, і форм роздрукування підсумкових сум в них може бути декілька.

Приклад 4.9. Редактор рядка



Як приклад об'єкта, що здійснює введення даних, розглянемо об'єкт Редактор рядка. Операції цього об'єкта повинні підтримувати процес редагування рядка тексту. На рис. 4.5. представлена схема реалізації інтерфейсу редактора рядка.

Розподіл ресурсу клавіатури означає приписування кожній з клавіш деякої операції редагування. Обмеження набору операцій полягає в блокуванні деяких груп клавіш.

Рис. 4.6. Схема реалізації інтерфейсу редактора рядка.

Клас: Редактор рядка

Тип: TstringEditor

Інформаційні поля		
Імена	Типи	Зміст
EditString	String	Текст, що редагується

Cursor	Byte	Позиція курсору	
Mode	(Insert, OwerWrite)	Режим вставки/заміни	
Методи			
Імена	Параметри	Типи	Зміст
Init	Str	String	Начало редагування Str
GoLeft			Курсор уліво
GoRight			Курсор вправо
DeleteRight			Видалення символу праворуч
DeleteLeft			Видалення символу ліворуч
ChangeMode			Переключення режиму Ins/Qwr
AddChar	Ch	Char	Вставка/надпис символу
Done			Кінець редагування
Monitor			Розпізнавання натиснутої клавіші

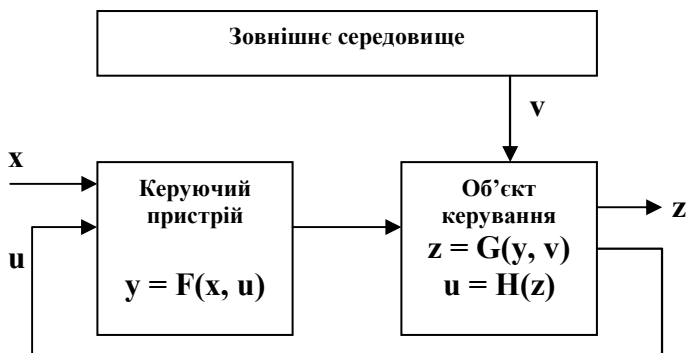
Метод Monitor виконує особливу роль: він розпізнає, яка клавіша натиснута і, у залежності від цього, доручає виконання своєї операції тому чи іншому методу.

4.5.7. Керуючі об'єкти і

методи

Керуючими об'єктами (методами), називають об'єкти, методи, що здійснюють керування системою чи її компонентами.

Для позначення керуючих об'єктів використовуються також терміни контролер, монітор, і т.д. Типовим прикладом реального об'єкта-монітора є панель керування будь-якою багатофункціональною інтерактивною системою. (Комп'ютер, телевізор, автомобіль, залізниця, і інші системи, що оснащені такими панелями, хоча вони можуть по-різному називатися). При проектуванні таких систем центральну роль відіграють об'єкти-монітори, що реалізують функції керування. Відзначимо, що на відміну від інших об'єктів, програмні об'єкти-монітори і системи керування часто вже не моделюють реальні керуючі об'єкти, а



самі виступають у цій ролі. У цьому і полягає комп'ютеризація керування.

Функція керування, як відомо, визначає керуючий вплив на об'єкт керування (систему або її компонент) у залежності від впливів зовнішнього середовища, об'єкта керування і свого власного стану. Правильний розподіл обов'язків у тріаді "зовнішнє середовище - монітор - об'єкт керування" - основна задача, що підлягає розв'язанню програмістом на етапі аналізу проектованої системи.

На рис. 4.7. показаний одна зі стандартних схем керування – схема керування зі зворотним зв'язком.

Рис 4.7. Керування з використанням зворотного зв'язку

Керуючі методи

Відзначимо, що в керуванні мають потребу не тільки складні системи чи їх компоненти, але часто і самі об'єкти. Характерним прикладом такого об'єкта є розглянутий вище редактор рядка. Керування ним здійснює метод Monitor. Зовнішнім середовищем тут є клавіатура, її сигнали - натискання клавіш. Об'єкт керування - рядок, що редагується. Керуючі впливи - зміни в рядку, що редагується. Результат натискання клавіши може бути різним у залежності від параметрів рядка, що редагується, (наприклад, її довжини) і стану самого монітора.

У нашому прикладі стан монітора визначається атрибутом (Insert - Owerwrite). У більш досконалих редакторах стан визначають ще і такі атрибути, як режим “прописні-рядкові букви”, мова, розкладка клавіатури, і т.і. Оскільки в редакторі рядка й об'єкт керування, і монітор реалізовані в одному об'єкті StringEdit, розподіл обов'язків “Керування - об'єкт керування” здійснюється на рівні атрибутів і методів. Метод Monitor керує редагуванням, атрибути типу (Insert - Owerwrite) визначають множину керуючих станів об'єкта. Атрибут EditString відіграє роль об'єкта керування, а керуючі впливи реалізовані схованими методами типу GoLeft.

Керуючі об'єкти

Важливим методологічним аспектом при проектуванні і реалізації керування є той факт, що операції керування мають більший ступінь загальності, ніж це необхідно для керування конкретною системою чи об'єктом.

Наприклад, основні операції керування рухом графічного об'єкта в ігровому полі конкретної комп'ютерної гри, власне кажучи, не залежать чи залежать у малому ступені від зовнішнього вигляду об'єкта. Справді, переміщення об'єкта - це зміна координат однієї (початкової) точки O , поворот - перетворення координат іншої точки A (поворот вектора OA). Тому можна визначити абстракції "траєкторія руху", "обертання", "зміна розмірів" і т.п.

Другий приклад - операції керування редагуванням рядка й операції керування однорівневим меню, які можна об'єднати в абстракції "редагування послідовності".

Тому один з основних методів проектування керування - використання загальних принципів керування у виді абстрактних об'єктів та їх модифікація стосовно до конкретних об'єктів і систем керування.

Вибір реалізації керування обумовлений багатьма факторами, але основні з них - ступінь складності керованої системи чи компоненти, а також - її ступінь залежності по керуванню від зовнішнього середовища.

Один з можливих підходів до реалізації керування полягає в описі так званого керуючого автомата.

Як ми вже відзначали, керуючий вплив залежить від зовнішнього середовища, об'єкта керування і стану монітора. Надалі ми не будемо виділяти об'єкт керування з множини об'єктів, що образують зовнішнє середовище. Справді, цей об'єкт, як і інші об'єкти, є зовнішнім відносно монітора, а в задачах керування системою в цілому він просто співпадає з цим середовищем.

Таким чином, можна визначити дві функції, значення яких визначають функціонування монітора: функцію виходів γ і функцію переходів δ .

$$\gamma : \langle S, X \rangle \rightarrow Y$$

$$\delta : \langle S, X \rangle \rightarrow S$$

Тут

X - множина впливів зовнішнього середовища (вхідних сигналів);

Y - множина керуючих впливів (вихідних сигналів);

S - множина керуючих станів монітора;

Елементи множини **X** називають *подіями*, а елементи множини **Y** - *діями*. Керування реалізують за допомогою спеціального атрибута, значення якого називають *керуючими станами* монітора.

В множині **S** виділений елемент s_0 , що називається початковим станом, у також підмножина S^* , заключних станів.

Таким чином, керуючий автомат визначається набором

$$\mathfrak{R} = \langle X, Y, S, \delta, \gamma, s_0, S^* \rangle$$

Множина **X** подій (сигналів зовнішнього середовища) описується скінченим набором імен властивостей, методи яких реалізують запити до зовнішніх об'єктів. У нашому прикладі монітора редактора рядка це системна функція ReadKey.

Множина дій **Y** описується скінченим набором імен операцій керування, методи яких реалізують повідомлення об'єкту чи керування декільком об'єктам, що утворюють керований компонент чи систему. У нашому прикладі монітора редактора рядка це - набір приватних методів типу GoLeft.

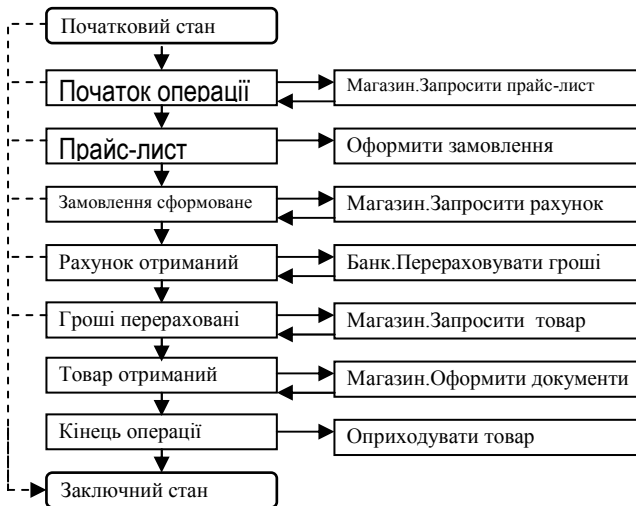
Нарешті, множина **S** керуючих станів призначено для "запам'ятовування" історії функціонування монітора. Виконавши дію **y**, керуючий автомат переходить у новий стан **s**, запам'ятовуючи тим самим факт виконання дії. Як ми вже з'ясували, керуючими станами в нашому прикладі є значення атрибута Mode.

Таким чином, проектування монітора полягає в описі функціонування керуючого автомата \mathfrak{R} . При цьому подіями (елементами множини **X**) є властивості об'єктів зовнішнього

середовища, а дії - елементи Y - суть імена методів (можливо, з параметрами). Множина S керуючих станів визначається скінченим набором атрибутів монітора.

Реалізація керування полягає у визначенні множини S і функції переходів δ на цій множині.

Розглянемо приклад реалізації керування процедурою покупки товару. У цій торговій операції беруть участь покупець,



магазин і банк. Керування операцією децентралізовано: кожна сторона контролює правильність виконання своїх дій. З погляду покупця операція представляється послідовністю станів, представлених на рис. 4.8.

Рис. 4.8. Автомат керування процедурою покупки товару

Вхідні сигнали – елементи множини X підрозділяються на дві групи: команди користувача і повідомлення магазину і банку.

- Команди користувача: продовжити операцію; скасувати операцію.
- **Повідомлення магазину:** прайс-лист представлений; товар відсутній; рахунок

представлений; гроші отримані; документи на одержання товару отримані.

- Повідомлення банку: рахунок клієнта заблокований; гроші перераховані; грошей на рахунку недостатньо.

Одна з теорем теорії автоматів твердить, що для будь-якого автомата, визначеного функціями переходів і виходів $\delta(s, x)$, $\lambda(s, x)$ можна визначити еквівалентний автомат, функція виходів якого має вигляд

$$\lambda(s, x) = \mu(\delta(s, x))$$

Це означає, що можна так визначити множину керуючих станів монітора **S** і функцію переходів $\delta(s, x)$, що вибір методу з множини **Y** буде цілком визначатися станом **s** монітора. Отже, у загальному вигляді керування можна реалізувати як вибір нового керуючого стану, перехід до якого супроводжується керуючим впливом:

```
Case  $\delta(s, x)$  of  
  s0 : Object1.y1;  
  s1 : Object2.y2;  
  
  s* : Objectk.yk;  
end;
```

Розглянемо деякі варіанти цієї загальної схеми:

1. Монітори з одним керуючим станом - найбільше простий і разом з тим такий варіант керування що часто зустрічається.

```
Case x of  
  X1 : Object1.y1;  
  .....  
  Xm : Objectl.ym;  
end;
```

Метод ExternalObject.GetVariant, звертаючись до зовнішнього середовища, генерує номер вихідного сигналу - елемента множини виходів

$$Y = \langle \text{Object1.Method1}, \dots, \text{Objectk.Methodk} \rangle.$$

У випадку, коли після звертання до одного з методів множини **Y** керування знову повинне здійснювати даний монітор, оператор вибору замикається в нескінченний цикл, причому принаймні один з методів повинний реалізувати вихід з цього циклу.

Repeat

Case ExternalObject.GetVariant(Parameter) **of**

Variant_1 : Object1.Method1(Parameter);

.....

Variant_k : Objectk.Methodk(Parameter);

end;

until False;

2. Монітори з вектором керуючих станів - ще один простий варіант керування, що часто зустрічається. У цьому варіанті керування описується вектором, кожна координата якого реалізує значення одного з параметрів керування. У розглянутому прикладі редактора рядка вектор **S** керування досконалішої версії редактора має три координати:

S = <Insert, Caps Lock, Language>

У принципі можливі наступні варіанти реалізації таких моніторів:

- хешування;
- вкладене розгалуження;
- комбінування хешування і вкладеного розгалуження.

Керування може здійснювати як сам керуючий об'єкт, так і керуючий метод керованого об'єкта. Ось деякі варіанти:

1. Монітор здійснює перехід до нового стану **S**, визначає об'єкт, якому необхідно передати керування, і передає вектор станів **S** як параметр відповідному методу-монітору з множини виходів **Y**, а цей метод вже розпізнає і здійснює керування у відповідності зі значенням **S**.
2. Монітор здійснює перехід до нового стану **S**, а потім здійснює вибір об'єкта за значенням **S**, а метод-монітор цього об'єкта здійснює вибір методу за значенням **X**.

3. Монітор здійснює перехід до нового стану **S**, а потім здійснює вибір об'єкта і методу за значенням **S**.

4.6. Вправи

1. Розгляньте документ *Паспорт громадянина України*. Здійсніть його опис за допомогою декомпозиції на найпростіші об'єкти-атрибути.
2. Розгляньте об'єкт *Запис у записній книжці*, у якій звичайно зберігають дані про знайомих. Визначте цей об'єкт у термінах об'єктів-атрибутів.
3. Розгляньте об'єкт *Корабель для гри "Морський бій"*. Визначте цей об'єкт у термінах об'єктів-атрибутів.
4. Визначте екранний об'єкт *Логотип з надписом, що грає роль ярлика файлу*. Визначте його атрибути і визначте протокол. Які методи відносяться до ярлика, а які – до файлу?
5. Визначте загальні операції керування системами "Телевізор", "Магнітофон", "Програвач" і опишіть абстрактний об'єкт для керування цими об'єктами.
6. Опишіть об'єкт «Система керування ліфтом», призначений для пасажирського ліфта багатоповерхового будинку.
7. Визначте загальні операції керування зовнішніми об'єктами "Вікно", "Меню", "Панель інструментів" і опишіть абстрактний об'єкт для керування цими об'єктами.
8. Опишіть абстрактний об'єкт *Миша*, що повинна використовуватися для керування вікнами.
9. Опишіть абстрактний об'єкт *Вікно*. Використовуйте для цього свій досвід керування вікнами в Windows.
10. Розглянете об'єкт *Найпростіший калькулятор*. Опишіть його обчислювальні можливості у формі протоколу об'єкта-обчислювача.
11. Розгляньте абстрактний тип даних, що описує методи внутрішнього сортування і пошуку. Опишіть цей тип у формі протоколу об'єкта-обчислювача.

12. Розгляньте абстрактний тип даних Алгебра підмножин. Опишіть цей тип у формі протоколу об'єкта-обчислювача.

4.7. Висновки

- Об'єкт характеризується структурою, станом, поведінкою і ідентичністю.
- Структура і поведінка однакових об'єктів описуються в загальному для них класі.
- Структура об'єкта – сукупність його атрибутів. Вона, як правило, схована від зовнішнього світу.
- Стан об'єкта визначає його статичні і динамічні властивості.
- Поведінка об'єкта описується його протоколом – сукупністю методів і властивостей, доступних іншим об'єктам.

5. ДИНАМІЧНІ ОБ'ЄКТИ

Нагадаємо, що динамічні структури даних широко використовуються в програмуванні для реалізації алгоритмів, що вимагають управління розподілом пам'яті в процесі виконання програми. Теорія і практика програмування використовує як неструктуровані динамічні дані, так і такі структури, як списки, стеки, черги, дерева і т.д. У мові програмування Паскаль динамічні дані і структури даних реалізуються засобами посилального типу.

Об'єкти, реалізовані динамічно, називають динамічними. Пам'ять для розміщення динамічних об'єктів розподіляється в процесі виконання програми.

5.1. Реалізація динамічних об'єктів

Відмітимо, що насправді пам'ять розподіляється для інформаційних полів динамічного об'єктного типу так само, як і при роботі з посиланнями на записи. Тому динамічні об'єкти описуються й обробляються звичайним чином. Саме, для створення динамічного об'єкта потрібно описати відповідний об'єктний тип статично, а потім описати об'єктний тип-посилання.

Розглянемо приклад.

Приклад 5.1.

Нехай об'єкт типу TLabel (етикетка) визначається атрибутами Name (ім'я) і Quantity (кількість) і методами Init, GetName, GetQuantity, AddQuantity. Додатково визначаємо посилальний тип TWare (товар) як посилання на TLabel. Розділ реалізації модуля Ware містить опис тіл методів.

**Unit Ware;
Interface**

```
    Type
TLabel = object
public
    constructor Init(S : String; N : Integer);
    procedure GetName(var S : String) ;
    function GetQuantity : Integer;
```

```

procedure AddQuantity(M : Integer);
private
    Name : String;
    Quantity : Integer;
end;
TWare = ^TLabel;
Implementation
constructor TLabel.Init(S : String; N : Integer);
begin
    Name := S; Quantity := N;
end;
procedure TLabel.GetName(var S : String) ;
begin
    S := Name
end;
function TLabel.GetQuantity : Integer;
begin
    GetQuantity := Quantity
end;
procedure TLabel.AddQuantity(M : Integer);
begin
    Quantity := Quantity + M
end;
end.

```

Розглянемо спочатку варіант використання динамічного об'єкта, що не використовує OO синтаксичних розширень мови.

```

uses CRT, Ware;
var Car : TWare;
    S : String;
begin
    ClrScr;
    New(Car);           { * }
    With Car^ do begin
        Init('Ford', 10);    { * }
        GetName(S);
    end;

```

```

        Writeln('TovarName : ', S);
        Writeln(' Q = ', GetQuantity);
        AddQuantity(5);
        Writeln(' Q = ', GetQuantity);
    end;
    Dispose(Car);
    ReadKey
end.

```

ОО розширення синтаксису мови Паскаль полягає в розширенні можливостей застосування операції New. По-перше, операція New може використовуватися як функція з аргументом - ім'ям посилального типу, об'єкт якого розміщується в пам'яті, по-друге, можливе додавання другого аргументу - звертання до конструктора відповідного об'єктного типу (у нашому прикладі конструктору TLabel.Init).

У нашому прикладі замість операторів, відзначених {*}, можна використовувати звертання

```
New(Car, Init('Ford', 10));
```

чи звертання до функції New з наступною ініціалізацією

```
Car := New(TWare); Car^.Init('Ford', 10);
```

чи звертання до функції New з одночасною ініціалізацією

```
Car := New(TWare, Init('Ford', 10));
```

Зауважимо, що таке розширення можливостей New коректно тільки тоді, коли протокол відповідного об'єктного типу містить конструктор (Init) у явному виді, а не процедуру (Init), що виконує дії по ініціалізації статичного об'єкта. Тому варто дотримуватися правила:

Опис будь-якого об'єктного типу повинний включати конструктор цього типу.

Поряд з ОО розширенням можливостей операції New реалізоване також відповідне розширення операції Dispose, що звільняє пам'ять, що динамічно розподіляється. Саме, у процедурі Dispose стало можливим використання другого параметра. Другим параметром може бути так званий деструктор.

Деструктором називають спеціальний метод, призначений для коректного завершення обробки динамічного об'єкта, що полягає

- у звільненні пам'яті, розподіленої для цього об'єкта (так званим збирання сміття);
- виконання інших дій, що супроводжують збирання сміття (наприклад, видалення посилань на екземпляри об'єктів, що видаляються, видалення графічного образу цього екземпляра з екрана, і т.п.)

Заголовок деструктора має вид

Destructor <Ім'я>;

Таким чином, синтаксично деструктори виділяються службовим словом **Destructor**, використовуваним замість **Procedure**. Імена деструкторів можуть бути другим аргументом операції **Dispose**. Наприклад:

Dispose(Car, Done).

Тут процедура **Dispose** збирає сміття, а метод **Done** виконує роботу, яка повинна супроводжуватися збиранням сміття. Відмітимо, що ім'я **Done** не є зарезервованим, але його використання загально-прийняте, так само, як і імена **Init**, **Get**, **Put**, і т.д.

Деструктори застосовуються тільки для динамічних об'єктних типів. При спадкуванні динамічних типів, що ми будемо вивчати пізніше, метод **Done** часто необхідно робити віртуальним, оскільки пізніше зв'язування, за допомогою якого реалізуються віртуальні методи, часто використовується при роботі з динамічними об'єктами. Для збирання сміття в цьому випадку необхідно використовувати операцію **Dispose(Car, Done)**. Звертання до методу **Done** не виконує цієї роботи. Якщо ніяких додаткових дій при зборці сміття не потрібно робити, тіло методу **Done** містить просто порожній розділ операторів. Наприклад:

```
Destructor TLabel.Done;
begin
end;
```

У цьому випадку звертання до деструктора ініціює пошук у таблицях ТДМ чи ТВМ реального розміру пам'яті, зайнятого динамічним об'єктом і передачі процедурі **Dispose** цього значення (розмір пам'яті, зайнятий об'єктом, визначеним за допомогою спадкування, може бути

різним у залежності від того, до якого класу цей об'єкт у дійсності належить). Ці системні дії називають *епілогом*.

У нашому прикладі віртуальні (динамічні) методи не визначені, тому і деструктор не визначений.

Закінчуючи розгляд нашого простого приклада здійснимо деякий критичний аналіз, відзначивши недоліки. Помітимо, що в розділі операторів основної програми ми використовували методи об'єктного типу TLabel, а не TWare. TWare узагалі не є об'єктним типом, а тільки посиланням на такий тип. Оскільки власного імені в екземпляра класу TLabel, з яким ми працювали, немає, нам довелося використовувати позначення Car[^]. І явне використання посилального типу, і, як наслідок, використання кваліфікованих імен, є порушенням канонів ОО програмування. Порушено принцип "Усе являється об'єктами". У нашому прикладі Car - не об'єкт. Порушені також принципи інкапсуляції Парнасу, відповідно до яких користувач об'єктного типу не повинний знати нічого, крім методів цього типу. З цієї точки зору знання того, що потрібно використовувати кваліфіковані посилальні імена, є зайвим, а значить - шкідливим.

Які практичні висновки випливають з цієї критики? По-перше, якщо Ваша програма працює, незважаючи на допущені в ній відступи від канонів ООП, ніхто не буде дивитися на висхідний текст. Отже, відступ від принципів (як, утім, і проявлена там геніальність) виявлена не буде. По друге, і це набагато цікавіше, зазначені недоліки можна виправити.

Можна, наприклад, визначити TWare як об'єктний тип з одним атрибутом - посиланням на Tlabel і визначивши в ньому методи, що здійснюють звертання до відповідним методам Tlabel.

Unit Goods;

Interface

uses Ware;

Type

TWare = **object**

Ware : ^TLabel;

constructor Init(S : String; N : Integer);

procedure GetName(var S : String) ;

```

    function GetQuantity : Integer;
    procedure AddQuantity(M : Integer);
end;

Implementation
constructor TWare.Init(S : String; N : Integer);
begin
    New(Ware);
    Ware^.Init(S, N);
end;
procedure TWare.GetName(var S : String) ;
begin
    Ware^.GetName(S)
end;
function TWare.GetQuantity : Integer;
begin
    GetQuantity := Ware^.GetQuantity
end;

procedure TWare.AddQuantity(M : Integer);
begin
    Ware^.AddQuantity(M)
end;
end.

```

Об'єктний тип TWare описаний тільки для того, щоб сховати посилання у викликах методів TLabel. Операції з об'єктами типу TWare менш ефективні за часом, оскільки їх виконання вимагає зайвих викликів процедур або функцій. Якому з варіантів реалізації віддати перевагу – справа програміста.

5.2. Об'єкти-залежності

Об'єкти системи можуть бути зв'язані між собою (залежати один від одного) за даними. Наприклад, такі об'єкти як Викладач і Кафедра зв'язані залежністю “працює”.

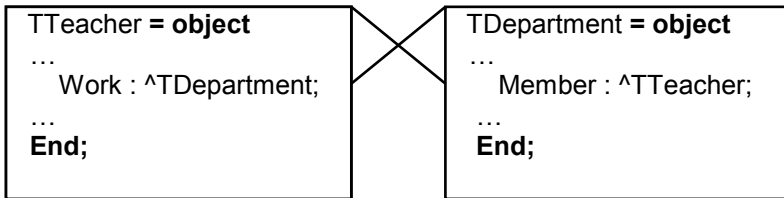
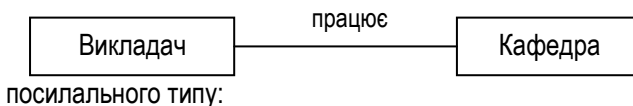


Рис. 5.1. Залежність Викладач-Кафедра.

Відмітимо, що з погляду математики термін “залежність” означає відношення, визначене на об’єктних типах. Докладний аналіз типів зв’язків-залежностей, що можуть бути реалізовані між об’єктами системи, буде здійснений пізніше. Зараз ми звернемо увагу тільки на ті аспекти цього поняття, що призводять до визначення об’єктів особливого типу – так званих об’єктів-залежностей. Зв’язок “Викладач-кафедра”, зображена на рис. 5.1, реалізується інформаційними полями



посилального типу:

Рис. 5.2. Зв’язок Викладач-Кафедра.

Зрозуміло, що зв’язки між об’єктами використовуються для визначення таких властивостей, що не зводяться тільки до функцій атрибутів.

Часто виникає необхідність вирішувати задачі, що відносяться не стільки до окремих об'єктів, скільки до зв'язків між ними. Наприклад, у задачах обробки родинних відносин між людьми ключовим поняттям є відношення споріднення. Декілька людей можуть бути зв'язані між собою родинними (сімейними) зв'язками. У цьому випадку природним рішенням задачі організації керування є визначення об'єкта "Родина", атрибути якого вказували б на об'єкти-члени родини. Природно, у такому випадку повинен бути реалізований і зворотний зв'язок. Зверніть увагу, що виділення об'єкта Родина дозволяє раціонально представити дані про родинний стан кожного члена родини. Зокрема, для кожного об'єкта типу Людина досить легко визначаються властивості, що описують сімейні зв'язки. Якщо ж об'єкт Родина не визначений, об'єкт Людина повинен містити атрибути Чоловік, Дружина, Дитина1, Дитина2. Це, по-перше, призводить до дублювання інформації, по-друге – ускладненню методів визначення властивостей, по-третє – сприяє виникненню помилок (У чоловіка, наприклад, не може бути чоловіка.)

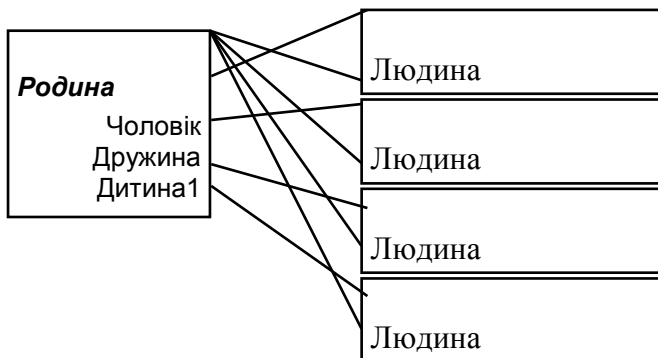


Рис. 5.3. Об'єкт Родина – зв'язок декількох об'єктів типу Людина

У розглянутому нами прикладі об'єкт Родина містить тільки допоміжні атрибути - посилання на членів родини. У більш загальній ситуації об'єкт-залежність може зберігати ще і дані інших типів, що є загальними для всіх об'єктів, зв'язаних залежністю. У нашому прикладі

об'єкт Родина може зберігати адресу будинку, у якому родина проживає.

Зв'язки між програмними об'єктами можуть бути реалізовані не тільки у виді об'єктів-залежностей. Об'єкти-залежності можуть бути включені (агреговані) в інші об'єкти, що призводить до зменшення кількості зв'язків у залежності. Розглянемо ще один приклад.

Приклад 5.2. Розклад занять факультету ВНЗ

У самому загальному виді такий розклад зв'язує наступні об'єкти:

- день тижня, номер пари;
- курс, група;
- дисципліна, вид заняття;
- викладач;
- аудиторія.

Однак на практиці розклад як реальний об'єкт представляють декілька інакше. Лист паперу, що відображає цей розклад, може бути визначений як двовимірний масив `Schedule[PairId, GroupId]`, елементами якого є залежності

<Дисципліна, вид заняття; Викладач; Аудиторія>

Таке представлення відношення між п'ятьма об'єктами зменшує кількість об'єктів-атрибутів залежності до трьох. Клітка розкладу відкриває доступ до атрибутів об'єктів Дисципліна, Викладач, Аудиторія. Тому, наприклад, легко вирішується задача заміни викладача, що читає дану дисципліну.

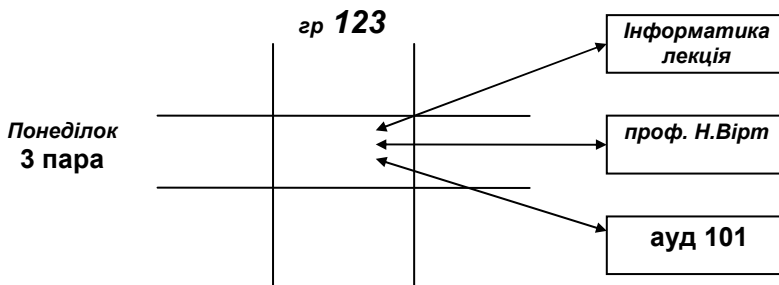
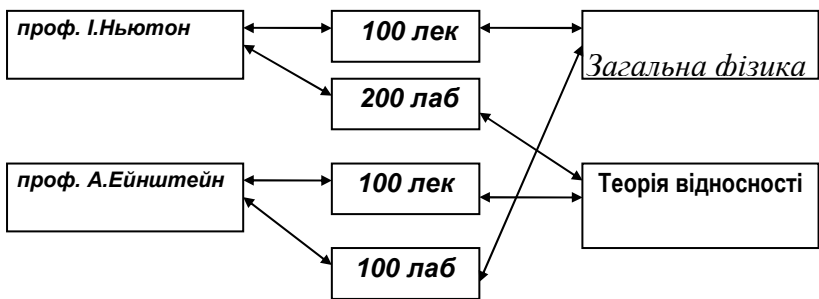


Рис. 5.4. Реалізація об'єкта Розклад занять.

Ще один варіант реалізації розкладу, більш прийнятний для задач планування навчального процесу, полягає в тім, щоб визначити об'єкт-залежність

Розподіл навантаження = <Викладач - Дисципліна>.

Залежність Розподіл навантаження, крім посилань на об'єкти



Дисципліна, Викладач, повинна містити ще кількість годин, що доручено виконати даному викладачу по даній дисципліні. Тоді задача розподілу навчального навантаження буде зв'язана з задачею Навчальний розклад природним чином.

Рис. 5.5. Варіант реалізації об'єкта Розклад занять

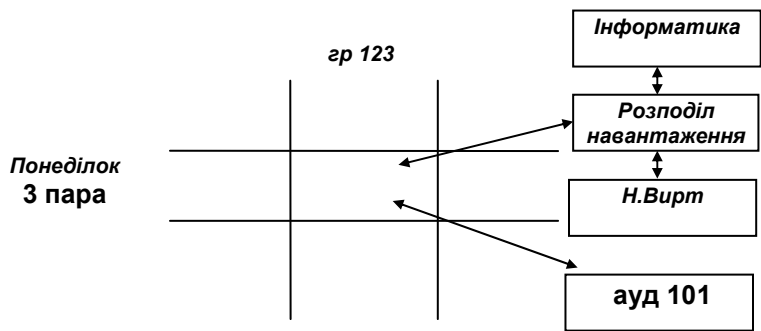


Рис. 5.6. Розклад занять і Розподіл навантаження.

5.3. Опис динамічних структур даних.

Зв'язні динамічні структури даних (списки, стеки, черги, дерева, графи і т.п.) завжди були в центрі уваги теорії і практики програмування. Велика увага цим структурам даних приділяється й в об'єктно-орієнтованому програмуванні. Зауважимо, що методи роботи з динамічними структурами даних точно визначені. Вони описуються фактичними стандартами, а в деяких мовах програмування - стандартизовані і на рівні синтаксису.

Наприклад, говорячи про стек, будь-який програміст має на увазі послідовність однотипних даних, до якої організований доступ за принципом *First in - Last Down* (першим увійшов - останнім вийшов). Операції над стеками - Push, Pop, isBottom фактично є стандартними не тільки семантично, але і на рівні імен. Таким чином, поняття об'єкта є ідеальним засобом опису і використання динамічних структур даних.

Описи динамічних структур як об'єктів відрізняються деякими істотними особливостями. Разом з тим динамічні структури часто використовуються для реалізації об'єктів. Ми розглянемо реалізацію об'єкта – двозв'язаного списку, що, наприклад, використовується для представлення списку найменувань і цін товарів (прайс-листа).

Перш, ніж приступити до проектування, уточнимо призначення прайс-листа. Цей об'єкт повинен використовуватися як його автором (продавцем), так і покупцями. Навіть якщо припустити для простоти, що опис товару складається тільки з його найменування і ціни, операції з прайс-листом для продавця і покупця повинні бути різними. Прайс-лист грає різні ролі, і це обумовлено різними ролями людей, що його використовують.

Визначимо в нашому прикладі прайс-листа для продавця, оскільки саме в цій ролі виявляється його динамічна природа.

Приклад 5.2. Прайс-лист

Атрибути: найменування, ціна.

Операції:

- Створити новий;
- Вставити новий товар;
- Вилучити товар;
- Перейти нагору (до попереднього товару);
- Перейти вниз (до наступного товару);
- Вилучити.

Ми перелічили тільки мінімально необхідний стандартний набір операцій над списком. Ми також не включили до цього протоколу ті операції, що, власне кажучи, працюють з окремим елементом списку. Це відноситься до таких операцій, як, наприклад, операції

- Призначити нову ціну товару;
- Роздрукувати дані по товару;

і іншим аналогічним операціям. Зрозуміло, що в повному протоколі ці операції повинні бути присутні. Методи цих операцій зі списком повинні звертатися до відповідного методам операцій з товаром (див. приклад 5.2.).

Визначивши атрибути об'єкта Прайс-лист, можна приступити до його реалізації. Необхідно розуміти, що реалізації можуть бути різними. Можна, наприклад, реалізувати Прайс-лист у виді масиву, пари стеків, двох зв'язкового списку, ще якимось. Наш вибір обумовлений, насамперед, навчальними цілями.

Першою особливістю реалізації, характерної не тільки для списків, але і для зв'язаних динамічних об'єктів узагалі, є те, що в описі беруть участь як об'єкт-елемент структури, так і об'єкт-структура. Ці об'єкти зв'язані рекурсивним описом, тому в реалізації методів об'єкт-структури використовуються інформаційні поля-зв'язки між об'єктами-елементами. Немає рації їх ховати один від одного. Тому об'єкт-елемент структури й об'єкт-структура описуються в одному модулі.

Один з можливих варіантів опису інтерфейсної частини відповідного модуля приведений нижче.

Unit ListUnit;

Interface

Uses UnWare;

Type

TLabel = ^TWare;

TList = ^TItem;

TItem = **object**

private

Content : TLabel;

Next : TList;

Prev : TList;

public {methods}

Constructor Init(WareName : String; WarePrice : Real);

Destructor Done;

end;

TPriceList = **object**

private

List : TList;

public

Constructor Init;

Procedure Insert(var Item : TItem);

Procedure Delete;

Procedure Up;

Procedure Down;

Procedure Done;

end;

Наступною важливою деталлю реалізації є опис елемента списку. Для посилань Next і Prev альтернативи немає. Поле Content відіграє ключову роль. Його можна описати як TWare чи TLabel. Розглянемо кожний з цих варіантів. Рис. 5.7. ілюструє розходження цих описів.

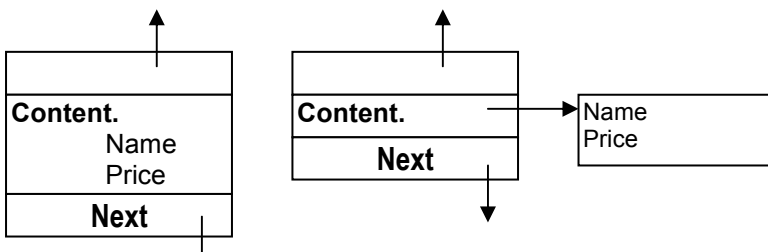
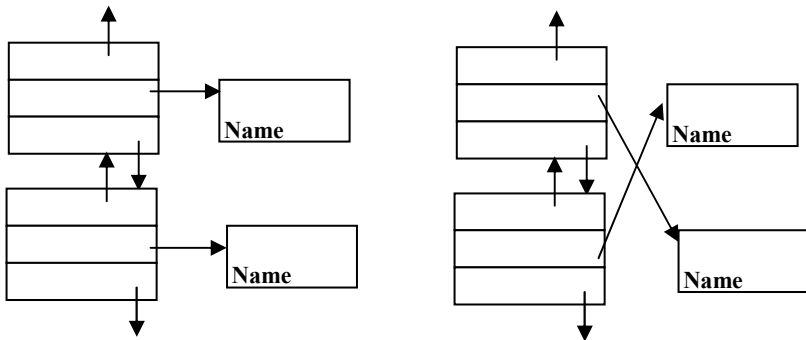


Рис. 5.7. Варіанти описів елемента списку.

У першому варіанті поле Content елемента списку – статичний об'єкт. В другому варіанті – динамічний об'єкт. Деякою перевагою першого варіанта є економія пам'яті на адресі поля Content. У другого варіанта переваг набагато більше. Головне з них - можливість динамічного керування обробкою елементів списку. Наприклад, для того, щоб поміняти місцями два елементи списку, необхідно усього лише перекинути два посилання, не торкаючися змісту (рис. 5.8.)



Мал. 5.8. Операція заміни місцями елементів списку.

З розділу реалізації приведемо лише визначення методів Insert і Delete.

Implementation {Роздел реалізації класу TPriceList}

```
.....  
procedure TPriceList.Insert(Ware : TLabel);  
var  
    Item : TList;  
    S : String;  
begin  
    Ware.GetName(S);  
    Item^.Init(S, Ware.GetPrice);  
    If (List<> Nil)  
        then begin  
            if (List^.Prev <> Nil) then begin  
                List^.prev^.next := Item;  
                Item^.prev := List^.prev;  
            end;  
            List^.prev := Item;  
            Item^.next := List;  
        end;  
    List := Item  
end;  
Procedure TPriceList.Delete;  
Var  
    Temp : TList;  
begin  
    Temp := List;  
    If (List^.Prev <> Nil) and (List^.Next <> Nil)  
        then begin  
            List^.Prev^.Next := List^.Next;  
            List^.Next^.Prev := List^.Prev;  
            List := List^.Next;  
        end;  
    If (List^.Prev <> Nil)and(List^.Next = Nil)  
        then begin  
            List := List^.Prev;  
            List^.Next := Nil;  
        end;  
    If (List^.Prev = Nil)and(List^.Next <> Nil)  
        then begin  
            List := List^.Next;  
            List^.Prev := Nil;
```



```

end;
If (List^.Prev = Nil)and(List^.Next = Nil)
    then List := Nil;
        Dispose(Temp, Done)
end;

```

.....
end. {кінець розділу і модуля}

5.4. Висновки

- Динамічні об'єкти використовуються для реалізації моделі реальних об'єктів, життєвий цикл яких визначається динамічно – у процесі виконання програми.
- Об'єктно-орієнтовані засоби мови Borland Pascal, що підтримують роботу з динамічними об'єктами – методи Constructor, Destructor, розширені операції New, Dispose.
- Динамічні об'єкти використовуються для реалізації таких спеціальних типів, як об'єкта-залежності, об'єкти – динамічні структури даних.

5.5. Вправи

1. Опишіть динамічно об'єкт "Паспорт громадянина України".
2. Опишіть динамічно екранний об'єкт "Поле введення тексту".
3. Визначите динамічно екранний об'єкт "Логотип з написом", що грає роль ярлика динамічного об'єкта - файлу.
4. Визначите об'єкт-залежність, що визначає користувача персонального комп'ютера в комп'ютерному класі.
5. Визначите об'єкт-залежність, що визначає читача книги бібліотеки.
6. Визначите об'єкт-залежність, що визначає постачальника і покупця товару, проданого магазином.
7. Опишіть динамічно екранний об'єкт "Логотип з написом". Реалізуйте динамічно об'єкт "Папка" як прямокутну область екрана з розташованими в ній декількома об'єктами типу "Логотип з написом". Основні операції – Виділити, Додати, Видалити.
8. Опишіть динамічно об'єкт "Запис у записній книжці", у якій звичайно зберігають дані про знайомих. Реалізуйте об'єкт "Записна книжка" у виді об'єкта – динамічної структури даних.

9. Опишіть динамічно об'єкт "Картка читача бібліотеки". Реалізуйте динамічно об'єкт "Список читачів бібліотеки". Основні операції визначите самостійно.
10. Опишіть динамічно об'єкт "Картка книги бібліотеки". Реалізуйте динамічно об'єкт "Каталог бібліотеки". Основні операції визначите самостійно.
11. Опишіть динамічно об'єкт "Рядок однорівневого меню". Реалізуйте об'єкт "Однорівневе меню" у виді об'єкта – динамічної структури даних.
12. Опишіть динамічно об'єкти "Тепловоз" і "Вагон потяга". Реалізуйте об'єкт "Потяг" у виді об'єкта – динамічної структури даних з основними операціями "Створити потяг", "Додати вагон", "Видалити вагон", "Знайти вагон".
13. Опишіть динамічно графічний об'єкт "Багатокутник" (замкнута ламана). Основні операції – "Виділити вершину", "Додати вершину", "Видалити вершину".

6. СПАДКУВАННЯ

6.1. Реалізація спадкування

Спадкування підтримується в середовищі Borland Pascal спеціальними засобами. Нехай об'єктний тип TClassA успадковується об'єктним типом TClassB. Цей факт виражається в такий спосіб:

Type

```
TClassA = object
    {Опис атрибутів і методів TClassA}
end;

TClassB = object(TClassA)
    {Опис власних атрибутів і методів}
end;
```

Цей опис означає, що в об'єктному типі TClassB визначені всі атрибути і всі методи типу TClassA, а також власні атрибути і методи, тобто ті, котрі описані в тілі опису TClassB. Тому повторне використання імен атрибутів типу TClassA у тілі визначення TClassB є помилкою, що відслідковується компілятором. Повторний опис методу, ім'я якого уже було використано в описі методу батьківського класу TClassA, допустимо й означає так назване перевизначення методу. Техніку перевизначення методів ми розглянемо нижче.

Приклад 6.1. Опис класу циліндрів TCilinder як спадкоємця класу TCircle (Коло).

```
TCircle = object
private
    Radius : Real;
public
    Constructor Init(R : Real);
    Procedure GetRadius;
```

```

        Procedure GetSquare;
        Destructor Done;
        { інші методи}
end;

TCilinder = object(TSquare)
private
    Height : Real;
public
    Constructor Init(R, H : Real);
    Procedure GetHeight;
    Procedure GetVolume;
    Destructor Done;
    {інші методи}
end;

```

Клас TCilinder визначений тепер атрибутами Radius, Height і методами GetRadius, GetSquare, GetHeight, GetVolume. Атрибут Radius, методи GetRadius, GetSquare успадковані, атрибут Height і методи GetHeight, GetVolume є власними. Конструктор Init і деструктор Done перевизначені.

Приклад 6.2. Геометричні перетворення площини.

У цьому прикладі приведений досить повний опис об'єктного типу TShape, що використовує спадкування. Нехай нам необхідно описати групу об'єктних типів - геометричних фігур, які необхідно піддавати геометричним перетворенням: паралельним переносам, обертанням, перетворенням масштабу. Як базисні операції ми будемо використовувати паралельний перенос фігури на вектор, поворот фігури навколо фіксованої точки, масштабування. Основна ідея в реалізації цих операцій полягає в тім, щоб кожен фігуру визначати координатами деякої початкової точки (на рис.6.1 - точка A) і відносними координатами ще однієї точки (на рис. 6.1. - точка B).

При такому підході перетворюватися будуть тільки координати цих двох точок, а координати інших точок,

використовуваних при рисуванні, можна переобчислювати. Тип TVector ми цілком не описуємо. Рисунок 6.1. ілюструє описаний підхід, а листінг, що приведений нижче, містить описи типів TVector, TInitPoint, TShape, які використовуються в описах і невеличкій демонстрації.

Рис.6.1. Геометричні перетворення площини – ілюстрація методу

Type

```
TVector = object
private
  X, Y : Real;
public
  { Опису операцій типу TVector}
end;
{-----}
TInitPoint = object
private
  CoordX,
  CoordY: Real;
public
  Procedure Init(a, b : Real);
  Procedure Move(V : TVector);
  Procedure Print;
End;
{-----}
TShape = object(TInitPoint)
```

private

InitVertex : TInitPoint;

public

Procedure Init(X0, Y0 : Real; X1, Y1 : Real);

Procedure Rotate(Alpha : Real);

Procedure ChangeSize(Coef : Real);

Procedure Print;

End;

{-----}

Procedure TInitPoint.Move(V : TVector);

begin

CoordX := CoordX + V.X;

CoordY := CoordY + V.Y;

end;

Procedure TInitPoint.Init(a, b : Real);

begin

CoordX := a;

CoordY := b;

end;

Procedure TInitPoint.Print;

begin

Writeln(' X = ',CoordX:1:1, ' Y = ', CoordY:1:1)

end;

Procedure TShape.Init(X0, Y0 : Real; X1, Y1 : Real);

begin

inherited Init(X0, Y0);

InitVertex.Init(X1, Y1);

end;

Procedure TShape.Rotate(Alpha : Real);

Var U, V : Real;

begin

U := Cos(Alpha);

V := Sin(Alpha);

```

    With InitVertex do begin
        CoordX := U*CoordX + V*CoordY;
        CoordY := -V*CoordX + U*CoordY;
    end;
end;

Procedure TShape.ChangeSize(Coef : Real);
begin
    With InitVertex do begin
        CoordX := Coef*CoordX;
        CoordY := Coef*CoordY;
    end;
end;

Procedure TShape.Print;
begin
    TInitPoint.Print;
    With InitVertex do Print;
    Writeln('=====')
end;
{----- Демонстрація -----}
Var
    V : TVector;
    S : TShape;

begin
    ClrScr;
    S.Init(1, 2, 1, 1);      S.Print;
    V.X := 1;
    V.Y := 1;
    S.Move(V);              S.Print;
    S.Rotate(Pi/2);         S.Print;
    S.ChangeSize(10);       S.Print;
    ReadKey
end.

```

Зауважте, що тип TShape успадковував метод Move. Справді, операція паралельного переносу при такому підході до опису геометричної фігури може застосовуватися тільки до початкової точки, тому її має сенс реалізувати в типі TInitPoint, а потім успадковувати. Об'єктні типи конкретних геометричних фігур тепер повинні успадковувати тип TShape, оскільки його властивості і методи є базовими. Всі обчислення, що описують власне геометричні перетворення, у ньому уже реалізовані.

Приклад 6.3. Інформаційна система Кафедра

Нехай для реалізації проекту “Інформаційна система Кафедра” необхідно реалізувати обробку даних про викладачів кафедри ВНЗ і про студентів, що навчаються на цій кафедрі. Таким чином, нам знадобляться об'єкти Викладач (TTeacher) і Студент (TStudent). На етапі аналізу проекту системи ми зауважуємо, що ці об'єкти мають загальні атрибути і загальні операції. Причина цієї спільності полягає в тому, що і ті, і інші - це люди, зайняті у ВНЗ. (Одні там працюють, інші - навчаються). Тому ми вводимо загальний об'єктний тип TPerson і визначаємо типи TTeacher і TStudent як дочірні типи TPerson.

```
TPerson = object  
    { Ім'я, домашня адреса, телефон, ...}  
End;
```

```
TTeacher = object( TPerson )  
    {кваліфікація, посада, ... }  
End;
```

```
TStudent = object( TPerson )  
    {спеціальність, курс, група, ...}  
End;
```

6.2. Перевизначення методів

Перевизначення в тілі опису дочірнього об'єктного типу деяких методів, вже описаних у батьківському класі, не тільки не

заборонено, але є одним з основних способів реалізації ідеї повторного використання програмного коду за допомогою механізму спадкування. Справді, успадковуючи властивості батьківського типу, об'єкти дочірнього типу повинні разом з тим модифікувати і свою власну поведінку, і взаємодії з іншими об'єктами. Ці модифікації, очевидно, полягають як у визначенні нових операцій, реалізованих власними методами, так і в зміні методів операцій, визначених у батьківському класі.

Типовим прикладом перевизначення методу є опис методу `Init` ініціалізації об'єкта. Оскільки в дочірньому класі визначаються нові атрибути, операція по своїй суті (семантиці) залишилася тією ж, а метод її реалізації повинен бути модифікований так, щоб ініціювати і значення цих нових, власних атрибутів.

Перевизначення успадкованого методу в обумовленому класі здійснюється у відповідності з наступними правилами:

- Використання імені методу успадкованого класу в описі дочірнього класу означає, що для дочірнього класу визначається власний метод з тим же ім'ям. Такий опис називають перевизначенням (пригнобленням) методу батьківського класу.
- Метод наслідуваного класу залишається доступним або під своїм повним ім'ям `<Ім'я батьківського типу>.<ім'я методу>`, або за допомогою словосполучення **inherited** `<ім'я методу>`. Службове слово **inherited** зарезервоване в мові для цієї мети.
- При перевизначенні методу успадковується тільки його ім'я. Список формальних параметрів методу може бути визначений заново - довільним чином.

У прикладі 6.2 перевизначені методи `TShape.Init` і `TShape.Print`. Ми використовували як перший, так і другий варіанти звертання до пригнобленого методу, хоча, на наш погляд, другий варіант виглядає більш природно. У тих же прикладах показаний прийом довизначення методу. Відзначимо, що перевизначення методу `TShape.Init` містить інший список формальних параметрів.

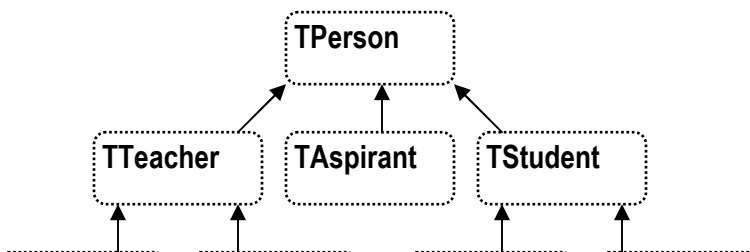


Рис.6.2. Фрагмент ієрархічної структури спадкування

Використання спадкування визначає на множині об'єктних типів деревоподібну структуру, яку називають ієрархічною структурою (спадкування) класів. Всі об'єктні типи - вузли цієї структури зв'язані воєдино відношенням прямого спадкування. Тому будь-яка вертикаль у цій структурі - шлях від кореня структури до її листів - являє собою послідовність класів, протоколи яких послідовно розширюються введенням власних атрибутів і методів.

Звертання до деякого методу, успадкованого одним із класів (скажемо, класом TClassA) такої послідовності під конкретним ім'ям, реалізований як пошук першого (найближчого) класу-предка, для якого цей метод є власним. Саме цей метод і виконується. Тому звертання-запити називають пошуком методу. Компілятор генерує повідомлення про помилку, якщо в результаті пошуку методу "знизу нагору" він не буде знайдений.

Припустимо, що в результаті пошуку метод буде знайдений у класі TClassB. Заголовок цього методу, отже, описаний у цьому класі. Тіло його опису може містити звертання до інших методів. Важливо знати і пам'ятати, що ці інші методи будуть інтерпретуватися як методи класу-предка TClassB незважаючи на те, що ці ж методи могли бути перевизначені для класу TClassA. Рис. 6.3. ілюструє пошук методу.

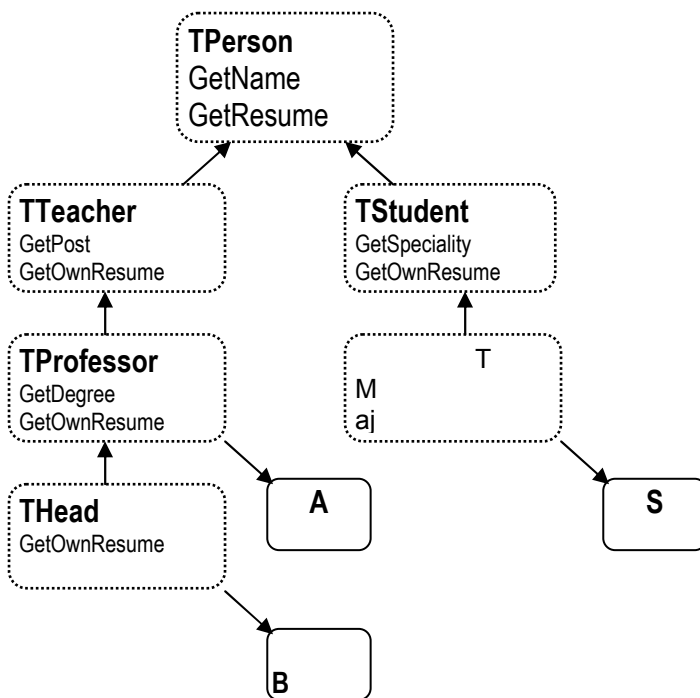


Рис. 6.3. Статичні і динамічні методи

На цьому рисунку зображений фрагмент ієрархії класів гіпотетичного проекту «Інформаційна система Кафедра» - розвиток приклада 3. Приналежність об'єктів A, B, S показана стрілками. Усі ці об'єкти можуть використовувати статичний метод `GetName`, визначений у класі `TPerson`. Зверніть увагу на те, що запит `B.GetName` буде шукати метод, починаючи з класу `THead` нагору по ієрархічному дереву і знайде його тільки в класі `TPerson`. Запит `B.GetDegree` знайде статичний метод у класі `TProfessor`, а запит `A.GetDegree` знайде цей метод у своєму власному класі.

Статична інтерпретація пошуку методу зв'язана з технічними деталями процесу генерації програмного коду реалізації пошуку. Справа в тім, що зв'язування програмного коду, що містить звертання до методу, з кодом підпрограми, що цей метод

реалізує, здійснюється на етапі компіляції і не може бути змінена в процесі виконання програми. Тому методи, що реалізовані таким чином, називають статичними.

Статичні методи інтерпретації пошуку методу іноді вступають у протиріччя з задумами програміста. Саме, часто зустрічаються ситуації, у яких природним способом реалізації пошуку методу було би спадкування тіла цього методу з власною інтерпретацією методів, до яких він звертається.

Приведемо приклад такої ситуації, використовуючи той же приклад (рис. 6.3.). Нехай однієї з операцій нашої інформаційно-пошукової системи повинна бути операція `GetResume`, що видає клієнту резюме - стислу інтегровану інформацію про будь-який об'єкт системи. За своїм змістом ця операція є загальною і, отже, повинна бути реалізована методом, що належить класу `TPerson`. Однак, крім тих відомостей, що є загальними для викладачів і студентів (Ім'я, вік і т.п.), резюме повинне містити специфічні відомості, що мають зміст тільки для конкретних класів – рейтинг для студентів молодших курсів, спеціалізацію – для старшокурсників, форму підвищення кваліфікації (магістратура, аспірантура, співшукачіство і т.п.) – для асистентів, ступені і звання – для лекторів. Ця специфічна інформація вибирається методами `GetOwnResume`, реалізованими в кожному класі індивідуально. Метод `GetResume` містить запит методу `GetOwnResume`: Власне резюме – це об'єкт типу `TResume`, атрибути якого використовуються методом `Tperson.GetResume`:

```
Procedure Tperson.GetResume(var R : TResume);
```

```
Begin
```

```
  With R do begin
```

```
    GetName(Name);
```

```
    Age := GetName;
```

```
    GetOwnResume(OwnResume)
```

```
  end
```

```
end;
```

Методи `GetResume` і `GetOwnResume` називають динамічними або віртуальними.

Приклад 6.2. (продовження)

Клас геометричних фігур TShape ми використовуємо для породження класів конкретних фігур TSegment (відрізок) і TCircle (окружність). Поставимо перед собою задачу моделювання складного руху цих фігур, що інтегрує перетворення паралельного переносу, повороту і подоби (стискання - розтягання). Метод моделювання полягає в тому, що рух фігури представляється у вигляді послідовності кроків, на кожному з яких здійснюються всі необхідні перетворення. Метод TransformStep, що виконує цю операцію, найвірніше описувати в класі TShape, оскільки в ньому визначені всі типи рухів, і, отже, можна визначити крок складеного руху незалежно від типів фігур, що рухаються.

```

Procedure TShape.TransformStep(V:TVector; Alpha:Real; Coef:Real);
begin
    Move(V);
    Rotate(Alpha);
    ChangeSize(Coef);
    Draw
end;
```

Для відображення цього руху на екрані монітора нам знадобиться операція рисунку кожної фігури, реалізацію якої виконує метод Draw. З одного боку, цей метод специфічний для кожного класу, а з іншого боку - логіка алгоритму пропонує включити операцію рисунку в тіло методу TShape.TpranformStep. Тому ми визначаємо в класі TShape приватний метод Draw, що повинний лише позначити свою присутність:

```

TShape = object(TInitPoint)
    .....
    Procedure Draw;
    .....

end;

Procedure TShape.Draw;
begin
end;
```

Зауважимо, що для визначення відрізка і окружності на площині не потрібні додаткові атрибути. Дві точки, визначені в класі TShape, цілком визначають і відрізок, і окружність. Тому описи відповідних класів мають вид:

```
Tsegment = object(TShape)
  private
    procedure Draw;
end;
```

```
TCircle = object(TShape)
  private
    procedure Draw;
end;
```

Тут наші задуми, цілком обґрунтовані з погляду логіки алгоритму, як і в попередньому прикладі, вступили в протиріччя з їх інтерпретацією. Справді, методи TSegment.TransformStep TCircle.TransformStep, успадковані від TShape, правильно перетворюють свої фігури, але не рисують їх. Розглянемо демонстрацію цього факту. У ній беруть участь наступні об'єкти, описані в розділі змінних

Var

```
V : TVector;
C : TCircle;
S : TSegment;
```

Коментарі до операторів наступного фрагмента прикладної програми демонструють розходження в інтерпретації методу Draw:

begin

```
ClrScr;
V.X := 1; V.Y := 1;
With C do begin
  Init(1, 2, 1, 1);           {Коло з даним радіусом і центром}
```

```

Draw;                                {Метод Draw із класу TCircle}
                                     {нарисоване коло у початковому положенні}
TransformStep(V, Pi/6, 2); {Метод звертається до TShape.Draw}
                               {Коло змінилося, але нічого не нарисовано}
end;

With S do begin
  Init(1, 2, 1, 1);
  Draw;                                {Метод Draw із класу TSegment}
                                     { нарисований відрізок в початковому положенні}
  TransformStep(V, Pi/6, 2);          {Метод звертається до Tshape.Draw}
                                     {Відрізок змінився. Нічого не нарисовано}
end;
end.

```

6.3. Динамічні методи.

Для реалізації операцій, повна інтерпретація яких полягає у використанні механізму спадкування одночасно з коректним перевизначенням пошуку методу, використовуються так звані віртуальні і динамічні методи.

В системі Borland Pascal динамічні і віртуальні методи відрізняються лише технічними деталями синтаксису і реалізації, але семантично вони вважаються еквівалентними.

Як ми бачили, причиною “неправильної” інтерпретації методу є статичне зв’язування програмного коду шуканого методу з програмним кодом методу, що здійснює пошук, тобто розміщення адреси викликуваної підпрограми в тілі програми, яка містить відповідний оператор виклику, на етапі компіляції. Такий зв’язок називають *раннім зв’язком*.

Альтернативою раннього зв’язку є динамічна реалізація звертання до підпрограми, яку називають *пізнім зв’язком*. Для динамічної реалізації пошуку методу компілятор створює спеціальні внутрішні таблиці - таблиці динамічних методів (ТДМ). У ТДМ розміщуються адреси звертань до різних інтерпретацій методу, що ідентифікується одним ім'ям, але реалізованого в різних об'єктних типах ієрархії спадкування. Звертання до динамічного методу в процесі виконання

програми ініціює пошук необхідної адреси в ТДМ. Для реалізації віртуальних методів компілятор також створює спеціальні таблиці - таблиці віртуальних методів (ТВМ), використовувани з тією же метою - для реалізації пізнього зв'язку. Оскільки обидва методи керування пошуком, власне кажучи, еквівалентні, ми вкажемо нижче їхні синтаксичні розходження, а для пояснення їхнього використання будемо використовувати терміни «динамічні» і «віртуальні» як синоніми.

Важливим для розуміння моментом реалізації динамічних методів є той факт, що в момент звертання до методу належність конкретного об'єкта конкретному об'єктному типу повинна бути однозначно розпізнана. З цією метою використовуються спеціальні методи, які називають конструкторами.

Конструктор - це спеціальний метод, виділений серед інших методів об'єктного типу синтаксично, що виконує необхідні початкові установки для реалізації пізнього зв'язку механізму динамічних методів.

У середовищі Borland Pascal кожен об'єктний тип, протокол якого містить динамічний метод, повинний включати також і конструктор об'єктів цього типу. Заголовок конструктора має вид

Constructor <Ім'я> (<список формальних параметрів>);

Таким чином, синтаксично різниця між методом-процедурою і методом-конструктором полягає у використанні службового слова Constructor замість Procedure.

Наступні правила регламентують описи і використання об'єктних типів з динамічними методами реалізації операцій:

1. Синтаксично віртуальні і динамічні методи ідентифікуються процедурною директивою **virtual**, що включається в заголовок методу. Для реалізації динамічного методу після службового слова **virtual** указується номер методу - ціле позитивне число (**virtual** <номер>). Номер динамічного методу повинен бути унікальним для даного методу. Директива **virtual** не дублюється в заголовку опису тіла методу.

2. Визначення динамічного методу в описі об'єктного типу означає, що у всіх типах, що успадковують даний, цей метод може бути визначений і оголошений динамічним.
3. Якщо в деякому класі метод оголошений динамічним, то при спадкуванні цього класу цей метод також успадковується.
4. Якщо у наслідуваному класі динамічний метод подавляється, то його перевизначення повинне бути оголошено динамічним.
5. Заголовок опису будь-якого динамічного методу при його перевизначенні в дочірньому класі повинен у точності відповідати заголовку цього методу у визначенні батьківського класу, разом з ім'ям і списком формальних параметрів.

Це означає, що всі динамічні методи в даному ланцюжку спадкування повинні мати один заголовок, визначений один раз - у відповідному батьківському класі. Цей заголовок просто копіюється для дублювання у всіх класах-нащадках. Помітимо, що при перевизначеннях статичних методів не існує ніяких обмежень на заголовки однойменних методів.

1. Визначення кожного класу, у якому описуються динамічні методи, повинне містити в собі конструктор цього класу. Конструктор класу може бути також успадкований.
2. Будь-який об'єкт класу (екземпляр класу), в якому визначені динамічні методи, повинен бути ініційований конструктором цього класу. Ініціалізація екземплярів класу здійснюється звертанням до конструктору цього класу.

Попередження: Якщо в програмі здійсниться пошук динамічного методу для об'єкта, що помилково не ініційований, компіляція пройде успішно, але процес виконання заблокує систему.

Приклад 6.2. (продовження)

Змінімо описи об'єктних типів відповідно до правил використання динамічних методів:

```
TShape = object(TInitPoint)
```

```

.....
Constructor Init(X0, Y0 : Real; X1, Y1 : Real);
Procedure Draw; virtual; {virtual 100;}
.....
end;

TSegment = object(TShape)
    procedure Draw; virtual; {virtual 100;}
    .....
end;

TCircle = object(TShape)
    procedure Draw; virtual; {virtual 100;}
    .....
end;

```

Помітимо, що власне конструктор Init визначений як статичний метод класу TShape, тому в дочірніх класах TSegment і TCircle він визначений коректно. Нижче приведений приклад використання конструктора Init.

Var

```

V : TVector;
C1, C2 : TCircle;
S1, S2 : TSegment;

```

begin

```

V.X := 1; V.Y := 1;
C1.Init(1, 2, 1, 1);    {ініційований екземпляр C1 класу TCircle }
C1.Draw;
C1.TransformStep(V, Pi/6, 2);
C2.Init(0, 0, 0, 0);    {ініційований екземпляр C2 класу TCircle }
C2 := C1;               {без ініціалізації присвоєння некоректне}
S1.Init(1, 2, 1, 1);    {ініційований екземпляр C1 класу TCircle }
S2.Init(0, 0, 0, 0);    {ініційований екземпляр C2 класу TCircle }
S1 := S2;
S2.Draw;

```

end.

6.4. Форми спадкування

Незважаючи на те, що механізм спадкування об'єктних типів досить простий, він може бути використаний програмістом для досягнення різних цілей. Нижче ми розглянемо деякі форми спадкування, що часто використовуються для опису ієрархії класів проектованої програмної системи.

Незалежно від форми спадкування, у визначенні беруть участь клас-батько і клас-нащадок. Власне форму спадкування визначає мета, що переслідує програміст при описі конкретної ланки ієрархії класів.

6.4.1. Визначення класу - спеціалізації батьківського класу.

При спеціалізації породжуваний клас має усі властивості й операції класу-батька. Спеціалізація означає визначення в дочірньому класі своїх власних властивостей і/або операцій. При цьому властивості і методи батьківського класу успадковуються в незмінному вигляді - тобто без перевизначень.

Наприклад, клас `TextField`, визначений у прикладі 4.5., має свою власну поведінку, яка визначена протоколом властивостей і методів. Спадкування може бути використано, наприклад, для визначення двох нових операцій - операції виведення тексту в поле й операції редагування і введення тексту.

В другому прикладі клас `TAdress` визначений атрибутами `Країна`, `Місто`, `Вулиця`, `Номер будинку`. Цих даних досить для того, щоб вказати адресу чи існування людини, що проживає в окремому будинку. Якщо ж адреса повинна містити ще і номер квартири, можна визначити клас `TAdressWithFlat`:

```
TAdressWithFlat = object(TAdress)
```

```
Flat : Word;
```

```
.....
```

```
procedure GetFlat;
```

.....
end;

Спеціалізація як форма спадкування означає породження об'єктного підтипу з повним виконанням принципу підстановки:

Будь-який об'єкт дочірнього типу може бути використаний у будь-якому програмному контексті як об'єкт батьківського типу. Його поведінка при цьому не відрізняється від поведінки об'єкта батьківського типу.

Всі методи, що містять як формальний параметр змінні типу TAddress, в якості фактичного параметру можуть використовувати об'єкти класу TAddressWithFlat.

Породження спеціалізованих класів - найбільш ідеальна форма спадкування, оскільки дочірній клас є підтипом батьківського класу.

Спеціалізація як форма спадкування використовувалася нами в прикладі 6.1. Відзначимо одну деталь цього прикладу, яку слід вважати якщо не помилкою, то недоліком. Для класів цього прикладу обрані імена TCircle (Коло) і TCilinder (Циліндр). Насправді важко уявити просторове тіло Циліндр як спеціальну форму плоскої фігури Коло. Тому термін "спеціалізація" погано погоджується з нашим інтуїтивним розумінням того, як з кола одержують циліндр. Насправді клас TCircle в контексті цього прикладу представляє не кола, а просторові тіла, проекція яких на площину XOY є кругом. Ім'я TRotationShape (фігура обертання) більш точно відбиває суть справи, оскільки циліндр і справді є фігурою обертання спеціальної форми.

6.4.2. Визначення класу - специфікації батьківського класу.

Специфікацією називають таку форму визначення дочірнього класу, при якій у дочірньому класі реалізована одна чи кілька операцій, що тільки описані, але не реалізовані в батьківському класі.

Таким чином, батьківський клас при цьому виступає в ролі абстрактного класу, що визначає єдиний протокол функціонування

одного чи декількох дочірніх класів. Класичний приклад специфікації – приклад 6.2 визначення класу TShape геометричних фігур, у якому реалізовані методи руху фігур, але не реалізований, а тільки визначений (оголошений) метод Draw зображення цих фігур.

Визначення абстрактних класів і наступні їхні специфікації для визначення більш конкретних класів - основний метод опису ієрархії об'єктних типів, у якому абстрагування як методологічний прийом приймає конкретні риси технології.

Відзначимо, що в прикладі 6.3 для побудови ієрархії використані як спеціалізації, так і специфікації.

Об'єктно-орієнтовані мови програмування підтримують специфікації такими засобами, як віртуальні методи і означення класів-шаблонів (контейнерних класів). На відміну від мови C++ і інших об'єктно-орієнтованих мов, Borland Pascal підтримує специфікацію як форму спадкування динамічними (віртуальними) методами. Класи-шаблони не реалізовані. Поняття класу-шаблону і деякі можливості моделювання класів цього типу будуть розглянуті нижче.

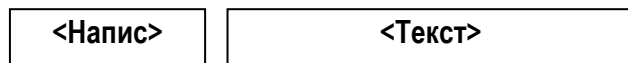
6.4.3. Визначення класу - конструкторії батьківського класу.

У деяких випадках обумовлений дочірній клас повинен успадковувати майже усі властивості і поведінку батьківського класу, але деякі деталі (імена чи параметри методів) вимагають зміни.

Нехай, наприклад, у проєктованій програмі потрібно виводити на екран текст в обрамленні двох написів



В розпорядженні програміста уже є об'єктний тип TCaptionText, який власне кажучи, майже співпадає із класом TTextField - текст із написом перед текстом виду



Тоді для визначення класу `TCaptionTextCaption` можна перевизначити статично методи `Init`, `Done` класу `TCaptionText`. Методи `Clear` і `Edit` операцій над текстовим полем `Content` при цьому успадковуються без змін. Програміст може також прийняти рішення про те, що ініціалізація і вилучення об'єкта з екрана монітора будуть не перевизначені, а реалізовані під іншими іменами. Тоді він отримає об'єктний тип, екземпляри якого мають дві форми представлення об'єкта на екрані.

У нашому прикладі дочірній клас не є ні спеціалізацією, ні специфікацією батьківського класу. Справді, не можна говорити про виконання принципу підстановки, оскільки не зберігається відношення "об'єкт В є об'єкт А": метод `Init` має інші параметри. Нарешті, дочірній клас не є специфікацією батьківського класу, оскільки його методи перевизначені статично.

Більш цікавим і важливим з погляду методології і техніки програмування є використання стандартних динамічних об'єктів – структур даних як шаблонів, до яких спадкування у формі конструювання застосовується з метою реалізації конкретних об'єктних типів.

Приклад 6.4.

Розглянемо об'єктний тип `TPriceList` приклада 5.2. Якщо атрибут `Content` класу `TItem` описати як `Pointer` і модифікувати відповідним чином метод `Insert`, ми одержимо об'єктний тип – двозв'язний список, елементами якого є покажчики типу `Pointer`. Назвемо цей клас ім'ям `TPointerList`. Для того, щоб одержати двозв'язний список `TPriceList` з елементами <Товар – Ціна> на базі класу `TpointerList`, потрібно перевизначити метод `Insert` так, щоб покажчик `Content` виставлявся на елемент списку, що конструюється. Оскільки покажчик типу `Pointer` універсальний, така обробка можлива. Відзначимо, що, по перше, клас `TPointerList` створено спеціально для того, щоб на його базі конструювати різні конкретні об'єкти, засновані на двозв'язних списках. По-друге, клас `TPointerList` не є абстрактним, оскільки об'єкти цього типу в принципі можна створювати. Суть справи, однак, полягає в тому, що об'єкти типу `TPointerList` просто не потрібні.

Приклад 6.5.

Розглянемо в якості ще одного приклада Клас T3DVector, елементами якого є 3-вимірні вектори. Якщо ми виберемо клас двовимірних векторів T2DVector у якості батьківського для визначення T3DVector, нам необхідно буде ввести новий атрибут Coord і перевизначити всі операції з векторами, доповнивши їх обробкою координати Z:

Interface

T3DVector = **object**(T2DVector)

private

CoordZ : Real;

public

Procedure GetXYProection(A: T3DVector; Axy: T2DVector);

Procedure Add(A, B : T3DVector);

....

end;

Implementation

Procedure T3DVector.GetXYProection(A: T3DVector; **var** Axy: T2DVector);

begin

Axy.CoordX := A.CoordX;

Axy.CoordY := A.CoordY;

end;

Procedure T3DVector.Add(A, B : T3DVector);

Var

Axy, Bxy : T2DVector;

begin

GetXYProection(A, Axy);

GetXYProection(B, Bxy);

Inherited Add(Axy, Bxy);

CoordZ := A.CoordZ + B.CoordZ;

end;

....

У цьому прикладі використовується трохи інша техніка, також характерна для спадкування. Розглянемо її деталі.

- Атрибут `Coord` - третя координата вектора. Додавання нової координати і є метою визначення класу `T3DVector`.
- Метод `T3DVector.Add` реалізує додавання тривимірних векторів, використовуючи при цьому метод додавання класу `T2Dvector`. Спадкування у визначенні класу `T3DVector` використовується саме для цих цілей.
- Операції класу `T3Dvector`, власне кажучи, прив'язані до операцій класу `T2Dvector` - успадковуються функціональні можливості.
- Метод `T3DVector.GetXYProection` реалізує зворотний зв'язок – приведення (редукцію) об'єктів дочірнього класу до об'єктів батьківського класу.

Відзначимо, що перевизначення основних операцій над векторами нетривіальні, а поле атрибута `Coord` не додає нічого істотного в структури даних.

Важливим методологічним аспектом цього прийому є погляд на визначення класу `T3DVector` як визначення методу інкрементації розмірності векторного простору. Ми одержуємо якісний як з теоретичної, так і практичної точок зору програмний код, що у принципі можна використовувати повторно - для опису просторів великих розмірностей. Якщо в попередньому прикладі роль еталона для повторного використання програмного коду грає клас - предок, то в цьому прикладі як такий еталон виступає клас-нащадок.

6.4.4. Визначення класу - узагальнення батьківського класу.

Визначення дочірнього класу, що є узагальненням батьківського класу, означає, що дочірній клас описує об'єкти більш загального типу, чим об'єкти батьківського класу. Для узагальнення батьківського класу до його атрибутів повинні бути додані нові атрибути, що і представляють суть узагальнення. Разом з тим функціональні можливості об'єктів

дочірнього класу прив'язані в основному до функціональних можливостей батьківського класу.

Породження класів у формі узагальнення можна використовувати в тих випадках, коли структурування даних у програмі відіграє основну роль, а функціональне поводження - підлеглу.

Розглянемо як приклад визначення нового класу TCaptionFormattedText, що визначається на базі TCaptionText, але виводить на екран текстовий рядок у деякому форматі (з установкою атрибутів шрифту, розташування тексту в поле і т.п.). Для реалізації цього задуму в дочірньому класі ми вводимо набір атрибутів форматування, методи їхньої установки і модифікації, а потім перевизначаємо метод Print, що використовує всю нову інформацію при виведенні тексту на екран. Відзначимо, що перевизначення методу Print полягає просто в написанні нового тіла методу, у якому ніяк не використовується програмний код методу Print класу TCaptionText.

Таким чином, новий клас цілком заміняє собою клас TCaptionText. Виявилося, що метод Print базового класу - зайвий. Цей недолік не грає особливої ролі, якщо таких методів не дуже багато, тобто базовий клас використовується в першу чергу для визначення даних, а функціональне поводження його об'єктів обмежено. У протилежному випадку для визначення дочірнього класу краще використовувати спеціалізацію. У нашому прикладі можна клас TCaptionText визначити без методу Print, успадковувати TCaptionText в описі TCaptionFormattedText, і в ньому визначити власний метод Print.

6.4.5. Визначення класу - розширення батьківського класу.

Розширення батьківського класу означає визначення в дочірньому класі нових методів, що розширюють функціональні можливості об'єктів.

Приклад 6.6. Нехай батьківський клас описує редактор рядка TStringEditor, у якому визначені тільки операції посимвольної обробки рядка, що редагується. Клас TExtendedStringEditor ми будемо для включення таких операцій редагування, як Copy, Cut, Paste.

Приклад 6.7. Клас TVector з операціями векторного простору (додавання, віднімання векторів і множення вектора на число) використовується для побудови класу TEvklidVector, у якому додатково визначаються операції скалярного добутку, обчислення довжини вектора, кута між векторами, проектування вектора на вектор.

Оскільки при узагальненні наслідувані методи не перевизначаються, новий клас є підтипом базового класу (принцип підстановки витримується). Відзначимо методологічно важливу особливість використання розширень: у проєктованій програмній системі використовуються об'єкти як базового, так і розширеного типів. Справді, далеко не кожен редактор рядка має потребу в операціях з підрядками: у редакторі для введення числа такі операції не потрібні, а в редакторі для роботи з реченнями природною мовою вони необхідні.

Другий приклад має і формально-математичне обґрунтування: у математиці векторні простори зі скалярним добутком утворюють спеціальний клас евклідових просторів зі своєю аксіоматикою і теорією.

6.5. Множинне спадкування

Часто виникає необхідність визначити клас, властивості і поведження об'єктів якого у великому ступені визначаються властивостями і методами одночасно декількох класів.

Форма спадкування, при якій успадковуються декілька класів одночасно (паралельно), називається множинною.

Адекватною метафорою для ілюстрації суті множинного спадкування є спадкування ознак і поведження батьків (матері і батька) дитиною в біологічній популяції. Як відомо, деякі біологічні ознаки, риси характеру, особливості поведінки дитина успадковує від батька, інші - від матері. Разом з тим вона може має і свої власні особливості, що з'явилися у результаті виховання, інших обставин.

З погляду теорії, множинне спадкування є адекватним відображенням існування декількох незалежних класифікацій на множині різноманітних об'єктів, визначених в системі на етапі її аналізу.

Приклад 6.8. Класифікації геометричних фігур.

Звернемося знову до задачі геометричних перетворень площини. Класифікація геометричних фігур - класичний приклад використання потенційно нескінченної множини незалежних властивостей для опису конкретних об'єктів. Один з можливих підходів, зв'язаних з геометричними перетвореннями – класифікація Г.Вейля, у якій абстрактними ознаками класифікації є властивості, обумовлені так званими інваріантами. Інваріантом класу геометричних фігур називається предикат, що зберігає логічне значення «істина» для всіх фігур цього класу.

Клас Перенос.

Інваріант – паралельний перенос. До одного класу належать усі фігури, що можуть бути перетворені друг до друга перетворенням паралельного переносу.

Атрибут класу - точка, що задає положення фігури.

Операція класу – паралельний перенос.

Клас Поворот

Інваріант - поворот. До одного класу належать усі фігури, що можуть бути перетворені друг до друга поворотом.

Атрибут класу - кут, що задає орієнтацію фігури.

Операція класу - поворот.

Клас Симетрія

Інваріант - симетрія. До одного класу належать усі фігури, що можуть бути перетворені друг до друга осью симетрії.

Атрибут класу - параметри осі симетрії.

Операція - осьова симетрія

Клас Гомотетія.

Інваріант - гомотетія. До одного класу належать усі фігури, що можуть бути перетворені друг до друга гомотетією.

Атрибут класу - коефіцієнт гомотетії.

Операція класу - гомотетія.

Інваріант - містити точку. До одного класу належать усі фігури, що містять фіксовану точку.

Атрибут класу - точка, що належить фігурі.

Операція класу - перевірка відношення "Точка належить фігурі."

Клас Рух (Перенос, Поворот, Гомотетія, Симетрія).

Інваріант - геометричні перетворення. Зазначимо, що форма геометричних фігур не змінюється при перетвореннях переносу, повороту, симетрії, гомотетії. Клас Рух успадковує усі властивості класів, що визначають види геометричних перетворень. Фрагмент ієрархії цих класів представлений на рис. 6.4.

Оскільки всі характеристичні властивості незалежні, не можна визначити однозначно, у якій послідовності потрібно реалізовувати спадкування. Отже, жоден з батьківських класів не має переваги.

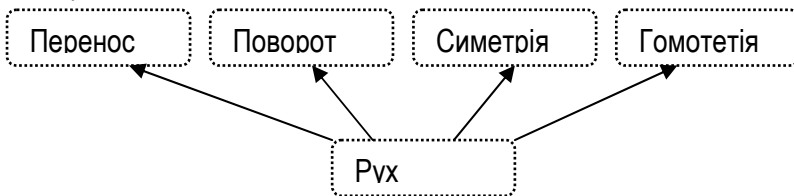


Рис. 6.4. Опис класу Рух множинним спадкуванням.

Комбінування батьківських класів Перенос, Поворот, Гомотетія, Симетрія дозволяє визначати різні спеціалізовані форми поведінки (рис. 6.5.). Наприклад:

Паралельний перенос і поворот - переміщення кулі, що обертається або переміщення кулі і її поворот в результаті зіткнення з іншою кулею.

Паралельний перенос і осьова симетрія - переміщення кулі і її відображення від борта.

Гомотетія і перенос - переміщення кулі, що імітує наближення чи віддалення від спостерігача.

Узагальнена симетрія - для моделювання різних груп симетрій. (Група симетрій правильного трикутника складається з 6-ти елементів - 3-х поворотів і 3-х осьових симетрій).

Таким чином, для проектування програмної системи, у якій існують різні спеціалізовані форми поведінки об'єктів, одне з можливих рішень - визначення класів з "чистим" поводженням, і на їхній базі - класів зі спеціалізованим "змішаним" поводженням.

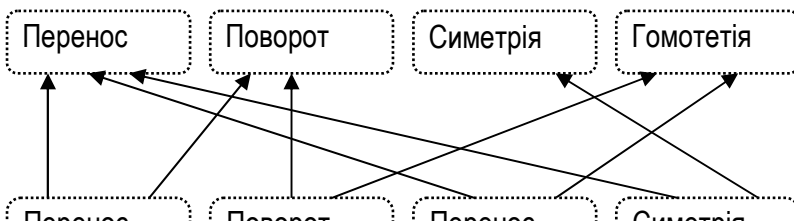


Рис. 6.5. Множинне спадкування як комбінування.

Приклад 6.9. Класифікації людей.

Кожна людина у своїй повсякденній діяльності грає кілька ролей, причому деякі з них - одночасно. Як ми вже бачили в прикладі (рис. 4.2.), кожна роль визначається сукупністю характерних властивостей, які можна визначити як об'єкти. Комбінування ролей в одному об'єкті "Людина багатобічна" природно визначити як множинне спадкування. Як і в першому прикладі, виділення абстрактних об'єктів-ролей дає можливість повторно використовувати програмний код, ґрунтуючись на природній класифікації. Ось деякі сполучення ролей, що, з одного боку, можуть бути складеними, а з іншого боку - входить до складу інших ролей: (Людина, студент, сім'янин), (Людина, Службовець, абонент), (Людина, мешканець, абонент). В інформаційно-довідковій системі навчального закладу всі ці ролі доцільно виділити в окремі класи для наступного варіювання.

Незважаючи на удавану простоту, реалізація множинного спадкування супроводжується деякими принциповими труднощами. Справа в тім, що при множинному спадкуванні порушується деревовидність ієрархії.

Якщо два класи, що використовуються для породження класу множинним спадкуванням, мають загальні імена атрибутів чи методів, (зокрема, загального предка), виникає неоднозначність інтерпретації їх загальних, успадкованих атрибутів і методів. Спадкування властивостей загального предка називають повторним.

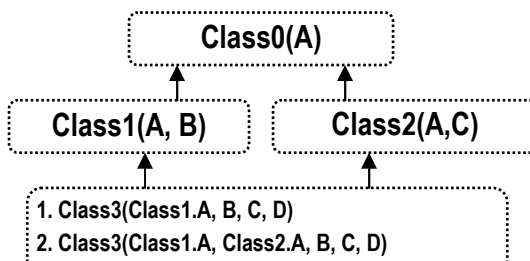


Рис. 6.6. Варіанти інтерпретації повторного множинного спадкування.

На рис. 6.6 показані два варіанти інтерпретації повторного множинного спадкування класів. У першому варіанті властивість А успадковується в єдиній копії - як властивість, що дісталася в спадщину від діда, у другому - у двох копіях - від матері, і від батька. І та, і інша інтерпретації реально зустрічаються на практиці.

Припустимо також, що ініційовано наступні об'єкти цих класів

Іванов І.І., доцент кафедри хімії, тел. 22-22-22.

Іванов І.І. вул. Будівельників 22, тел. 33-33-33.

У результаті ми хочемо одержати об'єкт виду

Іванов І.І., доцент кафедри хімії, тел. 22-22-22.

вул. Будівельників 22, тел. 33-33-33

Таким чином, ім'я повинне успадковуватися від діда, а телефони - від батьків.

Іншим джерелом подвійності є загальні елементи протоколів класів-батьків. Наступний рисунок 6.7. ілюструє цю ситуацію:

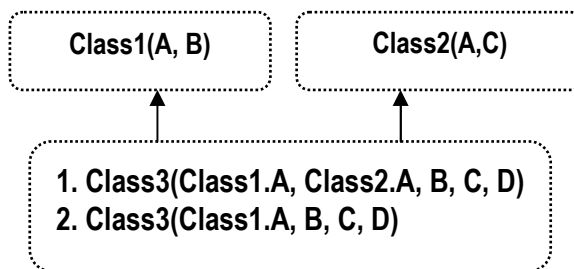


Рис. 6.7. Подвійність інтерпретації множинного спадкування

Подвійність інтерпретації множинного спадкування по-різному уточнюється в різних об'єктно-орієнтованих мовах програмування: в одних мовах успадковується тільки одна копія загальної властивості, в інших – обидві, у третіх, можливий вибір того чи іншого варіанта. У мові Borland Pascal обране найбільш радикальне рішення – множинне спадкування не підтримується взагалі (немає множинного спадкування – немає проблем).

Проте, множинне спадкування як метод опису об'єктних типів виникає природним образом. Тому важливо визначити по-перше, теоретичні передумови, по-друге, технологію його використання.

Підкреслимо ще раз, що спадкування як метод класифікації виражає відношення «об'єкт **a** є клас **A**». Наприклад, «Іванов І.І. є Людина», «дана фігура є коло». Оскільки кожен реальний об'єкт у різних ситуаціях по-різному співвідноситься з навколишнім світом, засобами мови проектування необхідно виражати відношення «об'єкт **a** є класи **A**, **B**, **C**,...». Якщо кожний з цих класів вже описаний, програміст попадає в ситуацію, коли він повинен використовувати множинне спадкування як адекватне рішення проблеми.

Конфліктна подвійність в іменах атрибутів і методів може виникнути:

1. Як результат неточного вибору імен атрибутів, властивостей і методів у протоколах батьківських типів, що власне кажучи, є різними і повинні успадковуватися в двох копіях.
2. Як наслідок того факту, що клас дійсно повинний успадковувати властивості, що є загальними на рівні абстракції батьківських класів, але такими, що розрізняються на рівні класу-спадкоємця.
3. Як результат принципової помилки при проектуванні класів, коли замість відношення «об'єкт **a** є клас **A**» було використане відношення «об'єкт **a** є частина об'єкта класу **A**».

Множинне спадкування засобами мови Borland Pascal найкраще реалізувати як послідовне спадкування - від кожного батьківського класу по черзі. При цьому конфлікт імен у випадку 1 усувається перевизначенням імен у батьківських класах.

На відміну від 1-го випадку, проблема у випадку 2 носить принциповий характер. Розглянемо два основних варіанти її походження і рішення.

- Варіант 1. Властивості і поводження об'єктів батьківських класів можна розглядати як змішані, складені з декількох «чистих» властивостей і ліній поведінки.

Рішення полягає у виділенні так званого базового класу і класів-домішок. Спочатку визначають базовий клас так, щоб атрибути цього класу містили в собі ідентифікатори об'єкта. Потім класи-домішки послідовно додаються до базового класу.

У прикладі 2 ці «чисті» властивості і лінії уже виділені у формі класів - Людина, Посада, Мешканець, Абонент. Клас Людина тут відіграє роль базового, а інші класи - роль домішок. Класи-домішки використовуються тільки для довизначення базового класу. Іншого призначення в них немає. При цьому клас Абонент відіграє роль домішки до домішки. Справді, у дияді Посада - Абонент роль базового класу грає Посада, об'єкт якої, крім всього іншого, повинний відігравати роль абонента телефонної мережі. Точно такий же розподіл ролей класів має місце в дияді Мешканець-Абонент. Тому у визначення класів потрібно внести зміни, що відповідають реальному розподілу ролей:

Людина {базовий клас}

Службовець = *посада*<абонент>, {домішка}

Мешканець = *Житло*<абонент>, {домішка}

ЛюдинаСлужбовець = *Людина*(*Службовець*);

ЛюдинаСлужбовецьМешканець = *ЛюдинаСлужбовець*(*Мешканець*).

Проблема спадкування загальних властивостей в одній копії тим самим вирішена.

Для рішення проблеми спадкування властивостей і поводження класу Абонент у двох копіях потрібно якимсь чином сховати атрибути класу Абонент у класах Мешканець і Службовець. Іншого способу повторно використовувати програмний код класу Абонент у класі ЛюдинаСлужбовецьМешканець за допомогою механізму спадкування в

мові Borland Pascal немає, оскільки перейменування атрибутів неможливо. (Той факт, що клас Абонент не успадковується класами Мешканець і службовець, відзначений у формулах визначень класів дужками іншого типу - гострокутними.) Методи рішення цієї проблеми за допомогою різних механізмів взаємодії класів будуть викладені нижче. Відзначимо тільки, що клас Абонент у розроблювальній системі, очевидно, буде тісно пов'язаний із класом Телефон, що описує реальний прилад. Об'єкти цього класу будуть існувати в системі самі по собі, тому будь-який опис класів Абонент, Службовець, Мешканець повинен бути заснований на взаємодії з класом Телефон. Зокрема, тут, швидше за все, допущено помилку в проектуванні, що вище класифікована як випадок 3. Саме, об'єкт Телефон є частиною об'єктів Службовець, Мешканець, а відношення Абонент - Службовець не є відношенням спадкування.

- Варіант 2. Властивості і поведінка об'єктів батьківських класів є чистими, їх неможливо розділити на базові класи і домішки.

Найчастіше розходження властивостей і поводження синтаксично оформляється на рівні імен у протоколах класів. Однак, існують ситуації, у яких це не так. Саме такі ситуації ми і розглянемо.

Приклад 6.10. Натуральні числа. (рис.6.8). Проектована спеціалізована математична система повинна підтримувати операції з натуральними числами довільної довжини (довгими цілими).

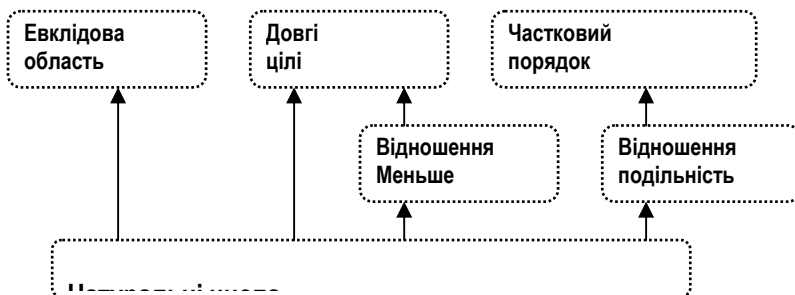


Рис.6.8. Ієрархія спадкування класу *Натуральні числа*.

У цьому прикладі абстрактний клас *Частковий порядок* описує властивості, загальні для відносин *Менше* і *Подільність*. У класі *Евклідова область* описані арифметичні операції над натуральними числами, включаючи операції обчислення неповної частки і залишку. Клас *Натуральні числа* успадковує класи *Менше*, *Подільність* і *Евклідова область*, повторно успадковуючи клас *Частковий порядок*.

Відзначимо наступні особливості цього прикладу: Об'єкт-атрибут *Довгі цілі* класу *Натуральні числа*, що визначає представлення натурального числа, повинний успадковуватися класом *Натуральні числа* безпосередньо. Інші класи верхніх рівнів ієрархії є абстрактними, їх методи варто визначити як віртуальні, оскільки ці класи відображують алгебраїчні (математичні) абстракції. Конфлікт полягає в тому, що в класі *Натуральні числа* повинні бути визначені дві операції порівняння натуральних чисел - по величині і по подільності, що у класі *Частковий порядок*, де визначається сигнатура операції порівняння, представлені однією абстрактною операцією.

Приклад 6.11. Амплітудна модуляція сигналу.

Технологія передачі радіосигналів на відстань полягає в тім, що переданий сигнал являє собою суміш двох гармонік різних за порядком величин частот. Гармоніку високої частоти називають несучою, а низькочастотну - обвідною. Наявність високочастотного компонента дозволяє передавати сигнал на великі відстані, а низькочастотна складова несе інформацію (часто це звукові частоти). Таким чином, клас *Змішаний сигнал* повинний успадковувати властивості класу *Сигнал* у двох копіях - *Несучий сигнал* і *Обвідний сигнал*.

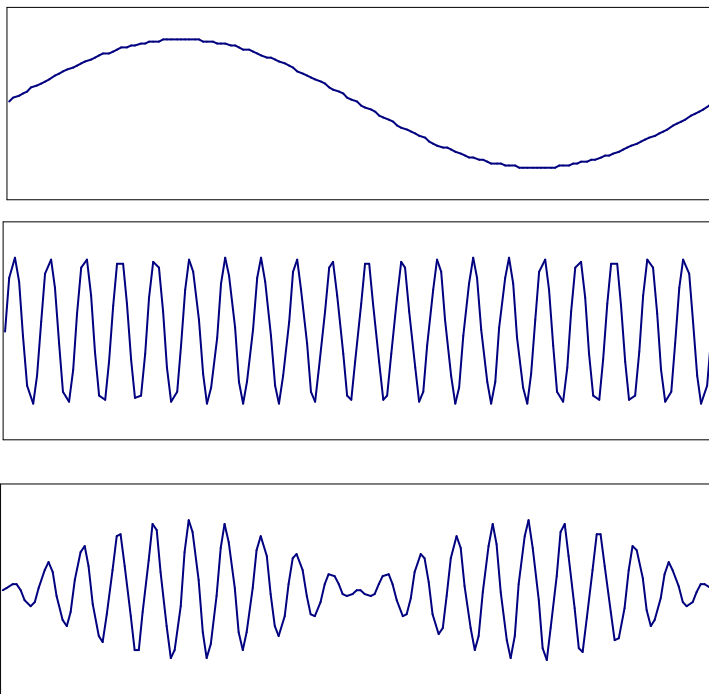


Рис. 6.9. Несучий, обвідний і змішаний сигнали

Для моделювання цього процесу визначимо класи TSignal, TSupportSignal, TExternSignal і означимо проблему опису типу TMixedSignal:

Type

TSignal = **object**

private

Omega : Real;

Amplitude : Real;

public

...

Function GetSignal(t : Real):Real;

...

```

end;
TSupportSignal = object(TSignal)
...
end;
TExternSignal = object(TSignal)
...
end;
TMixedSignal = object({--?--})  {Як визначити цей клас ?}
...
    Function GetSignal(t:Real):Real;
...
end;
Implementation
Function TSignal.GetSignal(t : Real):Real;
begin
    GetSignal :=Amplitude*Sin(Omega*t)
end;
Function TMixedSignal.GetSignal(t : Real):Real;
begin
    GetSignal := { як визначити цю функцію? }
end;

```

Розглянемо тепер цю же задачу вже не як фізичну, як технічну проблему. Несуча частота генерується спеціальним приладом - Генератором, а обвідний - іншим приладом, наприклад, Мікрофоном. Змішування відбувається в третьому приладі - Передавачі. Залишимо ім'я Сигнал для позначення абстрактного батьківського класу. У програмному коді зміняться тільки імена класів, а в програміста зміниться точка зору на конструювання об'єкта. Тепер об'єкт Передавач містить дві частини - Генератор і Мікрофон і замість спадкування тепер використовується так звана агрегація.

```

TGenerator = object(TSignal) ... end;
TMicrophone = object(TSignal) ... end;

```

```

{----- об'єктний тип Передавач -----}

```

```

TTransmitter = object
private
    Generator : TGenerator;
    Microphone : TMicrophone;
    ...
    Function GetSignal(t:Real):Real;
    ...
end;

```

Implementation

```

Function TTransmitter.GetSignal(t : Real):Real;
begin
    GetSignal:=Generator.GetSignal*Microphone.GetSignal
end;

```

Зробимо деякі висновки:

1. Розходження між спадкуванням і агрегацією, незважаючи на те, що ці два поняття виражають принципово різні типи відносин між об'єктами, насправді є суб'єктивним і відносним.

Справді, поведінка об'єкта Передавач подібна до поведінки об'єктів Мікрофон і Передавач тому, що функціонування цих частин спостерігається в зовнішньому світі і сприймається там як риси функціонування всього об'єкта. (Одноколісний велосипед успадковує колесо чи колесо є частиною одноколісного велосипеда?)

2. Множинне спадкування можна замінити агрегацією, оскільки агрегація дозволяє інкапсулювати атрибути і методи об'єктів-частин.

Справді, при агрегації імена атрибутів і методів стають кваліфікованими, тому загальнодоступні методи можна перевизначити. Перевизначення методів, схованих під ім'ям об'єкта-частини полягає в описі функції, що складається рівно з одного оператора. Наприклад:

```

Function TTransmitter.GetGeneratorSignal(t : Real) : Real;
begin

```

GetGeneratorSignal := Generator.GetSignal

end;

Такі перевизначення, по-перше, призводять до додаткових витрат обчислювальних ресурсів, по-друге - роблять висхідний текст програмного модуля занадто довгим і не пристосованим для читання.

3. Повна заміна множинного спадкування агрегацією призводить до того, що обумовлений клас випадає з ієрархії спадкування. Тим самим губиться корисна властивість поліморфізму методів.

Цей недолік є найбільш істотним. Розглянемо тому варіант реалізації множинного спадкування, у якому поєднуються і спадкування, і агрегація. Визначення класів TSignal, TGenerator, TMicrophone залишимо такими ж, як і вище. Клас TTransmitter визначимо як спадкоємця класу TMicrophone, а об'єкт Generator типу TGenerator зробимо частиною класу TTransmitter:

Type

{----- об'єктний тип Передавач -----}

TTransmitter = **object**(TMicrophone)

private

Generator : TGenerator;

...

Function GetSignal(t:Real):Real;

...

end;

Implementation

Function TTransmitter.GetSignal(t : Real):Real;

begin

GetSignal := Generator.GetSignal*TMicrophone.GetSignal

end;

Що змінилося? По-перше, клас Передавач тепер включений в ієрархію класів, отже методи, успадковані від класу Мікрофон є

поліморфними. По-друге, методи класу Мікрофон не потрібно «виводити на поверхню». Таким чином, половинчате рішення виявилось найбільш прийнятним.

Залишилося тільки відповісти на запитання, чому при спадкуванні батьківським обраний клас TMicrophone, а не TGenerator. Вибір тут обумовлений суб'єктивними мотивами: об'єкт Мікрофон має більш складну поведінку, ніж Генератор. В реальній системі він буде генерувати не одну гармоніку, а весь складний спектр звукових гармонік..

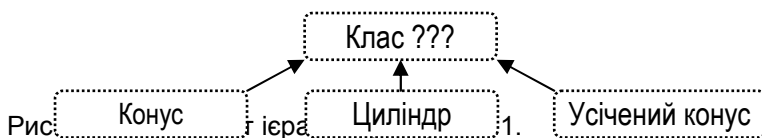
На закінчення відзначимо, що засобами мови Borland Pascal реалізувати поліморфізм повністю неможливо.

6.6. Висновки

- Спадкування – метод опису об'єктних типів, характерний для об'єктно-орієнтованого проектування.
- Відношення спадкування породжує на множині класів програмної системи структуру, яку називають ієрархією спадкування. Ця ієрархія багато в чому визначає логічну архітектуру системи.
- Спадкування в різних формах використовується з метою повторного використання програмного коду.

6.7. Вправи

1. У системі визначені класи Конус, Усічений конус і Циліндр. Класи Конус і Циліндр задані атрибутами R (радіус основи), H (висота), а клас Усічений конус – атрибутами R , r і H , де R і r – радіуси основ. У кожному із класів визначені методи обчислень об'єму V і повної поверхні S .
 - Реалізуйте ці класи за допомогою відношення спадкування у виді ієрархії (рис.6.10), побудувавши для цього абстрактний базовий клас. Визначите форму спадкування, що Ви використовували.



- Реалізуйте ці три класи, використовуючи один з них як базовий, а інші – як його дочірні. Визначте форму спадкування, що Ви використовували.
 - Порівняйте ці рішення і визначите, яке з них краще і чому.
2. У системі визначені об'єкти Викладач, Аспірант і Студент. У системі повинні використовуватися імена, дні народження й адреси цих людей, а також деякі відомості, специфічні для кожного з них. Для викладача – посада, учений ступінь і звання. Для аспіранта – спеціальність навчання, рік навчання, тема дисертації і науковий керівник. Для студента – спеціальність навчання, курс, група.
- Побудуйте ієрархію цих класів (рис. 6.11.), використовуючи класи-домішки Адреса і Дата, а також абстрактний клас Людина. Визначте форми спадкування, що Ви використовували. Вирішіть задачу реалізації множинного спадкування, показану на рис. 6.11.

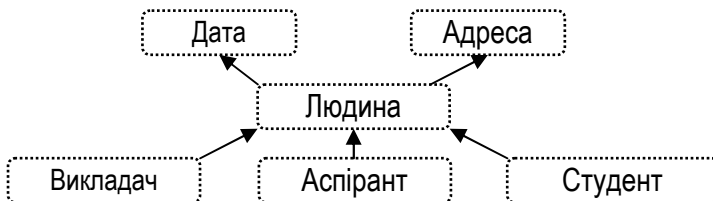


Рис. 6.11. Фрагмент ієрархії до впр. 6.6.2.

Запропонуйте свій варіант опису класів Викладач, Аспірант, Студент.

3. У системі описані об'єкти Продавець, Покупець, Банк. Всі ці об'єкти мають імена, юридичні адреси. Продавець і Покупець є клієнтами Банку зі своїми рахунками в Банку. Банк має свій власний капітал, що зберігається на спеціальному рахунку. Покупець має можливість знімати гроші зі свого рахунку, а Продавець – класти гроші на свій рахунок. Банк повинен відчислити N відсотків від кожної операції своїх клієнтів, збільшуючи на відповідну суму свій капітал.
- Реалізуйте ці класи за допомогою відношення спадкування у виді ієрархії (рис. 6.12), побудувавши для цього абстрактний базовий клас. Визначте форму спадкування, що Ви використовували.

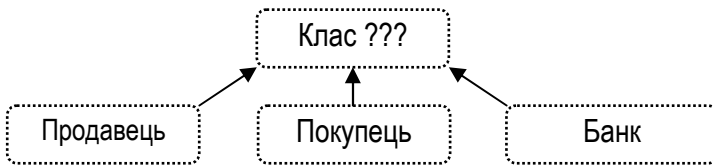


Рис. 6.12. Фрагмент ієрархії до впр. 6.6.3.

4. Для програмної системи Шахи визначені об'єкти – шахові фігури. Опис кожної фігури містить наступні атрибути: ранг, колір, стан (у грі, поза грою), позиція на дошці. Методи фігури:

- Перевірити можливість ходу на поле (x, y);
- Зробити хід на поле (x, y);
- Перевірити можливість здобуття на поле (x, y);
- Здійснити здобуття на поле (x, y);

Реалізуйте фрагмент ієрархії (рис. 6.13.), використовуючи спадкування у формі специфікації.

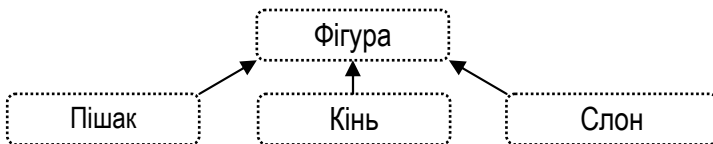


Рис. 6.13. Фрагмент ієрархії шахових фігур до впр. 6.6.4.

- Побудуйте і реалізуйте ієрархію всіх шахових фігур. Які методи потрібно додати до протоколів класів для того, щоб описати всі правила одного ходу гри в шахи (Задачу візуалізації не розглядайте).
5. Для програмної системи Бібліотека необхідно визначити об'єкти Книга, Газета, Журнал, Мікрофільм, КомпактДиск.
- Побудуйте ієрархію цих класів (рис. 6.14.). Як базовий клас виберіть клас Фонд (одиниця збереження). Атрибути класів визначите самостійно.
 - Реалізуйте методи-селектори атрибутів.
 - Сформулюйте принципи класифікації, відповідно до яких будуть проміжні класи.

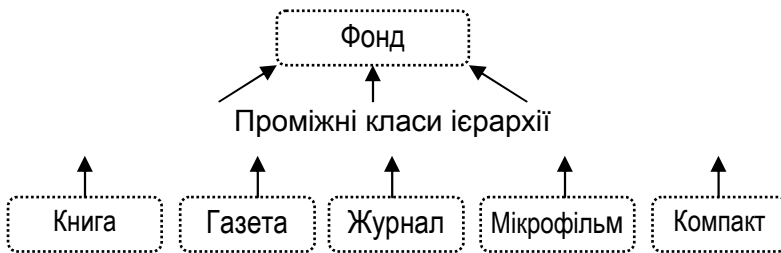


Рис. 6.13. Фрагмент ієрархії фонду бібліотеки до впр. 6.6.5.

6. Побудуйте ієрархію класів `Стек`, для чого:

- Опишіть абстрактний клас `Стек`, визначивши операції `Додати`, `Видалити`, `Порожній` як віртуальні.
- Побудуйте похідні класи `СтекМасив` (стек на основі масиву) і `СтекПосилання` (стек на основі посилань) з елементами типу `Pointer`.
- Опишіть клас-запис `Елемент = (Ім'я, Значення)`, конкретизувавши типи полів `Ім'я` і `Значення` за своїм розсудом.
- Побудуйте класи `СтекМасивЕлемент` і `СтекПосиланняЕлемент` як класи, похідні від класів `СтекМасив` і `СтекПосилання`, змінивши операцію `Додати` на операцію `Додати(Елемент)`.
- Розширте класи `СтекМасив` і `СтекПосилання`, додавши операцію інвертування стека.
- Розширте класи `СтекМасивЕлемент` і `СтекПосиланняЕлемент`, на основі розширених класів `СтекМасив` і `СтекПосилання`.

7. ВІДНОШЕННЯ МІЖ ОБ'ЄКТАМИ

Розглядаючи поняття об'єкта і класу (об'єктного типу), способи означення класів, приклади класів і об'єктів, ми увесь час мали справу не тільки з описуванням об'єктом, але і з іншими об'єктами, що складають оточення даного об'єкту. Самі по собі об'єкти не представляють ніякого інтересу: об'єкти реалізують себе тільки у взаємодії з іншими об'єктами.

Програмна система і будь-який її компонент є сукупністю взаємодіючих об'єктів, а кожен об'єкт цілком визначається способами взаємодії зі своїм оточенням.

Внутрішня побудова (структура) об'єкта визначається тільки його властивостями і поведінкою, тобто має сенс тільки тому, що є реалізацією його взаємодії з оточенням.

Ці методологічні аспекти дуже важливі: визначаючи способи взаємодії об'єкта з оточенням, ми тим самим визначаємо об'єкт як концептуальну сутність. На рівні технології проектування це означає, що інтерфейсна частина опису визначає реалізацію. На рівні мови програмування (Borland Pascal) це означає, що визначенням об'єктного типу є його протокол - та частина опису, яка записується за службовим словом `public`.

Розглядаючи приклади, ми мали справу з наступними типами відносин між об'єктами:

- Відношення зв'язку.
- Відношення агрегування.
- Відношення залежності.

Наприкінці розділу ми наведемо більш точну класифікацію типів відносин, у якій будуть відображені не стільки технічні, скільки концептуальні розходження.

Відносини між конкретними об'єктами вже готової програмної системи визначаються в описах відповідних об'єктних типів, що утворюють ієрархію абстракцій. Об'єктно-орієнтований підхід використовує для цього два основних типи абстрагування:

- Спадкування.
- Агрегування.

Таким чином, загальна схема проекту програмної системи, що відбиває відношення між об'єктами й об'єктними типами, що визначають ці об'єкти, є ніби то тривимірною: одна з координатних площин описує структуру відношень між об'єктами, друга - структуру спадкування, третя - структуру агрегування.

У цьому розділі ми розглянемо тільки структуру відношень між об'єктами, при необхідності звертаючись до структур абстрагування.

7.1. Відношення зв'язку

Зв'язок між об'єктами означає такий тип відношень, при якому об'єкти обмінюються повідомленнями, що містять запити по наданню послуг.

Об'єкти, що знаходяться у відношенні зв'язку, грають різні ролі: об'єкт-клієнт посилає об'єкту-серверу повідомлення, що містить прохання про виконання послуги.

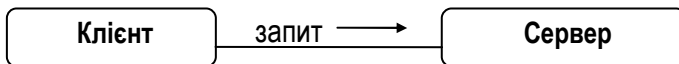


Рис. 7.1. Відношення зв'язку Клієнт-сервер.

Зв'язок Клієнт-сервер асиметричний, хоча дані можуть передаватися й у двох напрямках. Незважаючи на те, що повідомлення, що містить дані, послано клієнтом, сервер може повернути відповідь - результат виконаної роботи. У кожному одиничному акті зв'язку один з об'єктів грає активну роль клієнта, інший - пасивну роль сервера.

Деякі об'єкти завжди тільки активні: вони ініціюють роботу, але не призначені для виконання чужих доручень. Такі об'єкти називають акторами. Інші об'єкти завжди пасивні - вони призначені для виконання чужих доручень. Такі об'єкти називають серверами. Нарешті, деякі об'єкти виконують і чужі замовлення, і самі дають доручення. Вони називаються агентами.

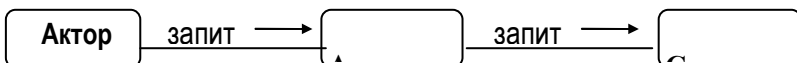


Рис. 7.2. Ролі об'єктів системи.

Приклад 7.1. Система Калькулятор.

Калькулятор призначений для виконання арифметичних дій над числами. Система в тому вигляді, що ми беремо за основу, являє собою три взаємодіючих об'єкти: Клавіатура, Вікно Результат, Обчислювач.

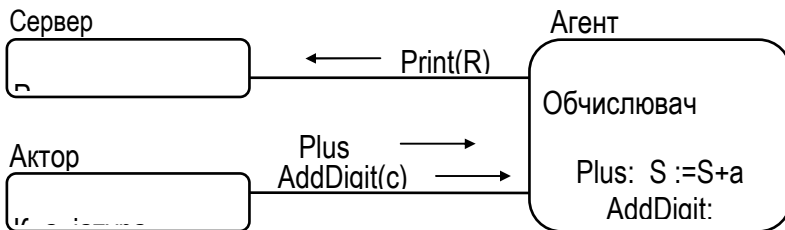


Рис. 7.3. Ролі і зв'язки об'єктів у системі калькулятор. Версія з неправильним розподілом ролей.

У цій системі Клавіатура - *актор*, Вікно Результат - *сервер*, Обчислювач - *агент*. Клавіатура зв'язана з Обчислювачем декількома операціями. (Протокол об'єкта Клавіатура визначається технічним завданням на проектування калькулятора.) На рис. 7.3 відзначені операції Обчислювача AddDigit(c) і Plus. Операція AddDigit(c) ініціюється натисканням цифрової клавіші, а операція Plus - натисканням клавіші "+". Операція Print визначена для об'єкта Результат і ініціюється об'єктом Обчислювач в описах операцій AddDigit, Plus.

Розглянемо макети методів, позначених на схемі:

```

Procedure TKeyboard.Monitor(Computer : TComputer);
var ch : Char;
begin
  Repeat
    Ch := ReadKey;
    Case Ch of
      '+' : Computer.Plus;
      '0'..'9' : Computer.AddDigit(Value(Ch))
      {інші клавіші}
    end;
  until False
end;
Procedure TComputer.Plus(Result : TResult);
begin
  S := S+a;
  Result.Print(S)
end;
Procedure TComputer.AddDigit(Result : TResult; c : Digit );
begin
  a := 10*a+c;
  Result.Print(a)
end;

```

Основною проблемою, яку вирішують при проектуванні зв'язків між об'єктами, полягає в правильному розподілі ролей. В нашому прикладі Обчислювач сам визначає об'єкт, якому потрібний результат обчислень. Але адже це не його справа! Обчислювач повинний бути сервером. Його справа – обчислювати і зберігати дані. Керування повинен здійснювати інший об'єкт. Розглянемо інший варіант:

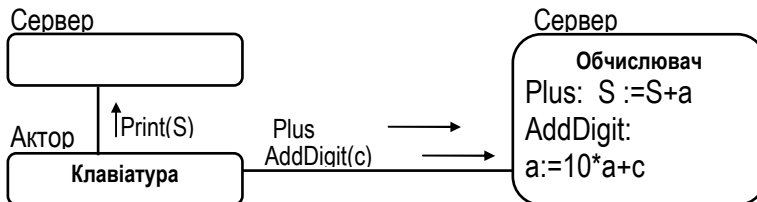


Рис. 7.4.Ролі і зв'язки об'єктів у системі Калькулятор.

У цій версії макет реалізації має вид:

```
Procedure TKeyboard.Monitor(Computer: TComputer; Result : TResult);
```

```
var ch : Char;
```

```
begin
```

```
  Repeat
```

```
    Ch := ReadKey;
```

```
    Case Ch of
```

```
      '+' : Result.Print(Computer.Plus);
```

```
      '0'..'9' : Result.Print(Computer.AddDigit(Val(Ch)))
```

```
      {інші клавіші}
```

```
    end;
```

```
  until False
```

```
end;
```

```
Function TComputer.Plus : Integer;
```

```
begin
```

```
  S := S+a;
```

```
  Plus := S;
```

```
end;
```

```
Function TComputer.AddDigit( c : Digit ) : Integer;
```

```
begin
```

```
  a := a + 10*c;
```

```
  AddDigit := a
```

```
end;
```

Важлива перевага цього варіанта реалізації полягає в тому, що керування системою тепер легко модифікувати, а об'єкти-сервери системи стали незалежними друг від друга. Обчислювач може використовуватися тепер і в сполученні з іншим екранним об'єктом.

У цій версії методу TKeyboard.Monitor аргументом методу TResult.Print є запит об'єкту Computer.

```
      '+' : Result.Print(Computer.Plus);
```

Результат обчислень у неявному виді повертається об'єкту KeyBoard і через нього транзитом - об'єкту Result для виведення

на екран. Але це не означає, що об'єкт Keyboard відіграє роль агента, оскільки в системі до нього немає звертань.

7.1.1. Видимість

Нехай у системі встановлений зв'язок між об'єктами **A** і **B**. Для того, щоб клієнт **A** міг послати повідомлення серверу **B**, об'єкт **B** повинний бути видимим об'єкту **A**. Керування областями видимості об'єктів підтримується засобами систем програмування. У системі програмування Borland Pascal системною областю видимості є модуль. Об'єктні типи, описані в одному модулі, видимі один одному. У протилежному випадку модуль опису сервера потрібно підключити до інтерфейсної частини модуля опису клієнта в розділі Uses. Якщо сервер обслуговує декілька клієнтів, його можна визначити в модулі Global, що підключається до всіх модулів, у яких визначаються клієнти.

Висновки

- Відношення зв'язку між об'єктами системи забезпечує гнучкість у керуванні системою й універсальність у використанні об'єктів системи. Це досягається правильним розподілом ролей і мінімізацією зв'язків.
- Керування системою зв'язаних об'єктів погіршується при збільшенні їх кількості і кількості зв'язків між ними.

7.2. Відношення агрегації

Агрегація описує відношення

“об'єкт **A** є частина об'єкта **B**”.

Таким чином, об'єкт **A** є атрибутом об'єкта **B**. Об'єкт **B** можна розглядати як агрегат, частинами якого, крім **A**, є й інші атрибути, у тому числі й об'єкти.

Приклад 7.2. (нова версія приклада 7.1)

У цьому прикладі ми розглянемо макет калькулятора, проєктованого у вигляді окремого приладу.

```
TCalculator = Object(TScreenObject)
```

```
private
```


Keyboard : TKeyboard;

Result : TResult;

Computer : TComputer;

public

{описи методів}

end;

Порівняємо цю конструкцію з конструкцією приклада 7.1., у якому калькулятор проектувався як система, що складається зі зв'язаних один з одним незалежних об'єктів.

- Зв'язки між частинами Калькулятора залишилися незмінними, змінилися тільки заголовки методів. Таким чином, *агрегація як відношення між об'єктами не заміняє собою відношення зв'язування. Проблему правильного розподілу ролей потрібно вирішувати і для частин одного об'єкта.*
- Об'єкти, що утворюють систему, виявилися інкапсульованими в агрегаті. Наш калькулятор має свій власний Обчислювач (Computer), пам'ять якого захищена від зовнішніх об'єктів. Вікно висновку Result також може використовуватися тільки даним екземпляром об'єктного типу.
- З'явилися нові, специфічні методи керування об'єктом як системою в цілому. Калькулятор тепер потрібно ініціювати, представити на екрані у виді окремого приладу, і т.д. Помітимо, що в попередньому прикладі клієнти системи Калькулятор усі ці дії повинні були б виконувати самі. Кожен об'єкт, що хоче користатися калькулятором, повинний зібрати його сам, своїм власним методом:

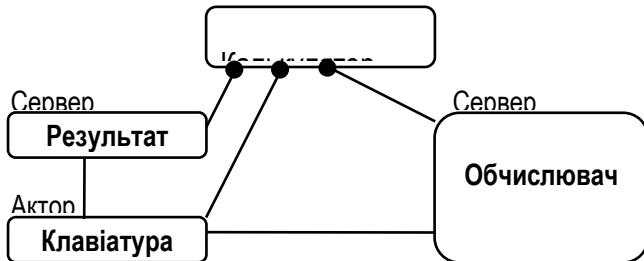
InitCalculator(K:TKeyBoard; C : TComputer; R:TRResult);

У цьому і полягає основне розходження між двома аналізованими варіантами: хочеш - збирай калькулятор сам, немає - користайся готовим, але не скаржся на його недоліки.

Зверніть увагу на те, як відзначено на малюнку відношення агрегації "частина-ціле". Сторона "частина" відзначена великою чорною крапкою.

Рис. 7.5. Проект об'єкта Калькулятор.

Висновки



- Агрегація є основним засобом структурування даних у структурному підході до проектування програм. Можливість включати в структури даних описи операцій зводить агрегацію як метод проектування на якісно новий рівень.
- Метод агрегації дає можливість створювати і керувати системами досить великої складності, однак ці системи спеціалізовані, їх структура є досить жорсткою, її важко модифікувати.

7.3. Відношення залежності

Відношення залежності між об'єктами A и B означає, що один з атрибутів об'єкта A є посиланням на об'єкт B.

У главі 5 ми вже розглядали цей тип відносин між об'єктами, приділивши увагу техніці описів об'єктних типів, орієнтованих на дані. Саме, ми розглянули приклади визначення об'єктів-залежностей і об'єктів - динамічних структур даних. Однак динамічний характер об'єктів використовується не тільки для цих цілей. Важливими перевагами динамічних об'єктів є:

- можливість керувати їхнім життєвим циклом у процесі виконання програми:
- можливість змінювати їхню структуру в процесі виконання програми.

Для об'єктів, орієнтованих на обчислення, ці переваги виявляються з нової сторони. Продовжимо розгляд приклада проектування калькулятора, використовуючи з цією метою відношення залежності.

Приклад 7.3. Калькулятор з динамічною структурою.

Як і в попередньому прикладі, опишемо калькулятор у виді окремого об'єкта, замінивши його статичні частини динамічними:

```
TCalculator = Object(TScreenObject)
```

```
private
```

```
    Keyboard : ^TKeyboard;
```

```
    Result : ^TResult;
```

```
    Computer : ^TComputer;
```

```
public
```

```
    {описи методів}
```

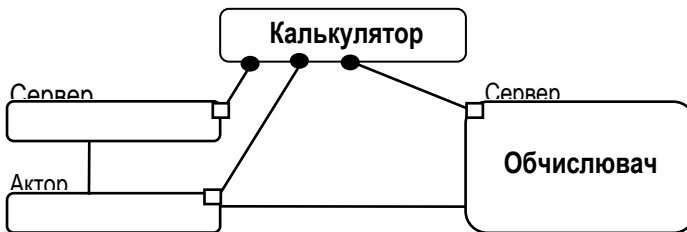
```
end;
```

Порівняємо тепер новий варіант проекту Калькулятор з колишніми проектами.

- Структура приладу залишилася власне кажучи незмінною. У ній збереглися і колишні відносини зв'язків, і структура агрегації.
- Недоліки, зв'язані з жорсткими обмеженнями функціональних можливостей приладу, усунуті: до його посилань, як до розеток (портів), можна підключати й інші екземпляри пристроїв-частин, і інші версії цих пристроїв, дотримуючи тільки правила сумісності, що повинні бути забезпечені правильною типізацією.
- Калькулятор тепер можна створювати, використовувати і знищувати в процесі роботи програми методами самого об'єкта Калькулятор, причому різними методами ініціалізації можна забезпечити включення до складу приладу і нові частини, і уже існуючі.
- Окремі пристрої калькулятора захищені від несанкціонованого доступу інкапсуляцією посилань.

Рис. 7.6. Об'єкт Калькулятор з динамічною структурою.

На рис. 7.6. приведена схема Калькулятора, у якій відношення залежності відзначені білим квадратиком на стороні “частина”. (Помітимо, що посилання вказує в той же бік – на частину.)



Висновки

- Відношення залежності є основним засобом проектування об'єктів, що мають керований життєвий цикл і гнучку, динамічну архітектуру.
- Відношення залежності використовується як при проектуванні об'єктів, орієнтованих на дані, так і при проектуванні об'єктів, орієнтованих на обчислення і складне поведіння.

7.4. Заключні зауваження

Проектування відносин між об'єктами програмної системи грає винятково важливу роль. Правильне визначення відносин між об'єктами, що утворюють систему, впливає на якість усієї системи в цілому в такій же, якщо не в більшій степені, ніж правильне проектування окремих її об'єктів.

Три типи відносин між об'єктами, розглянуті в цьому розділі, не є альтернативними. Вони доповнюють один одного.

- Відношення зв'язку визначає статичні правила функціонування системи через ролі її складових частин, відбиті в їхніх протоколах.
- Відношення агрегації визначає структуру системи в термінах “частина-ціле”.
- Відношення залежності визначає динамічний чи статичний характер структури системи.

Усі ці відносини в остаточному виді (синтаксично) визначена тільки тоді, коли об'єктна модель системи визначена цілком. Їхнє визначення і являє собою на логічному рівні статичну об'єктну модель.

Помітимо, що відношення залежності є уточненням відношення агрегації. Справді, на ранній стадії проектування системи важко віддати перевагу статичному чи динамічному характеру структури об'єкта. Відповідь на питання: що включити в опис об'єкта - інший об'єкт (агрегації) чи посилення на нього (залежність) можна відкласти до з'ясування переваг і недоліків цих варіантів. У розглянутому проекті калькулятора, наприклад, ми можемо зупинитися на проміжному варіанті - зовнішні пристрої (Клавіатура і вікно Результат) агрегувати в об'єкті Панель керування статично, а Обчислювач - динамічно, за допомогою відносини залежності. Тому класифікацію відношень між об'єктами можна уточнити:

Типи відношень між об'єктами:

- **відношення зв'язку (актор - агент - сервер);**
- **відношення агрегації:**
 - ⇒ **статичні;**
 - ⇒ **динамічні (залежності).**

7.5. Вправи

7.5.1. Програмна система Гра двох осіб

Система здійснює керування комп'ютерною грою *двох гравців*, у ході якої вони роблять ходи по черзі, змінюючи при цьому *позицію* гри.

Ходом гри керує суддя, що:

- починає гру, установлюючи початкову позицію і передаючи хід одному з гравців;
- визначає правильність кожного ходу і змінює позицію гри;
- визначає закінчення гри і встановлює переможця гри.

Побудуйте модель цієї гри, для чого:

- виділіть об'єкти;
- визначте й опишіть їхні протоколи;

- визначте відношення зв'язку між об'єктами і розподіліть ролі;
- визначте структуру програмної системи, уточнивши характер агрегації.

Гра в шахи задовольняє умовам цієї моделі. Перелічіть інші ігри, що відповідають цій моделі.

7.5.2. Комп'ютерна система

Керування пристроями

Програма являє собою систему керування двома виконавчими пристроями, що підключені до панелі керування через загальний пристрій сполучення.

Керування виконавчими пристроями здійснюється наступними командами:

- включити систему/виключити систему;
- переключити керування на інший пристрій;
- ініціювати роботу з виконавчим пристроєм;
- закінчити роботу з виконавчим пристроєм;
- кожен з виконавчих пристроїв, крім цього, виконує одну команду, визначену тільки для нього;
- пристрій сполучення для підтримки керування кожним з виконавчих пристроїв, повинний переключатися у свій режим.

Побудуйте модель цієї системи, для чого:

- виділіть об'єкти;
- визначте й опишіть їхні протоколи;
- визначте відношення зв'язку між об'єктами і розподіліть ролі;
- визначте структуру програмної системи, уточнивши характер агрегації.

Система керування двома логічними дисками A: і C: задовольняє цій моделі. Специфічними командами можуть бути, наприклад:

⇒ диска C: - відправити (виділений об'єкт) на A::

⇒ диска A: - оновити (відображення файлової системи).

Перелічіть системи, що відповідають цій моделі.

7.5.3. Комп'ютерна система

Перекладач

Програма являє собою систему, що здійснює автоматичне перетворення (переклад) об'єкта одного типу у відповідний йому об'єкт іншого типу. Типи вихідних і перетворених об'єктів задані сукупністю синтаксичних правил побудови об'єктів. Перетворення також задане сукупністю синтаксичних правил – правил перетворення об'єктів.

Система повинна забезпечити:

- введення об'єктів вихідного типу з клавіатури;
- перевірку правильності введеного об'єкта (приналежності об'єкта типу вихідних об'єктів);
- автоматичне перетворення (переклад) об'єкта в об'єкт вихідного типу;
- виведення на екран монітора об'єкта вихідного типу.

Побудуйте модель цієї системи, для чого:

- виділіть об'єкти;
- визначте й опишіть їхні протоколи;
- визначте відношення зв'язку між об'єктами і розподіліть ролі;
- визначте структуру програмної системи, уточнивши характер агрегації.

Система перекладу слів з однієї природної мови на іншу задовольняє цій схемі. Об'єктами тут є слова. Правила побудови об'єктів початкового і вихідного типу, а також правила перетворення (перекладу) задаються словником перекладу. Крім цього прикладу, описаній схемі задовольняють і системи трансляції мов програмування. Перелічіть інші системи, що відповідають цій моделі.

7.5.4. Комп'ютерна система

Виконавець алгоритмів

Програма являє собою систему, що здійснює виконання алгоритмів, що не розгалужуються, записаних у вигляді послідовності команд. Кожна команда визначає одну дію над даними. Алгоритм у виді послідовності команд зберігається і редагується в спеціальному об'єкті Алгоритм.

Оброблювані дані зберігаються і редагуються в іншому об'єкті – Пам'ять. Система використовує принцип адресності. Виконання алгоритму здійснюється об'єктом Виконавець. Виконавець має скінчений набір команд, що утворюють мову Виконавця.

Система повинна забезпечити:

- введення з клавіатури і редагування алгоритму в об'єкті Алгоритм.
- перевірку синтаксичної правильності кожної команди Алгоритму;
- введення з клавіатури і редагування даних в об'єкті Пам'ять.
- перевірку синтаксичної правильності кожного з даних;
- автоматичне виконання Алгоритму;
- покрокове виконання Алгоритму.

Побудуйте модель цієї системи, для чого:

- виділіть об'єкти;
- визначте й опишіть їхні протоколи;
- визначте відношення зв'язку між об'єктами і розподіліть ролі;
- визначте структуру програмної системи, уточнивши характер агрегації.

Система Виконавець алгоритмів реалізований у всіх системах програмування. Перелічіть інші системи, що відповідають цій моделі.

7.5.5. Комп'ютерна система

Масове обслуговування

Програма моделює систему масового обслуговування, у якій N виконавчих пристроїв обслуговують клієнтів.

Запит на обслуговування надходить від клієнтів системи диспетчеру через приймач замовлень. Диспетчер або призначає вільний виконавчий пристрій для обслуговування замовлення, або відхиляє замовлення, якщо усі виконавчі пристрої зайняті. Кожне замовлення, прийняте до обслуговування, реєструється в картці замовлень, що потім повинна бути збережена в картотеці. Кожен виконавчий пристрій, виконавши замовлення, сповіщає про це диспетчера.

Побудуйте модель цієї системи, для чого:

- виділіть об'єкти;
- визначте й опишіть їхні протоколи;
- визначте відношення зв'язку між об'єктами і розподіліть ролі;
- визначте структуру програмної системи, уточнивши характер агрегації.

Система обслуговування пасажирів невеликою фірмою, що має “на озброєнні” N автомобілів, телефон у ролі приймача замовлень і дівчину в ролі диспетчера, задовольняє цій моделі.

Перелічіть інші системи, що відповідають цієї моделі, постаравшись знайти новий зміст цієї моделі. (Зрозуміло, що заміна таксистів на рознощиків піцци нового змісту не вносить.)

7.5.6. Комп'ютерна система Хід рішення

Система призначена для підтримки ходу рішення деякої задачі.

- Задача визначена своєю умовою - фіксованим набором значень вихідних даних.
- Умову задачі генерує експерт по команді користувача – розв'язувач задачі.
- Кожен крок ходу рішення задачі полягає в застосуванні одного з методів перетворення даних до даних задачі.
- Метод перетворення може бути або застосований до конкретного набору значень даних, або ні.
- Питання про застосування методу до конкретного набору значень даних вирішує експерт.
- Процес вирішення задачі складається з послідовності кроків. На кожному кроці процесу рішення задачі крок рішення задачі запам'ятовується як крок ходу рішення.
- Кроки рішення задачі вибираються користувачем системи – розв'язувачем задачі з фіксованого списку методів задачі.
- Цей список специфічний для кожного класу задач.
- Ціль розв'язувача задачі полягає в тім, у процесі рішення задачі знайти хід рішення задачі у виді упорядкованої послідовності кроків.
- Процес рішення задачі може закінчитися або тупиком, або одержанням відповіді. Тупик - це стан задачі, при якому до даних

задачі не може бути застосований жоден з методів перетворення. Тупик визначається експертом. Відповідь - це стан задачі, яку визначає експерт.

- Процес рішення задачі може бути перерваний користувачем. Якщо рішення зайшло в тупик, процес рішення переривається експертом. Якщо рішення задачі досягнуте, хід рішення зберігається.

Всі істотні моменти процесу рішення задачі повинні бути відображені у вікні задачі.

Побудуйте модель цієї системи, для чого:

- виділіть об'єкти;
- визначте й опишіть їхні протоколи;
- визначте відношення зв'язку між об'єктами і розподіліть ролі;
- визначте структуру програмної системи, уточнивши характер агрегації.

Цієї моделі задовольняє, наприклад, картковий пасьянс. Порівняйте дійсний сценарій зі сценарієм гри двох гравців (впр. 7.6.1). Знайдіть у цих системах об'єкти, що володіють подібною поведінкою. Виділіть загальне і визначте розходження в їхньому поводженні. Спробуйте описати модель системи, у якій реалізовані обидва сценарії. Перелічіть інші системи, що відповідають цієї моделі, постаравшись знайти новий зміст цієї моделі.

7.5.7. Програмна система

Модель поводження популяції

Система моделює поводження співтовариства, що складається з великого числа однакових істот.

- Істота описується набором властивостей. Кожна істота має ідентифікуючу властивість (ім'я), характеристики C і параметр P .
- Життя співтовариства протікає в дискретному часі, кожен такт якого супроводжується зміною параметра P кожного члена співтовариства.
- Зміна параметра P кожного члена **а** співтовариства визначається загальною для всіх членів функцією F , аргументами якої, крім часу, є C и P :

$$P_a(t+1) = F(C_a, P_a, t)$$

- Взаємодія двох членів співтовариства **a** і **b** відбувається в той момент часу коли їхні властивості задовольняють співвідношенню $R(a, b)$.
- У результаті взаємодії змінюють свої значення характеристики кожного з членів співтовариства. Закон зміни характеристик пари об'єктів **a** і **b** визначається парою функцій

$$Ca = G(a, b); Cb = H(a, b)$$
- У початковий момент часу всі об'єкти, що утворюють співтовариство, генеруються спеціальним пристроєм.
- Користувач має можливість закінчити процес життя співтовариства й одержати результат моделювання в деякому виді.

Побудуйте модель цієї системи, для чого:

- виділіть об'єкти;
- визначте й опишіть їхні протоколи;
- визначте відношення зв'язку між об'єктами і розподіліть ролі;
- визначте структуру програмної системи, уточнивши характер агрегації.

Цій моделі задовольняє, наприклад, броуновський рух молекул у суміші з декількох газів у замкнутому об'ємі. Ідентифікатором молекули є її речовина і номер. Характеристики - вектор швидкості молекули. Параметри - координати молекули. Взаємодія молекул - їхнє зіткнення. У результаті взаємодії змінюються характеристики.

Порівняєте дійсний сценарій зі сценарієм гри в більярд. Знайдіть у цих системах об'єкти, що володіють подібним поведінням. Виділіть загальне і визначте розходження в їхньому поведінні. Спробуйте описати модель системи, у якій реалізовані обидва сценарії. Перелічіть інші системи, що відповідають цієї моделі, постаравшись знайти новий зміст цієї моделі.

7.5.8. Програмна система

Покупець – банк - продавець

Система підтримує процедуру покупки товару в магазині з оплатою по безготівковому розрахунку - через банк.

- У системі взаємодіють *покупець*, *продавець*, *банк*.
- Покупець посилає продавцю запит про покупку даного *товару*.

- Продавець або виставляє покупцю *рахунок*, у якому зазначена ціна товару, або сповіщає покупця про те, що даний товар відсутній. У цьому випадку *угода* закінчена.
- Покупець або дає *доручення* банку оплатити товар, або сповіщає продавця про те, що він відмовляється від *покупки*. У цьому випадку *угода* закінчена.
- Банк або оплачує товар, переводячи гроші з *рахунка покупця* на *рахунок продавця*, або сповіщає покупця про відсутність грошей на його рахунку. У цьому випадку угода закінчена. У випадку оплати товару про це сповіщаються і покупець, і продавець.
- Якщо покупець і продавець сповіщають банк про акт передачі товару, банк закриває угоду. Якщо покупець і продавець повідомляють банку про відмовлення від угоди, банк повертає гроші з рахунка продавця на рахунок покупця і закриває угоду.

Побудуйте модель цієї системи, для чого:

- виділіть об'єкти;
- визначте й опишіть їхні протоколи;
- визначте відношення зв'язку між об'єктами і розподіліть ролі;
- Особливу увагу приділіть проблемі контролю за ходом угоди, правильно розподіляючи відповідальності сторін.
- визначте структуру програмної системи, уточнивши характер агрегації.

Приведіть приклади систем, у яких повинний бути реалізований жорсткий контроль правильності взаємодії декількох незалежних об'єктів.

7.5.9. Програмна система

Буфер обміну

Система реалізує обмін об'єктами між Джерелом і Приймачем через спеціальний пристрій - буфер обміну.

- Джерело і приймач являють собою сховища однотипних об'єктів (речей), причому кожна річ займає в сховищі своє місце.
- Користувач може:
 - ⇒ знайти і вказати на будь-яку річ у сховищі - приймачі;
 - ⇒ вірізувати чи скопіювати відзначену річ у буфер обміну;

⇒ знайти чи створити вільне місце в сховищі-приймачі і відзначити його;

⇒ вставити у визначене місце приймача річ з буфера обміну;

⇒ поміняти ролями джерело і приймач.

- Джерело видаляє зі свого складу місце, що звільнилося після вирізки.
- Приймач може переповнитися, а джерело - спорожніти.

Побудуйте модель цієї системи, для чого:

- виділіть об'єкти;
- визначте й опишіть їхні протоколи;
- визначте відношення зв'язку між об'єктами і розподіліть ролі;
- визначте структуру програмної системи, уточнивши характер агрегації.

Описана система дуже схожа на буфер обміну, реалізований в операційних системах. Якими командами користувача і властивостями сховищ речей Ви б доповнили цю модель?

7.5.10. Програмна система керування конвеєром

Система призначена для керування конвеєром по виробництву деталей із заготовок.

- Заготовки зберігаються на заготівельному складі.
- Конвеєр складається з трьох установок, кожна з яких виконує одну спеціалізовану операцію по обробці заготовки.
- Результат обробки заготовки кожною установкою може бути обмірюваний.
- На виході кожної установки стоїть контролер якості – прилад, що перевіряє відповідний параметр і оцінює його на відповідність стандарту.
- Якщо заготовля деталь бракована, вона видаляється на склад браку, якщо деталь відповідає стандарту, вона надходить на вхід наступної установки чи на склад готових виробів.

Побудуйте модель цієї системи, для чого:

- виділіть об'єкти;
- визначте й опишіть їхні протоколи;
- визначте відношення зв'язку між об'єктами і розподіліть ролі;
- визначте структуру програмної системи, уточнивши характер агрегації.

Принцип конвеєризації застосовується в багатьох системах. Якими додатковими пристроями, властивостями й операціями можна доповнити цю модель, щоб зробити конвеєр універсальним?

7.5.11. Програмна система - комутатор

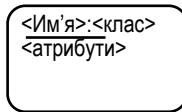
Система - комутатор призначена для керування з'єднанням декількох зовнішніх (вхідних вузлів) з декількома внутрішніми вузлами. Вона складається з $k = 2$ зовнішніх і $m = 4$ внутрішніх вузлів.

Система повинна задовольняти наступним вимогам:

- І на зовнішні, і внутрішні вузли можуть надходити запити на комутацію вхід-вихід.
- Для зовнішніх вузлів запит є загальним: якщо вільний хоча б один із зовнішніх вузлів, він приймає цей запит до обслуговування.
- Внутрішні вузли можна ототожнити з запитами: кожен внутрішній вузол може ініціювати свій запит.
- Обслуговування запитів полягає в наступному:
 - ⇒ Запит, що надходить на зовнішні вузли, містить номер внутрішнього вузла. Якщо група зовнішніх вузлів зайнята чи відповідний внутрішній вузол зайнятий, запит відкидається. У протилежному випадку система з'єднує вільний зовнішній вузол з відповідним внутрішнім вузлом. Обидва цих вузла стають зайнятими.
 - ⇒ Запит від одного з внутрішніх вузлів задовольняється, якщо хоча б один із зовнішніх вузлів вільний. Система з'єднує вільний зовнішній вузол з відповідним внутрішнім вузлом. Обидва цих вузла стають зайнятими.
 - ⇒ І на зовнішній, і на внутрішній вузли, якщо вони зайняті, може надійти команда Відбій, що звільняє вузол.

Побудуйте модель цієї системи, для чого:

- виділіть об'єкти;
- визначте й опишіть їхні протоколи;
- визначте відношення зв'язку між об'єктами і розподіліть ролі;



- визначте структуру програмної системи, уточнивши характер агрегації.

Зразком реалізації системи є комутатор внутрішніх і зовнішніх телефонів. Багато в чому ця система схожа на систему масового обслуговування (вправа 3). Опишіть їхні загальні риси і розходження. Як можна об'єднати ці системи в одну?

7.6. Діаграми об'єктів

Діаграма об'єктів – графічна форма опису об'єктів і їх зв'язків у логічному проєкті системи. Діаграми об'єктів описують, перш за все, статичну сторону взаємодії об'єктів. При аналізі і проєктуванні системи діаграми об'єктів використовуються для наочного опису семантики механізмів взаємодії об'єктів.

Рис.7.7. Значок об'єкта і деякі варіанти його заповнення.

Об'єкти на діаграмі зображуються у виді значків (рис.7.7.). На значку об'єкта вказують ту інформацію, що визначає істотні сторони семантики об'єкта: ім'я, клас, деякі атрибути. Синтаксис написів на значку об'єкта зазначений на рисунку. Значки об'єктів заповнюються в процесі аналізу чи проєктування. На стадії аналізу імена об'єктів, класів і атрибутів можуть писатися на будь-якій мові, що розуміють всі учасники проєкту. Не слід перевантажувати значок класу іменами атрибутів: діаграма відбиває ті істотні аспекти логічної моделі системи, що визначають

семантику зв'язків. Повні описи класів, яким належать об'єкти, відбиваються в CRC-картках класів.

Зв'язки між об'єктами зображують відрізком, що з'єднує значки цих об'єктів. Як ми бачили, зв'язки між об'єктами уточнюються в процесі аналізу системи. Тому спочатку цей зв'язок є асоціативним: він лише відбиває той факт, що між двома об'єктами існує канал зв'язку. (Відношення асоціації між класами, як і інші відносини між класами, ми розглянемо пізніше.)

Рис.7.8. Позначення зв'язку між об'єктами.

Якщо між об'єктами **A** і **B** встановлений зв'язок **L**, ці об'єкти можуть обмінюватися повідомленнями. Повідомлення (запит) здійснюється операцією **M**, елементи сигнатури якої позначаються на зв'язку. Напрямок повідомлення показується стрілкою. Як префікс до повідомлення може бути доданий його порядковий номер, що використовується для впорядкування потоку повідомлень. Власне порядок повідомлень описують діаграми переходів і станів, про які мова йтиме нижче. Якщо повідомлення може бути послане незалежно від попередніх подій, воно не маркірується.

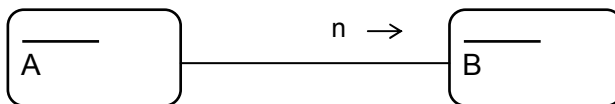
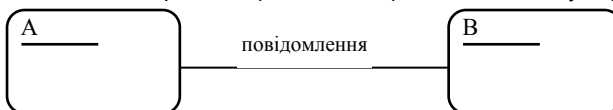
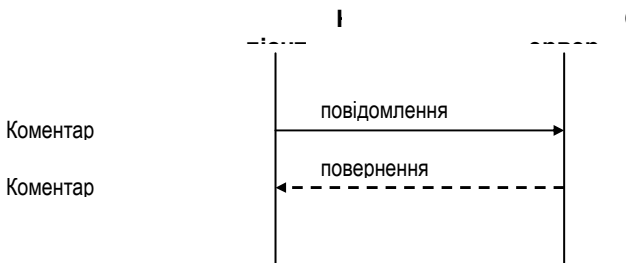


Рис.7.9. Позначення повідомлень.

Подібно до того, як об'єкти є екземплярами класів, зв'язки між об'єктами є екземплярами асоціацій. Асоціативні зв'язки уточнюються в процесі аналізу і проектування, перетворюючи у відношення, описувані в термінах мови реалізації. На кінцях лінії зв'язку ці уточнення





позначаються значками агрегації і залежності.

7.7. Діаграми взаємодії

Діаграми взаємодії, відображуючи об'єкти і їхню взаємодію, наочніше описують порядок взаємодії об'єктів. Об'єкти на ній зображені у виді прямих ліній, які можна інтерпретувати як лінії часу. Зв'язки, як і в діаграмах об'єктів, указуються лініями-стрілками, і маркуються повідомленнями. У діаграмі взаємодій указуються, якщо це необхідно, точки повернення керування до клієнта з поверненням результату акта взаємодії. При цьому допускається зображення повернення у виді пунктирної стрілки. Коментарі до повідомлення розташовують рядом з повідомленням (у тій же точці часу).

Рис 7.10. Основні елементи діаграми взаємодії

Діаграми взаємодій краще передають семантику сценаріїв системи. Немає ніяких перешкод до того, щоб включити в діаграму взаємодій і іншу інформацію: зв'язки, атрибути, ролі об'єктів. Однак ця інформація більш природно відповідає діаграмам об'єктів. Тому бажано використовувати і той і інший тип описів, погоджуючи при цьому специфікації продубльованної інформації.

Приклад 7.4. Аналіз системи Продавець – Покупець – Банк.

На рис.7.11 представлений можливий сценарій торгової операції по безготівковому розрахунку.



Рис. 7.11. Сценарій торгової операції по безготівковому розрахунку.

Укажемо на наступні аспекти цього приклада:

- Сценарій описує угоду, проведену до кінця. Насправді в текстовому описі торгової операції повинні бути зазначені варіанти розвитку подій, при яких одна зі сторін не може чи не хоче продовжити угоду. Усі ці варіанти не можуть бути адекватно відбиті в діаграмі взаємодій.
- Діаграми взаємодій, як і інші документи аналізу і проектування логічної моделі, модифікуються. Варіант, представлений на рис. 7.11, відбиває початкову стадію проектування, що підлягає змінам.

На рис. 7.11 вертикальними лініями відзначені групи повідомлень, що можуть бути впорядковані довільним чином, оскільки вони не залежать один від одного. На діаграмі їх не можна об'єднати, оскільки вони спрямовані різним адресатам.

- У сценарії, крім основних об'єктів, беруть участь ще такі об'єкти, як
 - рахунок на покупку,
 - платіжне доручення,
 - банківський Рахунок покупця,
 - банківський рахунок продавця,
 - доручення,
 - накладна,
 - внутрішні документи кожного з учасників угоди.

Усі ці об'єкти-документи використовуються об'єктами-учасниками угоди. Вони пересилаються в повідомленнях як фактичні параметри.

- Діаграма об'єктів, розглянутого сценарію взаємодії, виглядає тривіально: взаємодіючі об'єкти зв'язані за принципом “кожний з кожним”. Керування взаємодією покупець, продавець і банк – сервери. Асоціативні зв'язки переважані запитами-повідомленнями і поверненнями результатів.

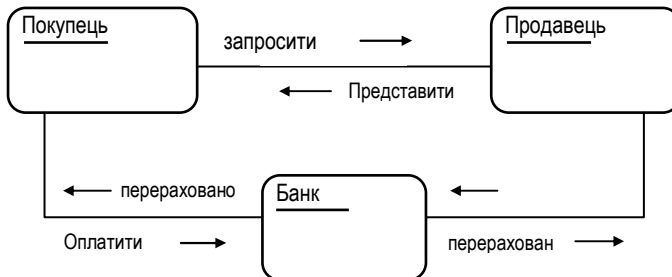


Рис 7.12. Діаграма об'єктів Покупець – Продавець - Банк.



Рис 7.13. Аналіз структури і поведінки об'єктів Покупець, Продавець, Банк.

Об'єкти Покупець, Продавець, Банк, у свою чергу, являють собою систему взаємодіючих об'єктів. Аналіз цих об'єктів супроводжується заповненням CRC-карток, побудовою діаграм об'єктів, діаграм взаємодії й інших документів супроводу проекту. Таким чином, аналіз проекту переноситься на периферію триади Покупець, Продавець, Банк (рис. 7.13.).

Побудовані діаграми об'єктів і взаємодій визначили обов'язки взаємодіючих сторін, суть яких полягає в тім, що вони:

- пересилають іншим об'єктам документи.
- дотримують правильної послідовності дій.

Інші (внутрішні) операції ми покладаємо на інші об'єкти, імена, обов'язки і взаємодії яких підлягають визначенню.

Таким чином, об'єкти Покупець, Продавець, Банк визначають інтерфейси взаємодії, а об'єкти периферії, асоційовані з ними – реалізацію.

Розглянемо тепер, наприклад, структуру системи, асоційованої з об'єктом Продавець (рис. 7.14).

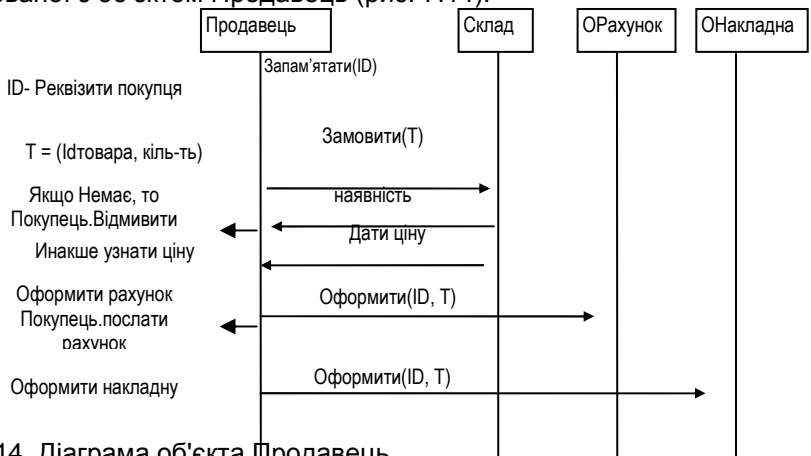


Рис. 7.14. Діаграма об'єкта Продавець

У цій структурі повинні бути представлені об'єкти-документи Рахунок, Накладна. Крім того, введемо ще один об'єкт – Склад, що буде вирішувати задачі, пов'язані з наявністю товару і ціною товару (Об'єкти, що здійснюють реєстрацію замовлень, рахунків, і т.п. ми в цьому навчальному прикладі ігноруємо, хоча в реальних системах вони присутні).

Опис сценарію роботи системи Продавець по обробці замовлення:

1. Запам'ятати ідентифікаційний номер замовлення.
2. Запросити склад про наявність товару в достатній кількості
3. Склад: Якщо товару немає, то
повідомляє Наявність = Немає
Інакше повідомляє Наявність = Так, Ценотовара
4. Якщо товару немає, послати повідомлення “Заказ відхилено”.
5. Інакше Оформити рахунок, послати повідомлення “Заказ прийнято”, послати покупцю рахунок

Опис сценарію роботи системи Продавець по оформленню угоди.

1. Оформити накладну.
2. Прийняти доручення.
3. Передати накладну покупцю

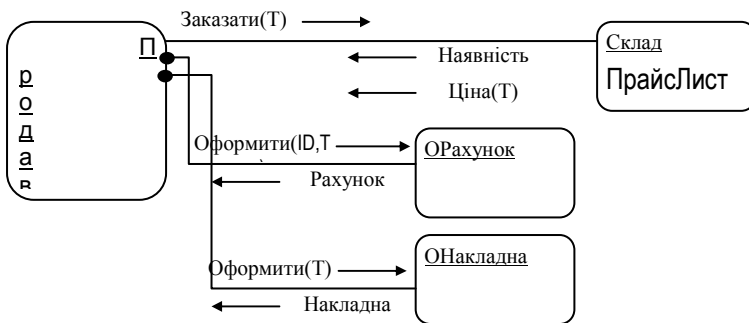


Рис. 7.15. Діаграма взаємодії Продавець по оформленню угоди.

7.8. Діаграми станів і переходів

Діаграма станів і переходів визначає структуру системи керування об'єктом чи цілою підсистемою (сукупністю взаємодіючих об'єктів)

Зберігши в цілому точку зору на систему керування як кінцевий автомат, методика опису керування, що представляється діаграмами станів і переходів містить кілька особливостей, що додають діаграмам простий і виразний вид.

Діаграма станів описує:

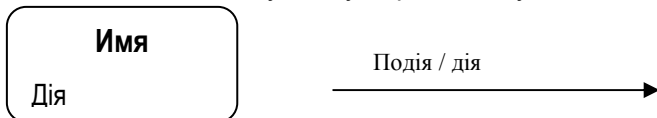
- множину А (керуючих) станів об'єкта;
- множину подій Х об'єкта;
- множину дій Y об'єкта;
- початковий і заключний стани;
- функції переходів і виходів.

Діаграми станів і переходів використовуються в ході аналізу, щоб визначити динаміку поведінки системи і реалізувати її на етапі проектування.

Основними графічними елементами діаграми є значки станів і переходів. Стани позначаються прямокутником із заокругленими куточками, у якому зазначені: ім'я стану і, якщо це необхідно, дія. Переходи позначаються стрілками, відзначеними подіями і діями. На рис. 7.16. показані значки стану і переходу.

Рис. 7.16. Значок стану і значок переходу

Таким чином, кожному стану варто дати унікальне для даної



діаграми (у даному автоматі) ім'я і приписати деяку дію. Імена потім будуть використовуватися як значення керуючих атрибутів, а дії будуть реалізовані методами.

Розрізняють дії, виконувані миттєво і дії, виконання яких вимагає часу. Дію, виконання якої вимагає часу, називають діяльністю.

Події – елементи безлічі входів повинні бути представлені формальним ім'ям, що позначає клас, метод чи дане. Початковий і заключний стани системи відзначаються спеціальними стрілками (рис. 7.17).

Рис. 7.17. Значка-оцінки початкового і заключного станів.

Приклад 7.5. Діаграма станів і переходів об'єкта Ліфт.

В цьому прикладі проектується управління пасажирським



ліфтом багатоповерхового будинку.

Технічне завдання на проектування складає частину сценарію програмної системи, в якому описується поведінка ліфта. У відповідності з цим описом можна виділити такі стани об'єкта Ліфт:

- Стан очікування на поверсі **N** (ліфт стоїть, пустий, двері зачинені)
- Стан руху за викликом з поверху **N** (ліфт рухається, пустий, двері зачинені)
- Стан руху по перевезенню на поверх **N** (ліфт рухається, наповнений, двері зачинені)
- Стан 1 наповнення пасажирями (ліфт стоїть пустий, двері відчинені)
- Стан 2 наповнення пасажирями (ліфт стоїть, наповнений, двері відчинені)
- Стан 3 наповнення пасажирями (ліфт стоїть переповнений, двері відчинено)

Зауважимо, що стани ліфта зазначеним переліком ґрунтуються на станах деяких об'єктів - частин ліфту: стан системи руху ліфта, підлоги ліфта і дверей ліфта. Таким чином, стан об'єкту ліфт описується вектором станів його частин Мотор, Двері, Підлога. Це – стандартна ситуація для об'єктів-агрегатів. Стан такого об'єкта визначається (можливо, сумісними або несумісними, залежними або незалежними) станами його частин. Якщо набір станів об'єкта в сценарії не визначений, його можна уточнити, описавши набори станів його частин, потім сформувавши простір всіляких станів цього об'єкта як простір векторів станів його частин. В цьому просторі потім необхідно виділити реалізуємі стани. Так, для об'єкта Ліфт стани його частин Мотор, Підлога, Двері наведено в таблиці.

Мотор (стан руху)	Підлога (стан наповнення)	Двері
Ліфт стоїть на поверсі N	Ліфт пустий	Двері відчинені
Ліфт рухається до поверху N	Ліфт наповнений	Двері зачинені
	Ліфт переповнений	

У нашому прикладі пристір всіляких станів містить $2 \cdot 3 \cdot 2 = 12$ елементів, тільки 6 з яких є реалізуємими. Вектор (Ліфт рухається, Ліфт наповнений, Двері відчинені) не є реалізуємим оскільки двері ліфта під час руху повинні бути зачиненими.

Крім станів частин ліфта, стани ліфту визначає його атрибут $N_{\text{поверху}}$, і, як ми побачимо пізніше, стан датчика часу, який відлічує час перебування ліфта в очікуванні наповнення (стан 1 наповнення пасажирами).

У тому ж описі вказані слідуючі події (в дужках вказані об'єкти, які посилюють відповідні повідомлення):

- Натискання кнопки **Виклик** (панель поверху **N**)
- Натискання кнопки **N поверху** (внутрішня панель ліфта)
- Ліфт спорожнів (датчик стану підлоги)

- Ліфт наповнився (датчик стану підлоги)
- Ліфт переповнився (датчик стану підлоги)
- Ліфт досягнув поверху N (датчик поверху N)
- Ліфт порожній і очікує дверей 10 сек. (Датчик часу)

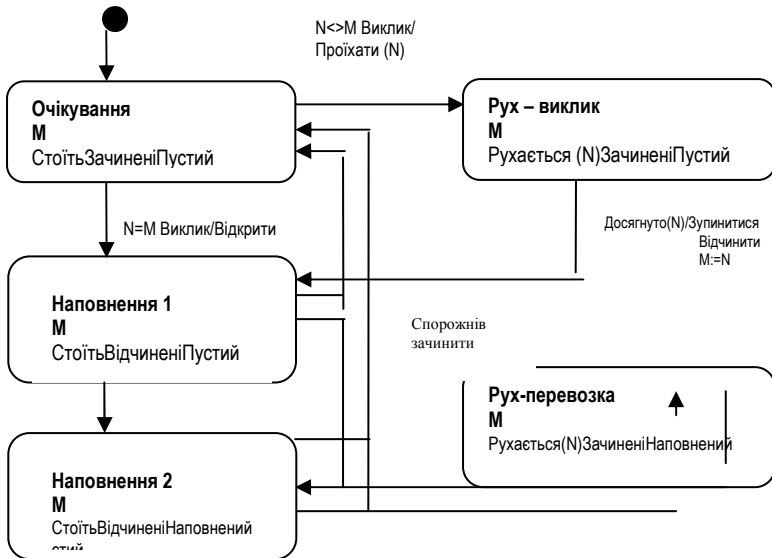
Крім цих подій переходами керують умови $M=N$ і $M \neq N$, де M – номер поверху, на якому знаходиться ліфт і N – номер поверху, на який ліфт повинен переїхати (за викликом або перевезенню).

Накінець, у сценарії вказані дії ліфта:

- Відчинити двері.
- Зачинити двері.
- Поїхати на поверх N.
- Зупинитись.

На діаграмі переходів і станів (рис. 7.18) в значках станів, крім власно імен станів, вказані стани частин ліфта. Уздовж ліній переходів позначені події без означення об'єктів – джерел відповідних повідомлень. Дії вказані також без позначення відповідних об'єктів – серверів. Це зроблено для того, щоб розвантажити діаграму від непотрібних деталей.

Відмітимо важливу деталь реалізації дії: кожна з них адресовано об'єкту, який керує однією з частин ліфта. Вона оцінює відповідну діяльність (наприклад, відчинення дверей). Результат цієї діяльності слід проаналізувати в об'єкті Ліфт. Не дивлячись на те, що двері повинні відчинитися, в результаті несправності механізмів вони можуть і не відчинитися. Це - так звана виключна ситуація. Тому метод *Двері.Відчинити* потрібно реалізувати в вигляді логічної функції, яка повертає результат, сигналізуючи успішне закінчення дії.



8. ВІДНОШЕННЯ МІЖ КЛАСАМИ

Основна задача, яку розв'язують на етапі проектування – побудова архітектури програмної системи на основі її об'єктної моделі – тобто побудова ієрархії класів, що описують об'єкти і їх взаємодію.

Опис класів і відношень між ними здійснюється для того, щоб виявити загальні риси й розходження в структурі та поведінці об'єктів системи, а потім застосувати результати такого аналізу для реалізації системи на основі загального програмного коду, використовуваного неодноразово.

8.1. Відношення спадкування

Ієрархія класів програмної системи відбиває ієрархію абстракцій, використовуваних для опису її функціонування.

Приклад 8.1. Система - Електрична схема.

На рис. 8.1. показана ієрархія абстракцій Приклад, розроблена для програмної системи Електрична схема. Система повинна моделювати роботу електричної схеми, зібраної з електричних приладів.

Будь-яка конкретна електрична схема, яку моделює система, складена з об'єктів - екземплярів класів нижнього рівня ієрархії. При цьому власне поняття “нижній рівень” також відносно: практично всі об'єктні типи нижнього рівня в нашому прикладі також абстрактні. Вони, у свою чергу, можуть бути класифіковані. Лампи, наприклад, можуть бути денного світла, накаливання та інші. Однак, на деякому рівні ця класифікація дає таку точність опису, при якій отримана інформаційна модель повинна бути визнана остаточним описом об'єктів системи, призначеним для реалізації.



Рис. 8.1. Ієрархія абстракцій Прилад системи Електрична схема

Один раз описавши клас Споживач, ми використовуємо його багаторазово - як загальну частину опису класів Нагрівач і Освітлювач, а описи класів Нагрівач і Освітлювач, у свою чергу, як загальну частину опису об'єктів Праска, Камін, Лампа.

Другий аспект використання ієрархії абстракцій - побудова різних конкретних систем як різних спеціалізацій загальної абстрактної системи. Справді, виділимо верхні рівні абстракцій у нашому прикладі (рис. 8.2.):

- Джерело - постійне, перемінне (по ємності);
- Вимірник - датчики потенціалу, швидкості, кількості;
- З'єднувач - вузол, провідник, вимикач, перемикач;
- Споживач - зовнішній, внутрішній.

Наповнимо нашу абстракцію новим змістом: будемо вважати, що ми маємо справу з водогінною мережею. Замість електричного струму в системі використовується вода. Усі абстракції тепер уточнюються в “водопровідних” термінах породженням нових класів нижніх рівнів ієрархії класів.

Незважаючи на те що в програмній системі Електрична схема, яку ми зараз проектуємо, така інтерпретація абстрактної моделі Прилад може не знадобитися, вона, безумовно, являє собою самостійний інтерес.

Уявіть собі, що Ваша програмна система повинна моделювати схему забезпечення будинку електрикою, водою й газом. У рамках цього проекту Ви обов'язково зштовхнетеся з різними, але породжуваними однією абстрактною моделлю підсистемами:

- Електрична схема будинку;
- Схема водопостачання будинку;
- Схема газопостачання будинку.



Рис. 8.2. Абстрактна модель - Прилад.

На етапі аналізу цієї набагато більшої системи Вам потрібно буде виділити в явному виді рівень абстракцій, приблизно відповідний класифікації, представленої на рис. 8.2. При цьому необхідно:

- ввести імена абстракцій як імена класів, атрибутів, властивостей, методів;

- описати методи.

Аналіз системи в термінах “загальне-часне” реалізується через *відношення спадкування*.

Однак одного цього виду відношення, як ми знаємо, недостатньо для *адекватного* опису системи.

8.2. Відношення агрегації

Так, у системі Забезпечення будинку з'явиться прилад Нагрівач води, у якому будуть агрегировані газовий прилад (або електричний прилад) і водяний прилад. Оскільки прототип нашої моделі - реальний газовий стовпчик дійсно являє собою агрегат, це рішення найбільш ймовірне. Методи взаємодії його газової й водяної частин відмінні від методів класу Прилад.

У реальних системах забезпечення будинків, крім цього агрегату, використовуються й інші агрегати, частини яких належать різним підсистемах забезпечення. Отже, їх також можна класифікувати й описувати на верхніх рівнях ієрархії абстракцій.

Аналіз системи в термінах “ціле-частина” реалізується через *відношення агрегації*. Відношення агрегації, встановлене між класами верхнього рівня, утворює ієрархію агрегації.

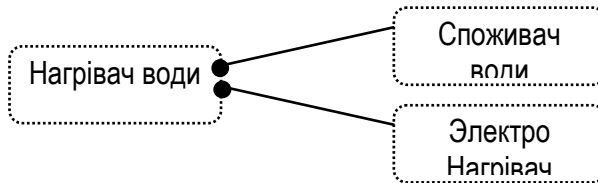


Рис. 8.3. Агрегат Нагрівач води

8.3. Відношення використання

Систему забезпечення будинку, очевидно, найкраще розглядати як систему з декількох відносно незалежних об'єктів-підсистем, що регулюють свою взаємодію, обмінюючись повідомленнями. У реальних

будинках системи забезпечення водою, газом і електрикою є відносно незалежними: відсутність води ніяк не пов'язана з відсутністю газу чи електрики. При цьому кожна з підсистем повинна адекватно реагувати на зміни в іншій: інакше електро- газо- водо- агрегати не будуть працювати.

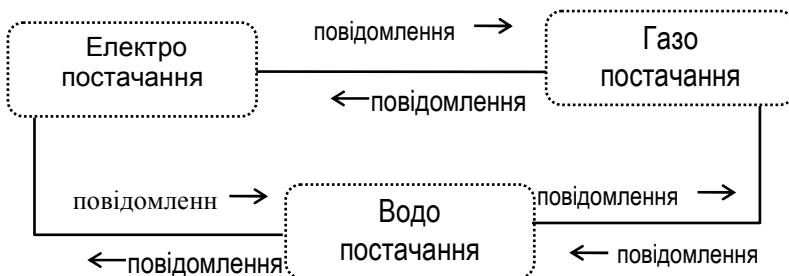


Рис. 8.4. Система забезпечення будинку

Аналіз і проектування системи в термінах “клієнт - сервер” реалізується через *відношення використання*.

Відношення зв'язку між об'єктами системи визначається в ієрархії класів системи як відношення використання.

Проектована програмна система складається тільки з конкретних екземплярів об'єктів нижнього рівня або як з частин, або як із незалежних об'єктів. У будь-якому випадку взаємодія класів реалізується як обмін повідомленнями, тобто як відношення зв'язку .

8.4. Відношення інстанцірування

Дуже специфічним видом відношення між класами є *інстанціювання*. Цей тип відносин ми ще не розглядали.

Повернемося ще раз до приклада нагрівача води. Звернемо увагу на той факт, що джерелом енергії ми вибрали електрику. Однак функції нагрівача можуть бути описані і на більш високому рівні абстракції. На рис. 8.5 представлені відповідні діаграми опису класів.

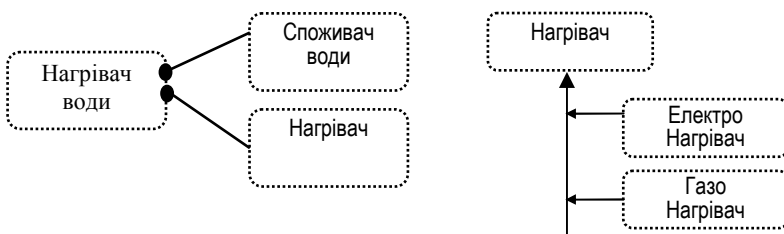


Рис. 8.5. Визначення класів нагрівачів води через відношення інстанціювання.

Клас Нагрівач є абстрактним - він створений спеціально для того, щоб описати функції, загальні для всіх нагрівачів. Атрибут Нагрівач відіграє роль формального параметра у визначенні класів за допомогою відношення інстанціювання. Спеціалізовані - класи Газо-Нагрівач, Електронагрівач і Сонцenaгрівач успадковують клас Нагрівач. Вони відіграють роль фактичних параметрів у визначеннях спеціалізованих класів Тен (Нагрівач води електричний), Бойлер (Нагрівач води газовий), Басейн (Нагрівач води сонячний).

Тен = НагрівачВоди < Електронагрівник >

Бойлер = НагрівачВоди < ГазоНагрівач >

Басейн = НагрівачВоди < СонцеНагрівач >

Класи Тен, Бойлер, Басейн знаходяться у відношенні інстанціювання з класом НагрівачВоди. Клас Нагрівач води при цьому відіграє роль класу-шаблону (параметризованого класу, контейнерного класу).

Відношення інстанціювання - специфічна форма узагальнення, що використовує абстрактні об'єктні типи як формальні параметри для визначення спеціалізованих класів через конкретні об'єктні типи як фактичні параметри.

Відзначимо деякі важливі аспекти інстанціювання як методу проектування:

- Формальним параметром класу-шаблону є саме тип, а не типізована перемінна.

- Клас - фактичний параметр повинний підтипом класу - формального параметра: повною мірою повинний бути дотриманий принцип підстановки.
- Інстанціювання додає способам опису ієрархії класів велику гнучкість. Зокрема, агрегировання в сполученні з інстанціюванням можна використовувати при описі верхніх рівнів ієрархії класів.
- Альтернативою інстанціюванню в багатьох випадках може бути використання віртуальних методів - спадкування у формі специфікації.
- Інстанціювання як метод програмування відіграє ключову роль для створення й використання так званих узагальнених об'єктів і алгоритмів. Бібліотеки таких об'єктів і алгоритмів використовують для реалізації конкретних алгоритмів і об'єктів як окремих випадків стандартних алгоритмічних і об'єктних схем.

Приклад 8.2.

Розглянемо алгоритми обчислення мінімального з двох чисел, реалізовані для цілих і для дійсних чисел у виді функцій. Тіла цих функцій абсолютно ідентичні, а заголовки відрізняються тільки типами формальних параметрів. Тіла опису являють собою один алгоритм, логіка якого не залежить від типу даних, що опрацьовуються.

```
Function Min(a, b : Integer) : Integer;
```

```
begin
```

```
  If a < b then Min := a else Min := b
```

```
end;
```

```
Function Min(a, b : Real) : Real;
```

```
begin
```

```
  If a < b then Min := a else Min := b
```

```
end;
```

У кожного програміста виникає природне бажання написати одну функцію, що годила б у всіх подібних ситуаціях для пошуку мінімуму.

Ця гіпотетична функція повинна виглядати так:

```
Function Min(a, b : LinearOrder) : LinearOrder;
```

```
begin
```

```
  If a < b then Min := a else Min := b
```

end;

При реалізації функцій Min для цілих і дійсних чисел імена типів Integer і Real повинні бути підставлені замість імені типу LinOrder. Ім'я типу LinOrder, отже, відіграє роль формального параметра, і імена Integer і Real використовуються як фактичні параметри.

Яким умовам повинні задовольняти типи, що відповідають формальним і фактичним параметрам в цій гіпотетичній конструкції? Функція порівняння “ < ” повинна бути описана в абстрактному класі-формальному параметрі, класи-фактичні параметри повинні успадковувати цей клас і перевизначити функцію “ < ”, інтерпретуючи її “для себе”.

Іграшковий приклад, розглянутий вище, демонструє тільки принцип інстанціювання. В реальних програмних системах цей принцип використовується в набагато більш складних ситуаціях. Так, наприклад, використовуються контейнерні класи зі стандартною реалізацією динамічних структур даних - стеків, списків, черг, дерев і т.д. Методи операцій над цими структурами, власне кажучи, не залежать від наповнення інформаційної частини їхніх елементів. Оскільки в кожній досить великій програмній системі класи цих структур застосовуються неодноразово, кожна конкретна реалізація використовує відповідний шаблон.

На жаль, мова Borland Pascal не підтримує відношення інстанціювання: поняття параметризованого об'єктного типу там не існує. Тому в кожній конкретній ситуації необхідно використовувати різні технічні прийоми, що замінюють параметризацію.

8.5. Відношення асоціації

Відношення *асоціації* встановлює лише семантичний зв'язок між класами. Воно є засобом попереднього усвідомлення програмістом того факту, що дані класи зобов'язані взаємодіяти.

Аналітик фіксує наявність цієї взаємодії, перетворюючи надалі асоціативні відносини в більш спеціалізовані, та зводить, в остаточному підсумку, асоціацію до форм, що мають бути реалізовані засобами мови. Так, розглядаючи приклад опису системи забезпечення будинку, ми неодноразово підкреслювали ту обставину, що ухвалення рішення

про вид відносини між класами не є однозначним. У багатьох ситуаціях існує вибір, який потрібно здійснити пізніше, зафіксувавши зараз тільки необхідність такого вибору. Це і є асоціація.

Незважаючи на те, що асоціативний зв'язок буде уточнюватися, установлюючи такий зв'язок, проектувальник виявляє деякі істотні риси зв'язку. По-перше, у цьому аналізі визначаються ролі учасників зв'язку. По-друге, визначається потужність відношення.

8.5.1 Ролі

Роль учасника відношення визначається тими цілями (потребами) і обов'язками (можливостями), яких повинний досягти (задовольнити) і зобов'язаний (здатний) забезпечити учасник взаємодії об'єктів.

Коротше кажучи, роль абстракції - це те, чим вона є для зовнішнього світу в даний момент.

Так, виділивши в системі керування торгової операції - покупки товару по безготівковому розрахунку об'єкти Покупець, Продавець, Банк, ми тим самим уточнюємо розходження їх ролей, та повинні уточнити зміст цих ролей, описавши властивості й методи, що забезпечують кожному об'єкту виконання своєї ролі. Наприклад:

Покупець хоче: вибрати товар, купити товар;

Покупець повинний: оплатити товар чи сповістити Продавця про відмовлення від угоди, документально оформити покупку.

Продавець хоче: продати товар.

Продавець повинний: представити рахунок чи сповістити Покупця про відсутність товару, документально оформити продаж.

Банк хоче: одержати прибутки від фінансового забезпечення торгової операції.

Банк повинний: переказати гроші з рахунка Покупця на рахунок продавця; повідомити Продавцю про оплату товару Покупцем; повідомити Покупцю про відсутність необхідної суми на рахунку Покупця; документально оформити переказ грошей.

Розподіливши ролі, ми уточнили ті властивості і методи, що відрізняють взаємодіючі об'єкти. Але це тільки один аспект аналізу

взаємодії. Другий аспект - з'ясування ступеня спільності взаємодіючих об'єктів. Аналізуючи цей аспект, ми повинні визначити загальні риси поведінки об'єктів і ті властивості й методи, що забезпечують цю спільність.

У розглянутому прикладі ця спільність визначається тим, що всі об'єкти здійснюють торгову операцію, правила якої зафіксовані юридично. Отже, усі її учасники - юридичні особи - учасники ринку. Усі юридичні особи, по визначенню, є учасниками ринку, але список ролей для кожної юридичної особи - учасника ринку - свій. Зокрема, деякі юридичні особи не мають права відігравати роль Продавця. Ще більш широке коло юридичних осіб не має права відігравати роль Банку. З іншого боку, крім юридичних осіб, учасниками ринку є також фізичні особи, що купують товари за готівку, тобто, не використовуючи Банк. Банк, у свою чергу, у торговій операції, відіграє роль посередника. Крім цієї ролі, банк грає й інші ролі - накопичувача, кредитора і т.д.



Рис. 8.6. Ролі юридичних осіб - учасників ринку.

Ми виділили абстрактний клас Юридична особа, властивості й методи повинні бути уточнені. Принцип нашої класифікації - право участі в безготівкових торгових операціях. До атрибутів цього класу відносяться Юридичне ім'я, Юридична адреса, Реквізити рахунка, і т.д. Клас забезпечує своє поведіння відповідними методами й властивостями. При цьому роль юридичної особи як учасника ринку забезпечує, зокрема, атрибут Реквізити рахунка. Інші реквізити -

Юридичне ім'я, Юридична адреса - є загальними для різних ролей юридичної особи.

У розглянутому прикладі поняття ролі використовувалося для побудови фрагмента ієрархії класів розподілом цілей і обов'язків по класах. Ще одна методологічна функція поняття ролі - побудова класів, що інтегрують у собі кілька рольових функцій. Справді, об'єкт Магазин, повинний інтегрувати в собі функції класів Покупець і Продавець, оскільки він ще і заковує оптом товари, що продає в роздріб. Таким чином, поняття ролі не зводиться до поняття класу, і навпаки, поняття класу не зводиться до поняття ролі. Конкретні класи системи можуть забезпечувати виконання декількох ролей, і навпаки, одну роль у різних актах взаємодії можуть підтримувати кілька класів.

Так, у прикладі Система забезпечення будинку клас Нагрівач грає дві ролі: роль споживача енергії (електрики, газу, сонячної енергії) і роль джерела енергії (для підігріву води). Тут, виявляється, і споживач води грає дві ролі - він споживає не тільки воду, але і тепло. Фізичний аспект цього приладу полягає в тому, що енергія поточної води визначається двома параметрами: потужністю плинину й температурою.

Ролі взаємодіючих об'єктів (і відповідно, класів) можуть бути визначені в абстрактних термінах. Деякі терміни цієї класифікації ми уже використовували. Це найбільш загальні поняття, що визначають акт взаємодії: клієнт, сервер. Більш специфічні рольові поняття - постачальник інформації, споживач інформації, співробітники по зборі інформації, носій інформації (документ), і т.д.

Клас Продавець - споживач інформації "своїх" серверів і постачальник інформації для Покупця, для того, щоб пред'явити рахунок Покупцю, повинний скористатися послугами об'єкта Склад (постачальник інформації про наявність товару), об'єкта Генератор розрахунку (постачальник інформації для грошових полів рахунка), і створити документ - об'єкт Рахунок (носіє інформації), об'єднавши в ньому всю необхідну інформацію для посилки Покупцю.

От як виглядає документ Рахунок, що поєднує в собі всі необхідні дані (у фігурних дужках зазначений постачальник інформації):

Структура документа Рахунок. Оформляється Продавцем.

```
Рахунок ::= (  
  Номер рахунка  
  Реквізити Продавця  
  Ім'я покупця          {Покупець}  
  Список (  
    Товар              {Покупець, Склад}  
    Ціна                {Генератор розрахунку}  
    Кількість          {Покупець, Склад}  
    Вартість           {Генератор розрахунку}  
  )  
  Разом сума            {Генератор розрахунку}  
  Сума прописом  
  Податки              {Генератор розрахунку}  
  Підпис продавця  
  Дата оформлення  
)
```

8.5.2. Ключі

Ключ - це атрибут, значення якого унікально ідентифікує об'єкт. Ключі використовуються для пошуку необхідної інформації. Ключів може бути трохи, але їхні значення повинні бути унікальними.

У нашому прикладі ключами є атрибути Номер рахунка, Ім'я покупця, Товар. Номер рахунка буде використаний при оформленні документа Накладна, Ім'я Покупця - для адреси посилки рахунка, Товар - для пошуку товару в необхідній кількості і для визначення цін у розрахунку вартості.

8.5.3. Потужності

Потужність відносини асоціації встановлює можливу кількість його учасників. Розрізняють три значення цього поняття:

- один-до-одного;
- один-до-багатьох;

- багато-до-багатьох.

Значення “один-до-одного” означає, що кожен об’єкт-клієнт має рівно одного “свого” об’єкта-сервера і навпаки, кожен об’єкт-сервер обслуговує тільки одного “свого” клієнта (Дане відношення взаємне однозначно).

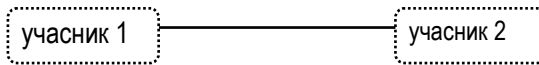


Рис. 8.7. Відношення потужності “один-до-одного”

Значення “один-до-багатьох” означає, що один з об’єктів-учасників асоціації представлений у єдиному числі, а партнерів у нього може бути декілька (мовою математики це означає, що дане відношення зворотне чи до нього відношення функціональне).

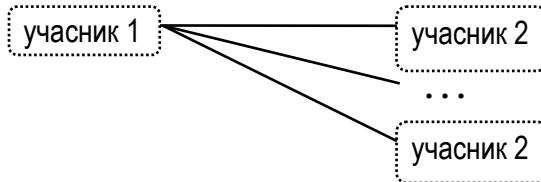


Рис. 8.8. Відношення потужності “один-до-багатьох”

Значення “багато-до-багатьох” означає, кожний з об’єктів-учасників асоціації представлений у множині.

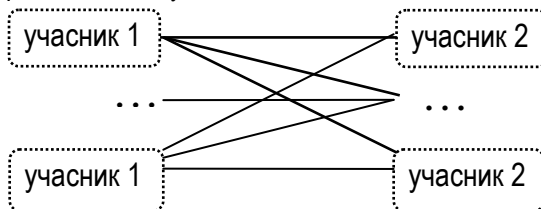


Рис. 8.9. Відношення потужності “багато-до-багатьох”

У розглянутому прикладі Покупець і Продавець знаходяться у відношенні “багато-до-багатьох”, Покупець і Банк - у відношенні “один-до-багатьох” (тим самим ми визначили, що Покупець тримає рахунок тільки в одному банку), Продавець і Склад - у відношенні “один-до-одного” (тим самим ми визначили, що в кожного продавця є тільки один розрахунковий відділ).

Типи відношень “один-до-багатьох”, “багато-до-багатьох”, у свою чергу, допускають уточнення по діапазону значень потужності сторони “багато”.

8.6. Діаграми класів

Діаграма класів графічно відображає класи і їхні відносини, представляючи тим самим логічний аспект проекту. Окрема діаграма класів представляє визначений ракурс ієрархії класів. На стадії аналізу діаграми класів використовуються для того, щоб виділити загальні ролі й обов'язки об'єктів, що забезпечують необхідне поводження системи. На стадії проектування діаграми класів використовуються для визначення архітектури системи. Система позначень елементів на діаграмах класів погоджена із системою позначень елементів на діаграмах об'єктів. Система позначень, описана нижче, являє собою систему Г.Буча [2], адаптовану для цілей цієї книги.

Два головних елементи діаграми класів - це значки класів і ліній, що показують їхні відносини.

Як і значок об'єкта, значок класу повинний містити ім'я. На деяких значках класів корисно перелічувати декілька атрибутів і операцій класу. Атрибути й операції на діаграмі представляють прообраз повної специфікації класу, у якій з'являються всі його елементи. Якщо ми хочемо побачити на діаграмі більше атрибутів класу, ми можемо збільшити значок; якщо ми зовсім не хочемо їх бачити - ми видаляємо поділяючу рису і пишемо тільки ім'я класу.

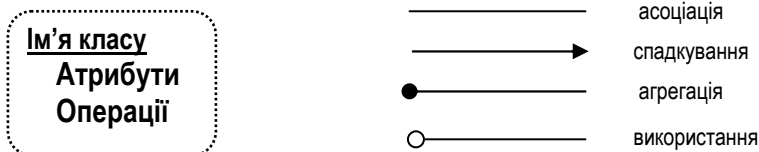


Рис. 8.10. Значки класу і відношень.

Атрибут на значку класу може позначатися ім'ям чи класом, чи і тим і іншим, і, можливо, мати значення за замовчуванням:

- **A** - тільки ім'я атрибута;
- **.C** - тільки клас;
- **A.C** - клас і ім'я;
- **A.C = E** – клас, ім'я і значення за замовчуванням.

Операції звичайно зображуються на значку класу або своїм ім'ям з дужками, або ім'ям разом із сигнатурою.

- **N()** - тільки ім'я операції;
- **N(A) :R** – ім'я (**N**), формальні параметри (**A**), клас значення, що повертається, (**R**)

Загальний принцип системи позначень: їхній синтаксис аналогічний синтаксису обраної мови реалізації.

Верхні рівні ієрархії спадкування, як правило, є абстрактними класами. Тому що цей факт важливий, для позначення абстрактних класів використовується спеціальний значок, називаний прикрасою (рис. 8.11). Загальний принцип використання прикрас: вони представляють вторинну інформацію про деяку сутність у системі.

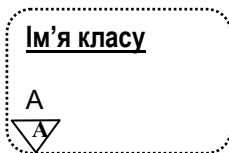


Рис. 8.11. Значок класу з прикрасою.

При зображенні конкретного зв'язку їй можна співставити текстову позначку, що документує ім'я цього зв'язку чи підказує її роль.

Біля значка асоціації, можна вказати її потужність, використовуючи наступні позначення:

- **1** - в точності один зв'язок;
- **N** - необмежене число (0 чи більше);
- **0..N** - нуль чи більше;
- **1..N** - одна чи більше;
- **0..1** - нуль чи одна;
- **3..7** - зазначений інтервал;
- **1..3, 7** - зазначений інтервал чи точне число.

Позначення потужності пишеться біля кінця лінії асоціації й означає число зв'язків між кожним екземпляром класу на початку лінії з екземплярами класу в її кінці. Якщо потужність явно не зазначена, то мається на увазі, що вона не визначена.

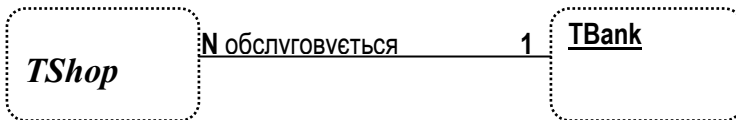


Рис. 8.10. Асоціація з позначками потужності й роллю.

Позначення інших типів зв'язку уточнюють рисунок асоціації додатковими позначками. Це зручно, оскільки в процесі розробки асоціативні зв'язки, звичайно, уточнюються. Спочатку документується наявність зв'язку, а потім, після ухвалення рішення про тип відносини, асоціація уточнюється як спадкування, агрегація чи використання.

Значок спадкування - це стрілка, що вказує від підкласу до суперкласу. Значок агрегації виходить із значка асоціації додаванням зафарбованого кружка на кінці, що позначає агрегат. Екземпляри класу

на іншому кінці стрілки будуть у якомусь змісті частинами екземплярів класу-агрегату. Відзначимо, що можливі рефлексивна й циклічна агрегації. Нарешті, значок використання позначає відношення "клієнт/сервер" і зображується як асоціація з порожнім кружком на кінці, що відповідає клієнту. Цей зв'язок означає, що клієнт має потребу в послугах сервера, тобто операції класу-клієнта викликають операції сервера чи мають сигнатуру, у якій значення, що повертається, чи аргументи належать класу сервера.

Незважаючи на те, що мова Borland Pascal не підтримує параметризованих класів, відношення інстанціювання можна і потрібно використовувати при проектуванні класів. Клас-шаблон відзначається спеціальною прикрасою - пунктирним прямокутником у правому верхньому куті, у якому зазначені формальні параметри. Інстанційований клас зображується прямокутником із суцільною границею, що містить фактичні параметри. Зв'язок між параметризованим класом і його інстанціюванням зображується пунктирною лінією, що вказує на параметризований клас.

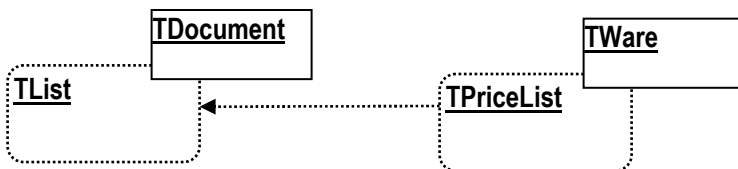


Рис. 8.11. Позначення інстанціювання

8.6.1. Утиліти

Мова Borland Pascal дозволяє розробнику застосовувати як процедурний, так і об'єктно-орієнтований стиль програмування. У програмі можна використовувати процедури і функції, що не належать якому-небудь класу. Їх називають вільними підпрограмами. Часто кілька вільних підпрограм об'єднані семантично – вони, наприклад, реалізують абстрактний тип даних (див. приклад 3.1.). Такі об'єднання називають утилітами. У гл. 4, п. 6.5. ми розглядали утиліти як об'єкти-обчислювачі. Якщо в проекті використовуються утиліти, це в діаграмі класів позначають звичайним значком класу з прикрасою у виді тіні

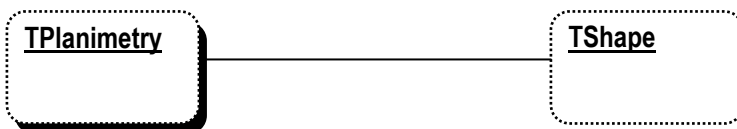


Рис. 8.11.1 Позначення утиліти і класу, її, що використовує.

Зв'язок класів з утилітою може бути відношенням використання, але не спадкування чи агрегація. У свою чергу, утиліта класу може використовуватися іншими класами, але не може від них успадковувати.

8.6.2. Позначення для агрегації

Як ми бачили в гл. 7, відношення агрегації породжує на множині класів ієрархію "ціле-частина". Клас-агрегат не обов'язково містить у собі фізично клас-частину. Агрегація може бути реалізована динамічно: клас-агрегат містить посилання на клас-частину.

Фізичний зміст відзначається на діаграмі прикрасою на кінці лінії, що позначає агрегацію; відсутність цієї прикраси означає, що рішення про фізичний зміст не визначено. Щоб відрізнити фізичну присутність об'єкта від посилання на нього, у діаграмах використовується зафарбований квадратик для позначення агрегації за значенням і порожній квадратик - для агрегації по посиланню.

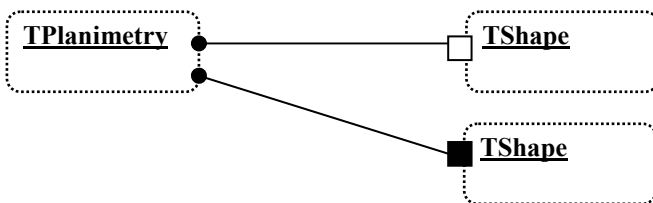


Рис. 8.12. Позначення агрегації за значенням і по посиланню.

8.6.3. Позначення для інваріантів

Як говорилося в п. 1.3. гл. 4, інваріант - це вираження деякої умови (співвідношення), що повинне зберігатися, якщо система знаходиться в стабільному стані. На діаграмі класів інваріанти, якщо це необхідно,

вказуються на відповідному значку у виді вираження, укладеного у фігурні дужки.

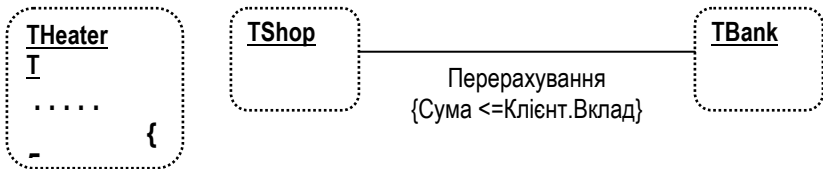


Рис. 8.13. Вказівка інваріантів на значках класів і зв'язків.

На рис. 8.13. показані позначення інваріантів. Клас THeater (нагрівач) повинний підтримувати значення параметра T (температури) у межах від 0 до 100.

Операція Перерахування (грошей з рахунка клієнта на рахунок магазину) повинна бути обмежена сумою, що не перевершує розмір внеску цього клієнта.

8.7. Специфікації

Специфікація - це неграфічна форма, використовувана для повного опису елемента діаграми: класу, асоціації, окремих операції чи цілої діаграми. Специфікації призначені для доповнення діаграм тією інформацією, яку необхідно мати для подальшої роботи, але яка на діаграмах не представлена.

Таким чином, безліч усіх синтаксичних і семантичних фактів, що знайшли своє відображення на діаграмі, повинна бути підмножиною фактів, описаних у специфікації моделі й узгоджуватися з ними.

Усі специфікації мають як мінімум наступні компоненти:

Ім'я : ідентифікатор
Визначення : текст

Визначення - це текст, що ідентифікує представлене елементом чи поняттям функцію і, що включається в словник проекту.

У кожній специфікації містяться мінімальні відомості. Незалежно від того, скільки граф містить у собі специфікація, їх не потрібно заповнювати усі відразу і зараз. Позначення повинні полегшувати розробку, а не створювати додаткові труднощі.

Специфікації класу. Кожен клас у моделі має рівно одну специфікацію, у якій містяться як мінімум наступні пункти:

Обов'язки : **текст**

Атрибути : **список атрибутів**

Операції : **список операцій**

Інваріанти : **список інваріантів**

Перелічені основні поняття можуть бути представлені в термінах обраної мови реалізації.

Як говорилося в главі 4, часте поводження деяких класів можна представити скінченим автоматом. У цьому випадку специфікація класу містить додатково поле виду

Автомат : **посилання на автомат**

Параметризовані й інстанційовані класи повинні включати наступний пункт:

Параметри : **список (формальних, фактичних) параметрів**

Специфікації операцій. Для всіх методів класів і вільних підпрограм наші специфікації включають наступні основні пункти:

Клас значення, що повертається : **посилання на клас**

Аргументи : **список формальних аргументів**

Ці графи можна заповнити обраною мовою реалізації. Відповідно до правил мови можна включити ще один пункт:

Кваліфікація : **текст**

Цей пункт може містити твердження про те, чи являється метод статичним чи динамічним (віртуальним).

8.8 Висновки

1. Тип відносин між класами означає характер взаємодії цих класів. Конкретний тип відносини визначає ієрархію класів відповідного типу.
2. П'ять типів ієрархій класів включають: асоціювання, спадкування, агрегація, використання, інстанціювання.

3. Відносини між класами системи описуються діаграмами класів і специфікаціями.
4. Діаграми класів, доповнені специфікаціями класів і інших сутностей системи, представляють логічну архітектуру розроблюваної системи.

9. МЕТОДОЛОГІЯ ОБ'ЄКТНО-ОРІЄНТОВАНОГО ПРОЕКТУВАННЯ

Розробка програмних комп'ютерних систем, як цілеспрямована діяльність, що носить творчий характер, не може бути цілком формалізована й автоматизована. Всі стратегічні і тактичні рішення проекту - результат розумової діяльності. Вони носять евристичний характер. Проектування як творчий процес, проте, не повинен бути хаотичним. Проектування завжди носить системний характер, воно спрямовано на досягнення цілей проекту як по його технічних характеристиках, так і за часом і витратами власне на процес проектування. Процес проектування комп'ютерної системи, отже, повинен бути структурованим і керованим. Тому проектування завжди здійснюється в рамках визначеної методології.

Під методологією проектування розуміють єдину систему основних концепцій, правил, методів і методик, що надають процесу проектування системний і цілеспрямований характер.

Методологія проектування, як правило, підтримується технологіями проектування - системою технічних засобів, що забезпечують реалізацію (втілення в життя) цієї методології.

Основою об'єктно-орієнтованої методології проектування програмних систем є система концепцій, описана в главі 2. Принципи, викладені там, представляють точку зору програміста на проектувану програмну систему. Ці принципи, як ми знаємо, визначають об'єктно-орієнтований підхід (об'єктно-орієнтований стиль).

Технологічну підтримку цього підходу забезпечують об'єктно-орієнтовані системи програмування і ті обчислювальні середовища, для яких ці системи реалізовані. В якості такої технології ми розглянули систему програмування Borland Pascal у середовищі MS DOS.

Вивчаючи основні поняття ОО програмування, ми власне кажучи розглядали техніку об'єктно-орієнтованого програмування. В центрі нашої уваги були методи рішень тактичних задач проектування: як визначити об'єкт, як реалізувати взаємодію об'єктів, як описати фрагмент структури об'єктів (підсистеми), як описати фрагмент ієрархії об'єктних типів (класів) системи. При цьому ми використовували поняття і методики, що не залежать від системи програмування, розглядаючи окремо технологічні проблеми реалізації. В цій главі ми розглянемо

коротко основні стратегічні аспекти процесу проектування програмної системи.

9.1. Життєвий цикл розробки програмної системи

Життєвий цикл програмної системи складається з наступних фаз (етапів):

- аналіз вимог до програмної системи;
- проектування системи;
- реалізація проекту системи;
- тестування і впровадження системи;
- супровід і розвиток системи.

Відзначимо, що ця класифікація відноситься скоріше до роду і логічної послідовності робіт, а не до часу їхнього виконання: на деякому часовому інтервалі розробки системи можуть виконуватися роботи, що відносяться до різних фаз її життєвого циклу.

Процес розробки завжди носить ітеративний, циклічний характер: якщо в системі виявлена помилка чи недолік, знайдена можливість її поліпшення чи з'явилася необхідність у її модифікації, роботи, пов'язані з цим, можуть відноситися до різних фаз.

Зокрема, розвиток системи починається зі змін вимог до неї. Потім (повторно) здійснюється аналіз, проектування, реалізація, тестування, впровадження і супровід нової версії.

9.2. Моделі системи

Моделлю системи називається формальний опис системи, у якому виділені основні об'єкти, що складають систему, і визначені відносини між цими об'єктами.

Моделі використовуються для перевірки працездатності програмної системи на ранніх етапах її розробки, спілкування з замовниками системи з метою уточнення вимог до системи, внесення змін у проект системи на будь-якому етапі її життєвого циклу.

Проект досить складної програмної системи неможливо представити однією моделлю так само, як неможливо отримати точне уявлення про деякий предмет, тільки глянувши на нього.

Г. Буч наглядно [2] описує об'єктну модель як систему спеціалізованих моделей, кожна з яких представляє проект в одному з ракурсів.

Динамічна модель

Статична модель

Логічна модель

Структура класів

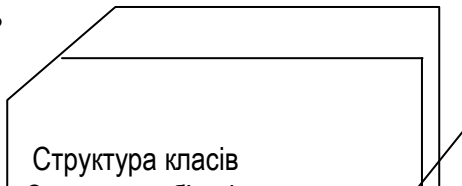


Рис. 9.1. Типи об'єктних моделей

Проектована програмна система, таким чином, представляється у виді спеціалізованих моделей двох типів:

- статичної моделі, що представляє статичні, структурні аспекти системи, в основному пов'язані з даними;
- динамічної моделі, що описує функціонування окремих частин системи в часі.

Кожен тип моделі описує проектовану систему або на логічному, або на фізичному рівні.

Ці типи моделей дозволяють отримати такі ракурси представлення системи, що взаємно доповнюють один одного й у сукупності дають достатньо повний формальний її опис в єдиній системі позначень.

Моделі, розроблені і налагоджені на першій фазі життєвого циклу системи, продовжують використовуватися на всіх наступних фазах її життєвого циклу, полегшуючи програмування системи, її налагодження і тестування, супровід і подальшу модифікацію.

9.3. Об'єктно-орієнтоване проектування

Об'єктно-орієнтоване проектування – це методологія проектування, що з'єднує в собі процес об'єктної декомпозиції і прийоми представлення логічної і фізичної, а також статичної і динамічної моделей проектованої системи.

Саме об'єктно-орієнтована декомпозиція відрізняє об'єктно-орієнтоване проектування від структурного: у першому випадку логічна

модель будується в термінах класів і об'єктів, у другому - у термінах структур даних і потоків управління.

9.4. Аналіз вимог і попереднє проектування системи

9.4.1. Об'єктно-орієнтований аналіз

Проектування прикладної програмної системи починається з аналізу вимог, яким вона повинна задовольняти. Мета аналізу – дати настільки повний опис функціонування проектованої системи, щоб на його підставі можна було скласти попередній проект. В процесі аналізу необхідно також визначити призначення й умови експлуатації системи. Аналіз повинен виявити тільки те, *що* робить система, а не то, як вона це робить.

При об'єктно-орієнтованому підході аналіз вимог до системи зводиться до розробки логічної об'єктної моделі цієї системи. Етап аналізу вимагає спільної діяльності замовників, користувачів і розроблювачів по складанню так званого *словника предметної області*.

Об'єктно-орієнтований аналіз – це методологія, при якій вимоги до системи сприймаються з погляду класів і об'єктів, виявлених у предметній області.

9.4.2. Побудова логічної об'єктної моделі. Метод об'єктної декомпозиції

Тепер у нас є всі необхідні поняття, щоб описати процес побудови логічної об'єктної моделі. Цей процес містить наступні етапи:

- визначення об'єктів і класів;
- підготовка словника предметної області;
- підготовка CRC-картотеки;
- визначення структури системи;
- побудова сценаріїв системи;
- визначення ієрархії класів спадкування.

9.4.3. Визначення об'єктів і класів

Об'єктно-орієнтована декомпозиція починається з аналізу зовнішніх вимог до проєктованої системи.

Аналіз зовнішніх вимог до проєктованої прикладної системи дозволяє визначити конкретні об'єкти і, можливо, деякі абстрактні класи об'єктів, зв'язані з предметною областю системи. Усі класи повинні бути абстраговані в цій прикладній області. Проблеми реалізації зараз не розглядаються.

Аналіз починається з виділення можливих класів з письмової постановки прикладної задачі - технічного завдання й іншої документації, наданої замовником.

Це дуже складний і відповідальний етап розробки, від нього багато в чому залежить подальша доля проєкту.

При визначенні можливих класів потрібно постаратися виділити максимально велику кількість класів, складаючи список імен кандидатів у класи. Зокрема, кожному іменнику, що зустрічається в попередній постановці задачі, може відповідати клас.

Далі список можливих слів – кандидатів у класи треба проаналізувати з метою виключення з нього зайвих слів - класів. Такими класами є:

- надлишкові класи: якщо два чи кілька слів позначають один об'єкт, варто зберегти тільки один клас;
- класи, що не мають прямого відношення до проблеми: для кожного імені можливого класу оцінюється, наскільки він необхідний у майбутній системі;
- класи, нечітко визначені (з погляду розглянутої проблеми);
- властивості: деяким словам будуть відповідати не класи, а атрибути чи властивості;
- операції: деяким іменникам більше відповідають не класи, а операції;
- ролі: деякі іменники визначають імена ролей в об'єктній моделі;
- конструкції, що відносяться до програмної чи апаратної реалізації.

Після аналізу текстів і виключення непотрібних імен буде отриманий попередній список класів, що складають проєктовану систему.

Визначення класу на цьому етапі містить у собі його ім'я, а також попередній протокол основних методів і властивостей, що описують його функціонування на логічному рівні. Результати аналізу повинні бути відображені в документації проекту.

9.4.4. Словник даних предметної області проекту

Текстові, графічні й інші попередні описи систем, як правило, містять дуже багато неточностей і надмірностей. Тому необхідно на самому початку проектування підготувати словник даних проекту, що містить точні визначення всіх об'єктів (класів), атрибутів, операцій, ролей і інших сутностей, розглянутих у проекті. Цей словник зобов'язані використовувати всі учасники проекту, включаючи представників замовника.

9.4.5. CRC – картотека проекту

Список класів і словник даних предметної області проекту є основою для складання іншої документації проектування. Зокрема, на його основі формується CRC-картотека проекту – пакет CRC-карток, кожна з яких містить описи тих атрибутів і методів, які виявлені на даний момент. Надалі CRC-картотека підтримується у відповідності з усіма змінами проекту.

Оскільки основними семантичними одиницями розроблювальної системи є об'єкти, їхні описи рекомендується документувати особливо ретельно. Методика документування описів об'єктів заснована на створенні, веденні і використанні спеціальної картотеки, що складає з CRC-карток. Абревіатура CRC розшифровується як C – Component (компонента), R – Responsibility (обов'язок), C – Collaborator (співробітники).

Власне CRC-картка являє собою листок паперу розміру приблизно 15 X 20 см. (формат A5), що (обов'язково олівцем) заповнюють інформацією про об'єкт. Спочатку до такої інформації відносять: ім'я класу, його обов'язки, імена класів – його співробітників.

Усі зміни в структурі класів, фіксуються (олівцем і гумкою) у відповідних CRC-картках. Таким чином, у процесі роботи над проектом зміст CRC-картки стає усе більш повним і формальним. В остаточному підсумку цей зміст перетворюється в інтерфейс класу, описаний мовою, близькою до мови реалізації.

Банк	
Обов'язки	Співробітники
<ul style="list-style-type: none"> ▪ Подтримує рахунок Покупця ▪ Подтримує рахунок Продавця ▪ Переводить гроші з рахунку Покупця на рахунок Продавця ▪ Платить податки ▪ Забирає собі проценти з угоди 	<ul style="list-style-type: none"> ▪ Покупець ▪ Продавець ▪ Рахунок ▪ Податкова служба

Рис. 9.2. Структура CRC-картки.

9.4.6. Визначення структури

СИСТЕМИ

Структура системи це, перш за все, її статична модель. Вона представлена сукупністю діаграм об'єктів. Логічна модель системи повинна містити опис всіх об'єктів і їх зв'язків “з точністю до програмної реалізації”.

9.4.7. Побудова сценаріїв

СИСТЕМИ

Попередній опис функціонування системи повинен бути представлений в документації замовника. Практика показує, однак, що докладний і несуперечливий опис її поведінки можна одержати тільки в процесі постійної і творчої взаємодії розроблювачів із представниками замовника і користувачами. Уточнення функціонування системи документально фіксується в попередньому описі у вигляді системи сценаріїв.

Формальний опис динамічної поведінки системи представляють в термінах діаграм взаємодій і діаграм станів і переходів. Кожна така діаграма взаємодій, як ми бачили, являє собою графічний образ одного сценарію. Перехід системи від одного сценарію до іншого описується в термінах діаграм станів і переходів. Кожен клас системи, що керує її поведінкою, може мати власну діаграму станів переходів. Вона показує, як об'єкт класу переходить з одного стану до іншого під впливом подій.

Сукупність діаграм взаємодій і діаграм станів і переходів утворює логічну динамічну модель проектованої системи.

9.4.8 Уточнення й удосконалення логічної моделі

Основний час на початковому етапі аналізу системи, як показує досвід, займають узгодження розроблювальної документації. Всі позиції словника даних предметної області, CRC-картотеки, наборів різного типу діаграм повинні бути вивірені, погоджені між собою, а сценарії - погоджені з замовником. Тільки після цього документацію системи можна використовувати для подальшого аналізу, побудови прототипу (макету) системи, поліпшення її характеристик і внесення змін.

Основними критеріями оцінки якості моделі системи на цьому етапі проектування є: точна відповідність один одному усіх видів документації, відповідність її прогнозованого поведіння попередньому опису. Важливу оцінку якості аналізу дає рівень складності об'єктів системи: об'єкт не повинен мати занадто багато чи занадто мало методів. Крім того, об'єкт не повинен бути пов'язаний з великою кількістю інших об'єктів.

9.5. Проектування системи

Метою проектування є створення архітектури системи для її реалізації і вироблення єдиних тактичних прийомів, що використовуються різними елементами системи.

9.5.1. Архітектура системи

Об'єктна модель, побудована методом об'єктно-орієнтованої декомпозиції як результат аналізу вимог, на цьому етапі повинна бути перетворена в ієрархію спадкування класів. У результаті цих перетворень визначається логічна архітектура системи.

Як показує досвід системного проектування взагалі і досвід проектування програмних систем зокрема, наявність ієрархії абстракцій є характерною ознакою будь-якої складної системи. Іншими словами, складні системи складаються з численних підсистем і елементарних частин (об'єктів), які можна класифікувати (типізувати) таким чином, що кількість різнотипних підсистем і частин буде відносно невеликою.

Складні системи можуть бути представлені у виді композицій деяких типів підсистем і елементарних об'єктів.

Приклади побудови ієрархій спадкування класів ми уже розглядали. Однак, приведений вище постулат системного аналізу є настільки важливим, що має потребу в додаткових ілюстраціях і коментарях.

Приклад. 9.1. Енергетичні системи.

Під енергетичною системою ми розуміємо систему, призначену для забезпечення потреб в енергії іншої системи. Вже в цьому визначенні присутнє абстрагування. Конкретними прикладами енергетичних систем є:

- електричні схеми;
- схеми теплопостачання;
- схеми газопостачання.

Крім того, це означення є *конкретизацією* більш загального поняття транспортної системи як системи транспортувань деяких ресурсів від постачальника споживачам.

Типи елементарних об'єктів енергетичної системи:

- джерело енергії;
- провідник енергії;
- споживач енергії.

Типи композиційних підсистем структурованої енергетичної системи:

- паралельне з'єднання;
- послідовне з'єднання.

Таким чином, три типи елементарних об'єктів і два типи композиції являють собою мову, у термінах якої описуються конкретні структуровані енергетичні системи довільної складності.

Побудова ієрархії класів з використанням методу спадкування концептуально відрізняє об'єктно-орієнтоване проектування від інших методологій проектування.

Ієрархії класів будуються на основі аналізу властивостей і поведінки об'єктів логічної моделі системи. Пошук загальних властивостей і ліній поводження об'єктів, звичайно, не може бути формалізований цілком. Наступні рекомендації можуть бути використані на практиці:

В основі ієрархії програмної системи лежить ієрархія абстракцій тієї реальної предметної області, для якої розробляється система. Цей факт обов'язково відбивається в документації, що представлена замовником.

Часто в попередньому описі вимог до системи вже містяться в явному вигляді описи деяких абстрактних класів.

Деякі класи проміжних рівнів ієрархії можуть бути отримані в результаті вивчення нормативно--довідкової документації й іншої літератури, що відноситься до даної предметної області.

Велику роль для правильної класифікації об'єктів грають консультації з представниками замовника й експертами в даній предметній області.

Класи-елементи ієрархії можуть бути визначені методами побудови спадкування в різних формах: програміст, вивчаючи групу класів з подібними властивостями і поведінням, виявляє їхні загальні риси, намічає батьківські класи і форми спадкування, використання яких дає бажаний результат. Цей метод є основним "внутрішнім методом" проектування. Він спирається тільки на логічну модель і не вимагає залучення ніякої іншої "зовнішньої" інформації.

Ієрархія спадкування повинна охоплювати не тільки примітивні об'єкти (об'єкти нижнього рівня структури об'єктної моделі), але і всі інші її структурні рівні.

Остання рекомендація має потребу в коментарі. Як ми вже знаємо, спадкування й агрегація технічно (на рівні реалізації) взаємозамінні. Одна методологічна крайність – використання тільки агрегації, друга – тільки спадкування. У першому варіанті проектування системи обмежується реалізацією мовою програмування її об'єктної моделі. В другому варіанті ігноруються проміжні структурні рівні об'єктної моделі: при реалізації агрегації використовується лише на рівні примітивних об'єктів. Практика показує, що висока якість проекту досягається збалансованим об'єднанням цих методик, коли спадкування використовується і для структурованих об'єктів.

Основним документом, що формується на цій стадії проектування, є діаграми класів. Відповідно до побудованих діаграм класів вносяться зміни й в іншу документацію проекту. У CRC-картотеку додаються нові картки, вносяться зміни в уже існуючі картки. Модифікуються діаграми взаємодій, об'єктів, станів і переходів.

9.5.2 Тактичне проектування

Тактичне проектування полягає в прийнятті рішень про використання загальних прийомів, що повинні бути покладені в основу

реалізації. До числа таких прийомів відносяться, наприклад, використання єдиних схем розробки, прийнятих для проекту як стандартні, використання єдиних середовищ розробки, єдиних прийомів управління пам'яттю, обробкою помилок, і т.і.

Кожне рішення про використання загального прийому повинне бути оформлене у вигляді документа, стандартного сценарію, або реалізовано на рівні прототипу готового інструмента і поширено серед учасників проекту.

На етапі проектування приймаються основні рішення про побудову фізичної моделі системи.

Формальним нормативним документом у результаті повинен бути план, у якому визначені розклад і розподіл завдань учасникам проекту.

9.6. Реалізація проекту системи

Процес реалізації системи полягає в побудові фізичної моделі і її реалізації у виді комп'ютерної програми. Незважаючи на те, що реалізація – один із заключних етапів життєвого циклу системи, він звичайно починається на перших етапах проектування у виді експериментів, спрямованих на перевірку правильності та ефективності прийнятих рішень.

На перших етапах життєвого циклу експериментальної реалізації (макетуванню або прототипуванню) підлягають окремі методи, об'єкти, основні сценарії взаємодії, невеликі фрагменти системи, у яких реалізовані типові рішення. Надалі розміри і складності реалізованих сценаріїв і фрагментів системи збільшуються. Тим самим функціонування системи ускладнюється. Така методика реалізації відповідає принципу “від простого до складного” чи “знизу вгору”. Об'єктно-орієнтоване програмування найбільше відповідає цій методиці: кожен об'єкт може бути реалізований окремо від інших частин системи – у вигляді модуля, що виконується. Перехід вгору, ускладнення, здійснюється “зборкою” декількох об'єктів і полягає в реалізації інтерфейсів – зв'язків між об'єктами і підсистемами.

Відзначимо, однак, що логічна архітектура системи, що виявляється в результаті проектування, може призвести до перевизначення практично всіх класів. Ієрархію спадкування природно будувати “зверху вниз”, “від загального до спеціального”. Це протиріччя розв'язується збалансованим сполученням двох підходів. З одного боку, проектування

системи повинне супроводжуватися експериментами, з іншого – роботи з макетування мають бути спрямовані на підтримку процесу проектування.

Нарешті, аналіз і проектування як теоретичні роботи повинні передувати реалізації вже не експериментального проекту (прототипу), а промислової системи.

10. ЛАБОРАТОРНИЙ ПРАКТИКУМ КУРСУ “ОБ’ЄКТНО-ОРІЄНТОВАНЕ ПРОГРАМУВАННЯ”

10.1. Теми завдань

10.1.1. Математика

Наступні системи є реалізацією схеми розробки “Хід рішення”. Крок рішення являє собою рішення однієї конкретної задачі зі списку т.зв. найпростіших задач.

1. **Геометрія: Задачі на побудову.** Комп'ютерне середовище призначене для демонстрації ходу рішення шкільних геометричних задач на побудову.
2. **Геометрія: Ієрархія геометричних фігур. Задачі на обчислення.** Система призначена для підтримки ходу рішення геометричних задач на обчислення.
3. **Аналітична геометрія: Задачі лінійної аналітичної геометрії на площині й у просторі.** Система призначена для підтримки ходу рішення задач лінійної аналітичної геометрії на площині й у просторі.
4. **Аналітична геометрія: Задачі квадратичної геометрії на площині й у просторі.** Система призначена для підтримки ходу рішення задач аналітичної геометрії, зв'язаних з фігурами і тілами 2-го порядку на площині й у просторі.

Наступні системи є реалізацією схеми розробки “Керування пристроями”. Кожен пристрій – розв’язувач задачі своїм конкретним методом.

1. **Алгебра: Ієрархія розширень числових систем.** Система являє собою ієрархію числових типів:

- Натуральні числа;
- Скінченні поля;
- Раціональні числа;
- Алгебраїчні числа;
- Дійсні числа;
- Комплексні числа;
- Кватерніони;
-

Основна задача системи: обчислити значення

арифметичного виразу.

2. **Чисельні методи: Інтегрування, рішення рівнянь.** Система призначена для рішення і демонстрації ходу рішення задач на інтегрування і розв’язування рівнянь одного невідомого стандартними чисельними методами.
3. **Чисельні методи: Інтерполяція, апроксимація.** Система призначена для рішення і демонстрації ходу рішення задач інтерполяції й апроксимації стандартними чисельними методами.
4. **Чисельні методи лінійної алгебри: обчислення визначника, рішення системи лінійних рівнянь, обертання матриці.** Система призначена для рішення і демонстрації ходу рішення стандартних задач лінійної алгебри стандартними чисельними методами.

- 5. Шкільна математика: Рівняння, системи рівнянь.** Система призначена для рішення рівнянь і систем рівнянь типів, які вивчаються у середній школі. Особливістю системи є те, що задачі повинні розв'язуватися в тих алгебраїчних структурах, в яких їх вирішують у школі (у раціональних числах, радикалах).
- 6. Шкільна математика: графіки функцій.** Система призначена для побудови графіків функцій, що задаються формулами і вивчаються у середній школі. Система повинна підтримувати відображення декількох графіків в одній системі координат.

10.1.2. Фізика.

1. Система фізичних приладів. Розділ Електрика.

Програмна система призначена для конструювання простих електричних схем (постійного струму) і проведення різних вимірювальних експериментів.

Список фізичних приладів: джерела електричної енергії; провідники; резистори; лампочки; вольтметри; амперметри; омметри.

Типи з'єднань приладів: послідовне, паралельне.

Вимірювані параметри: струм через прилад, спадання напруги на ділянці схеми, опір ділянки схеми, споживана потужність приладу.

Експеримент полягає в конструюванні схеми і вимірі різних параметрів приладів і ділянок цієї схеми.

2. Система фізичних приладів. Розділ Оптика.

Програмна система призначена для конструювання простих оптичних схем і проведення різних демонстраційних експериментів.

Список оптичних приладів: джерела світла (моно і поліхромні, різної форми), дзеркала (плоскі, опуклі, вогнуті), лінзи (призматичні, опуклі, вогнуті), екран.

З'єднання: послідовне.

Експеримент полягає в конструюванні оптичної схеми і демонстрації зображення джерела світла на екрані.

3. Моделі статистичної фізики.

Система є реалізацією схеми розробки “Модель поведінки популяції”. Вона призначена для моделювання броуновського руху молекул у суміші декількох газів. Акт взаємодії молекул – їх зіткнення. У результаті зіткнення змінюється вектор руху кожної молекули.

4. Модель сонячної системи.

Система реального часу повинна моделювати (з деяким ступенем точності) рух основних тіл сонячної системи: Сонця, великих планет, їх супутників. Користувач повинний одержати можливість вивчати кожен планетну систему окремо.

10.1.3. Ігри

1. “Гра двох осіб”. Кожна з наведених нижче тем є конкретною реалізацією схеми розробки програмної системи “Гра двох осіб”. Її сценарій потрібно уточнити як систему правил конкретної гри, візуалізацію початку, ходу і закінчення гри.

- Гра в доміно.
- Гра в шашки.
- Карткова гра.
- Гра “Морський бій”
- Гра “Хрестики-нулики”.

2. Лотерея. Програмна система Лотерея призначена для реалізації гри в лотерею. Лотерейна картка являє собою об’єкт, у якому зберігаються серія і номер.

Система повинна:

- Керувати процесом продажів лотерейних карток, створюючи в результаті своєї роботи список карток, що беруть участь у лотереї;
- Моделювати роботу лотерейного барабана, що випадковим чином вибирає виграшну картку для кожного призу зі списку призів, встановленого в лотереї;
- Здійснювати пошук виграшних лотерейних карток і виграних призів за списком карток, що беруть участь у лотереї;
- Реєструвати усі виграші, видані по пред’явлених картках.

3. Гра “Життя”. Відома математична гра “Життя” являє собою конкретну реалізацію схеми розробки “Модель поведінки популяції”.

- Життєвий простір популяції – потенційно нескінченний в усі сторони лист білого паперу в клітинку.
- Кожен член популяції в цій грі – клітка, пофарбована в чорний колір.
- У початковий момент часу популяція являє собою скінченну множину членів популяції, кожний з яких займає свою клітку життєвого простору.
- Кожен такт часу може призвести або до народження нового члена популяції, або до продовження життя, або до смерті члена популяції.
- Умови народження, продовження життя і смерті члена популяції визначаються його оточенням (взаємодією набору сусідніх кліток) у життєвому просторі. Один з конкретних варіантів умов еволюції популяції складається в наступному:

Оточенням клітки називаються 8 кліток, сусідніх даній клітці. Нехай в оточенні клітки, зайнятої даним членом популяції знаходиться M інших членів популяції $\{0 \leq M \leq 8\}$. Нехай, далі, a і b – два числа $\{0 \leq a \leq b < 8\}$. Тоді при

- $M \leq a$ член популяції гине (від самотності)
 - $M \leq b$ член популяції продовжує жити
 - $M > b$ член популяції гине (від тісноти)
 - Якщо дана клітка вільна і кількість M її сусідів задовольняє умові $a < M \leq b$, у даній клітці народжується новий член популяції.
- Можливі й інші варіанти умов еволюції популяції.

10.1.4. Ділові системи

1. **Покупець - банк – продавець.** Система є конкретною реалізацією схеми розробки програмної системи “Покупець - банк – продавець”. Її сценарій потрібно уточнити, описавши структури основних об’єктів і уточнивши форми документів, що супроводжують торгові операції.
2. **Замовлення таксі по телефону.** Система є конкретною реалізацією схеми розробки “Система масового обслуговування”. Її

сценарій потрібно уточнити, описавши структури основних об'єктів і уточнивши форми документів, що супроводжують обслуговування клієнтів.

3. **Товари - поштою.** Система призначена для керування продажами товарів поштою. Система є конкретною реалізацією схеми розробки "Система масового обслуговування". Її сценарій потрібно уточнити, описавши структури основних об'єктів і уточнивши форми документів, що супроводжують обслуговування клієнтів.
4. **Календар.** Система призначена для рішення наступних задач, зв'язаних із з обробкою календарних дат.

- Визначити поточну дату і день тижня.
- Визначити день тижня по даті, що вводиться з клавіатури.
- Визначити дату, що приходить на день <Поточна дата> + N днів
- Визначити дату, що приходить на день <Поточна дата> - N днів

Система може бути розширена додаванням нових функцій, зв'язаних з обробкою календарних дат і інформації, що супроводжує календарні дати.

Рис. 10.1. Зовнішній вигляд системи "Календар".

- 5. Перекладач.** Система призначена для перекладу слова з однієї мови на іншу. Система відповідає схемі розробки Перекладач. Система заснована на списку однобічних словників виду <мова, мова>. Кожен словник – упорядкований лексикографічно за алфавітом вихідної мови список статей. Кожна стаття має вид <слово вихідною мовою, список слів-значень перекладу>.

Система повинна передбачати:

- редагування словників;
- вибір і установку вихідної мови і мови перекладу;
- введення і редагування перекладного слова;
- переклад слова;
- вибір слова зі списку слів-значень перекладу.

The diagram illustrates the user interface of the 'Calendar' system. It consists of several input fields and a list of options:

- At the top, there are four input fields labeled 'Д', 'М', 'Р', and 'Д' (likely representing Day, Month, Year, and Day of the week).
- Below these, there are four more input fields labeled 'Д', 'М', 'Р', and 'Д'.
- On the left side, there is a vertical input field labeled 'К'.
- On the right side, there are four horizontal input fields.

- редагування списків елементів синтаксичних конструкцій, синтаксичних конструкцій речень і прикладів;

- ініціалізацію Тренажера, систему підказок.

10.2. Теми лабораторних робіт

Лабораторний практикум курсу націлений на вивчення методів і технологій об'єктно-орієнтованого проектування й одночасне вивчення об'єктно-орієнтованої мови програмування. Тому кожна його лабораторна робота складається з двох частин.

Завдання 1-ої частини зв'язані з придбанням навичок програмування в конкретній системі об'єктно-орієнтованого програмування. Вони носять характер окремих вправ на відповідну тему і виконуються кожним студентом самостійно.

Завдання 2-ї частини являють собою окремі етапи роботи над одним проектом, що повинен виконуватися студентами в складі робочої групи проекту. Оскільки робота над проектом носить ітераційний характер, звіт про виконання лабораторного практикуму представляється робочою групою наприкінці практикуму – у виді готового програмного виробу і його проектної і технічної документації.

Лабораторна робота № 1. Поняття об'єкта.

Мета: Одержати навички проектування і реалізації об'єктних типів.

Основні поняття: Об'єктний тип (клас), атрибут і властивість класу, операція і метод класу. Опис класів і використання об'єктів-екземплярів класів.

Зміст:

- Формулювання призначення програмної системи. Розробка сценарію – текстового опису програмної системи.
- Виконання вправ по реалізації статичних об'єктів.

Лабораторна робота № 2. Динамічні об'єкти.

Мета: Одержати навички проектування і реалізації динамічних об'єктних типів.

Основні поняття: Динамічний об'єктний тип (клас), конструктор і деструктор класу, віртуальні (динамічні) методи класу. Описи класів як модулів. Інкапсуляція.

Зміст:

- Аналіз сценарія програмної системи і складання словника предметної області. Розробка технічних вимог і умов експлуатації програмної системи.
- Виконання вправ по реалізації динамічних об'єктів.

Лабораторна робота № 3. Відношення між об'єктами.

Мета: Одержати навички проектування і реалізації взаємодії об'єктів.

Основні поняття: Відношення зв'язку між об'єктами. Клієнти, агенти і сервери. Області видимості об'єктів. Типи відносин. Відношення агрегування. Відношення залежності. Діаграми об'єктів, діаграми взаємодії, діаграми станів і переходів.

Зміст:

- Розподіл обов'язків і підсистем у групі розроблювачів проекту.
- Виконання вправ по реалізації зв'язків між об'єктами системи

Лабораторна робота № 4. Спадкування.

Мета: Одержати навички проектування і реалізації відношення спадкування класів.

Основні поняття: Відношення між об'єктними типами (класами). Спадкування. Батьківські і дочірні класи. Пошук методу. Перевизначення методів. Поліморфізм. Статичні і динамічні (віртуальні) методи.

Зміст:

- Аналіз підсистем і розробка діаграм об'єктів, взаємодій, станів і переходів.
- Виконання вправ по проектуванню ієрархії спадкування.

Лабораторна робота № 5. Форми спадкування.

Мета: Одержати навички проектування і реалізації спадкування в різних формах.

Основні поняття: Форми спадкування. Класи і підтипи. Множинне спадкування і його реалізації. Класи-домішки. Агрегація і спадкування.

Зміст:

- Аналіз підсистем і розробка діаграм об'єктів, взаємодій, станів і переходів.
- Виконання вправ по реалізації різних форм спадкування.

Лабораторна робота № 6. Методологія об'єктно-орієнтованого аналізу

Мета: Одержати представлення про методологію об'єктно-орієнтованого аналізу.

Основні поняття: Моделі функціонування програмної системи. Мета і задачі ОО аналізу. Формалізація словника предметної області. Дії і події. Формалізація сценаріїв. CRC-картки, діаграми об'єктів, взаємодії, станів і переходів.

Зміст:

- Аналіз системи, проектування структури класів системи.
- Розробка і налагодження ієрархії класів системи.

Лабораторна робота № 7. Методологія об'єктно-орієнтованого проектування.

Мета: Одержати представлення про методологію об'єктно-орієнтованого проектування.

Основні поняття: Проект програмної системи, її архітектура і загальні тактичні рішення. Мети і задачі ОО проектування. Проектування структури класів і ієрархії модулів. Планування процесу проектування.

Зміст:

- Проектування й оптимізація структури класів системи.
- Розробка і налагодження ієрархії класів системи.

Лабораторна робота № 8. Технології об'єктно-орієнтованого проектування.

Мета: Одержати представлення про сучасні технології об'єктно-орієнтованого проектування.

Основні поняття: Структура робочої групи проекту і розподіл обов'язків. Документація проекту програмної системи. Сучасні технології об'єктно-орієнтованого проектування. Середовища розробки і схеми розробки. Тестування і впровадження програмної системи.

Зміст:

- Проектування технічної документації системи.

- Тестування й оптимізація системи.

Лабораторна робота № 9. Експлуатація і супровід об'єктно-орієнтованих систем.

Мета: Одержати представлення про експлуатацію, супровід і еволюцію програмної системи.

Основні поняття: Життєвий цикл програмної системи. Технічні характеристики програмної системи. Супровід програмної системи, його мета і задачі.

Зміст:

- Проектування інструкції з експлуатації системи
- Тестування, впровадження і супровід системи.

Литература

1. Бадд Т. Объектно-ориентированное программирование в действии / пер. с англ. - СПб.: Питер, 1997.- 464 с. ил.
2. Буч Г. Объектно-ориентированный анализ и проектирование с примерами приложений на С++, 2-ое изд./пер. с англ.- М.: «Издательство Бином», СПб.: «Невский диалект», 1998 - 560 с., ил.
3. Шлеер С., Мэллор С.. Объектно-ориентированный анализ: моделирование мира в состояниях: пер. с англ.-Киев: Диалектика, 1993. – 240 с. ил.
4. Марченко А.И, Марченко Л. М. Программирование в среде Turbo Pascal 7.0. К.:Диалектика, 1999, 430 с. ил.
5. Львов М.С., Співаковський О.В. Основи алгоритмізації та програмування. Навч. посібник. – Херсон: Айлант. 2000. – 214 с.