Модуль 1. Організація процедуро-орієнтованих програм

Тема 1. Теоретичні та методологічні засади організації програм і даних

Введення в .NET Framework-технологію програмування

1.1. Основні концепції і термінологія

Базова технологія безпосередньо пов'язана з мовою C#, має назву .NET (вимовляється як "дот нет").

.NET – це загальний термін для багатьох служб, які надаються й використовуються під час створення та виконання програми на C#.

Особливості інфраструктури .NET-платформи

C# повністю залежить від .NET і тому походження багатьох концепцій C# уходить своїми коренями в. NET.

Нижче перераховані особливості інфраструктури .NET-платформи:

.NET надає засоби для виконання інструкцій, що містяться в програмі, написаної на C#. Ця частина .NET називається середовищем виконання (**execution engine**).

.NET допомагає реалізувати так зване середовище, безпечне до невідповідності типів даних (**type safe environment**).

.NET звільняє програміста від стомлюючого процесу і такого, що нерідко веде до помилок при керуванні комп'ютерною пам'яттю, яка використовується програмою.

До складу .NET-платформи входить **бібліотека**, котра містить масу готових програмних компонентів, які можна використовувати у власних програмах. Вона заощаджує чимало часу, тому що програміст може скористатися готовими фрагментами коду. Фактично він повторно використовує код, створений та ретельно перевірений професійними програмістами Microsoft.

У .NET спрощена підготовка програми до використання (**розгортання**).

.NET забезпечує перехресну взаємодію програм, написаних на різних мовах. Будь-яка мова, підтримувана .NET, може взаємодіяти з іншими мовами цієї платформи.

У даний момент на платформу .NET перенесено близько 20 мов. Оскільки для виконання коду, написаного на будь-якій мові, що підтримується платформою .NET, використовується те саме середовище виконання, його часто називають єдиним середовищем виконання (**Common Language Runtime, CLR**).

Програма, при створенні якої була передбачена можливість повторного використання, називається компонентом (**програмним компонентом**).

1.1.1. Мови програмування та компілятори

Програмувати на перших комп'ютерах, виготовлених у сорокових роках минулого сторіччя, було дуже непросто. Для цього використовувалася **машинна мова**, що складалася з послідовностей бітів, прямо керуючих простими діями процесора. Програмування на рівні машинної мови – заняття надзвичайно трудомістке та стомлююче.

Незабаром програмісти почали шукати альтернативи машинній мові, більш близькі людській мові, щоб підвищити продуктивність своєї праці.

Першим результатом пошуків стали так звані **асемблери** – мови, в яких використовувалися більш зрозумілі людині команди, наприклад move, getint або putint. Але, незважаючи на те, що асемблери були трохи простіші для читання й розуміння, їх поєднувала з машинним кодом одна важлива загальна риса: розроблювач при написанні програми повинен був мислити в термінах низькорівневих операцій процесора і пам'яті.

Однак еволюція комп'ютерів і зростаючі потреби у все більш складних програмах привели до появи повністю **машинно-незалежних мов** програмування. Першою з них стала FORTRAN, створена в середині п'ятидесятих років. Незабаром з'явилися інші мови високого рівня. Сьогодні їхня кількість за деякими оцінками перевищує дві тисячі.

Одним із останніх доповнень у родині мов високого рівня став C#. Однак як би високо ми не відходили від базових інструкцій процесора,

нам, як і раніше, потрібний машинний код, який апаратне забезпечення комп'ютера може розуміти – а виходить, й виконувати.

Традиційно перетворення вихідного коду, написаного мовою високого рівня, в машинний код здійснювали системні програми, які називаються *компіляторами*.

На рис. 1.1. наведена ілюстрація того, як вихідний код типової мови високого рівня перетворюється у програму, що виконується.

Написаний текст, який містить інструкції мови високого рівня, називається *вихідним кодом*.

У випадку C# цей вихідний код зберігається в файлі з розширенням .**cs**.

Результатом компіляції стає *програма, що виконується*, яка складається з інструкцій машинної мови.

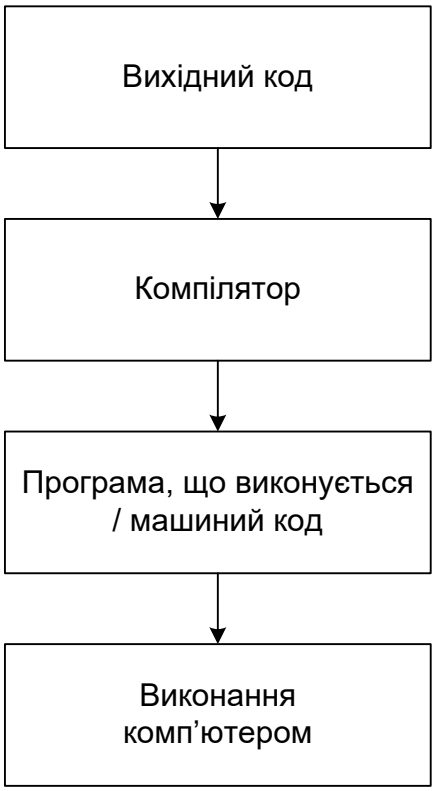

Рис. 1.1. **Традиційний процес компіляції**

1.1.2. Компіляція в .NET вихідного коду C#

Традиційний процес компіляції вихідного коду, написаного мовою програмування високого рівня, в програму, що виконується, мав кілька недоліків. Найбільш істотними з них є наступні.

Недолік 1. Для конкретної апаратної платформи, що характеризується типом процесора й т. д., потрібен свій компілятор, оскільки в кожної з них – своя машинна мова. Відповідно, якщо потрібно виконувати програму, написану на FORTRAN, на чотирьох комп'ютерах із процесорами різних виробників, буде потрібно чотири різних компілятори FORTRAN. Більш того, щораз, коли виробник апаратного забезпечення випускає нове покоління своїх процесорів, в компілятор доводиться вносити зміни та доповнення.

Недолік 2. У більшості програмістів є улюблена мова програмування, якой вони віддають перевагу перед усіма іншими. Можливо, кращим рішенням було б дозволити кожному члену команди писати на своїй улюбленій мові, але це нелегко, якщо додержуватися процесу компіляції, розглянутому на рис. 1.1.

Різні мови на машинному рівні реалізують ту саму функціональність різними способами – частково це залежить від особливостей компіляторів. В свою чергу, це унеможливлює взаємодію різних мов між собою.

Цю проблему були покликані вирішити так звані **компонентні системи** (такі, як **CORBA** і **COM**). Вони оговорювали стандарти взаємодії між різними частинами програм.

Програміст А писав компонент X, скажімо, мовою Visual Basic, і цей компонент міг взаємодіяти з компонентом Y, що був розроблений програмістом В на C++.

Компонентні системи мали комерційний успіх. Однак їхнє поширення викликало до життя інші проблеми, однією з яких стала неможливість забезпечити взаємодію "зовнішнього" компонента з іншими частинами програми на такому ж рівні, якби всі частини програми були написані на одній мові.

В C# і .NET реалізовані рішення описаних вище двох проблем. Розглянемо всі складові процесу компіляції програми в .NET (рис. 1.2).

Насамперед, варто звернути увагу на появу на рис. 1.2 ще двох мов – C++ і Visual Basic. І поза залежністю від того, на якій мові (з підтримуваних .NET) написана програма, це ніяк не впливає на процес її компіляції в .NET.

Після того як написаний вихідний код, його потрібно відкомпілювати в машинний код. Однак спочатку він компілюється в іншу мову, що називається **Microsoft Intermediate Language** (**MSIL**). Більш того, всі компілятори, орієнтовані на .NET-платформу, повинні генерувати на виході код даної проміжної мови MSIL.

Як ясно з назви, MSIL є проміжною ланкою між мовами високого рівня (вихідний код) і машинними мовами (називаними також **природним кодом**).

Код MSIL можна швидко та ефективно транслювати в машинну мову за допомогою **JIT-компілятора** (**Just in Time-Compiler**).

Код, що генерується JIT-компілятором, нічим не відрізняється від машинного коду, що генерується звичайним компілятором, однак JIT-компілятор використовує трохи іншу стратегію. Замість того, щоб інтенсивно використовуючи пам'ять і, затрачаючи значний час, перетворити в машинний код відразу весь код MSIL, він компілює в машинний код лише ті частини додатка, які реально необхідні в даний момент. В результаті код компілюється "на ходу", безпосередньо перед виконанням, і JIT-компілятор не витрачає час на компіляцію MSIL-коду, що не використовується.

У чому ж переваги архітектури .NET?

1. Ввівши MSIL (рис.1.2) між мовою високого рівня та машинною мовою, ми фактично відокремили ці мови одну від одної.

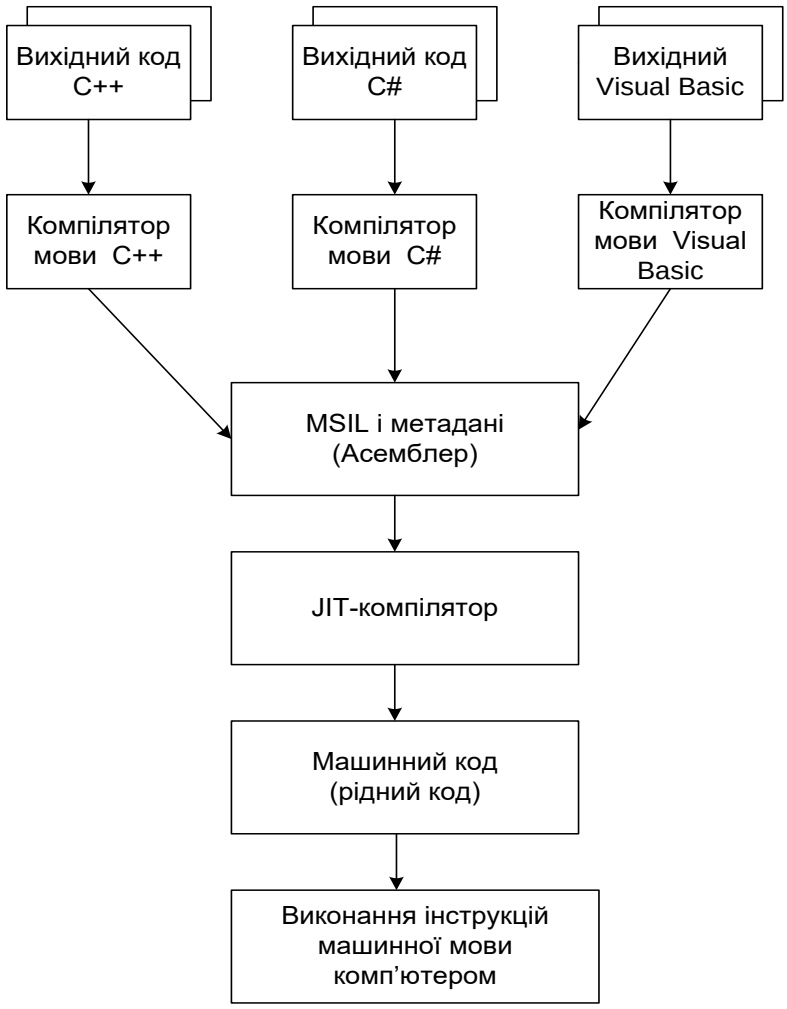

Рис. 1.2. **Процес компіляції в .NET**

Код MSIL залишається незмінним, на якій би апаратній платформі він не використовувався. Єдиним машинно-залежним елементом є JIT-компілятор, і при зміні обладнання лише він має потребу в модифікації.

На кожному комп'ютері застосовується свій JIT-компілятор, що перетворює код MSIL в машинний код, сумісний з даною конкретною конфігурацією. В результаті все, що потрібно, – це компілювати код, написаний мовою високого рівня, в універсальний код, мова якого залишається незмінною. Це і є рішенням згаданої вище першої проблеми.

Розглянемо блок «MSIL і метадані», наведений на рис. 1.2. Термін *"метадані"* можна перевести як "дані про дані". Метадані генеруються компілятором мови високого рівня і містять докладний опис елементів вихідного коду. Опис цей настільки докладний, що вихідний код інших мов зможе використовувати даний код так, якби він був написаний на тій же мові. Тепер програмісти, що пишуть на C++, C# и Basic, реально

зможуть працювати в рамках одного проекту і, отже, у такий спосіб долається недолік 2.

Важливо знати про існування MSIL, але в повсякденному програмуванні стикатися з ним прямо не доводиться. Звичайно застосовуються дві команди – одна компіляції програми в MSIL-код і метадані, а інша – для виконання програми (при цьому буде викликатися JIT-компілятор). Фактично виконання програми – це виконання кінцевого результату роботи компіляторів. MSIL у цьому процесі залишається "невидимим" для користувача.

1.2. Розробка програмного забезпечення

Процес розробки програмного забезпечення (ПЗ) – це послідовність дій, кінцевим продуктом якої є комп'ютерна програма. За минулі 20 років були виділені чітко визначені етапи процесу розробки ПЗ. В даному розділі розглядаються лише найбільш важливі з них:

1. Створення специфікації програми**.** Визначення вимог до програми.

2. Проектування програми. Розробка концепції, що дозволяє втілити вимоги специфікації в працюючій програмі. Вона визначає шляхи реалізації функціональності, описаної в специфікації, засобами мови програмування високого рівня, наприклад C#.

3. Написання програми. Розробка та написання вихідного коду програми.

4. Тестування й налагодження програми. Необхідно перевірити, чи відповідає програма вимогам, визначеним у специфікації.

Ці етапи повинні виконуватися в зазначеній тут послідовності. Однак іноді виникають ситуації, коли для продовження роботи над програмою необхідне повернення до попереднього етапу.

Розглянемо кожний з етапів більш докладно.

Створення специфікації програми

Навіщо визначати вимоги до програми? По-перше, якщо чітко невідомо, що програма повинна робити, важко навіть почати думати про те, як її реалізувати. Визначення мети – перший крок до її досягнення.

Часто рішення сховане в самій специфікації. (Відомо, що правильно сформульована проблема вже містить своє рішення.)

Ось приклад дуже простої специфікації програми: "Програма повинна вміти обчислювати середнє значення двох чисел".

Проектування програми

Проектування програми може включати множину різних дій залежно від розміру й характеру конкретного проекту. Великий проект може включати декілька стадій, описаних нижче:

а). розділення всього проекту на різні функціональні підсистеми. Прикладами підсистем можуть служити користувальницькі графічні інтерфейси, генератори звітів, інтерфейси до баз даних. Слід зазначити, що проекти, розглянуті далі, занадто малі, і їх не варто розбивати на підсистеми. Таким чином, наведені далі програми можна потенційно розглядати як частину функціональної підсистеми великого проекту;

б). розбивка кожної підсистеми на модулі. Термін "модуль" дуже гнучкий і приймає різні значення в різних контекстах. Тут *модуль* – це сукупність даних і функцій, які можуть працювати із цими даними. Функції, реалізовані в модулі, дозволяють йому у взаємодії з іншими модулями даної підсистеми виконувати вимоги, адресовані до неї.

Функція звичайно має одну дуже вузьку мету. Вона складається з набору конкретних інструкцій, що виконуються одна за другою. Прикладами функцій можуть бути "Знайти квадратний корінь числа" або "Знайти найбільше значення в списку". В різних високорівневих мовах цю концепцію називають по-різному: процедури, функції, підпрограми. В C# функції називаються *методами;*

в). визначення даних і методів у кожному модулі. Служби, пропоновані модулями, діляться на зручні частини, досить компактні, щоб їх можна було реалізувати в рамках одного методу. Кожній частині модуля призначається свій метод, і, відповідно, кожний метод одержує певну функціональність. І, нарешті, творці програми визначають дані, що можуть бути представлені цим модулем;

г). внутрішнє проектування методів. Проектуванням на цьому рівні звичайно займається програміст, що розробляє конкретний метод. На даному етапі розробляються алгоритми виконання дуже вузьких, конкретних задач і пишуться перші оператори мовою високого рівня, – наприклад, C#.

Написання програми

Дана частина процесу створення ПЗ обов'язково присутня в проектах будь-якого масштабу.

Тестування й налагодження програми

Програма піддається різним тестам (як під час написання, так і після нього). Тести необхідні, щоб переконатися, що програма виконує саме те, що повинна виконувати. **Тестування**, зокрема, дозволяє виявити (і виправити) помилки, допущені в процесі проектування та написання програми.

Процес пошуку й усунення помилок називається **налагодженням** (англійський термін – debugging від bug).

Невеликі, неформальні проекти (наприклад, з лабораторного практикуму по даній дисципліні) часто включають лише рівні 3 і 4. Програміст може виконати їх, сидячи перед комп'ютером. Стадія проектування програми реалізується або в голові у програміста, або на листку паперу у формі декількох діаграм, або у вигляді пошуку декількох стандартних алгоритмів у підручнику.

1.3. Процедурно-орієнтоване програмування

Одним із традиційних підходів до проектування комп'ютерних програм був процедурно-орієнтований стиль. В ньому найвищий пріоритет належить діям, що виконує елемент програми, у той час як дані, з якими працює програма, залишаються ніби на другому плані.

Типова *процедурно-орієнтована програма* – це послідовність інструкцій, виконуваних одна за одною. В такій програмі, як правило, є численні точки розгалуження, в яких вибирається лише один із декількох можливих напрямків виконання, залежно від умов в програмі. Більша частина інструкцій маніпулює даними.

У старомодних, процедурно-орієнтованих програмах доступ до даних був можливий з будь-якої частини програми (як це показано на рис. 1.3.), а операції над ними – за допомогою будь-якої інструкції.

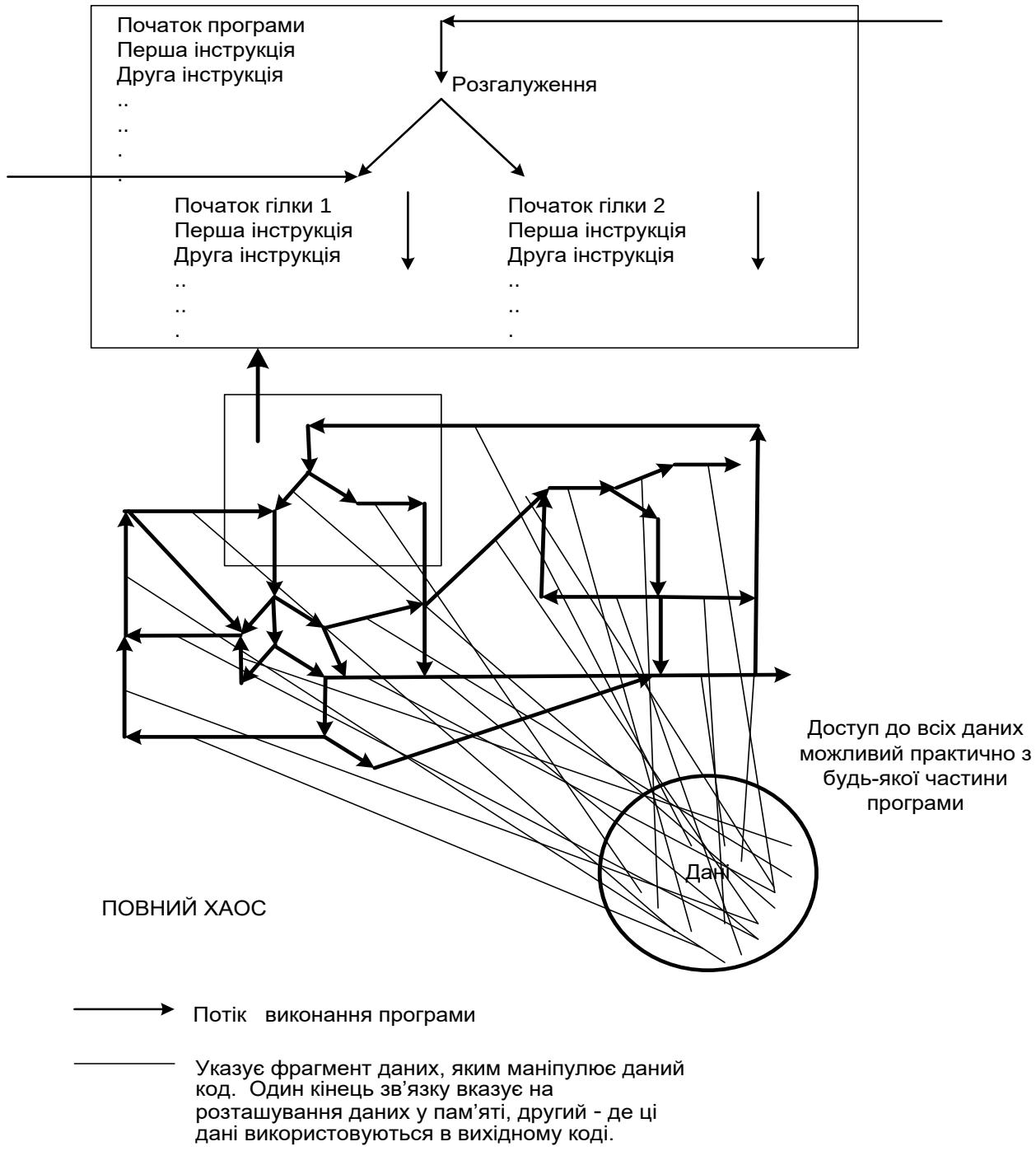

Рис. 1.3. **Структура процедурно-орієнтованої програми**

Слід зазначити, що багато з великих фрагментів вихідного коду, написаних на таких об'єктно-орієнтованих мовах, як C#, створено з використанням примітивних процедурно-орієнтованих мовних конструкцій.

Цей підхід гарний для дуже малих проектів, але при створенні великих та складних програм розроблювачеві доводиться відслідковувати всі можливі розгалуження в коді програми, оскільки всі

частини її більш-менш взаємозв’язані. Гірше того, різні фрагменти коду програми можуть одержати доступ до того ж самого значення даних – причому вони можуть не тільки читати, але й змінювати його.

При розробці однієї частини програми можна й не знати про те, що дані, з якими вона працює, можуть бути змінені іншими частинами програми.

Ситуація дуже швидко наближається до повного хаосу, коли над проектом спільно працюють декілька програмістів.

1.4. Об’єктно-орієнтоване програмування

Як же подолати розглянуту вище проблему процедурно-орієнтованого стилю програмування? Типовий людський підхід до подолання складної проблеми – розбити її на більш прості частини.

Спробуємо розбити попередній приклад на чотири менших, самостійних фрагменти. Результат показаний на рис. 1.4.

Тут весь набір інструкцій розділений на чотири окремих модулі. В об’єктно-орієнтованому світі такі модулі називаються **об'єктами**. Дані також розділені на чотири частини, так що кожний об'єкт містить лише ті, з якими він працює. Тепер при виконанні програми ці чотири об'єкти взаємодіють, **пересилаючи один одному повідомлення,** активізуючи інструкції та обмінюючись даними. Це не тільки значно знизило складність програми, але й дозволило створити чотири самодостатніх модулі, кожний з яких може бути "витягнутий" із програми й "вставлений" у неї знову.

Тепер не так уже складно добитися, щоб модулі створювалися й підтримувалися різними програмістами. Як можна бачити з рисунка, об'єкти використовуються для об'єднання даних з методами (інструкціями), що оперують над цими даними.

В об’єктно-орієнтованому світі дані і процедури мають рівне значення.

Але звідки ж знати, які об'єкти повинна містити програма? Які дані потрібні? Які інструкції необхідні?

Щоб відповісти на ці питання, насамперед необхідно з'ясувати, що таке об'єкт.

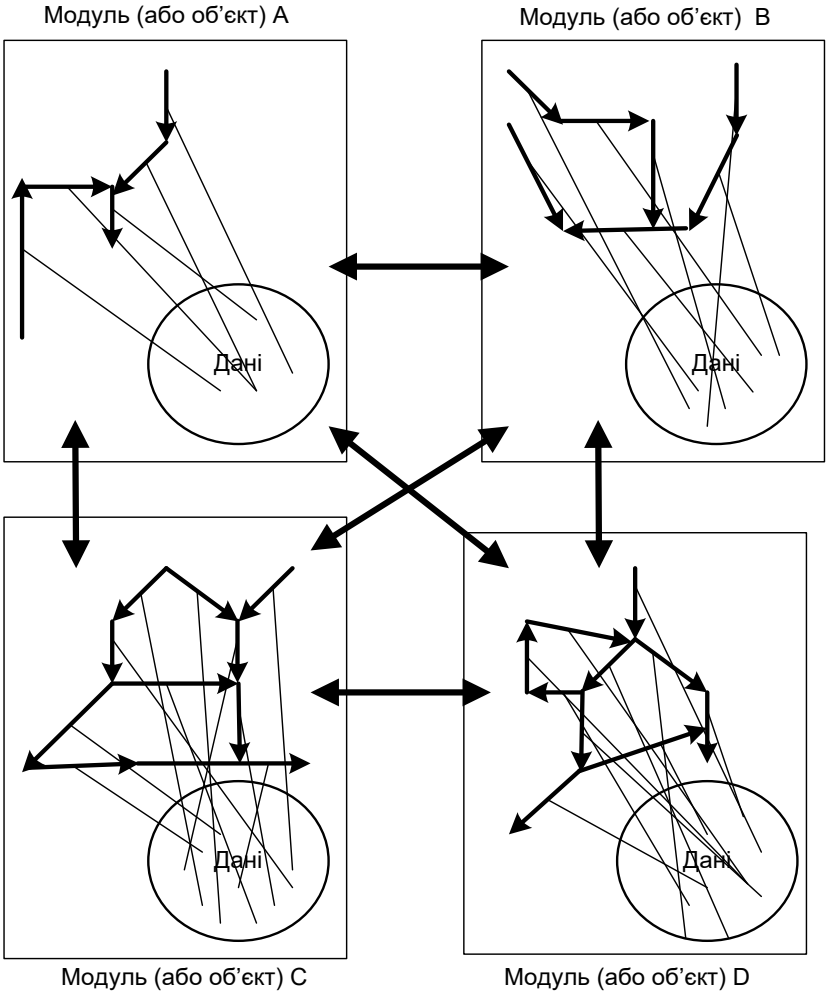

Рис. 1.4. **Розподіл процедурно-орієнтованої програми на самодостатні модулі**

Поняття об'єкта

В повсякденному житті нас оточують об'єкти: книги, будинки, машини, собаки, люди ... . Часто об'єкти здатні брати участь у певних діях. Автомобіль, наприклад, здатний "відкривати двері", "закривати двері", "запускати двигун", "рухатися вперед", "набирати швидкість", "розвертатися" та "гальмувати".

Кожний об'єкт взаємодіє зі своїм оточенням і впливає на інші об'єкти. Прикладами подібних взаємодій можуть бути об'єкт "людина", що підходить до об'єкта "автомобіль" та відкриває його двері, або об'єкт "автомобіль", який містить об'єкт "людина" та транспортує його із точки А в точку Б. При написанні програми за участю автомобілів і людей з використанням процедурно-орієнтованої методології вся увага буде приділена діям ("відкрити двері", "закрити двері" і т. д.). З іншого боку, при використанні об’єктно-орієнтованого програмування (ООП) програма буде будуватися навколо об'єктів "автомобіль" і "людина".

Таким чином, ООП – це методологія, в якій при моделюванні конкретних ситуацій використовуються об'єкти. **Комп'ютерне моделювання** намагається імітувати процеси, що відбуваються в реально існуючій або теоретичній системі. Збираючи й аналізуючи дані, отримані в цій штучній системі, можна одержати цінні відомості про внутрішні особливості системи реальної. Для успішного моделювання необхідно розробити модель та реалізувати її в комп'ютері. Часто модель є спрощенням реальної системи, однак вона цілком відповідає внутрішнім процесам і станам реальної системи, що модулюються.

Щоб заповнити пролом між реальним світом і моделюванням в комп'ютері, необхідно, насамперед, ідентифікувати об'єкти, що беруть участь у даному процесі. Звичайно, це один із перших кроків при розробці програми в ООП, і він прямо відповідає кроку б) (розбивка кожної підсистеми на модулі) процесу розробки програми, що обговорювався раніше.

Як приклад розглянемо ліфтову систему будівлі в дії. Які об'єкти в цьому беруть участь? Непоганий підхід: звертати увагу на те, які іменники використовуються при описі системи, оскільки іменники прямо відповідають об'єктам. Нижче приводиться опис системи, в якому реальні об'єкти виділені напівжирним шрифтом.

Опис системи: кілька **ліфтів** розташовано в **будівлі** з десятьма **поверхами**. Кожний ліфт має доступ до всіх десяти поверхів. Коли **людина** бажає викликати ліфт, їй необхідно нажати **кнопку** на поверсі, де вона перебуває в даний момент, і т. д.

Наступний етап відноситься до кроку в) (визначення даних і методів в кожному модулі) процесу розробки ПЗ. Він полягає у визначенні атрибутів (даних) і функцій (методів) кожного модуля (об'єкта).

Наприклад, об'єкт "Ліфт" може мати наступні атрибути: "максимальна швидкість", "поточне місце розташування", "поточна швидкість", "максимальне число людей, що може вмістити ліфт", і т. д. Функції об'єкта можуть бути наступними: "підніматися", "опускатися", "зупинитися" та "відкрити двері". Визначивши правильні атрибути і функції кожного об'єкта в реальному світі, їх можна представити в комп'ютерній моделі. Тут потрібно розширити запас термінології ООП ще декількома важливими термінами.

Кожна частина програми на C#, що представляє об’єкт реального світу, відповідно називається **об’єктом**.

Істотні атрибути об'єкта реального світу повинні бути представлені у відповідному об'єкті програми на *C#*. Ці атрибути називаються **змінними екземпляра** й відбивають поточний стан об'єкта. Змінні екземпляра еквівалентні даним на рис. 1.4.

Поведінка об'єкта реального світу представлена в об'єкті програми C# у формі методів. Кожний метод містить інструкції. Методи об'єкта виконують дії зі змінними екземпляра цього ж об'єкта. Методи еквівалентні потокам виконання на рис. 1.4.

Поняття класу

Ще один важливий термін ООП — клас. **Клас** визначає загальні риси (змінні екземпляра і методи) групи подібних один одному об'єктів. Отже, всі об'єкти одного класу мають ті ж змінні екземпляра і методи. Які змінні екземпляра і методи включати в створюваний клас, програміст вибирає сам: все залежить від потреб створюваної програми.

Як приклад класу, розглянемо автомобіль із концептуальної точки зору. В реальному житті люди використовують конкретні автомобілі. Прикладами можуть бути блакитний Volvo, що стоїть на стоянці, максимальна швидкість якого 100 миль на годину, або чорний BMW з максимальною швидкістю 150 миль на годину. Обидва ці реально існуючі відчутні автомобілі можна вважати об'єктами.

Щоб забезпечи опис конкретного автомобілю як об'єкта програми на C#, програміст приймає рішення про включення чотирьох змінних екземпляра в клас **Автомобіль: Марка, Поточне місце розташування, Максимальна швидкість** і **Поточна швидкість**. Слід зазначити, що атрибут **Колір** (Color) не включений у клас, оскільки програміст вважав його непотрібним для даної програми.

Далі програміст вирішує ввести до складу класу методи **Відкрити двері, Закрити двері, Рухатися вперед, Рухатися назад, Прискорюватися, Гальмувати** та **Повертати.**

Кожний екземпляр класу **Автомобіль** можна розглядати як порожню коробку, що має бути наповнена вмістом у конкретному об'єкті.

Вміст кожної змінної класу **Автомобіль** може в різних об'єктах мати як однакові, так і різні значення, але кожний метод, визначений класом **Автомобіль**, ідентичний у всіх об'єктах **Автомобіль**.

Концепції об'єкта і класу представлені на настільки ранній стадії вивчення C#, оскільки будь-яка, навіть найпростіша, C#-програма є об’єктно-орієнтованою.

1.5. Повторне використання програмного забезпечення

Повторне використання програмного коду закладене в основу програмування в .NET і C#. На ранній стадії написання кожна програма представляється зовсім не залежним проектом, розроблювальним з нуля. Однак це не зовсім раціональний спосіб створення програм, і сьогодні більшість із них створюється за іншою методологією, основу якої складає повторне використання заздалегідь розроблених і налагоджених програм або їхніх частин.

Високий ступінь повторного використання коду означає, що при створенні програми розроблювач написав тільки частину коду, а інша частина – це вставлені в програму компоненти, написані та налагоджені досвідченими програмістами. Навіть в найпростіших програмах на C# варто завжди дотримуватися концепції повторного використання коду.

Одна з найбільш привабливих сторін об'єктно-орієнтованих мов (в тому числі C#) – це ретельно продумана підтримки повторного використання коду. Зокрема, **class** виявився дуже гарним для повторного використання коду. Елементом повторного використання коду в .NET є також складання (**assembly**). Будь-яка програма в .NET і C# складається з одного або більше складань.

Складання – це логічний пакет, що містить свій опис. Він складається з коду MSIL, метаданих і, якщо необхідно, ресурсів, наприклад зображень.

Складанням є будь-яка програма, написана для .NET, будь то компонент для повторного використання або самодостатня программа, що виконується.

Складання можна розглядати з двох точок зору. З погляду розроблювачів складання, що розглядають його зсередини, на рівні вихідного коду і з погляду користувачів складання, що розглядають його зовні, коли потрібно підібрати підходящий компонент для повторного використання в тім або іншім проекті.

1.6. Бібліотека класів .NET Framework

Протягом багатьох років у різних типах програм постійно використовувалися ті самі функції та алгоритми. Прикладом може служити сортування списку, спеціалізовані інженерні розрахунки,

математичні обчислення й т. д. Цей факт усвідомило багато компаній і розроблювачів ПЗ, що почали створювати бібліотеки класів, в яких містилися подібні широко використовувані функції. Зараз важко знайти додаток, в якому б не використовувалися частини з повторно використовуваних бібліотек класів.

Компанія Microsoft включила до складу середовища .NET бібліотеку класів. Вона називається бібліотекою класів .NET Framework, або бібліотекою базових класів (**Base Class Library, BCL**). Бібліотека містить сотні класів і надає доступ до множини функцій.

В числі основних механізмів, використовуваних в BCL, – принципи ООП, технологія складань і пов'язані з нею концепції. В результаті BCL проста у використанні і забезпечує широкі можливості повторного використання коду.

У мові C# відсутня власна бібліотека класів – він повністю покладається на BCL і тісно інтегрований з нею. Відповідно, програму, написану на C#, неможливо запустити без BCL та середовища виконання .NET.

В основі всіх класів, написаних в C#, лежить один конкретный класс BCL, і багато конструкцій C# є лише представленням тих класів і їхніх функцій, що містяться в BCL. Наявність BCL значно спрощує доступ до служб операційної системи. Замість того, щоб використовувати малозрозумілі команди і складні вираження, служби ОС стають доступними у формі, набагато більш дружній користувачеві. Прикладом може служити надавана BCL підтримка віконних графічних інтерфейсів користувача.

1.7. Можливості мови C#

Як уже відмічалося вище, в теперішній час існує множина мов програмування. І, незважаючи на те, що будь-яку програму можна написати на практично будь-якій мові, багато мов було пристосовано для рішення більш-менш конкретного кола проблем.

C# и .NET працюють винятково в операційних системах Microsoft, але при цьому вони надають програмістові в рамках цих операційних систем дуже потужні засоби. Важливо, що C# – багатоцільова мова, і в рамках платформи Windows він може використовуватися для створення найрізноманітніших програм.

Нижче наведено короткий список лише декількох категорій програм, при створенні яких комбінація достоїнств C# і .NET забезпечить високу ефективність праці програміста C#.

Консольні додатки

У консольних додатках для взаємодії з користувачем застосовується одне просте вікно. В них немає вражаючих графічних інтерфейсів і анімації; інформація виводиться в символьному режимі. При цьому програми виходять більш простими, тоді як для насичених графікою додатків складність – звичайна справа.

Незважаючи на те, що комерційних програм з консольним інтерфейсом не так багато, створення їх – чудовий спосіб вивчення мистецтва програмування. Він дозволяє зосередитися на мовних концепціях і допомагає швидко набути розуміння мови, необхідне для створення більш складних графічних програм і, що, мабуть, найважливіше для спеціальностей напрямку 6.092, – дозволяє відносно легко адаптуватися до процесу написання «C-подібних» скриптів, широко використовуваних у різних програмних середовищах розробки засобів мультимедіа (Flash, Director і т. п.). Саме з цієї причини всі додатки, що вивчаються в рамках дисципліни «Основи програмування», є консольними.

Однак програмування для консолі аж ніяк не є уділом лише новачків. Професійні програмісти часто використовують вікно консолі для тестування додатків і компонентів.

Віконні додатки на базі WinForms

На відміну від консольних в них застосовується графічний користувальницький інтерфейс користувача, де команди користувача передаються шляхом клацань миші на екранних кнопках і піктограмах, а введення інформації здійснюється через різні текстові вікна і вікна списків.

Щоб написати віконний додаток з нуля, не використовуючи ніяких готових компонентів, доведеться витратити на розробку місяці й навіть роки. Бібліотека класів .NET Framework містить великий набір компонентів за назвою **WinForms**. Ці компоненти застосовуються для створення складних віконних додатків. **WinForms** забезпечує програмісту на C# простий доступ до віконних служб операційної системи Windows.

Додатки ASP.NET

ASP.NET (Active Server Pages .NET) – це група компонентів, використовуваних для спрощення створення додатків на базі браузера.

Браузер – це додаток, що дозволяє користувачеві в зручній формі переглядати документи в форматі HTML (HyperText Marcup Language). Браузери широко використовуються для відображення інформації в Internet.

Web-служби

Web-служби – це важлива частина нової технології, що обіцяє змінити шляхи використання Internet, а з ними – представлення про те, як розробляти додатки й використовувати їх. Web-служби представляють просто компоненти або додатки, доставлені на комп'ютер з Internet. З Web-служб, розміщених на різних комп'ютерах, зв'язаних з Internet, можуть бути сформовані нові Web-служби.

Це відкриває для програмістів різні цікаві можливості. Зокрема, з'являється можливість збирати компоненти і додатки не тільки з класів .NET Framework і вихідного коду, написаного самим програмістом, але й з Web-служб, знайдених в Internet.

Питання для самоконтролю

1. Що означає .NET? У чому особливість .NET-платформи?
2. Перерахуйте основні етапи одержання программи, що виконується.
3. Навіщо необхідний етап компіляції?
4. У чому особливість компіляції в .NET вихідного коду C#?
5. У чому суть процедурно-орієнтованого стилю програмування? Яка структура процедурно-орієнтованої програми?
6. Укажіть недоліки процедурного стилю програмування та шляхи їхнього подолання.
7. Дайте визначення об'єкта і наведіть приклад з описом його властивостей і поводження.
8. Що таке клас?
9. Навіщо необхідно повторне використання програмного забезпечення? Приведіть приклади повторного використання.
10. Призначення бібліотеки класів .NET Framework.
11. Опишіть можливі типи C#-додатків і області їхнього застосування.