Курс "Об'єктно-орієнтоване програмування" (частина друга)

OOP

Лекцій – 24 години

Лабораторних робіт – 24 години

Екзамен – 5 н/с (осінь)

Курсова робота – 6 н/с

Мета – вивчення й практичне освоєння **методів і засобів об'єктно-орієнтованого програмування**, знайомство з **методологією об'єктно-орієнтованого аналізу та проектування (OOA & OOD)** складних програмних систем; вивчення сучасної об'єктно-орієнтованої **мови програмування C#.**

Головні теми

- Нові об’єктно-орієнтовані можливості мови C#
- Лямбда-вирази. Анонімні методи
- Мова запитів LINQ
- Регулярні вирази
- Динамічне зв’язування
- Паралелізм. Асинхронні функції
- Зборки .NET Framework
- Об’єктно-орієнтована парадигма програмування
- Побудова об’єктної моделі предметної області

Лабораторні роботи

МОСКОВСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ
имени М.В. ЛОМОНОСОВА

Факультет вычислительной математики и кибернетики

Н.И. Березина

Лабораторные работы по курсу

Объектно-ориентированное программирование: язык программирования C#

Учебное пособие

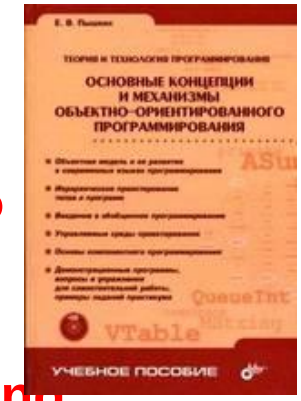
МАКС Пресс

Москва-2010

Лабораторні роботи (поточний н/с)

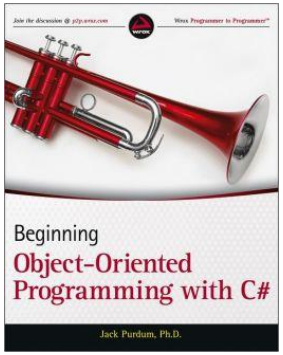
1. Класи, властивості, індексатори. Масиви. **Лаб. №1**
2. Спадкування, виключення. Інтерфейси. Ітератори. **Лаб№2**
3. Узагальнені типи. Класи колекцій. LINQ. **Лаб№3**
4. Делегати. Події. **Лаб№4**
5. Класи для роботи з файлами. Серіалізація. **Лаб№5**
6. Побудова обєктної моделі предметної області (для варіанту курсової роботи). **Лаб№6**
7. Демонстрація нових можливостей C# (рефлексія, динамічні типи, асинхронна робота з потоками). **Лаб№7**

Література

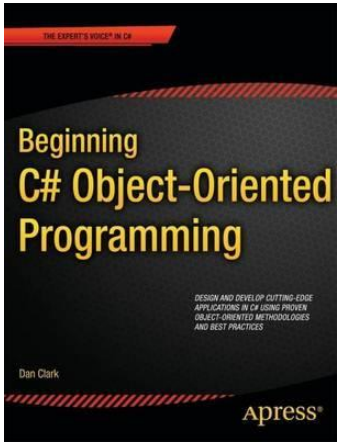


1. **Пышкин Е.В. Основные концепции и механизмы объектно-ориентированного программирования** . – СПб.: BHV, 2005.

2. **Beginning Object-Oriented Programming with C#.** – Wrox, 2012



3. **Beginning C# Object-Oriented Programming.** – Apress, 2013



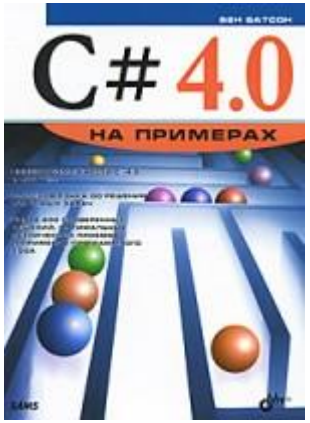
Література

4. **Шилдт Герберт C# 4.0 Полное руководство**, 2011

5. **Эндрю Троелсен - ЯЗЫК ПРОГРАММИРОВАНИЯ C#5.0 И ПЛАТФОРМА .NET 4.5**, 2013

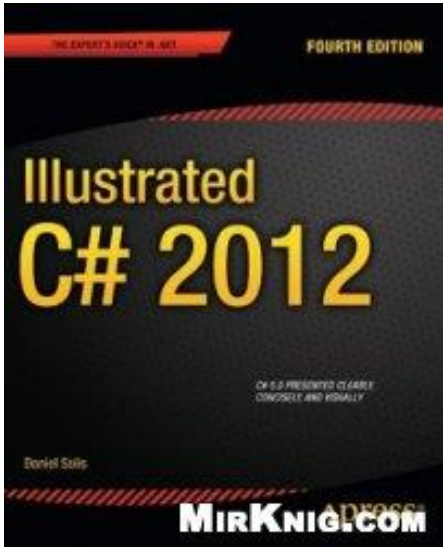
6. Зиборов В.В. - Visual C# 2012 на примерах.

7. Ватсон Б. - C# 4.0 на примерах.

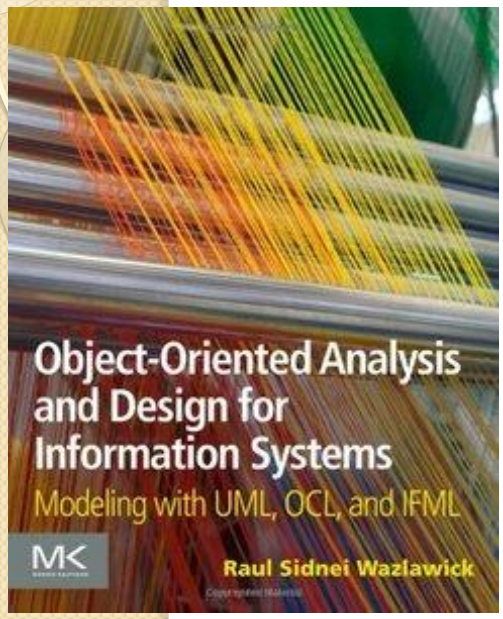


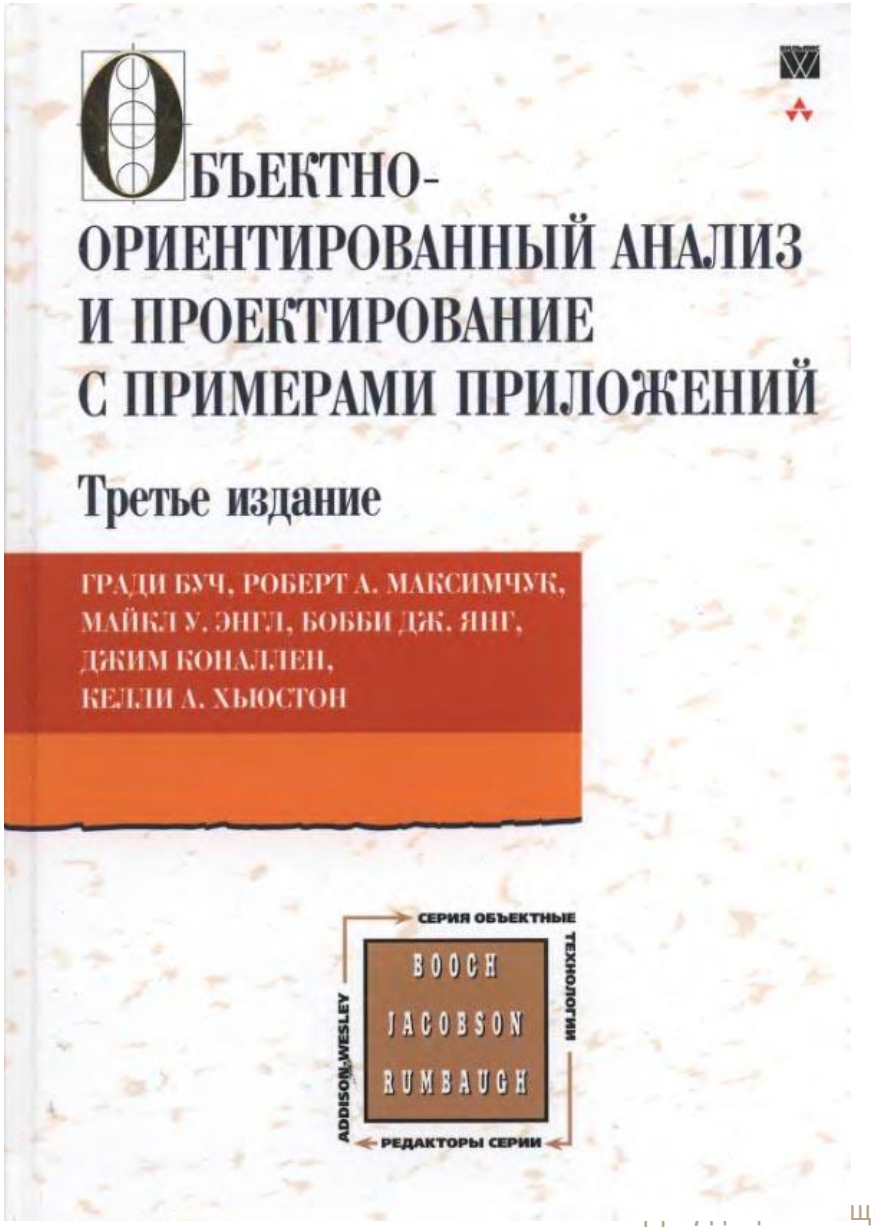
Література

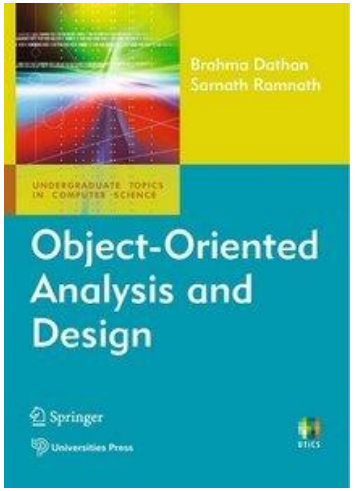
8. **Трей Нэш C# 2010. Ускоренный курс для профессионалов.**
9. К. Нейгел C# 2008 и платформа .NET 3.5 для профессионалов.
10. Illustrated C# 2012

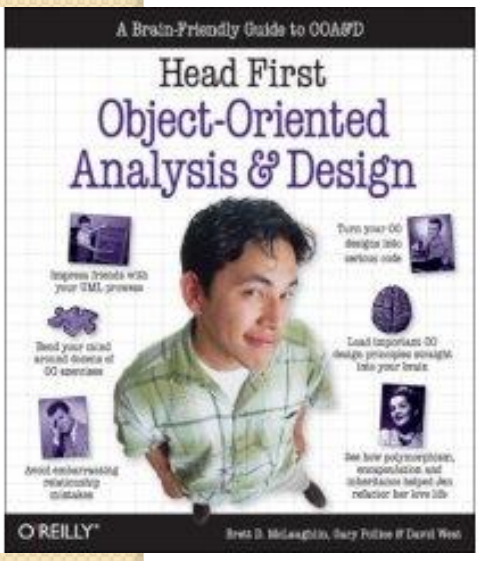


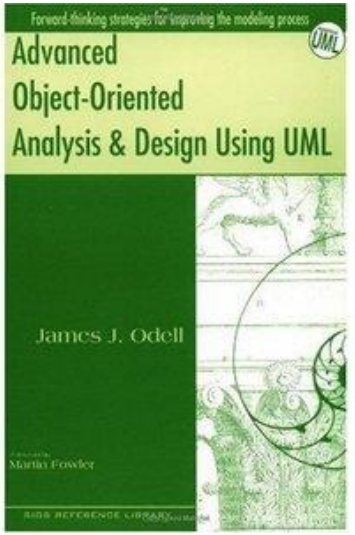
Література


Object-Oriented Analysis and Design for Information Systems
Modeling with UML, OCL, and IFML
Raul Sidnei Wazlawick


ОБЪЕКТНО-ОРИЕНТИРОВАННЫЙ АНАЛИЗ И ПРОЕКТИРОВАНИЕ С ПРИМЕРАМИ ПРИЛОЖЕНИЙ
Третье издание
ГРАДИ БУЧ, РОБЕРТ А. МАКСИМЧУК, МАЙКЛ У. ЭНГЛ, БОББИ ДЖ. ЯНГ, ДЖИМ КОНАЛЛЕН, КЕЛЛИ А. ХЬЮСТОН
СЕРИЯ ОБЪЕКТНЫЕ ТЕХНОЛОГИИ
ADDISON-WESLEY
BOOCH
JACOBSON
RUMBAUGH
РЕДАКТОРЫ СЕРИИ


Brahma Dathan
Sarnath Ramnath
Object-Oriented Analysis and Design
Springer
Universities Press


A Brain-Friendly Guide to OOA&D
Head First
Object-Oriented Analysis & Design
O'REILLY


Advanced Object-Oriented Analysis & Design Using UML
James J. Odell
Martin Fowler

Відеокурси

- **Специалист М2555 Разработка Windows - приложений для Microsoft .NET на Visual C# [2011]**
- **Специалист М10266 (М2124) Программирование на C# с использованием Microsoft .NET Framework 4 [2011]**
- Udemy - Object Oriented Programming (2014)
- AppDev - Object Oriented Programming
- AppDev - Learning To Program Using Visual C# 2008
- Lynda.com - Foundations of Programming Object-Oriented Design
- TekPub - C# Design Strategies with Jon Skeet
- Tekpub - Mastering C#4 with Jon Skeet
- thenewboston.org - C# Beginners Tutorial
- **Асинхронное программирование**

C# - один из наиболее популярных объектно-ориентированных языков программирования в мире. Став специалистом, пишущим на C#, Вы будете востребованы в любой стране.

Застосування засобів введення-виведення

- Шилдт Г. C# 4.0 Полное руководство, гл.14, стр.431

- **Введення-виведення в C# здійснюється через потоки**
- **Байтові та символьні потоки.**
  - На самому низькому рівні – байтами
  - Для символів Unicode потрібне приведення типів
- **Вбудовані потоки:**
  - **Console.In**
  - **Console.Out**
  - **Console.Error**
- **Класи потоків**
  - **Вони визначені в System.IO**
- **Клас Stream**
  - **Основним для потоків є клас System.IO.Stream**
  - **Визначає байтовий поток і є базовим для інших потоків**

Клас Stream

Метод	Опис
void Close()	Закриває поток
void Flash()	Виводе вміст потока на фізичний пристрій
int ReadByte()	Повертає цілочисельне представлення наступного байту (або -1)
int Read(byte[] buffer, int offset, int count)	Читає в буфер count байтів, починаючи з зміщення offset
long Seek(long offset, SeekOrigin origin)	Встановлює поточне положення в потоці
void WriteByte(byte value)	Виводе байт в поток виведення
void Write(byte[] buffer, int offset, int count)	Запис з буфера в поток
bool CanRead	Властивість
bool CanSeek	Властивість
bool CanWrite	Властивість
long Length	Властивість
long Position	Властивість

Класи байтових потоків

- BufferedStream
- FileStream
- MemoryStream
- UnmanageMemoryStream
- Інші потоки для виведення в стиснуті файли, сокети та канали

Класи-оболонки символьних потоків

- **Для створення символьного потоку достатньо заключити байтовий поток в один з класів-оболонок символьних потоків**
- На вершині ієрархії класів символьних потоків знаходяться абстрактні класи TextReader та TextWriter

Методи введення, визначені в TextReader

Метод	Опис
int Peek()	Отримує наступний символ, не видаляючи його
int Read()	Повертає ціле представлення символа
int Read(char[] buf, int i, int count)	Введення в масив
string ReadLine()	Введення рядка
string ReadToEnd()	Введення всіх символів

- Ці методи спроможні генерувати виняток IOException
- В класі TextWriter визначені варіанти методів Write() та WriteLine(), призначені для виведення даних усіх вбудованих типів

Спадкоємці класів TextReader та TextWriter

Клас потока	Опис
StreamReader	Призначений для введення символів з байтового потоку. Цей клас є оболонкою для байтового потоку введення
StreamWriter	Призначений для виведення символів в байтовий поток.
StringReader	Призначений для введення символів з символьного рядка
StringWriter	Призначений для виведення символів в рядок

Двійкові (бінарні) потоки

Окрім байтових та символьних потоків ще існують два класи двійкових потоків:

- BinaryReader
- BinaryWriter

Клас FileStream і байтове введення-виведення в файл

- Цей клас є похідним від класу Stream

Відкриття та закриття файлу.

В класі визначено декілька конструкторів:

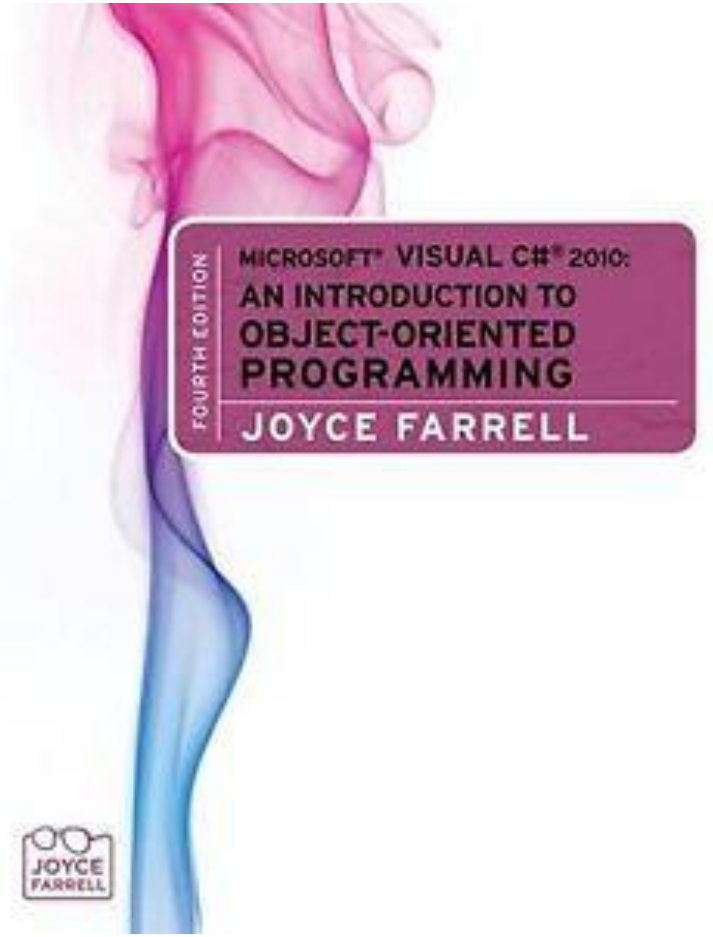
- FileStream(string шлях, FileMode режим)
  - Шлях – ім'я файлу
  - Режим – FileMode.Append (або Create, CreateNew, Open, OpenOrCreate, Truncate)
  - Приклад.
  - FileStream fin;
  - try{ fin=new FileStream("test.dat", FileMode.Open);}
  - catch(Exception e){ Console.WriteLine(e.Message);}
- FileStream(string шлях, FileMode режим, FileAccess доступ)
  - Доступ: FileAccess.Read (Write,ReadWrite)

Приклад

```
using System;
using System.IO;
public class WriteSomeText{
   public static void Main(){
    FileStream outFile = new FileStream( "SomeText.txt" , FileMode.Create,
                                  FileAccess.Write);
    StreamWriter writer = new StreamWriter(outFile);
    Console.Write("Enter some text >> ");
      string text = Console.ReadLine();
    writer.WriteLine(text);
          // Error occurs if the next two statements are reversed
    writer.Close();
    outFile.Close();
    }
}
```

Запис в файл довільного доступу (1)

(Joyce Farrell Microsoft Visual C# 2010: An Introduction to Object-Oriented Programming, Fourth Edition, 2011, p.695)




```
using System;
using System.IO;
class Employee{
    public int EmpNum {get; set;}
    public string Name {get; set;}
    public double Salary {get; set;}
}
public class WriteSequentialFile{
   public static void Main(){
      const int END = 999;
      const string DELIM = ",";
      const string FILENAME = "EmployeeData.txt";
```

Запис в файл довільного доступу (2)

```
Employee emp = new Employee();
FileStream outFile = new FileStream(FILENAME,
                              FileMode.Create, FileAccess.Write);
StreamWriter writer = new StreamWriter(outFile);
Console.Write("Enter employee number or " + END +
                              " to quit >> ");
emp.EmpNum = Convert.ToInt32(Console.ReadLine());
    while(emp.EmpNum != END){
        Console.Write("Enter last name >> ");
        emp.Name = Console.ReadLine();
        Console.Write("Enter salary >> ");
        emp.Salary = Convert.ToDouble(Console.ReadLine());
        writer.WriteLine(emp.EmpNum + DELIM + emp.Name +
                              DELIM + emp.Salary);
```

Запис в файл довільного доступу (3)

```
    Console.Write("Enter next employee number or " +
          END + " to quit >> ");
    emp.EmpNum = Convert.ToInt32(Console.ReadLine());
}
writer.Close();
outFile.Close();
}
}
```

Читання з файлу довільного доступу (1)

```
using System;
using System.IO;
class Employee{
    public int EmpNum {get; set;}
    public string Name {get; set;}
    public double Salary {get; set;}
}
public class ReadSequentialFile{
   public static void Main(){
          const char DELIM = ',';
          const string FILENAME = "EmployeeData.txt";
          Employee emp = new Employee();
          FileStream inFile = new FileStream(FILENAME,
                                FileMode.Open, FileAccess.Read);
```

Читання з файлу довільного доступу (2)

```
StreamReader reader = new StreamReader(inFile);
string recordIn;
string[] fields;
Console.WriteLine("\n{0,-5}{1,-12}{2,8}\n",
                  "Num", "Name", "Salary");
recordIn = reader.ReadLine();
  while(recordIn != null){
     fields = recordIn.Split(DELIM);
     emp.EmpNum = Convert.ToInt32(fields[0]);
     emp.Name = fields[1];
     emp.Salary = Convert.ToDouble(fields[2]);
```

Читання з файлу довільного доступу (3)

```
Console.WriteLine("{0,-5}{1,-12}{2,8}",
        emp.EmpNum, emp.Name,
        emp.Salary.ToString("C"));
        recordIn = reader.ReadLine();
        }
reader.Close();
inFile.Close();
}
}
```

Класи колекцій в C#

- **Visual C# 2008. Базовый курс**. – 2009 (Уотсон, Нейгел) – глава 11, с. 289 – є глибоке копіювання, ітератори, тощо
- **Шилдт Г. C# 4.0: полное руководство**. – с. 923 (глава 25)

Платформа .NET Framework надає спеціалізовані класи для зберігання груп об'єктів. Це класи колекцій.

Класи колекцій забезпечують підтримку стеків, черг, списків, словників, хеш-таблиць, тощо.

5 типів колекцій:

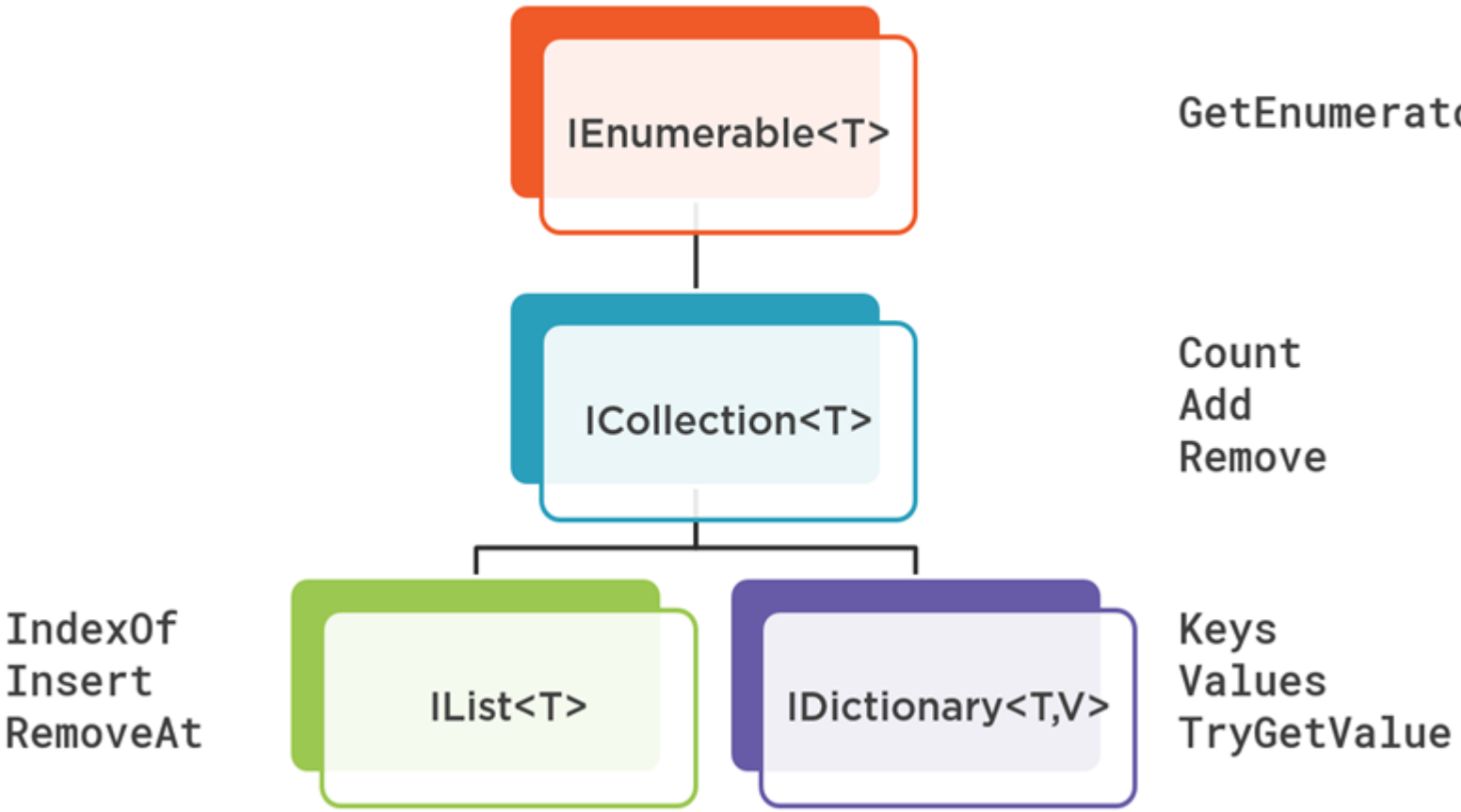
- **Неузагальнені** (ArrayList, Hashtable, SortedList, Stack, Queue) – простір імен **System.Collections**
- Спеціалізовані
- З порозрядною організацією
- **Узагальнені** (List<T>, Dictionary<T>, HashSet<T>, …) - **System.Collections.Generic**
- Паралельні

Інтерфейси, що використовуються колекціями

Більшість класів колекцій є похідними від інтерфейсів ICollection, IComparer, IEnumerable, IList, IDictionary та їх узагальнених еквівалентів.

- **IEnumerable** – надає ітератор, що підтримує простий перебір елементів колекції
- **ICollection** – визначає методи, що дозволяють визначити кількість елементів, а також методи синхронізації для колекцій
- **IList** – забезпечує доступ за індексом та методи модифікації колекцій
- **IDictionary** – це інтерфейс колекції пар "ключ-значення"
- **ICloneable** – визначає метод для створення копії об'єкта

Generic Collection Interfaces



Додаткові інтерфейси для колекцій

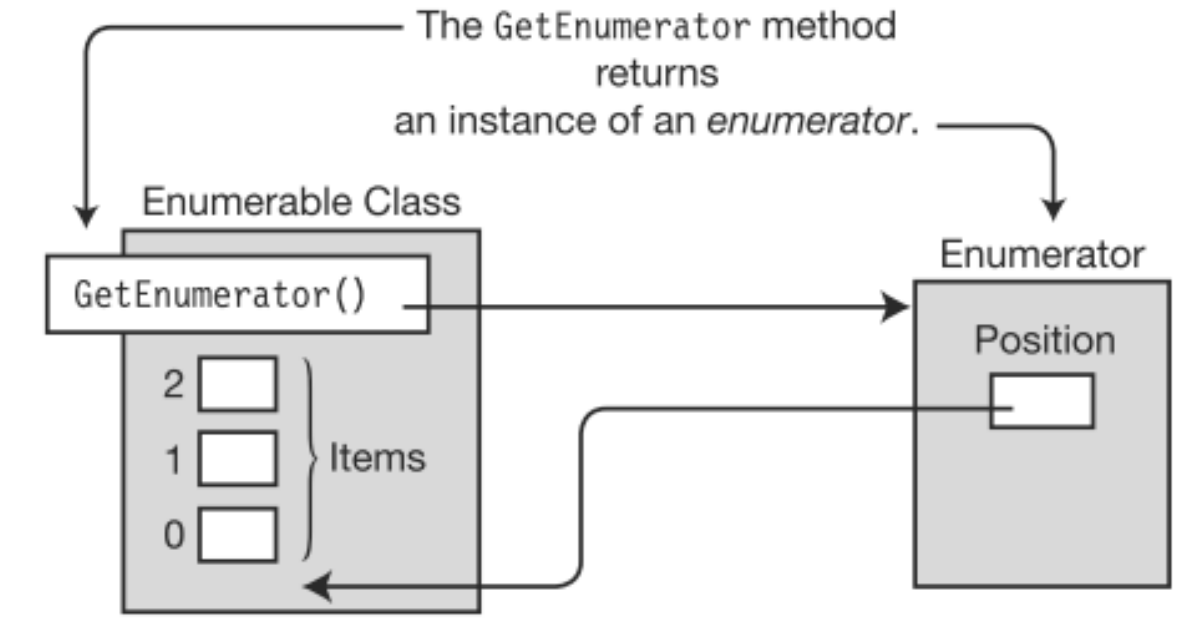
- **IComparer** – визначає метод порівняння об'єктів
- **IEnumerator** – визначає методи, що дозволяють здійснити простий перебір елементів колекції.
  - (Повертається методом GetEnumerator інтерфейса IEnumerable)
- **IComparable** – використовується при пошуку та сортуванні об'єктів

Інтерфейс IEnumerable

- **Усі колекції його реалізують. Цей інтерфейс дозволяє перебрати елементи колекції в циклі.**
- Інтерфейс описує усього один метод GetEnumerator()
  - IEnumerator GetEnumerator();
  - Цей метод повертає посилання на інтерфейс IEnumerator (перераховувач), за допомогою якого можна здійснити перебір
- Інтерфейс IEnumerator має методи MoveNext(), Reset() та властивість Current

```
int[] arr1 = { 10, 11, 12, 13 };  // Define the array.
foreach (int item in arr1)   // Enumerate the elements.
   Console.WriteLine("Item value:  {0} ", item);
```

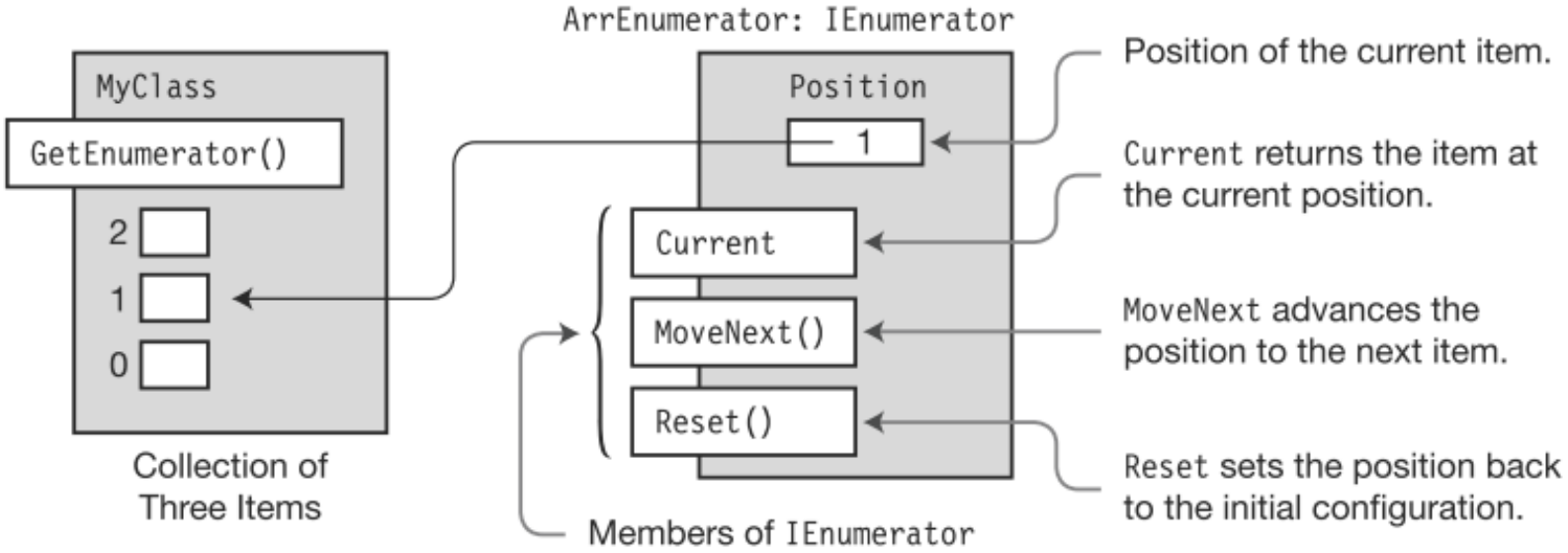
Інтерфейс IEnumerable



An *enumerable* is a type that has a method called `GetEnumerator` that returns an enumerator for its items.

An *enumerator* is an object that can return each item in a collection, in order.

The IEnumerator Interface



Simulation of a foreach loop

```
static void Main()
  {
    int[] MyArray = { 10, 11, 12, 13 };          // Create an array.
                          Get and store the enumerator.
                                          ↓
    IEnumerator  ie = MyArray.GetEnumerator();
              Move to the next position.
                    ↓
    while ( ie.MoveNext() )
    {                 Get the current item.
                              ↓
      int i = (int) ie.Current;
      Console.WriteLine("{0}", i);               // Write it out.
    }
  }
```

Інтерфейс ICollection

- Він спадкує від IEnumerable
- Він визначає властивість для отримання числа елементів колекції
  - **int Count { get; };**
- Та ще визначає метод CopyTo() для копіювання елементів колекції в масив
  - **void CopyTo (Array array , int index);**

Інтерфейс IList

```
public interface IList: ICollection, IEnumerable
{
    object this [int index] { get; set; } //індексатор
    int  Add( object value); //повертає індекс доданого елемента
    void  Insert( int index, object value);
    void  Remove( object value);
    void  Remove( int index);
    void  Clear();
    bool  Contains( object value);
    int  IndexOf( object value);

    + властивість IsReadOnly
    + властивість IsFixedSize
}
```

Інтерфейс ICollection<T>

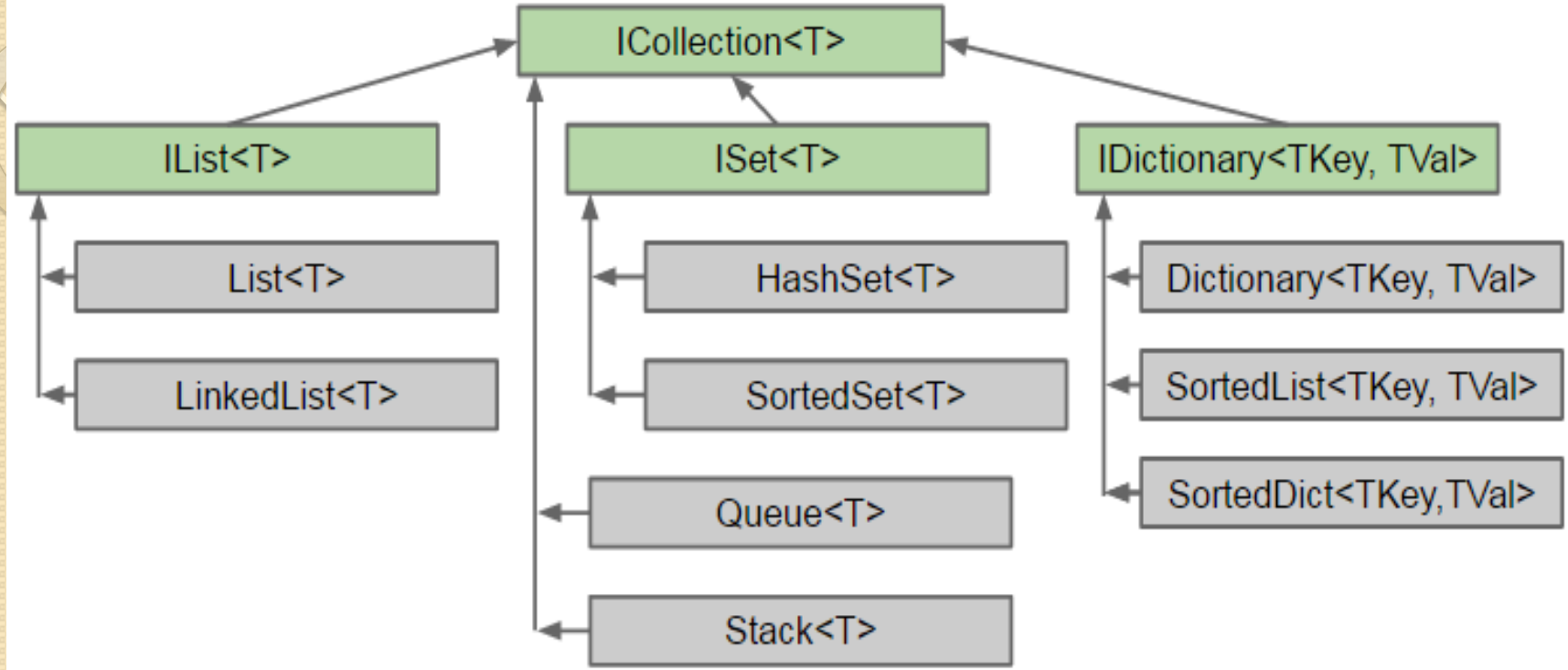
```
public interface ICollection<T>: IEnumerable<T>, IEnumerable
{
    int Count { get;}
    bool  IsReadOnly { get; }
    void  Add( T item);
    void  Clear();
    bool  Contains( T item);
    void  CopyTo( T[ ] array, int arrayIndex);
    bool  Remove( T item);

    }
```

Узагальнені типи строго типізовані, що дає компілятору більше можливості для забезпечення безпеки типів.

Також при зберіганні типів-значень generic-колекції більш ефективні тому що виключають необхідність в упаковці (boxing).

Інтерфейс IList<T>

```
public interface IList<T>: ICollection<T>,
   IEnumerable<T>, IEnumetable
{
   T  this [ int index] { get; set;}
   int IndexOf( T item);
   void Insert( int index, T item);
   void RemoveAt( int index );
}
```


ICollection<T>
IList<T>
ISet<T>
IDictionary<TKey, TVal>
List<T>
LinkedList<T>
HashSet<T>
SortedSet<T>
Queue<T>
Stack<T>
Dictionary<TKey, TVal>
SortedList<TKey, TVal>
SortedDict<TKey,TVal>

Ітератори

- **Нэш C# 2010. Ускоренный курс для проф-лов**. Стр.272
- **Illustrated C# 2012** - p.446
- **Шилдт Г. C# 4.0: полное руководство**. – с. 1003

Ми розглядали класи, що реалізують IEnumerable (їх називають Enumerable classes). Вони створюють "енумератори" (перераховувачі, enumerators) для перебору елементів.

Починаючи з C# 2.0 з'явився більш простий механізм для створення Enumerable classes. Цей механізм – ітератори.

Ітератори примушують компілятор самому створювати enumerators для класу.

Що таке ітератор?

Ітератор – це метод, операція або аксесор, що повертає по черзі члени сукупності об'єктів (Шилдт).

Визначивши ітератор, можна звертатися до об'єктів в циклі foreach

Ітератори

Ітератор – це об’єкт, який дозволяє перебирати всі елементи колекції без врахування особливостей її реалізації.

(Wikipedia) Ітератор – це об’єкт, який абстрагує за єдиним інтерфейсом доступ до елементів колекції.

(MSDN) Ітератор – метод доступу get або оператор, що виконує ітерацію класу масива або колекції за допомогою ключового слова yield.

При використанні return з yield елемент колекції негайно повертається об’єкту, що його викликав, до того як буде отримано доступ до наступного елемента.

Приклад ітератора

The following method declaration implements an iterator that produces and returns an enumerator.

- The iterator returns a generic enumerator that returns three items of type string.
- **The yield return statements declare that this is the next item in the enumeration**.

Return an enumerator that returns strings.

↓

```
public IEnumerator<string> BlackAndWhite()          // Version 1
{
   yield return "black";
   yield return "gray";
   yield return "white";
}
```

```
public IEnumerator<string> BlackAndWhite()          // Version 2
  {
     string[] theColors = { "black", "gray", "white" };
     for (int i = 0; i < theColors.Length; i++)
        yield return theColors[i];                  // yield return
  }
```

Iterator Blocks

An iterator block is a code block with one or more yield statements.

Any of the following three types of code blocks can be iterator blocks:

- A method body
- An accessor body
- An operator body

Iterator blocks are treated differently than other blocks. Other blocks contain sequences of statements that are treated imperatively. .....

The code in the iterator block describes how to enumerate the elements.

Iterator blocks have two special statements:

- The yield return statement specifies the next item in the sequence to return.
- The yield break statement specifies that there are no more items in the sequence.

Приклад використання ітератора

```
class MyClass
 {
    public IEnumerator<string> GetEnumerator()
    {
       return BlackAndWhite();  // Returns the enumerator
    }

    public IEnumerator<string> BlackAndWhite()  // Iterator
    {
       yield return "black";
       yield return "gray";
       yield return "white";
    }
 }
```

Приклад використання ітератора

```
class Program
  {
    static void Main()
    {
      MyClass mc = new MyClass();

      foreach (string shade in mc)
        Console.WriteLine(shade);
    }
  }
```

This code produces the following output:

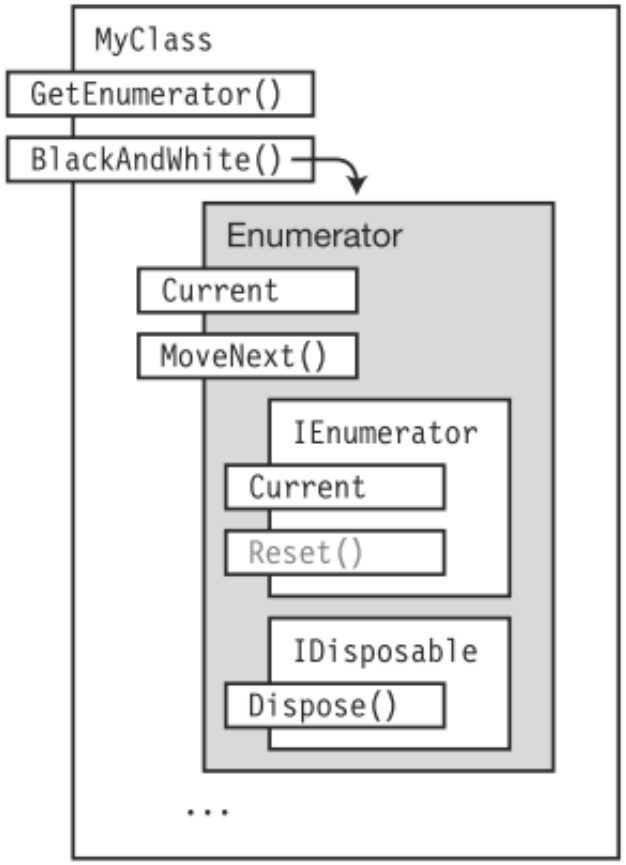
```
black
gray
white
```

Приклад використання ітератора

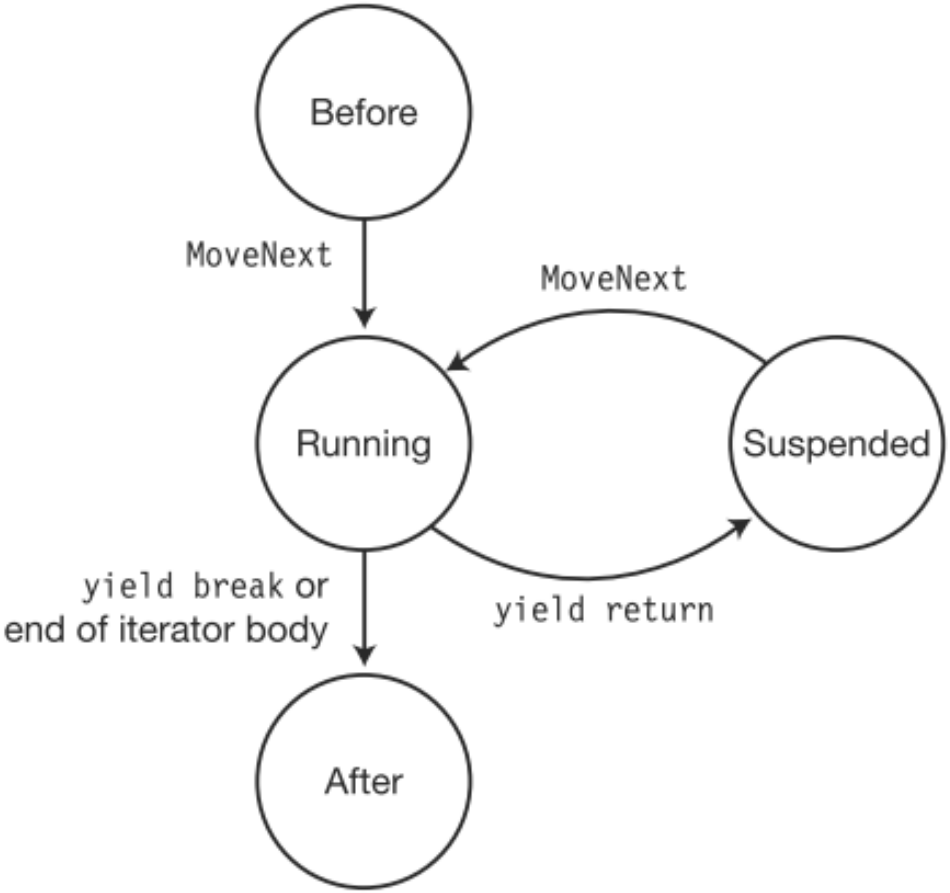
```
class Myclass
{
   public IEnumerator<string> GetEnumerator()
   {
      return BlackAndWhite();
   }

   public IEnumerator<string> BlackAndWhite()
   {
      yield return "black";
      yield return "gray";
      yield return "white";
   }
}
```

The iterator construct

produces a method that returns an enumerator.



An iterator state machine



Ітератори

- **Вони дозволяють створити перераховувач (“перечислитель”, enumerator), що блокує доступ до контейнера на період перерахування.**
- У Нэша на стор.269 розглянуто приклад власного класу колекції, яка реалізує IEnumerable<T> та IEnumerator<T> без ітераторів (це складно)
- З використанням ітераторів буде так:

```
public class MyColl<T>: IEnumerable<T>
{
    public MyColl( T[] items){ this.items = items;}
    public IEnumerator<T> GetEnumerator(){
        foreach ( T item in items){ yield return item;}
        }
    IEnumerator IEnumerable.GetEnumerator(){
        return GetEnumerator();
    }
    private T[] items;
}
```

Ітератори

```
public class EntryPoint
{
    static void Main()
    {
        MyColl<int> integers = new MyColl<int>(
                new int[] {1, 2, 3,4 });
        foreach ( int n in integers){
                Console.WriteLine(n);
        }
    }
}
```

При виклику GetEnumerator() **код в методі, що містить yield, в цей момент часу не виконується. Натомість компілятор генерує клас перераховувача,** який містить yield-блок.

Екземпляр цього класу і повертається.

Таким чином, коли спрацьовує foreach – викликається код yield-блоку через методи IEnumerator<T>.

Прямі, зворотні та двонаправлені ітератори

Вони є у багатьох бібліотечних контейнерних типах.

Прямі – це звичайні (перераховують в прямому порядку).

Зворотні – в зворотному порядку.

Реалізувати можна легко за допомогою створення блоку yield.

(Дивись: Нэш, стор.277)

LINQ : мова інтегрованих запитів

- **Нэш C# 2010. Ускоренный курс для проф-лов**. Стр.524
- **Illustrated C# 2012** - p.477
- **Шилдт Г. C# 4.0: полное руководство**. – Гл.19 с.637
- **Троельсен Э. Язык пр-я C#2008 и платформа .NET 3.5** – с.478

Набір технологій LINQ (Language Integrated Query) з'явився в .NET 3.5.

LINQ – це строго типізована мова запитів до довільних сховищ даних безпосередньо вбудована в граматику мови C#.

Різновиди LINQ:

- **LINQ to Objects**
- LINQ to SQL
- LINQ to XML
- **LINQ to Entities**
- Parallel LINQ

Для застосування засобів LINQ у вихідний текст програми слід включити простір імен **System. Linq**.

LINQ

Через LINQ ви можете запитувати дані з баз даних, колекцій програмних об'єктів, XML-документів тощо.

Простий приклад.

```
static void Main()
{
    int[] numbers = { 2, 12, 5, 15 };          // Data source
    IEnumerable<int> lowNums =                 // Define and store the query.
                    from n in numbers
                    where n < 10
                    select n;
    foreach (var x in lowNums)    // Execute the query.
        Console.Write("{0}, ", x);
}
```

This code produces the following output:

2, 5,

Трошки теорії

- (Нэш) **Мови, подібні C** (включаючи C#), **імперативні за природою** – в тому сенсі, що наголос робиться на станах системи і їх змінах на протязі певного часу.
- **Мови запитів даних, подібні SQL, за своєю природою функціональні** – в тому сенсі, що наголос робиться на операціях.
- **LINQ заповнює прогалину між імперативним та функціональним стилями програмування.**
- **LINQ дозволяє програмістам зосередитися на бізнес-логіці і менше втрачати часу на кодування рутинних деталей.**
- Примітка. Функціональні мови програмування: Lisp, Scheme, F#, …

LINQ

Мова C# використовує наступні пов'язані з LINQ засоби:

- **Неявно типізовані локальні змінні**
- **Систаксис ініціалізації об'єктів в колекції**
- **Лямбда-вирази**
- **Розширюючі методи (Extension Methods)**
- **Анонімні типи (Anonymous Type)**

Неявно типізовані локальні змінні

Ключове слово var дозволяє визначити локальну змінну без явного вказування типу.

Компілятор визначить тип під час першого присвоювання.

```
var myInt = 0;          var myBool = true;
```

Ініціалізатори колекцій

```
using System;
using System.Collections.Generic;
class Employee{
      public string Name{ get; set;}
}
class CollInitializerExample{
   static void Main(){
      var devTeam = new List<Employee> {
        new Employee { Name = “Michael Bolton”},
         new Employee { Name = “Satir Nagh”},
         new Employee { Name = “Peter Gibbons”}
};
. . . . . .
```

Анонімні функції

- Метод, на який посилається делегат, нерідко окремо (без делегата) взагалі не використовується.
- В подібних випадках можна скористатися анонімною функцією, щоб не створювати окремий метод.

Анонімна функція, по суті, представляє безіменний кодовий блок, що передається конструктору делегата.

- Починаючи з версії C# 3.0 передбачені два різновиди анонімних функцій:
  - **Анонімні методи**
  - **Лямбда-вирази**

Анонімний метод – це метод, який оголошується inline в точці створення екземпляра делегата.

Анонімні методи

```
class Program
{
  public static int Add20(int x)
  {
    return x + 20;
  }

  delegate int OtherDel(int InParam);
  static void Main()
  {
    OtherDel del = Add20;


    Console.WriteLine("{0}", del(5));
    Console.WriteLine("{0}", del(6));
  }
}
```

Named Method

```
class Program
{


  delegate int OtherDel(int InParam);
  static void Main()
  {
    OtherDel del = delegate(int x)
                   {
                     return x + 20;
                   };
    Console.WriteLine("{0}", del(5));
    Console.WriteLine("{0}", del(6));
  }
}
```

Anonymous Method

Syntax of Anonymous Methods

The syntax of an anonymous method expression includes the following components:

- The type keyword **delegate**
- The **parameter list**, which can be omitted if the statement block doesn't use any parameters
- The **statement block**, which contains the code of the anonymous method

Parameter

Keyword **list** **Statement block**

↓ ↓ ↓

delegate (Parameters) { ImplementationCode }

Анонімний метод без параметрів

```
using System;
delegate void CountIt();
class AnonMethDemo
{
    static void Main()
    {
            CountIt count = delegate{
             for( int i=0; i <=5; i++)
                Console.WriteLine(i);
             };

    count();
}
}
```

Лямбда-вирази

- Незважаючи на всю цінність анонімних методів, їм на зміну в C# 3.0 прийшов більш досконалий підхід : лямбда-вирази.
- І хоча лямбда-вирази в основному застосовуються в роботі з LINQ, вони часто використовуються і разом з делегатами та подіями.

Лямбда-операція

- У всіх лямбда-виразах застосовується нова лямбда-операція “=>”, яка розділяє лямбда-вираз на дві частини
- В лівій частині вказується вхідний параметр (або декілька параметрів)
- В правій частині вказується тіло лямбда-виразу
- Операція “=>” інколи описується такими словами, як “переходе” або “стає” (“goes to”)

Лямбда-вирази

Можна легко трансформувати анонімний метод в лямбда-вираз, виконавши наступне:

- **Видалити ключове слово delegate**
- **Помістити лямбда-операцію, =>, між списком параметрів та тілом анонімного метода**

```
MyDel del = delegate(int x){ return x + 1; } ; // Anonymous method
MyDel le1 =    (int x) => { return x + 1; } ;     // Lambda expression
```

Отриманий лямбда-вираз можна спростити, врахувавши, що компілятор з оголошення делегата знає тип параметра та іншу інформацію

```
MyDel del = delegate(int x) { return x + 1; };     // Anonymous method
  MyDel le1 =  (int x) => { return x + 1; };     // Lambda expression
  MyDel le2 =   (x) => { return x + 1; };     // Lambda expression
  MyDel le3 =    x  => { return x + 1; };     // Lambda expression
  MyDel le4 =    x  =>  x + 1;     // Lambda expression
```

Лямбда-вирази

- C# підтримує дві форми лямбда-виразів: поодинокі і блоковані.
- У поодиноких лямбда-виразах тіло складається з одного виразу, тіло не укладається в фігурні дужки.

  параметр => вираз
- У блокованих лямбда-виразах тіло складається з ряду операторів і укладається в фігурні дужки

  (список_параметрів) => { оператори }

Лямбда-вираз застосовується в два етапи:

- **Спочатку оголошується тип делегата, сумісний з лямбда-виразом, а потім створюється екземпляр делегата, якому присвоюється лямбда-вираз.**
- **Після цього лямбда-вираз обчислюється при звертанні до екземпляра делегата**

Приклад

```
using System;
delegate bool IsEven ( int v);
delegate int Incr ( int v);
class SimpleLambdaDemo{
    static void Main () {
        Incr  incr = count => count+2;
        int  x = -10;
        while(x<=0){
            Console.Write(x + " ");
            x = incr(x);
        }
        Console.WriteLine("\n");
        IsEven  isEven = n => n%2 == 0;
        for( int i=1; i<=10; i++)
            if( isEven(i) ) Console.WriteLine( i + "  парне.");
    }
}
```

Блоковані лямбда-вирази

```
using System;
delegate int  IntOp( int end);
class StatementLambdaDemo
{
  static void Main()
  {
    IntOp  fact = n => {
                  int r = 1;
                  for( int i =1; i<=n; i++)
                        r = i*r;
                  return r;
                  };
       Console.WriteLine( “ 3! дорівнює  “ + fact(3));
   }
}
```

Лямбда-вираз з декількома параметрами

```
delegate bool InRange( int lower, int upper, int v);
. . . . . .
InRange rangeOK = (low, high, val) => val>=low && val<=high;
. . . . . .
if( rangeOK(1, 5, 3)) Console.WriteLine( “In range !”);
```

Зауваження.

Термін lambda expression походить від lambda calculus (*Лямбда-числення*), що розроблено в 1920-х та 1930-х роках математиком **Алонзо Черчем** (Alonzo Church).

Лямбда-числення є системою для представлення функцій і використовує грецьку літеру лямбда (λ) для позначення безіменних функцій.

Лямбда-вирази

Лямбда-вираз спрощує роботу з делегатами (зменшує об'єм коду, що треба писати вручну)

Приклад.

```
List<int> list = new List<int>();
list.AddRange( new int[]{20, 1, 4, 8, 9, 44});
List<int> evenNumbers = list.FindAll( i => (i%2)==0);
foreach( int i in evenNumbers){
    Console.Write( "{0}\t", i);
}
Console.WriteLine();
```

Лямбда-вирази дуже корисні при роботі з об'єктною моделлю LINQ. Як ми далі побачимо,

операції запитів C# LINQ – це просто скорочена нотація викликів методів класу System.Linq.Enumerable.

Ці методи мають параметри-делегати.

Розширюючі методи

- **Extension Methods дозволяють додавати нову функціональність існуючим класам без необхідності спадкування (А також до запечатаних (sealed) класів)**
- Extension Method – це статичний метод статичного класу, який можна викликати як метод екземпляра іншого класу.
- Перший параметр Extension Method має модифікатор this та визначає тип, який розширюється.
- Приклад.

```
public static class StringHelper {
 public static bool IsCapitalized( this string s) {
        if( string.IsNullOrEmpty(s)) return false;
          return char.IsUpper( s[0]);
 }
}
```

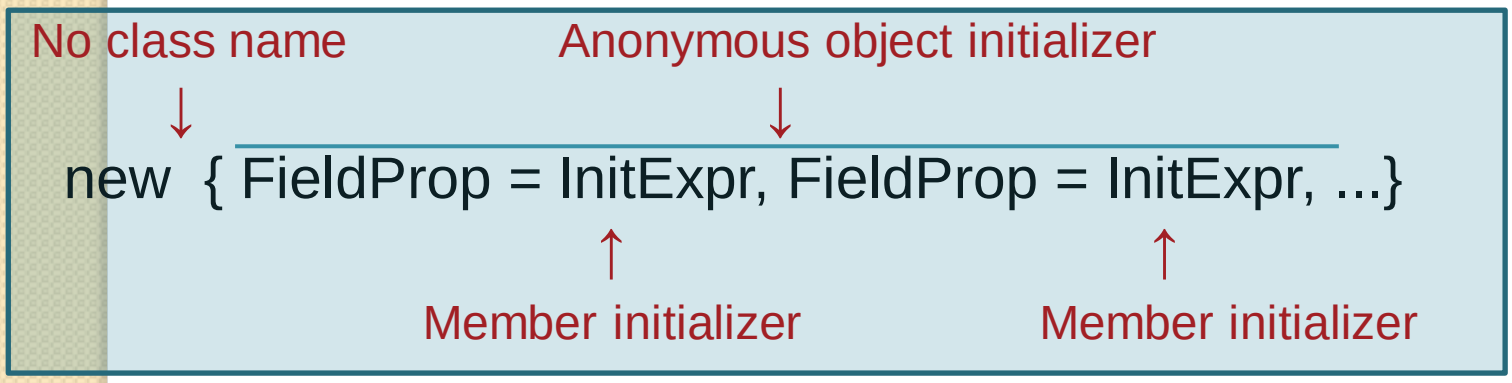
Застосування: Console.Write(“Perth”.IsCapitalized());

Розширюючі методи

- При роботі з LINQ нам рідко треба буде створювати власні розширюючі методи, а треба буде використовувати численні Extension Methods уже створені розробниками фірми Microsoft

Анонімні типи

- Вони застосовуються для швидкого моделювання “форми” даних, дозволяючи компілятору генерувати нове визначення класу під час компіляції на основі вказаного набору пар “ім’я / значення”




```
static void Main( )
 {
   var student = new {Name="Mary Jones", Age=19, Major="History"};
    Must use var                                  Anonymous object initializer
   Console.WriteLine("{0}, Age {1}, Major: {2}",
                     student.Name, student.Age, student.Major);
 }
```

Більш складний приклад

```
class Other
{
   static public string Name = "Mary Jones";
}
class Program
{
   static void Main()
   {
      string Major = "History";

      var student = new { Age = 19, Other.Name, Major};

      Console.WriteLine("{0}, Age {1}, Major: {2}",
                  student.Name, student.Age, student.Major);
   }
}
```

Assignment form

Identifier

Member access

Приклад

```
class MyData { public string FirstName {get;set;}
    public string LastName { get;set;}
    public DateTime DOB { get;set;}
    public string MiddleName { get; set;}
}
static void Main(string[] args) {
 List < MyData > data = new List < MyData > ();
data.Add(new MyData { FirstName = "Jignesh", LastName =
    "Trivedi", MiddleName = "G", DOB = new DateTime(1990,
    12, 30) });
  data.Add(new MyData { . . . . . });
var anonymousData = from pl in data   select new {
pl.FirstName, pl.LastName   };
foreach(var m in anonymousData) {
Console.WriteLine("Name : " + m.FirstName + " "
+ m.LastName);   }   }
```

LINQ

- **LINQ – це строго типізована мова запитів до довільних сховищ даних безпосередньо вбудована в граматику мови C#.**
- Гарним способом експериментувати з LINQ є завантаження LINQPad (http://www.linqpad.net/)

LINQ надає дві синтаксичні форми для задання запитів: синтаксис запиту (query syntax) та синтаксис методу (method syntax).

- Method syntax використовує стандартні виклики методів інтерфейсу IEnumerable<T> (більше 40)
- Query syntax дуже схожий на вирази запитів SQL.
- Ви можете комбінувати обидві форми в одному запиті.

Приклад

```
static void Main( )
  {
    int[] numbers = { 2, 5, 28, 31, 17, 16, 42 };
    var numsQuery = from n in numbers          // Query syntax
                    where n < 20
                    select n;
    var numsMethod = numbers.Where(x => x < 20);   // Method syntax
    int numsCount = (from n in numbers          // Combined
                    where n < 20
                    select n).Count();
    foreach (var x in numsQuery)
       Console.Write("{0}, ", x);
    Console.WriteLine();
    foreach (var x in numsMethod)
       Console.Write("{0}, ", x);
    Console.WriteLine();
    Console.WriteLine(numsCount);
  }
```

Результат:

2, 5, 17, 16,
2, 5, 17, 16,
4

Приклад

Дано масив рядків. Вивести в алфавітному порядку ті рядки, які містять пробіл.

```
string[] a = { “Morrowind”, “Uncharted 2”, “System Shock 2”};
IEnumerable<string> subset = from g in a
        where g.Contains(“ “)
        orderby g
        select g;
foreach( string s in subset)
        Console.WriteLine( “Item: {0}”, s);
```

Результат:

System Shock 2

Uncharted 2

Без LINQ буде в 2-3 рази довше (дивись Троельсена)

LINQ та розширюючі методи

- Хоча в попередньому прикладі не використовували явно розширюючі методи, вони насправді працюють на задньому плані
- **Вирази запитів LINQ можна застосовувати для ітерації по вмісту контейнера даних, який реалізує інтерфейс IEnumerable<T>**
- Однак клас System.Array не реалізує цей інтерфейс
- Незважаючи на це клас System.Array отримує необхідну функціональність через статичний тип класу System.Linq.Enumerable
  - методи Aggregate<T>, First<T>, Max<T> та інші
- Це можна перевірити, натиснувши крапку після імені змінної в Visual Studio 2012

Приклад запиту LINQ через розширюючі методи

```
Product[] products = { new Product{Name=…, Category=…,
   Price=…},
…… };
var results = products
                .OrderByDescending( e => e.Price)
                .Take(3)
                .Select( e => new { e.Name, e.Price});
foreach( var p in results){
   Console.WriteLine( “Item : {0}, Cost : {1}”, p.Name, p.Price);
```

Кожен з застосованих методів (розширюючих) приймає IEnumerable<T> і повертає IEnumerable<T>, це дозволяє об’єднувати методи в ланцюжки для створення складних запитів.

Роль відкладеного виконання

- **Запити LINQ не виконуються до тих пір, поки не буде почата ітерація по послідовності**
- Формально це називають відкладеним виконанням
- Переваги такого підходу в можливості **застосування одного і того ж запиту LINQ багаторазово** до одного і того ж контейнера з гарантією отримання актуальних результатів

```
int[] numbers = {10, 20, 30, 40, 1, 2, 3, 8};
var subset = from i in numbers where i<10 select i;
//Оператор LINQ тут виконується!
foreach( var i in subset)
    Console.WriteLine("{0} < 10", i);
    Console.WriteLine();
numbers[0] = 4;
//Оператор LINQ знову виконується!
foreach( var j in subset)
    Console.WriteLine("{0} < 10", j);
```

Роль негайного виконання

Для того, щоб виконати запит LINQ за межами foreach, можна викликати один з методів класу System.Linq.Enumerable, таких як

- ToArray<T>( )
- ToDictionary<TSource, TKey>( )
- ToList<T>( )

Всі ці методи змушують запит LINQ **виконуватися в момент виклику**, щоб отримати "знімок" даних (яким потім можна маніпулювати незалежно)

```
int[] numbers = {10, 20, 30, 40, 1, 2, 3, 8};
int[] subsetAsIntArray = (from i in numbers where i<10
    select i).ToArray<int>();

/*  або

List<int> subsetAsListOfInts = (from i in numbers where
    i<10 select i).ToList<int>();

*/
```

Застосування запитів LINQ до об'єктів колекцій

- Методика нічим не відрізняється. 1)Узагальнені колекції (Нэш)

```
class Employee {
    public string FirstName { get; set;}
    public string LastName { get; set;}
    public Decimal Salary{ get; set;}
    public DateTime StartDate { get; set;}
}
. . .
var employees = new List<Employee> {
    new Employee { FirstName="Joe", LastName="Bob",…},
    new Employee { FirstName="Jane", LastName="Doe",…},
…. };
var query = from e in employees
            where e.Salary>10000
            orderby e.LastName, e.FirstName
            select new { LastName = e.LastName,
                         FirstName = e.FirstName };
    foreach( var item in query){
        Console.WriteLine("{0}  {1}", item.LastName, item.FirstName);}
```

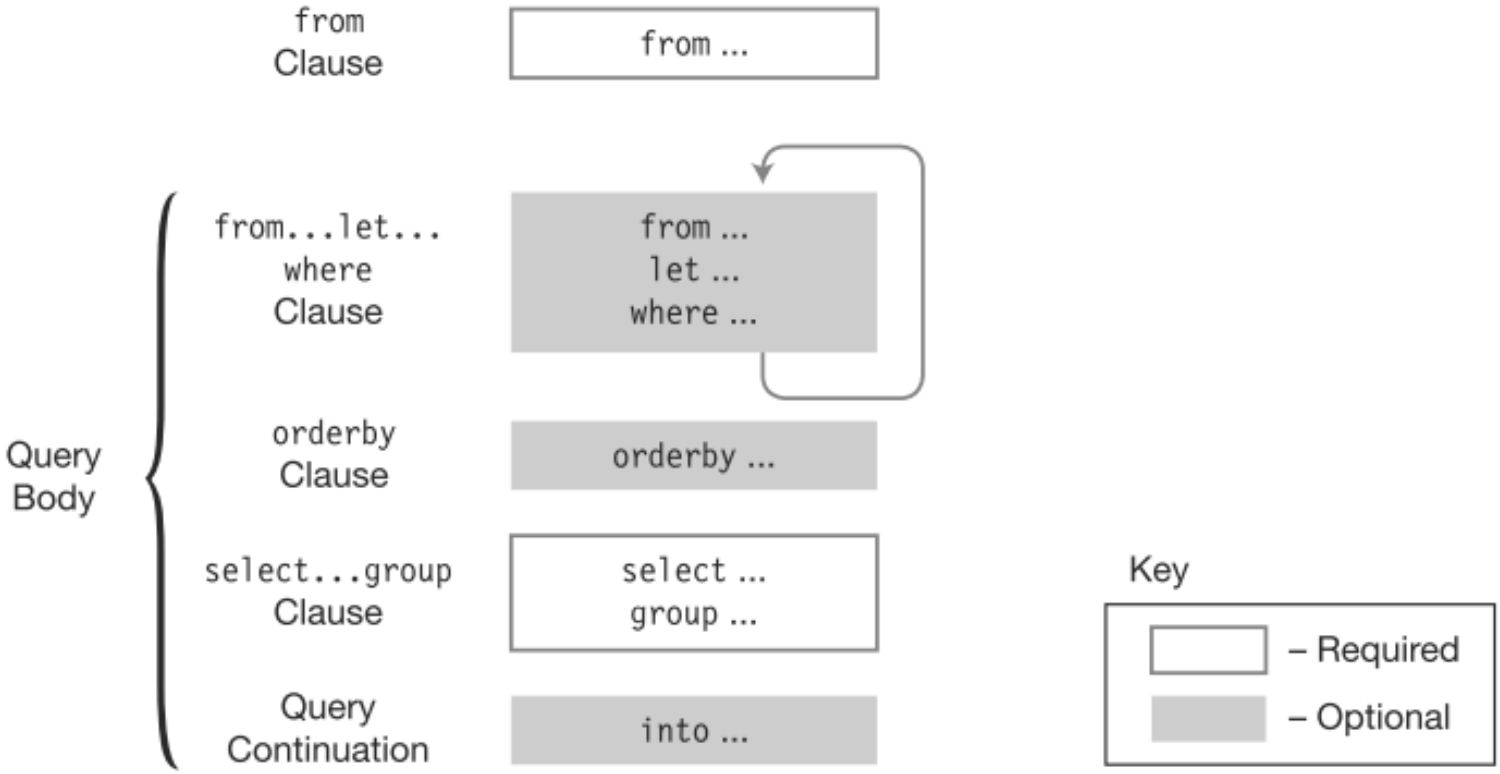
Неузагальнені колекції

- Вони не реалізують IEnumerable<T>
- Тому LINQ треба застосовувати через розширюючий метод Enumerable.OfType<T>()

Приклад.

```
ArrayList myCars = new ArrayList(){……..};
//Трансформуємо ArrayList в тип, сумісний з IEnumerable<T>
var myCarsEnum = myCars.OfType<Car>();
var fastCars = from c in myCarsEnum where c.Speed>55
    select c;
……..
```

Структура запиту (query expression)



- Обов'язковими є пункти from та select або group
- **Запит повинен починатися з ключового слова from і закінчуватися select або group**

Загальна форма from

Iteration variable declaration

↓

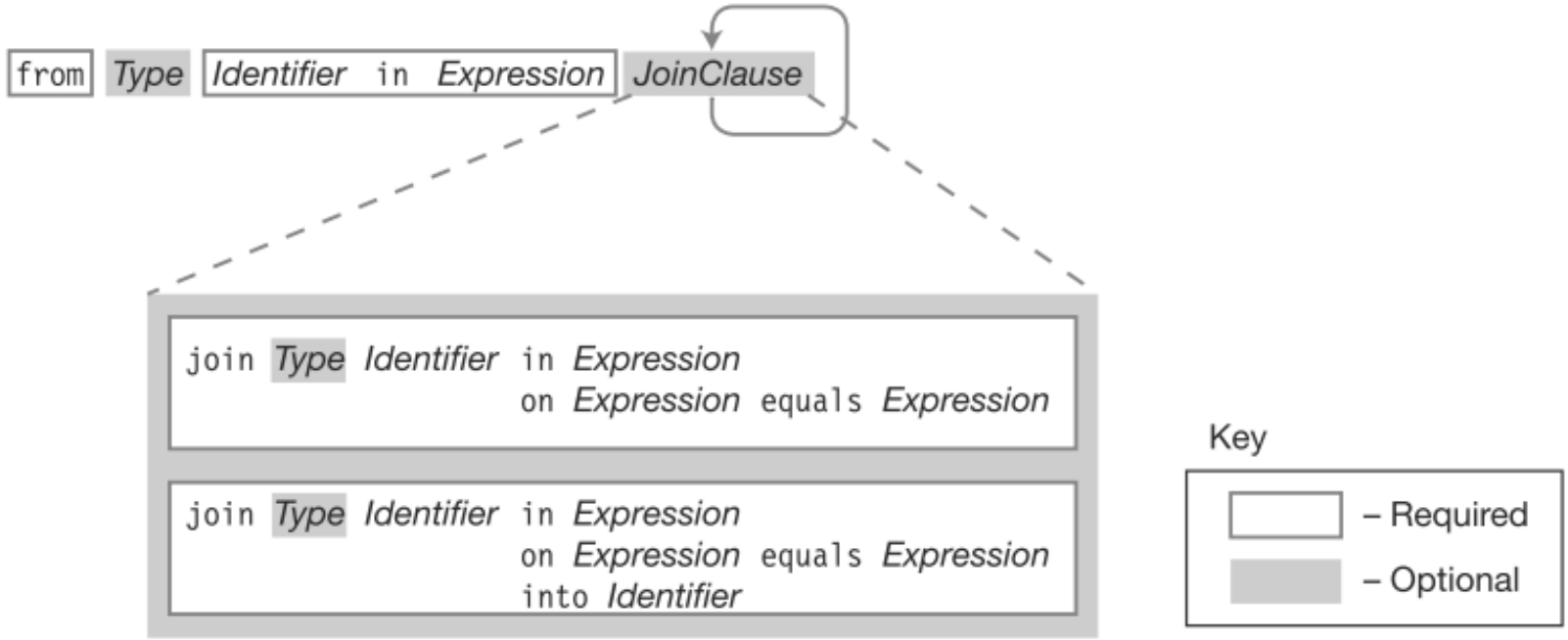
from Type Item in Items

- Item – це “змінна діапазону” (iteration variable) представляє кожен елемент в джерелі даних (data source)
- Type задавати не обов’язково
- Items – ім’я колекції, з якої запитуються дані

```
int[] arr1 = {10, 11, 12, 13};
var query = from item in arr1
            where item < 13
            select item;
foreach( var item in query )
  Console.Write("{0}, ", item );
```

Примітка.
Конструкцій from може бути декілька (Дивись: Нэш, стор.531)

The syntax of the from clause



- Конструкцію join (з'єднання) використовують для співставлення даних з двох різних джерел

Приклад використання join

Нэш, стор. 532 (фрагмент)

```
class EmployeeId{ public string Id{ get; set;} public string Name{ get; set;}}
class  EmployeeNationality{ public string Id{ get; set;}
        public string Nationality{ get; set;}}
……….
var employees = new List< EmployeeId>(){……};
var empNationalities = new List< EmployeeNationality>(){…….};
var query = from emp in employees
            join n in empNationalities
                on emp.Id equals n.Id
            orderby n.Nationality descending
            select new {
                Id = emp.Id,
                Name = emp.Name,
                Nationality = n.Nationality};
foreach (var person in query){ Console.WriteLine("{0}, {1}, /t{2}",
   person.Id, person.Name, person.Nationality);}
```

Конструкція select та проекція

Загальна форма оператора select

select Вираз

- В запиті LINQ конструкція select визначає **конкретний тип елементів, що отримуються по запиту**.
- Її називають **проектором**, тому що вона **проекціює**, або **транслює**, дані всередині запиту в форму, потрібну для використання.
- Часто в якості **Виразу** використовують просто **змінну діапазону**. Тоді трансформація не відбувається

```
double[] nums = { 16.4, 12.125, …..};
var sqrtRoot = from n in nums
               where n>0
               select Math.Sqrt(n);
```

Групування результатів за допомогою group

group змінна_діапазону by ключ

Цей оператор повертає дані, що згруповані в послідовності

Приклад.(Шилдт, стр.655). Дано масив рядків з адресами сайтів. Згрупувати сайти по імені домена верхнього рівня.

```
string[] websites  = { "hsNameA.com“, "hsNameD.com“, "hsNameG.tv",
    "hsNameB.net", "hsNameE.org", "hsNameH.net", "hsNameC.net",
    "hsNameF.org", "hsNamel.tv" };
var webAddrs = from addr in websites
                    where addr.LastIndexOf('.') != -1
                    group addr by addr.Substring(addr.LastIndexOf('.'));
foreach(var sites in webAddrs) {
      Console.WriteLine(“Ім’я домена " + sites.Key);
       foreach(var site in sites) Console.WriteLine(" " + site);
Console.WriteLine();
}
```

Групування результатів за допомогою group

Приклад (пояснення)

Тут в змінній webAddrs зберігається список груп (Key) (підтримується інтерфейс IGrouping<string, string>), тому для доступу до кожної групи потрібно два цикли foreach.

Конструкція into і продовження запиту

При використанні в запиті операторів select або group іноді треба сформувати тимчасовий результат, а потім продовжити запит.

Це можна зробити, використавши

into змінна_діапазону тіло_запиту

Конструкція into і продовження запиту

Приклад (невелика модифікація попереднього).

Виключити всі групи, що складаються менше, ніж з трьох елементів.

```
string[] websites = {……};
var webAddrs = from addr in websites
                where addr.LastIndexOf(‘.’) != -1
               group addr by addr.Substring(addr.LastIndexOf('.'))
                into ws
                where ws.Count() > 2
                select ws;
```

В результаті буде вибрана тільки група .net

Конструкція let

- Конструкція let дозволяє створювати в запиті локальну змінну, яку можна в подальшому використовувати

let ім'я = вираз

Приклад.

```
string[] strs = { “alpha”, “beta”, “gamma”};
// Запит на отримання символів в відсортованому вигляді
var chrs = from str in strs
           let chrArray = str.ToCharArray();
           from ch in chrArray
           orderby ch
           select ch;
foreach( char c in chrs) Console.Write(c + “ “);
```

Результат:

a a a a a b e g h l m m p t

Конструкція let

Приклад 2 (Удосконалений зі списком сайтів).

```
var webAddrs = from addr in websites
                    let idx = addr.LastIndexOf('.')
                    where idx != -1
                   group addr by addr.Substring(idx)
                   into ws
                   where ws.Count() > 2
                   select ws;
```

Тут індекс останнього входження символа ‘.’ присвоюється змінній idx. (Виграш – не треба другий раз його шукати)

Стандартні операції LINQ

Кількість операцій – 47. Їх можна розділити на 12 категорій.

Категорія	Опис
Filtering	Повертають subset of elements, що задов. умову
Projecting	Перетворюють кожний елемент у відпов. з виразом
Joining	Змішують (meshes) ел-ти колекції з іншою
Ordering	Переупорядковують послідовність
Grouping	Розбивають елементи на групи
Set	Об'єднують дві послід-ті одного типу, ...
Element	Вибирають один елемент
Aggregating	Виконують обчислення, повертають скалярну величину
Quantifiers	Виконують обчислення, повертають true або false
Conversion: Import	Конвертують не-generic в generic
Conversion: Export	Конвертують в масив, список, словник
Generation	Генерують прості послідовності

Операції, що фільтрують

Метод	Опис
Where	Повертають subset of elements, що задов. умову
Take	Повертає перший елемент
Skip	Ігнорує перший елемент
TakeWhile	Виділяє елементи, поки задовол. умова
SkipWhile	Ігнорує елементи, поки задовол. умова
Distinct	Повертає колекцію без дублікатів

Об'єднання, різниця, конкатенація та перетин даних

```
List<string> myCars = new List<string>{“Yugo”, “Aztec”, “BMW”};
List<string> yourCars = new List<string>{“BMW”, “Saab”, “Aztec”};
var carDiff = ( from c in myCars select c)
                    .Except( from c2 in yourCars select c2);
foreach( string s in carDiff) Console.WriteLine( s);
```

Виведе: Yugo

(Примітка: “Except” – за винятком)

Метод Intersect() виведе спільні елементи (Aztec, BMW)

Метод Union() дає об'єднання (BMW – один раз)

Метод Concat() дає конкатенацію (BMW – два рази)

Повернення результатів запиту LINQ

(Троельсен, с.476)

В класі можливо визначити поле, значенням якого буде результат запиту LINQ.

Для таких полів var використовувати не можна (треба IEnumerable<T>)

```
class Program{
  static void Main(){
  IEnumerable<string> subset = GetStringSubset();
    ……  }
```

```
static IEnumerable<string> GetStringSubset() {
   ……..
 }
```

Другий варіант – повернення через негайне виконання

```
static string[] GetStringSubsetAsArray(){
var r = from ….; return r.ToArray();
}
```

Динамічна ідентифікація типів та рефлексія

- **Динамічна ідентифікація типів** (RTTI, Run-Time Type Information) – представляє собою механізм, що дозволяє визначати тип даних під час виконання програми.
- **Рефлексія** (Reflection) – це засіб для отримання відомостей про тип даних.
- Отримавши ці дані, можна конструювати і застосовувати об'єкти під час виконання.

- Більшість програм написані, щоб працювати з даними
- Дані про програми та їх класи називаються метаданими і зберігаються в програмних зборках (складення, assembly)
- Програма може дивитися на метадані інших зборок або на свої, в той час як вона працює

Коли працююча програма заглядає у власні метадані, або метадані інших програм (зборок), це називається відображенням або рефлексією

Динамічна ідентифікація типів

(Шилдт, стор. 537)

- Для її підтримки в C# передбачені три ключових слова: is, as та typeof

- Перевірка типу за допомогою операції is

  Вираз is тип

- Приведення до типу

  Вираз as тип

Застосування операції typeof

- Операції as та is лише перевіряють сумісність типів
- Операція typeof дає інформацію про сам тип
- Вона формує (повертає) об'єкт класу System.Type для заданого типу

  Type t = typeof(Тип);
- Цей об'єкт містить ("відображає") інформацію при Тип

Приклад застосування операції typeof

```
using System;
using System.IO;
class UseTypeof
{
  static void Main()
  {
    Type t = typeof( StreamReader);
    Console.WriteLine( t.FullName);
    if( t.IsClass) Console.WriteLine("Є класом");
    if( t.IsAbstract) Console.WriteLine("Є абстрактним класом");
    else Console.WriteLine("Є конкретним класом");
  }
}
```

Рефлексія

- **Це засіб (інструмент), що дозволяє отримати відомості про тип.**
- Термін “рефлексія”, або “відображення” походить від принципу дії цього засобу
- **Об’єкт класу Type відображає тип, який нас цікавить**
- Для отримання інформації про тип об’єкту класу Type подають запити, а він повертає (“відображає”) зворотну інформацію
- Для застосування рефлексії треба підключити відповідний простір імен:

```
using System.Reflection;
```

Клас System.Type

- Це ядро підсистеми рефлексії
- Він є абстрактним класом і є похідним від абстрактного класу System.Reflection.MemberInfo
- В класі MemberInfo визначені такі властивості (тільки для читання)

Властивість	Опис
Type DeclaringType	Тип класу, в якому оголошено член
MemberTypes MemberType	Тип члену (Поле, метод, властивість, подія, …)
int MetadataToken	Значення, пов'язане з конкретними метаданими
Module Module	Модуль (файл), в якому знаходиться тип
string Name	Ім'я типу
Type ReflectionType	Тип відображуваного об'єкта

В клас MemberInfo входять два абстрактних методи: GetCustomAttributes() та IsDefined()

Методи класу Type

Клас Type додає немало своїх власних методів та властивостей.

Метод	Призначення
ConstructorInfo[] **GetConstructors()**	Отримати список конструкторів
EventInfo[] **GetEvents()**	Отримати список подій
FieldInfo[] **GetFields()**	Отримати список полів
MetodInfo[] **GetMethods()**	Отримати список методів
PropertyInfo[] **GetProperties()**	Отримати список властивостей
MemberInfo[] **GetMembers()**	Отримати список членів
Type[] **GetGenericArguments()**	Список generic-аргументів

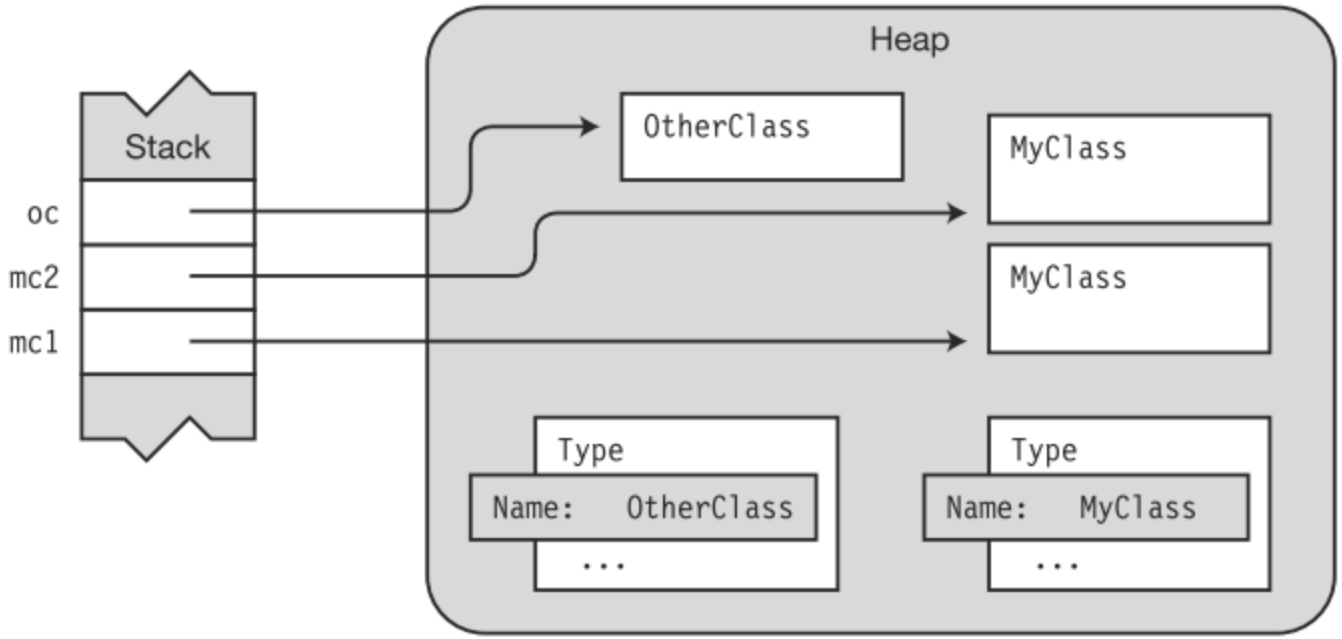
Властивості класу Type: Assembly, Attributes, BaseType, FullName, IsAbstract, IsArray, IsClass, IsEnum, ….

Через те, що Type є абстрактним класом, не можуть існувати його екземпляри (об'єкти, instances).

Під час виконання CLR створює екземпляри класу, похідного від Type (RuntimeType), а повертає посилання на Type

Клас Type

- **Для кожного типу, використовуваного в програмі, CLR створює об'єкт Type, який містить інформацію про цей тип.**
- Кожен тип, що використовується в програмі, пов'язаний з окремим об'єктом типу.
- Незалежно від кількості екземплярів типу, які створюються, є тільки один об'єкт Type, пов'язаний з усіма екземплярами.



Отримання об’єкта Type

Ви можете отримати об'єкт Type за допомогою методу GetType() або за допомогою операції typeof

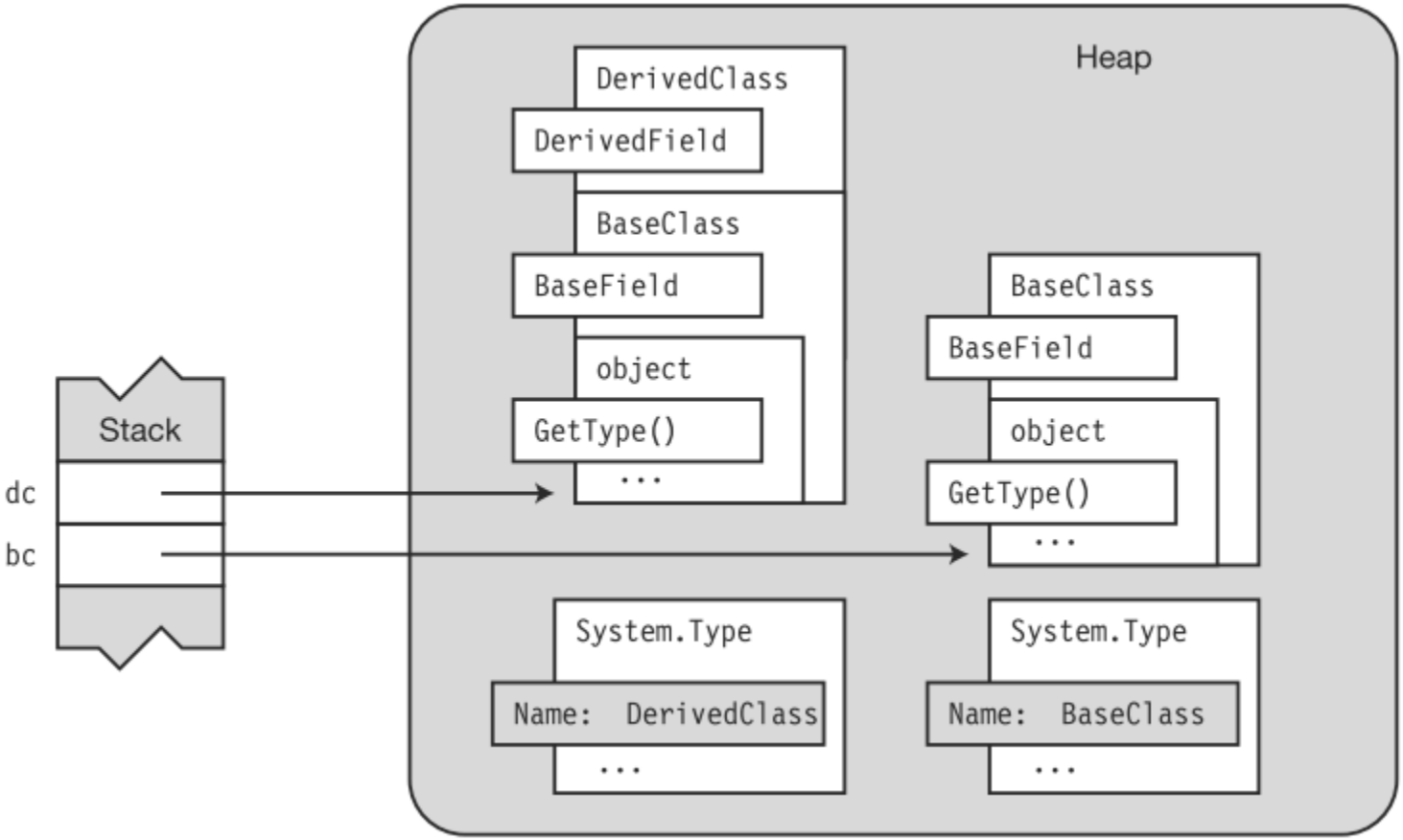
```
using System;
using System.Reflection;

   class BaseClass
   {
      public int BaseField = 0;
   }

   class DerivedClass : BaseClass
   {
      public int DerivedField = 0;
   }

   class Program
   {
      static void Main( )
      {
         var bc = new BaseClass();
         var dc = new DerivedClass();
```

Отримання об’єкта Type

```
BaseClass[] bca = new BaseClass[] { bc, dc };
        foreach (var v in bca)
        {
          Type t = v.GetType();     // Get the type.
          Console.WriteLine("Object type : {0}", t.Name);
          FieldInfo[] fi = t.GetFields();    // Get the fields
          foreach (var f in fi)
            Console.WriteLine("      Field : {0}", f.Name);
          Console.WriteLine();
        }
    }
  }
```

```
Object type : BaseClass
     Field : BaseField
Object type : DerivedClass
     Field : DerivedField
     Field : BaseField
```

Отримання об’єкта Type



Застосування рефлексії

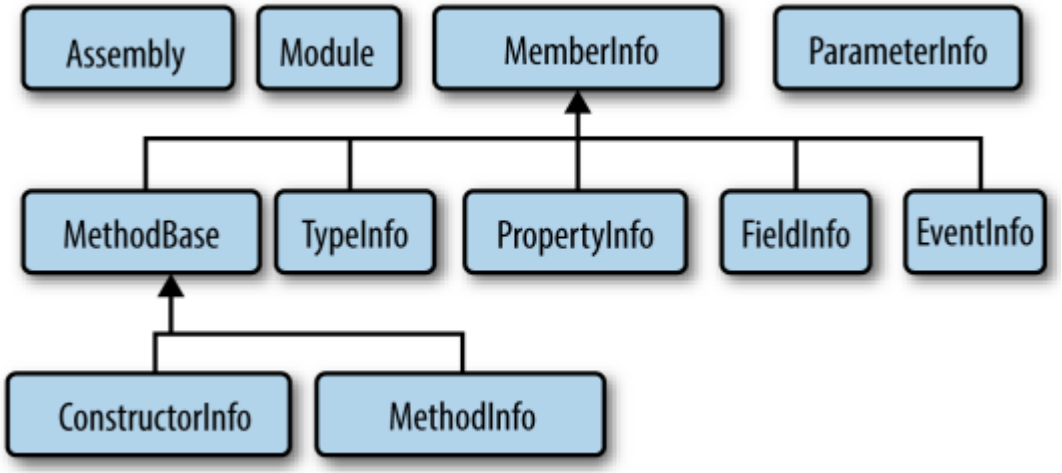
- (Шилдт, стор. 543) **Рефлексія дозволяє використовувати код, що був недоступним під час компіляції**
- “Щоб все перерахувати потрібна окрема книжка“

Далі ми розглянемо 4 основні способи застосування рефлексії:

- **Отримання інформації про методи**
- **Виклик методів**
- **Конструювання об’єктів**
- **Завантаження типів даних із зборок**

Отримання інформації про методи

- Треба застосувати метод GetMethods() класу Type
- Він повертає масив об'єктів класу MethodInfo
- Цей клас є похідним від абстрактного класу MethodBase, що в свою чергу спадкує від MemberInfo
- Це дає можливість використати методи і властивості цих трьох класів



Отримання інформації про методи

- Для отримання імені метода слугує властивість Name
- Клас MethodInfo має два цікаві члени: ReturnType та GetParameters()
  - ParameterInfo[] GetParameters();
- Відомості про параметри містить об'єкт класу ParameterInfo

Приклад (Шилдт, стор. 544).

```
using System;
using System.Reflection;
class MyClass {
    int x;
    int y;
    public MyClass(int i, int j) {
            x = i;
            У = j;
    }
```

Приклад (продовження)

```
public int Sum() {
    return x+y;
}
public bool IsBetween(int i) {
    if(x < i && i < y) return true;
    else return false;
}
public void Set(int a, int b) {
    x = a;
    y = b;
}
public void Set(double a,
    double b) {
    x = (int) a;
    y = (int) b;
}
```

```
public void Show() {
   Console.WriteLine(" x: {0}, y:
{1}", x, y);
}
}
class ReflectDemo {
  static void Main() {
    Type t = typeof(MyClass);
     Console.WriteLine("Аналіз
методів, визначених" +
"в класі " + t.Name);
      Console.WriteLine() ;

Console.WriteLine(«Підтримува
ні методи: ");
```

Приклад (продовження)

```
MethodInfo[] mi = t.GetMethods();
   // Вивести методи, підтримувані в класі MyClass
foreach(MethodInfo m in mi) {
   // Вивести тип, що повертається та ім'я кожного метода
 Console.Write(" " + m.ReturnType.Name + " " + m.Name
   +"(");
   // Вивести параметри
 ParameterInfo[] pi = m.GetParameters() ;
 for (int i=0; i < pi.Length; i++) {
    Console.Write( pi[i].ParameterType.Name + " " +
   pi[i].Name);
    if(i+I < pi.Length) Console.Write(", ");
 }
 Console.WriteLine(")");
 Console.WriteLine();
}
}
}
```

Приклад (закінчення)

Аналіз методів, визначених в класі MyClass

Підтримувані методи:

Int32 Sum ()

Boolean IsBetween(Int32 i)

Void Set(Int32 a, Int32 b)

Void Set (Double a, Double b)

Void Show()

String ToString()

Boolean Equals(Object obj)

Int32 GetHashCode()

Type GetType()

Виклик методів за допомогою рефлексії

- **Отримання інформації про методи**
- **Виклик методів**
- **Конструювання об'єктів**
- **Завантаження типів даних із зборок**

Як тільки методи даного типу стають відомими, ми можемо їх викликати. Для цього слугує метод Invoke(), що входить до складу класу MethodInfo

Одна з форм цього метода:

object Invoke(object obj, object[] parameters)

- Де obj – посилання на об'єкт, для якого викликається метод (null для статичного методу)
- parameters – параметри, які передаються методу

Приклад

```
using System;
using System.Reflection;
class MyClass {
    int x;  int y;
……….
}
class InvokeMethDemo {
    static void Main() {
            Type t = typeof(MyClass);
            MyClass reflectOb = new MyClass(10, 20);
            int val;
            Console.WriteLine("Виклик методів, визначених в
    класі " + t.Name),
            Console.WriteLine();
            MethodInfo[] mi = t.GetMethods();
```

Приклад (продовження)

```
// Викликати кожен метод
foreach(MethodInfo m in mi) {
    // Отримати параметри
    ParameterInfo[] pi = m.GetParameters() ;
    if (m.Name.CompareTo("Set")==0 &&
          pi[0].ParameterType == typeof(int)) {
          object[] args = new object[2];
          args[0] = 9;
          args[l] = 18;
          m. Invoke(reflectOb, args);
          }
……….
```

Отримання типів даних зі зборок

- **Отримання інформації про методи**
- **Виклик методів**
- **Конструювання об'єктів**
- **Завантаження типів даних із зборок**

Зборка (англ. assembly) – це двійковий файл EXE або DLL, який містить виконуваний код програми. Зборка містить номер версії, метадані та інструкції IL (intermediate language)

В попередніх прикладах ми всі відомості про клас MyClass отримували за допомогою рефлексії, а саме ім'я класу було відомо заздалегідь.

Рефлексія дозволяє завантажувати зборку і дізнаватися про всі типи даних.

Отримання типів даних зі зборок

- **Зборці відповідає клас *Assembly***
- Щоб завантажити зборку, треба викликати статичний метод **LoadFrom()** цього класу
- static Assembly LoadFrom(string файл_зборки)
- Для отримання типів слід скористатися методом **GetTypes()**

Приклад (Шилдт)

```
……….
Assembly asm = Assembly.LoadFrom(“MyClasses.exe”);
Type[] alltypes = asm.GetTypes();
foreach( Type t in alltypes)
   Console.WriteLine( “Знайдено: “ + t.Name);
……….
```

Зборки (assembly) в .NET

Зборка – це двійковий (відкомпільований) файл, що містить керований код. Зборки мають розширення .EXE або .DLL.

Формат зборки (Троельсен, стор. 502):

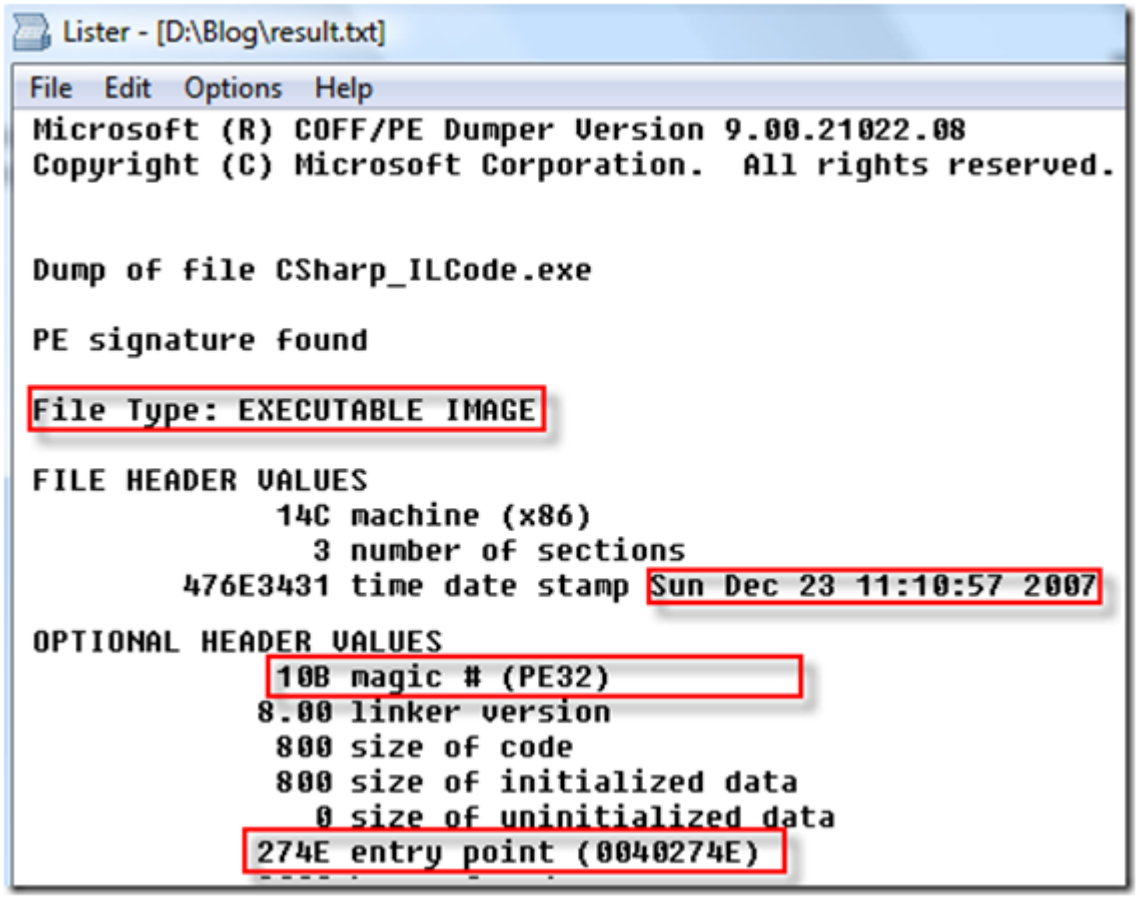
- **Заголовок** файлу Windows
- **Заголовок** файлу CLR
- **CIL-код**
- **Метадані** типів
- **Маніфест** зборки
- Додаткові вбудовані **ресурси**

Щоб подивитись заголовки – утиліта ***dumpbin.exe*** (в вікні командного рядка Visual Studio):

```
dumpbin  /headers CarLibrary.dll
```

Формат зборки

dumpbin /all assembly_name > result.txt



Для вивчення маніфесту, CIL-коду та метаданих можна використати утиліту ildasm.exe

Зборки (assembly) в .NET

- **Зборки дають можливість повторного використання коду**
  - Типи даних б-ки .NET містяться в зборках *mscorlib.dll* та *System.Windows.Forms.dll*
- **Зборки визначають межі (границі) типів**
- **Зборки підтримують версії**
  - Формат <ст.номер>.<мол.номер>.<номер_зборки>.<редакція>
  - Наприклад 1.0.0.0
- **Зборки є самоописувані**
  - ***Маніфест*** – це блок метаданих, що описує саму зборку
  - Тому CLR не звертається до системного реєстру
- **Зборки конфігуруються**
  - Зборки бувають **приватні** (private) і **загальні** (shared)
  - Приватні – в каталозі застосунку, загальні – в каталозі з назвою ***глобальний кеш зборок*** (Global Assembly Cache - GAS)
  - Для конфігурування – XML-файли

Тип dynamic в мові C# 4.0

(Шилдт, стор. 703)

- Мова C# є строго типізованою
- Всі операції перевіряються під час компіляції на відповідність типів
- **Тип dynamic дозволяє *пропускати перевірку типів* під час компіляції операцій, в яких він застосовується. Це відкладається на період виконання**

Тип dynamic:

- Спрощує доступ до API моделей COM, бібліотекам IronPython, до моделі DOM (HTML), …

Тип dynamic в мові C# 4.0

- **В більшості випадків тип dynamic поводить себе однаково з типом object** (їх можна використати для посилань на довільні об'єкти)
- ***Але для типу object відповідність типів перевіряється при компіляції***

```
using System;
class Program{
   static void Main(){
     dynamic dyn = 1;
     object obj = 1;
     Console.WriteLine( dyn.GetType());
     Console.WriteLine( obj.GetType());
   }
}
```

Якщо додати obj = obj + 3, то буде помилка

Якщо додати dyn = dyn + 3, то все буде працювати

Результат: System.Int32 System.Int32

Тип dynamic в мові C# 4.0

- **Тип dynamic може спростити використання рефлексії:** ***метод об'єкту можна викликати за його іменем, а не за допомогою методу Invoke()***

Приклад (Шилдт, с. 705) Файл MyClass.cs :

```
public class DivBy{
    public bool IsDivBy( int a, int b){
            if( (a%b) == 0) return true;
            return false;
    }
    public bool IsEven( int a){
            if( (a%2)==0) return true;
            return false;
    }
}
```

Скомпілюємо цей файл в бібліотеку DLL (MyClass.dll)

Приклад (продовження)

Далі складаємо програму:

```
using System;
using System.Reflection;

class DynRefDemo {
   static void Main() {

   Assembly asm = Assembly.LoadFrom("MyClass.dll") ;

   Type[] all = asm.GetTypes() ;
     // Знайти клас DivBy
   int i;
   for(i =0; i < all.Length; i++)
         if(all[i].Name == "DivBy") break;
   if(i == all.Length) {
     Console.WriteLine("Клас DivBy не знайдено в сборці");
     return;
   }
```

Приклад (продовження)

```
Type t = all[i];
 //Знайти конструктор за замовчуванням
ConstructorInfo[] ci = t.GetConstructors();
int j ;
for(j =0; j < ci.Length; j++)
    if(ci[j].GetParameters().Length == 0) break;
if(j == ci.Length) {
    Console.WriteLine("Конструктор за замовчуванням не
          знайдено");
    return;
}
   // Створити об'єкт класу DivBy динамічно
dynamic obj = ci[j].Invoke(null);
```

Приклад (закінчення)

```
// Далі можемо викликати за іменем методи для змінної obj
if( obj.IsDivBy(15, 3) )
    Console.WriteLine("15 ділиться націло на 3");
else
    Console.WriteLine("15 НЕ ділиться націло на 3");
if( obj.IsEven(9) )
    Console.WriteLine("9 парне число.");
else
    Console.WriteLine("9 НЕ парне число.");
}
}
```

Робота з потоками в C#

(Шилдт, стор. 833)

- **Процес – це програма в стадії виконання**
- **В процесі є головний поток – виконавець коду програми**
- **Головний поток може створювати інші потоки, які будуть виконуватися паралельно**
- Підтримка потоків в C# була ще в старих версіях
- **В C# 4.0 з'явились два важливих доповнення:**
  - ***TPL – Task Parallel Library***
  - ***PLINQ – Parallel LINQ***
- Перш, ніж розглядати TPL, зупинимось на традиційному підході
- Класи, що підтримують багатопоточність, визначені в просторі імен System.Threading

Клас Thread

Клас Thread інкапсулює поток виконання. Це *герметичний* клас, від нього не можна спадкувати.

```
using System;
using System.Threading;
class ThreadTest{
    static void Main(){
            Thread  t = Thread( WriteY);
            t.Start();
            for (int i = 0; i < 1000; i++) Console.Write ("x");
    }
    static void WriteY(){
    for (int i = 0; i < 1000; i++) Console.Write ("y");
    }
}
```

Результат: xxx…xyyy…yxxx…xyyy…y….

Клас Thread

// Typical Output:

xxxxxxxxxxxxxxxyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyy

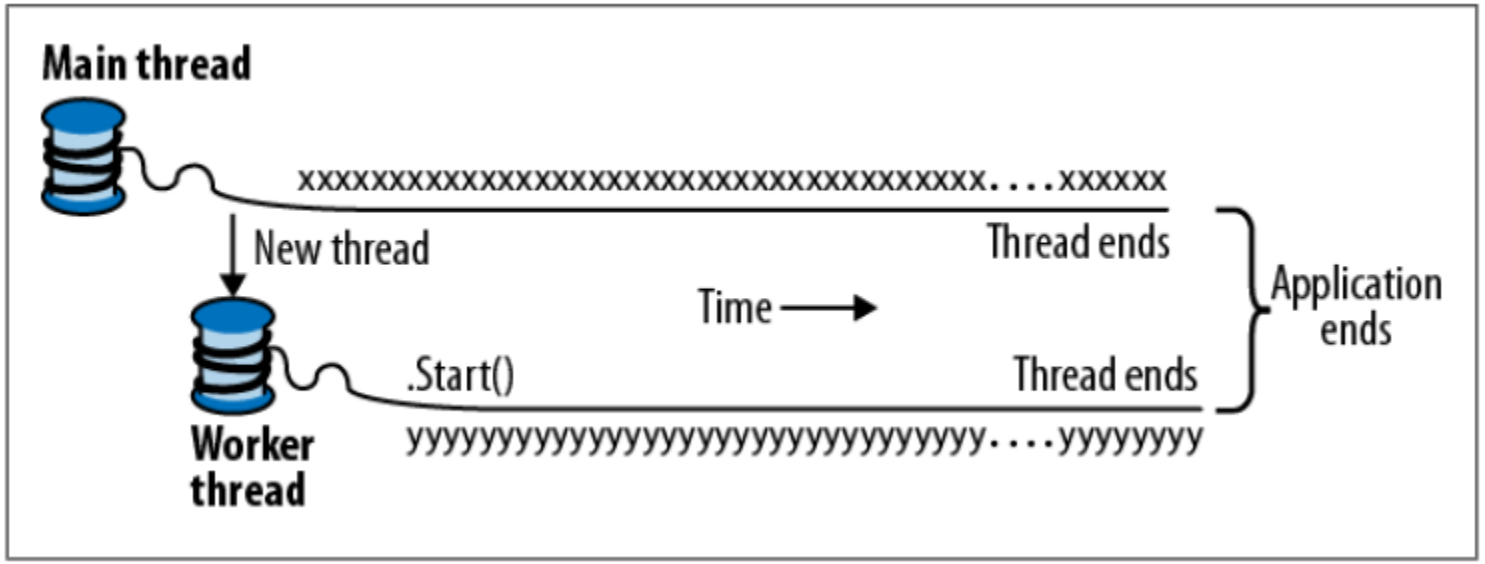
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxyyyyyyyyyyyyy

yyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyxxxxxxxxxxxxxxxxxxxxx

xxxxxxxxxxxxxxxxxxxxyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyy

yyyyyyyyyyyyyxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx

...



Клас Thread

Найпростіший конструктор класу Thread:

public Thread(ThreadStart запуск)

де запуск – це ім'я методу, що викликається для виконання в потоці,

а ThreadStart – делегат, визначений в середовищі .NET :

public delegate void ThreadStart()

Методи Join() та Sleep()

Метод Join() чекає до тих пір, поки поток, для якого він був визваний, не завершиться

Метод Sleep() здійснює затримку на задане число міллісекунд

Join and Sleep

```
static void Main()
{
  Thread t = new Thread (Go);
  t.Start();
  t.Join();
  Console.WriteLine ("Thread t has ended! ");
}
static void Go() { for (int i = 0; i < 1000; i++)
   Console.Write ("y"); }
```

Це друкує "y" 1000 разів, а потім " Thread t has ended!" відразу ж після цього.

Потоки

- Потокам можна **передавати параметри**
- Потоки можуть мати **пріоритети**
- Потоки можна **синхронізувати**
- Можна організувати **багатозадачність на основі процесів**
  - Для запуску нового процесу – клас Process
- Читайте Шилдта, …
- Це все ми ще будемо вивчати в курсі “Операційні системи”

Бібліотека TPL (Task Parallel Library)

Шилдт Г. C# 4.0: полное руководство. – Гл.24 с.885

Троельсен Э. Язык пр-я C# 2010 и платформа .NET 4 – с.700

- *TPL – самий важливий* з нових засобів .NET 4
- TPL представляють типи з простору імен System.Threading.Tasks
- **За допомогою TPL можна будувати паралельний код *без необхідності безпосередньої роботи з потоками***
- Починаючи з .NET 4 використання TPL є *рекомендованим способом* побудови багатопоточних застосунків

Два підходи до паралельного програмування

- **Паралелізм даних**
  - **Одна операція над сукупністю даних розбивається на декілька паралельно виконуваних потоків**
  - Кожен поток обробляє свою частину спільних даних
- **Паралелізм задач**
  - **Дві операції або більше виконуються паралельно**

Клас Task

- **Створити нову задачу у вигляді об'єкта класу Task і почати її виконання можна різними способами**.
- Для початку створимо об'єкт типу **Task** за допомогою конструктора і запустимо його, викликавши метод Start ().
- Скористаємось конструктором :

  public Task (Action дія)

  де **дія** – точка входу в код, який представляє задачу,

  Action – делегат, визначений в просторі імен **System**

  public delegate void Action ()

Приклад (Шилдт)

```
using System;
using System.Threading;
using System.Threading.Tasks;

class DemoTask
{
      static void MyTask() {  //Статичний метод
      Console.WriteLine("MyTask() запущено");
      for(int count = 0; count < 10; count++) {
      Thread.Sleep(500);
      Console.WriteLine("В методі MyTask(), рахунок
          дорівнює" +  count);
      }
       Console.WriteLine("MyTask завершено");
     }
```

Приклад (продовження)

```
static void Main()
{
  Console.WriteLine("Основний поток запущено");
  // Сконструювати об'єкт задачі
  Task tsk = new Task(MyTask);
  // Запустити задачу на виконання
  tsk.Start () ;
  // метод Main() активний до завершення методу MyTask()
  for(int i = 0; i < 60; i++) {
        Console.Write(".");
        Thread.Sleep(100);
  }
  Console.WriteLine("Основний поток завершено");
 }
}
```

Приклад (закінчення)

Основний поток запущено

.MyTask() запущено

.....В методе MyTask(), рахунок дорівнює 0

..... В методі MyTask(), рахунок дорівнює 1

..... В методі MyTask(), рахунок дорівнює 2

..... В методі MyTask(), рахунок дорівнює 3

..... В методі MyTask(), рахунок дорівнює 4

..... В методі MyTask(), рахунок дорівнює 5

..... В методі MyTask(), рахунок дорівнює 6

..... В методі MyTask(), рахунок дорівнює 7

..... В методі MyTask(), рахунок дорівнює 8

..... В методі MyTask(), рахунок дорівнює 9

MyTask завершено

Основний поток завершено

Модернізований приклад (Шилдт)

MyTask() – не обов'язково статичний метод

```
class MyClass {
    public void MyTask() {
            Console.WriteLine("MyTask() запущено");
            ……
    }
}
class DemoTask {
    static void Main() {
            Console.WriteLine("Основний поток запущено");
            MyClass mc = new MyClass();
            Task tsk = new Task(mc.MyTask);
            tsk.Start();
            …….  }
}
```

Ідентифікатор задачі

- На відміну від класу **Thread**, в класі **Task** відсутня властивість **Name** для зберігання імені завдання.
- Але в класі **Task** є властивість **Id** для зберігання ідентифікатора завдання:

```
public int Id {get; }
```

- Кожна задача отримує ідентифікатор, коли вона створюється.
- Значення ідентифікаторів унікальні, але не впорядковані.
- Ідентифікатор виконуваного зараз завдання можна виявити за допомогою властивості **CurrentId**.

```
public static Nullable <int> CurrentID {get; }
```

Приклад

```
using System;
using System.Threading;
using System.Threading.Tasks;
class DemoTask {
  static void MyTask() {
    Console.WriteLine("MyTask() №"+Task.CurrentId + " запущено");
    for (int count = 0; count < 10; count++) {
    Thread.Sleep(500);
    Console.WriteLine("В методі MyTask() #" + Task.CurrentId +
       ", рахунок дорівнює" + count );
     }
    Console.WriteLine("MyTask №" + Task.CurrentId + "завершено");
  }
```

Приклад

```
static void Main() {
  Console.WriteLine("Основний поток запущено");
  Task tsk = new Task(MyTask);
  Task tsk2 = new Task(MyTask);
  tsk.Start();
  tsk2.Start();
  Console.WriteLine(«Ідентифікатор задачі tsk: " + tsk.Id);
  Console.WriteLine(«Ідентифікатор задачі tsk2: " + tsk2.Id);
  for(int i = 0; i < 60; i++) {
    Console.Write(".");
    Thread.Sleep(100);
  }
  Console.WriteLine("Основний поток завершено");
 }
}
```

Застосування методів очікування

- В попередніх прикладах основний поток очікував завершення завдань за допомогою Thread.Sleep()
- Є більш досконалий спосіб: **метод Wait() класу Task**

Переробка попереднього прикладу:

Замість

```
for(int i = 0; i < 60; i++) {
      Console.Write(".");
      Thread.Sleep(100);
   }
```

краще

```
tsk.Wait() ;
tsk2.Wait() ;
```

Застосування класу TaskFactory для запуску завдання

- **Задачу можна створити і відразу ж почати її виконання, викликавши метод StartNew (), визначений у класі TaskFactory.**
- Нижче приведена найпростіша форма його оголошення:

```
public Task StartNew(Action action)
```

- **За замовчуванням об'єкт класу TaskFactory може бути отриманий *з властивості Factory*, доступного тільки для читання в класі Task.** Використовуючи цю властивість, можна викликати будь-які методи класу TaskFactory.
- Наприклад, наступний виклик методу StartNew () в що розглядалися раніше програмах призведе до створення і запуску завдання tsk однією дією.

```
Task tsk = Task.Factory.StartNew(MyTask);
```

Застосування лямбда-виразів як завдань

- Нагадаємо, що лямбда-вирази є особливою формою анонімних функцій.

```
using System;
using System.Threading;
using System.Threading.Tasks;
class DemoLambdaTask {
    static void Main() {
    Console.WriteLine("Основной поток запущено");
    // Далі лямбда-вираз використовується для визначення завдання.
    Task tsk = Task.Factory.StartNew( () => {
            Console.WriteLine("Задача запущена");
            for(int count = 0; count < 10; count++) {
                    Thread.Sleep (100);
                    Console.WriteLine("Рахунок в задачі дорівнює" + count );
            }
            Console.WriteLine("Задача завершена");
            } );
tsk.Wait ();
tsk.Dispose();
Console.WriteLine("Основний поток завершено");
}
}
```

Застосування лямбда-виразів як завдань

```
// Далі лямбда-вираз використовується для визначення завдання.
Task tsk = Task.Factory.StartNew( () => {
        Console.WriteLine("Задача запущена");
        for(int count = 0; count < 10; count++) {
          Thread.Sleep(100);
          Console.WriteLine("Рахунок в задачі
          дорівнює" + count );
        }
        Console.WriteLine("Задача завершена");
      } );
```

Повернення значення з завдання

- **Завдання може повертати значення**.
- Це дуже зручно з двох причин.
  - По-перше, це означає, що **за допомогою завдання можна обчислити деякий результат** (Подібним чином підтримуються паралельні обчислення)
  - І по-друге, **головний процес виявиться блокованим доти, поки не буде отриманий результат** (Це означає, що для організації очікування результату не потрібно ніякої особливої синхронізації)

Повернення значення з завдання

Для того щоб повернути результат з задачі, досить створити цю задачу, використовуючи узагальнену форму Task<TResult> класу Task

Ми маємо два конструктори:

public Task(Func<TResult> функція)

public Task(Func<Object, TResult> функція, Object стан)

Варіанти методу StartNew():

public Task<TResult> StartNew(Func<TResult> функція)

public Task<TResult> StartNew(Func<0bject,TResult> функція, Object стан)

Значення, що повертається завданням, виходить з **властивості Result** в класі Task

public TResult Result { get; internal set; }

Приклад

```
using System;
using System.Threading;
using System.Threading.Tasks;
class DemoTask {
  static bool MyTask() {
   return true;
  }
  static int SumIt(object v) {
   int x = (int) v;
   int sum = 0;
   for(; x > 0; x --)
   sum += x;
   return  sum;
  }
```

```
static void Main() {
Console.WriteLine("Основний
   поток запущено");
// Сконструювати об'єкт першої
   задачі
Task<bool> tsk =
   Task<bool>.Factory.StartNew(
   MyTask);
Console.WriteLine("Результат
   після виконання задачі
   MyTask: " +
tsk.Result);
```

Приклад

```
// Сконструювати об’єкт другої задачі
Task<int> tsk2 = Task<int>.Factory.StartNew(SumIt, 3);
Console.WriteLine(" Результат після виконання SumIt: " +
    tsk2.Result);
tsk.Dispose();
tsk2.Dispose();
Console.WriteLine("Основний поток завершено");
}
}
```

```
Основний поток запущено
Результат після виконання задачі MyTask: True
Результат після виконання SumIt: 6
Основний поток завершено
```

Клас Parallel

- Є одним з головних в TPL
- **Спрощує паралельне виконання коду і надає методи, що раціоналізують обидва види паралелізму: даних і задач**
- **Клас Parallel є статичним, і в ньому визначені методи For(), ForEach() і Invoke()**
- У кожного з цих методів є різні форми.
- Метод **For()** виконує розпаралелювання циклу for
- Метод **ForEach()** - розпаралелює цикл foreach (обидва методи підтримують **паралелізм даних**)
- Метод **Invoke()** підтримує паралельне виконання двох методів або більше.

Розпаралелювання задач методом Invoke()

- Метод **Invoke ()**, визначений у класі **Parallel**, дозволяє **паралельно виконувати один або кілька методів**, що вказуються у вигляді його аргументів.
- Він також **масштабує виконання коду**, використовуючи доступні процесори, якщо є така можливість.
- Нижче приведена найпростіша форма його оголошення:

  public static void Invoke (params Action [] actions)
- делегат Action оголошується наступним чином:

  public delegate void Action ()
- Після Invoke() не треба викликати метод Wait()

Приклад

```
using System;
using System.Threading;
using System.Threading.Tasks;
class DemoParallel {
    // Метод, що виконується як задача,
    static void MyMeth() {
        Console.WriteLine("MyMeth запущено");
        for(int count = 0; count < 5; count++) {
            Thread.Sleep(500);
            Console.WriteLine("В методі MyMeth
                рахунок дорівнює " + count );
        }
        Console.WriteLine("MyMeth завершено");
    }
```

Приклад

```
static void MyMeth2() {
   Console.WriteLine("MyMeth2 запущено");
   for(int count = 0; count < 5; count++) {
     Thread.Sleep(500);
     Console.WriteLine("В методі MyMeth2, рахунок" +
         count );
   }
   Console.WriteLine("MyMeth2 завершено");
}
static void Main() {
   Console.WriteLine("Основний поток запущено");
   Parallel.Invoke(MyMeth, MyMeth2);
   Console.WriteLine("Основний поток завершено");
 }
}
```

Зауваження. Виконання методу Main() тут призупиняється

Застосування методу For ()

- В TPL **паралелізм даних** підтримується за допомогою методу **For ()**, визначеного в класі **Parallel**
- Цей метод існує в декількох формах
- Одна з форм:

```
public static ParallelLoopResult For(int fromInclusive,
    int toExclusive, Action<int> body)
```

- де fromInclusive позначає *початкове значення індексу*
- toExclusive значення, *на одиницю більше кінцевого*
- body – метод, який буде циклічно виконуватись (сумісний з делегатом Action<int>)

```
public delegate void Action<int>(int obj)
```

Головна особливість методу For () полягає в тому, що він дозволяє, коли така можливість є, розпаралелити виконання коду в циклі.

Приклад

```
using System;
using System.Threading.Tasks;
class DemoParallelFor {
   static int[] data;
   static void MyTransform(int i)
    {   //імітує обробку
   data[i] = data[i] / 10;
if(data[i] < 10000) data[i] = 0;
if(data[i] > 10000 & data[i] <
    20000) data[i] = 100;
if(data[i] > 20000 & data[i] <
    30000) data[i] = 200;
if(data[i] > 30000) data[i] = 300;
    }
```

```
static void Main() {
   Console.WriteLine("Основний
          поток запущено");
   data = new int[100000000];
   // Ініціалізувати дані в звичайному циклі for
   for(int i=0; i < data.Length; i++)
         data[i] = i;
   // Розпаралелити цикл методом For()
   Parallel.For(0, data.Length,
                    MyTransform);
   Console.WriteLine("Основний
          поток завершено");
 }
}
```

Застосування методу ForEach ()

Використовуючи метод ForEach (), можна створити розпаралелений варіант циклу foreach.

```
public static ParallelLoopResult
ForEach<TSource>(IEnumerable<TSource> source,
Action<TSource> body)
```

де **source** позначає колекцію даних, оброблюваних в циклі, а **body** - метод, який буде виконуватися на кожному кроці циклу.

How to: Write a Simple Parallel.ForEach Loop (from MSDN)

```
using System;
using System.Drawing;
using System.IO;
using System.Threading;
using System.Threading.Tasks;

class SimpleForEach
{
    static void Main()
    {
        string[] files =
    System.IO.Directory.GetFiles(@"C:\Users\Public\Pictures\Sa
    mple Pictures", "*.jpg");

        string newDir = @"C:\Users\Public\Pictures\Sample
    Pictures\Modified";

        System.IO.Directory.CreateDirectory(newDir);
```

```
Parallel.ForEach(files, currentFile =>{
    string filename = System.IO.Path.GetFileName(currentFile);
    System.Drawing.Bitmap bitmap = new
    System.Drawing.Bitmap(currentFile);
    bitmap.RotateFlip(System.Drawing.RotateFlipType.Rotate180
    FlipNone);
    bitmap.Save(System.IO.Path.Combine(newDir, filename));
    Console.WriteLine("Processing {0} on thread {1}", filename,
    Thread.CurrentThread.ManagedThreadId);
      } //close lambda expression
        ); //close method invocation

      // Keep the console window open in debug mode.
       Console.WriteLine("Processing complete. Press any key to
    exit.");
       Console.ReadKey();
    }
}
```

Асинхронне програмування в C# 5.0

- **[1] C# 5.0 Карманный справочник, 2013. - стр. 252**
- [2] Multithreading in C# 5.0 , 2013 , p. 112
- [3] Illustrated C# 2012 , p. 535
- **[4] Троельсен Язык пр-я C# 5.0 и платф. .NET 4.5 6-е изд. стр. 694**

- **Що таке асинхронний виклик довго виконуваного метода? Це коли основна програма не блокується на час виконання методу, а сам метод виконується в окремому потоці.**

- **В версії C# 5.0 з'явились нові ключові слова await та async, призначені для підтримки асинхронного програмування.**

Приклад [1]

Розглянемо синхронний метод, пов'язаний з довгими обчисленнями:

```
int ComplexCalculation()
{
  double x = 2;
  for (int i = 1; i < 100000000; i++)
          x += Math.Sqrt (x) / i;
           return (int)x;
}
```

Цей метод блокує програму на кілька секунд

```
int result = ComplexCalculation();
// Sometime later:
Console.WriteLine (result);   // 116
```

Приклад [1]

Використовуючи TPL, можна створити об’єкт Task<TResult> для паралельного виконання довгої операції в окремому потоці

```
Task<int> ComplexCalculationAsync()
{
  return Task.Run (() => ComplexCalculation());
}
```

Цей метод є асинхронним, тому що він негайно повертає керування викликаючій стороні, паралельно продовжуючи свою роботу.

Однак нам потрібен якийсь механізм, що дозволяє викликаючій стороні вказувати, що має статися, коли операція закінчиться і стане відомим її результат.

Клас Task<TResult> вирішує цю проблему, надаючи доступ до методу GetAwaiter, який **дозволяє** викликаючій стороні **приєднувати продовження**.

Приклад [1]

```
Task<int> task = ComplexCalculationAsync();
var awaiter = task.GetAwaiter();
awaiter.OnCompleted (() =>        // Continuation
{
  int result = awaiter.GetResult();
  Console.WriteLine (result);      // 116
});
```

Ключові слова await та async

Ключове слово await спрощує приєднання продовження.

Компілятор розкриває конструкції

```
var result = await expression;
statement(s);
```

в код, функціонально подібний наступному:

```
var awaiter = expression.GetAwaiter();
awaiter.OnCompleted (() =>
{
  var result = awaiter.GetResult();
statement(s);
);
```

Ключові слова await та async

Отже, ми можемо викликати метод ComplexCalculationAsync, визначений вище, наступним чином:

```
int result = await ComplexCalculationAsync();
Console.WriteLine (result);
```

Для того, щоб скомпілювати цей код, необхідно додати модифікатор async в метод, який його містить,

```
async void Test()
{
  int result = await ComplexCalculationAsync();
  Console.WriteLine (result);
}
```

Модифікатор async інструктує компілятор інтерпретувати await як ключове слово, а не як ідентифікатор

Перехоплення локального стану

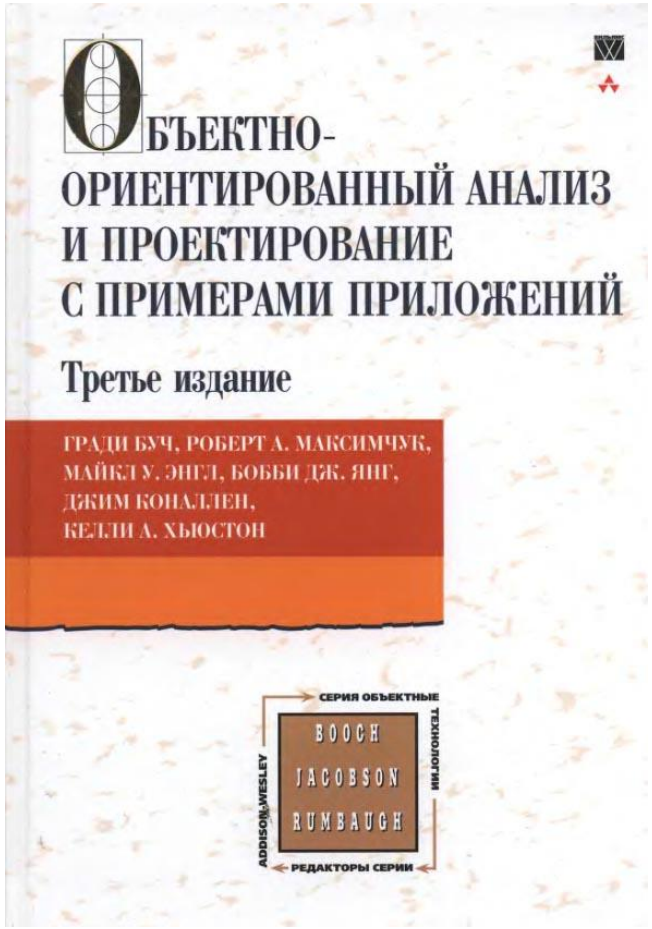
Реальна *міць* виразів await проявляється в тому, що вони можуть з'являтися практично в будь-якому місці коду.

```
async void Test()
{
  for (int i = 0; i < 10; i++)
  {
    int result = await ComplexCalculationAsync();
    Console.WriteLine (result);
  }
}
```

При першому виконанні методу ComplexCalculationAsync виконання повертається викликаючій стороні завдяки виразу await.

Коли метод завершується, виконання відновлюється з точки переривання з збереженими значеннями локальних змінних і лічильників. Компілятор досягає цього, транслюючи цей код в кінцевий автомат.

Object-oriented analysis and design, OOA&D



Об'єктно-орієнтований аналіз і проектування

Об'єктно-орієнтоване проектування (Object-oriented design, OOD) - це метод проектування, що поєднує *процес об'єктно-орієнтованої декомпозиції* і *систему позначення* для представлення логічної і фізичної, а також статичної та динамічної моделей проектованої системи (Граді Буч).

- **Логічна модель**: класи і об'єкти
- **Фізична модель**: модулі і процеси

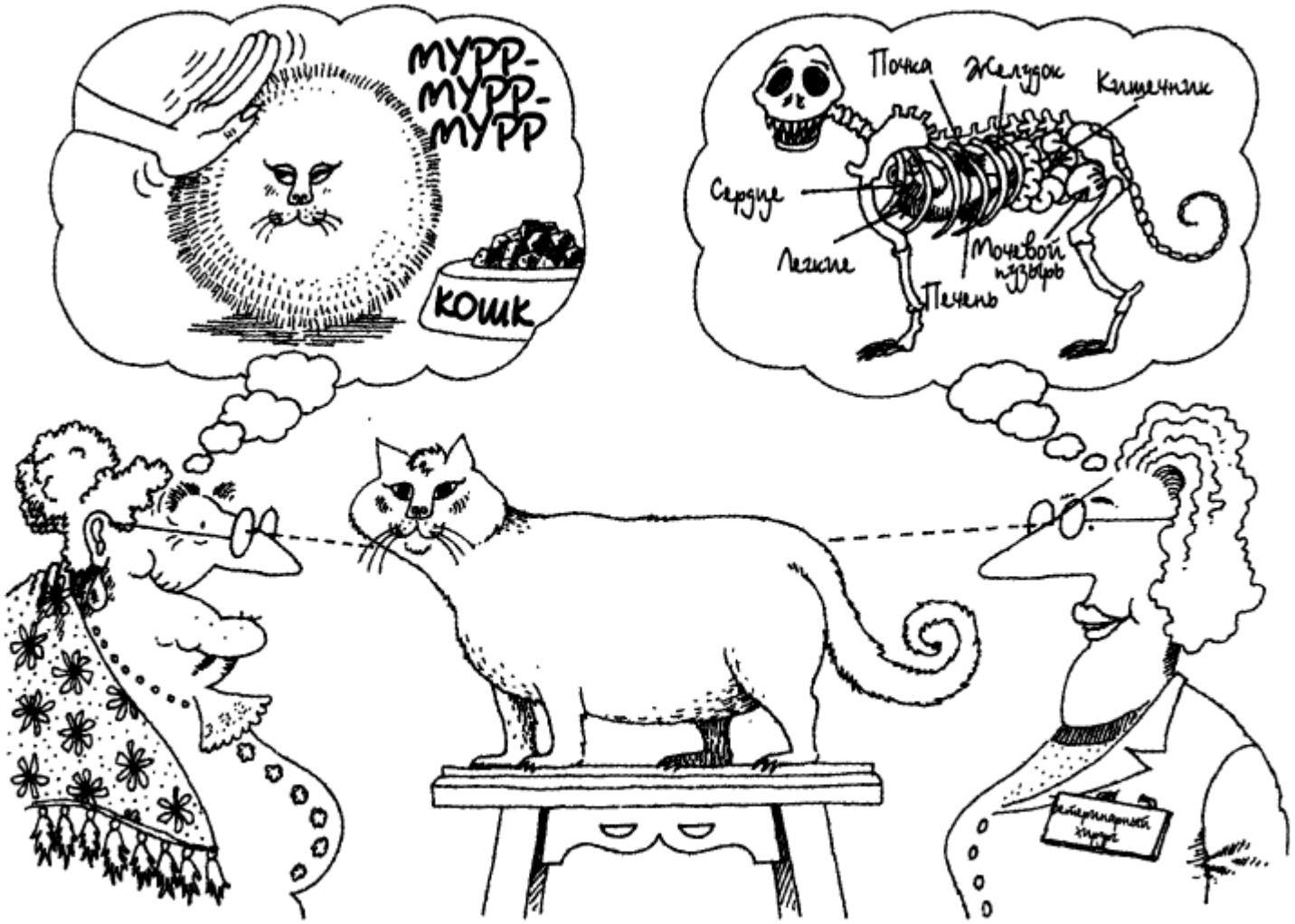
Об'єктно-орієнтований аналіз

Об'єктно-орієнтований аналіз - це метод аналізу, який досліджує вимоги до системи з точки зору *класів* та *об'єктів*, що відносяться до словника предметної області (Граді Буч).

Як співвідносяться між собою об'єктно-орієнтований аналіз, проектування та програмування?

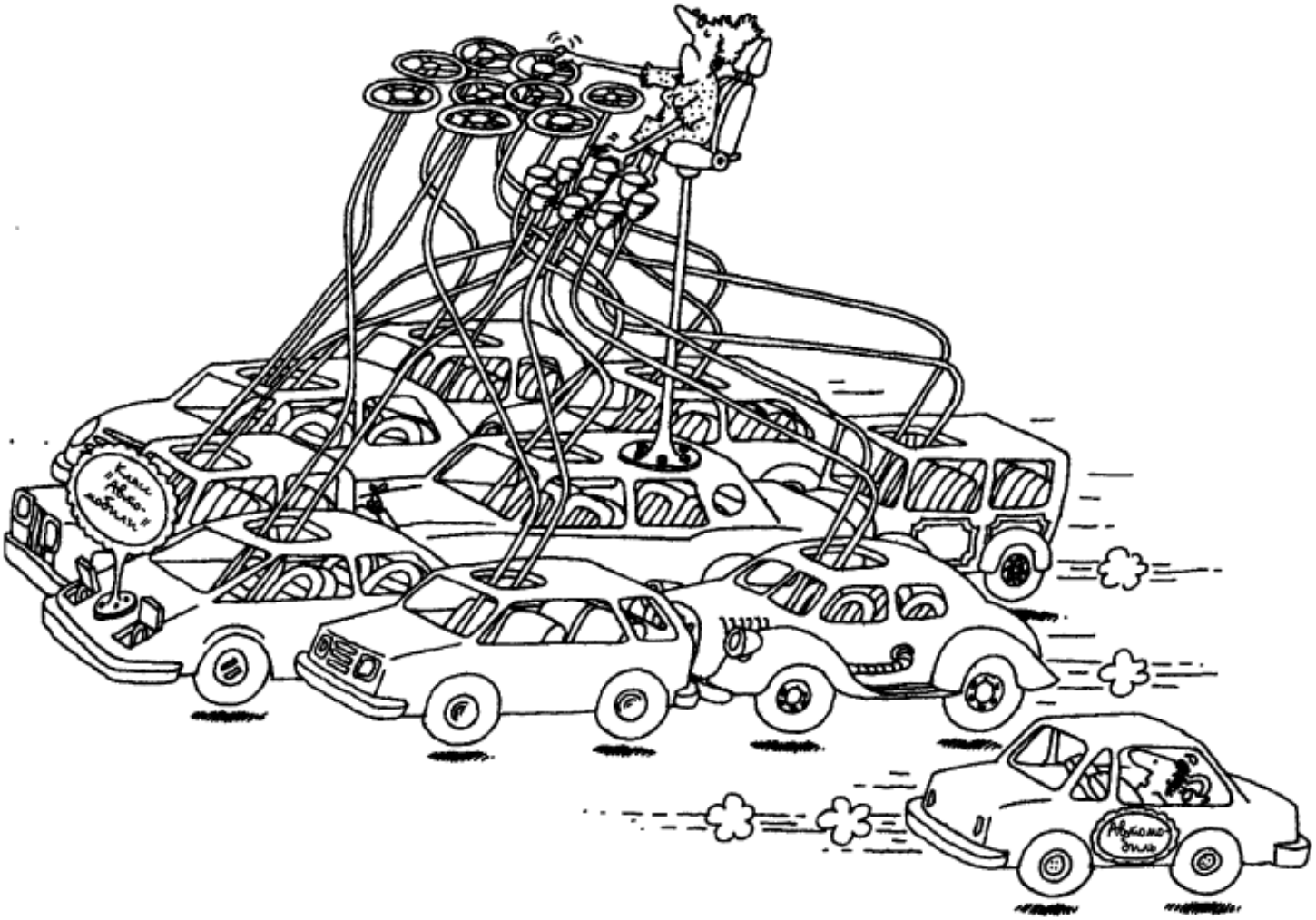
Результатами об'єктно-орієнтованого аналізу є моделі, що лежать в основі об'єктно-орієнтованого проектування, яке в свою чергу дозволяє розробити схему повної реалізації системи з використанням об'єктно-орієнтованого програмування.

Складові об'єктно-орієнтованого підходу



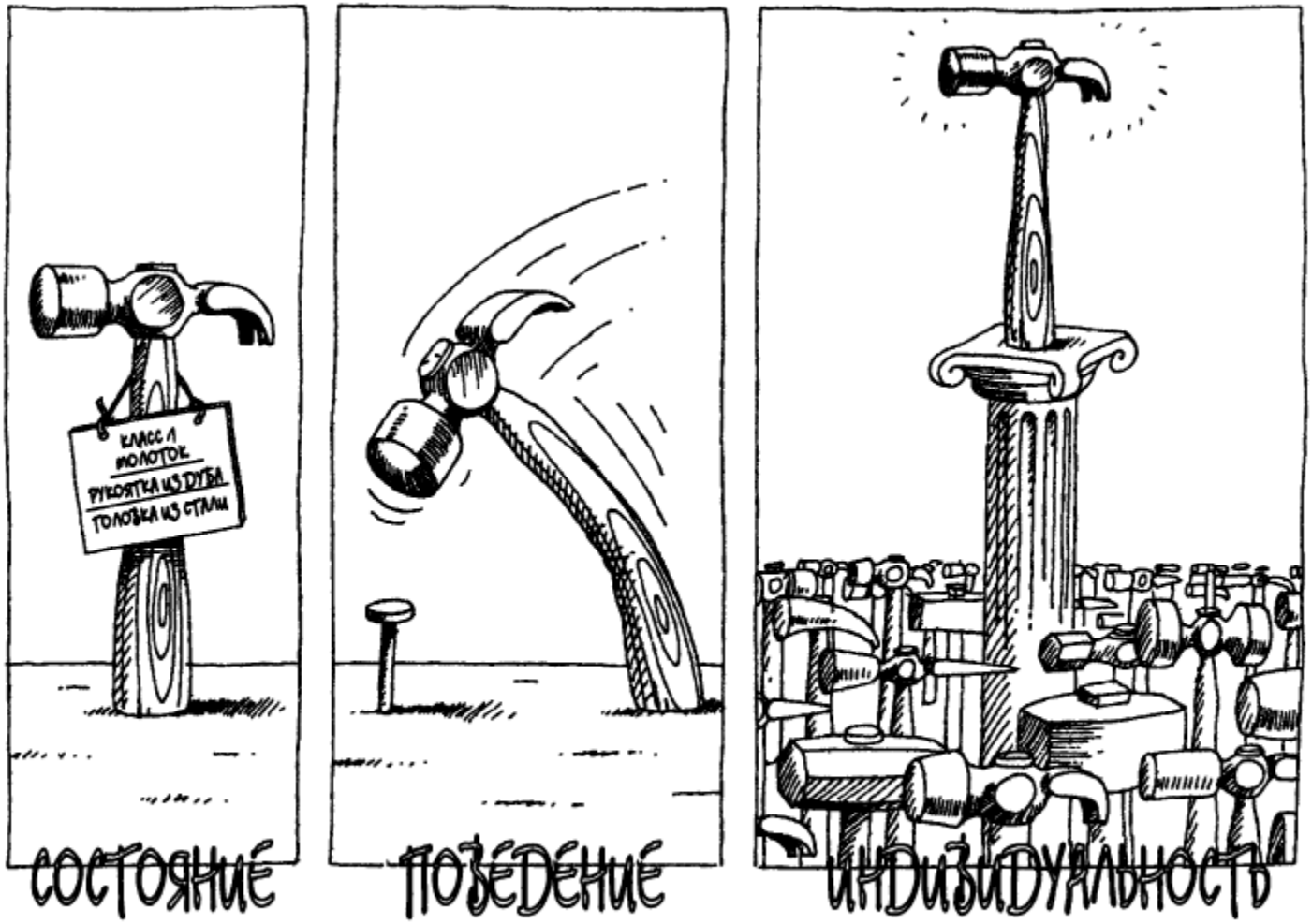
Абстракция концентрирует внимание на существенных свойствах объекта с точки зрения наблюдателя

Класи і об’єкти



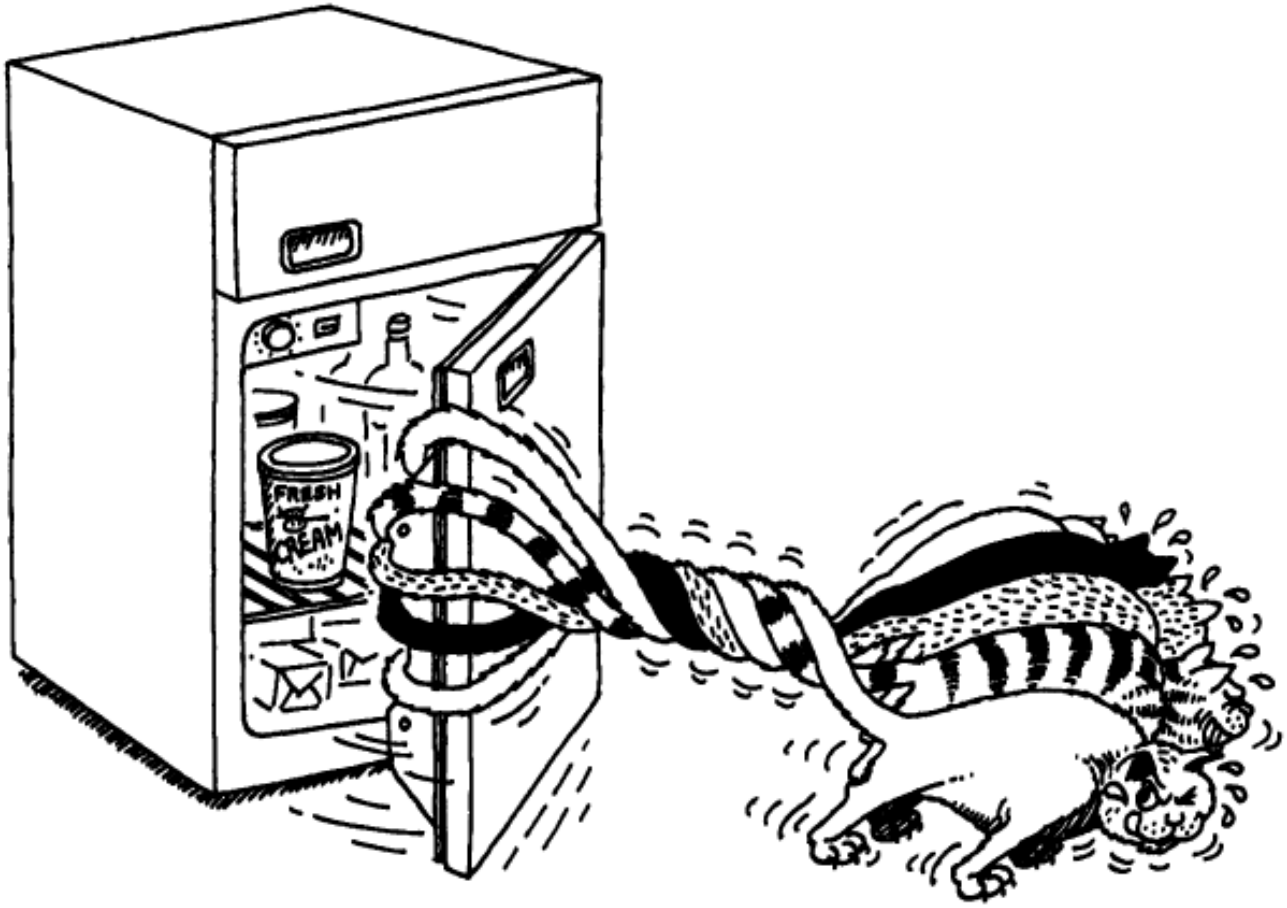
Класс описывает совокупность объектов, обладающих общей структурой и одинаковым поведением

Класи і об’єкти



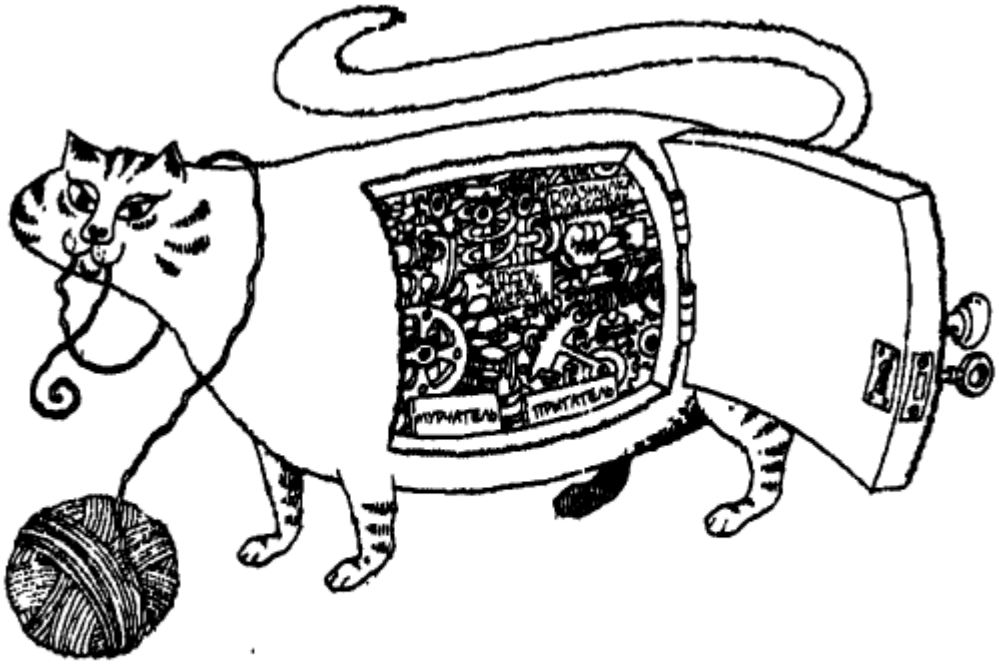
Объект имеет состояние, демонстрирует точно определенное поведение и обладает уникальными особенностями

Взаємодія об'єктів



Объекты взаимодействуют между собой, обеспечивая требуемое поведение

Інкапсуляція



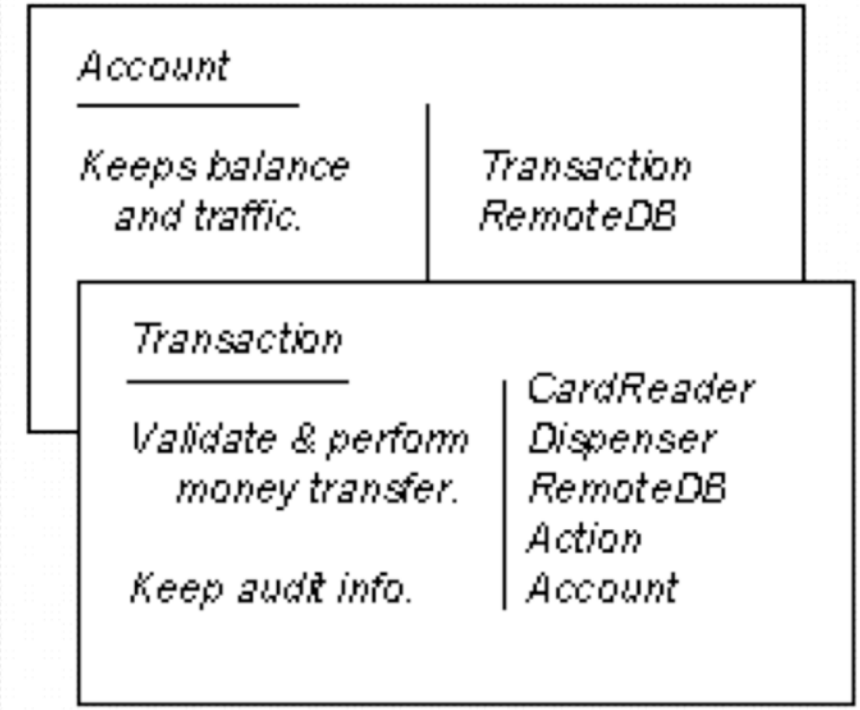
Инкапсуляция скрывает детали реализации объекта

Модульність

Модульность позволяет упаковать абстракции в отдельные единицы программы

CRC-картки

Class / Responsibilities / Collaboration



CRC-картки

- Знайти класи
- Визначити відповідальності класів
- Визначити співпрацю класів
- Визначити елементи Use Case (варіанти використання)
- Розкласти картки на столі

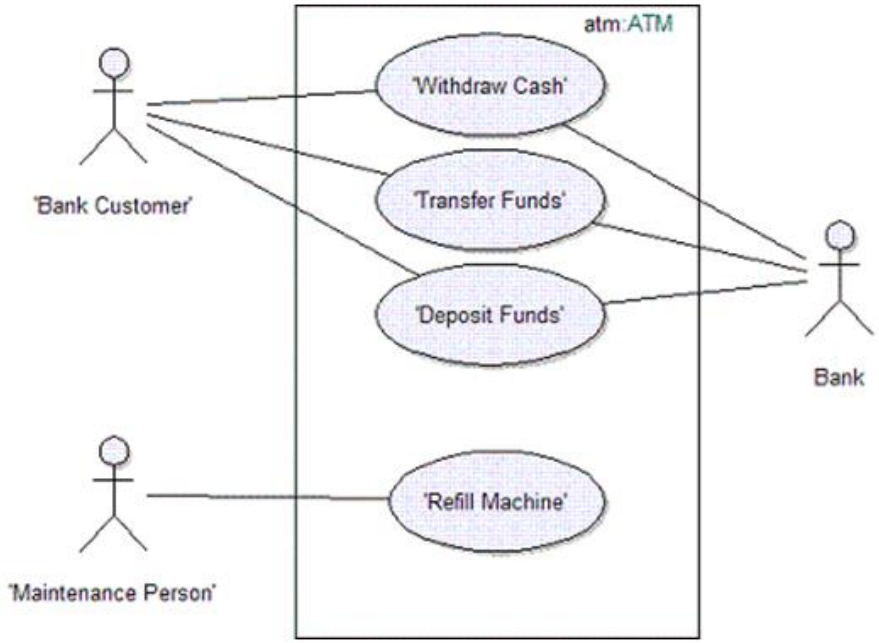
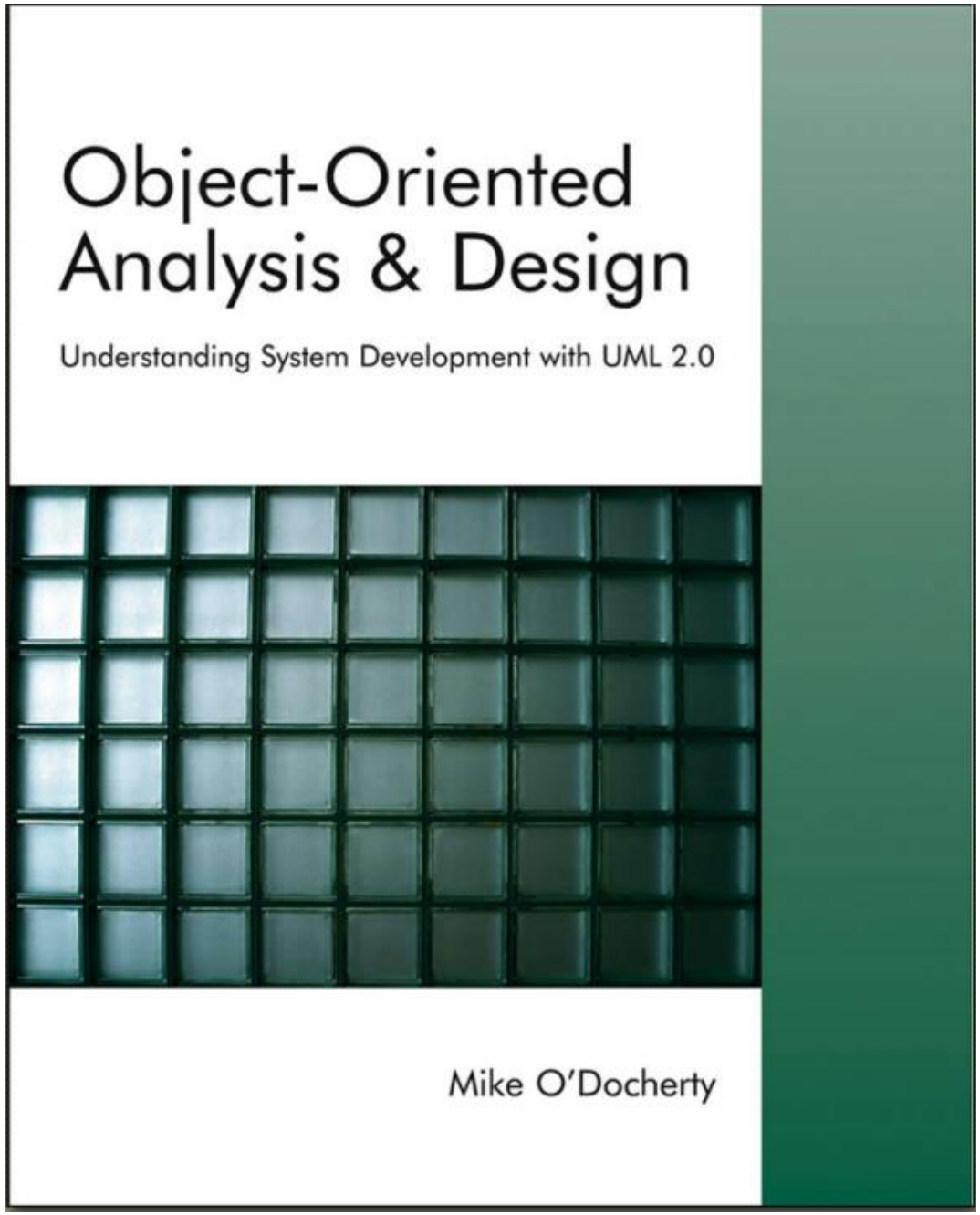
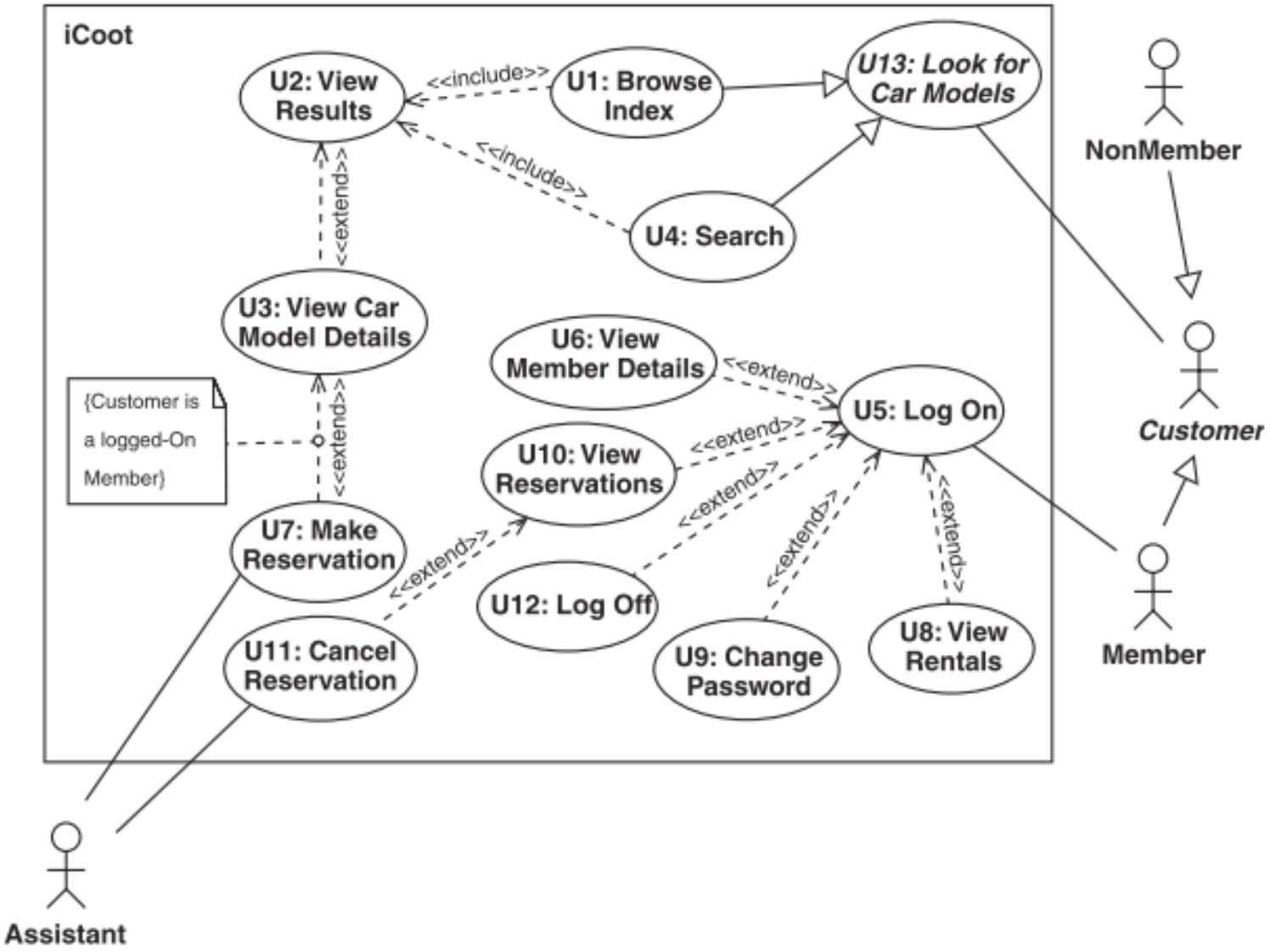


Figure 1: ATM Use-Case Example

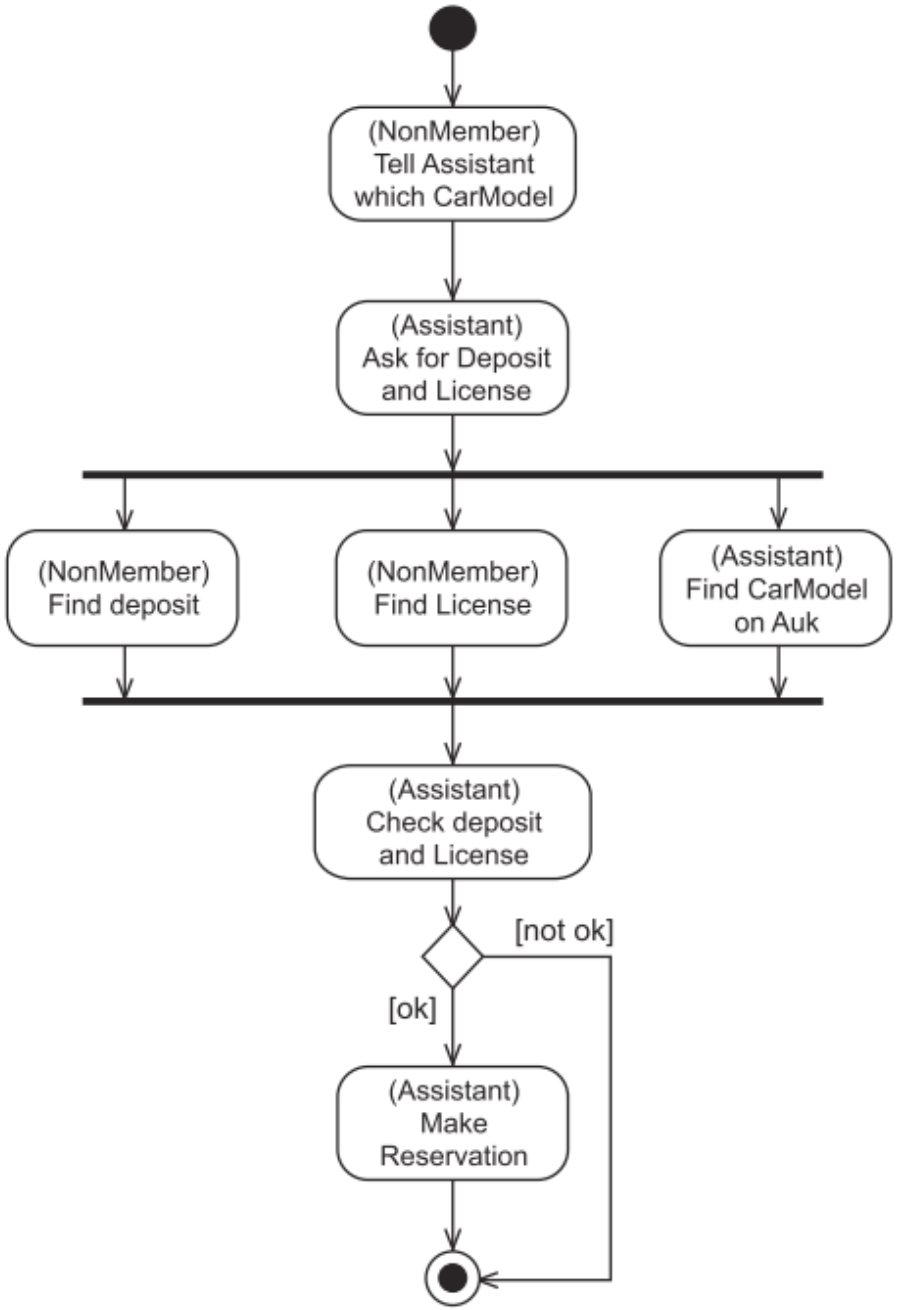

Object-Oriented
Analysis & Design
Understanding System Development with UML 2.0
Mike O'Docherty

A use case diagram



Пункт прокату автомобілів, доступний через Internet

Illustrating Use Cases on an Activity Diagram



Визначення класів

Граді Буч запропонував для побудови об'єктної моделі метод неформального опису, в якому:

- **Виділяються іменники та дієслова в описі предметної області**
- **Іменники** розглядаються як кандидати для утворення **класів**
- **Дієслова** - кандидати в **операції** над класами.

Більш детально про це –

Гайсарян С.С. Объектно-ориентированное проектирование

Визначення класів

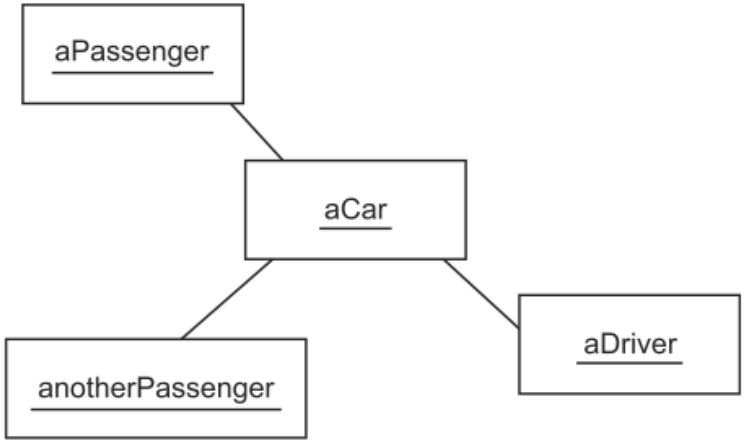
- При визначенні можливих класів потрібно постаратися **виділити якомога більше класів**
- Далі список можливих класів має бути проаналізований з метою **виключення з нього непотрібних класів**. Такими класами є:
  - **надлишкові** класи
  - **нерелевантні** (що не мають прямого відношення до проблеми)
  - **атрибути** (наприклад, ім'я, вік, вага, адреса і т.п.)
  - **операції** (наприклад, телефонний_виклик)
  - **ролі**
  - **реалізаційні конструкції**

Підготовка словника даних

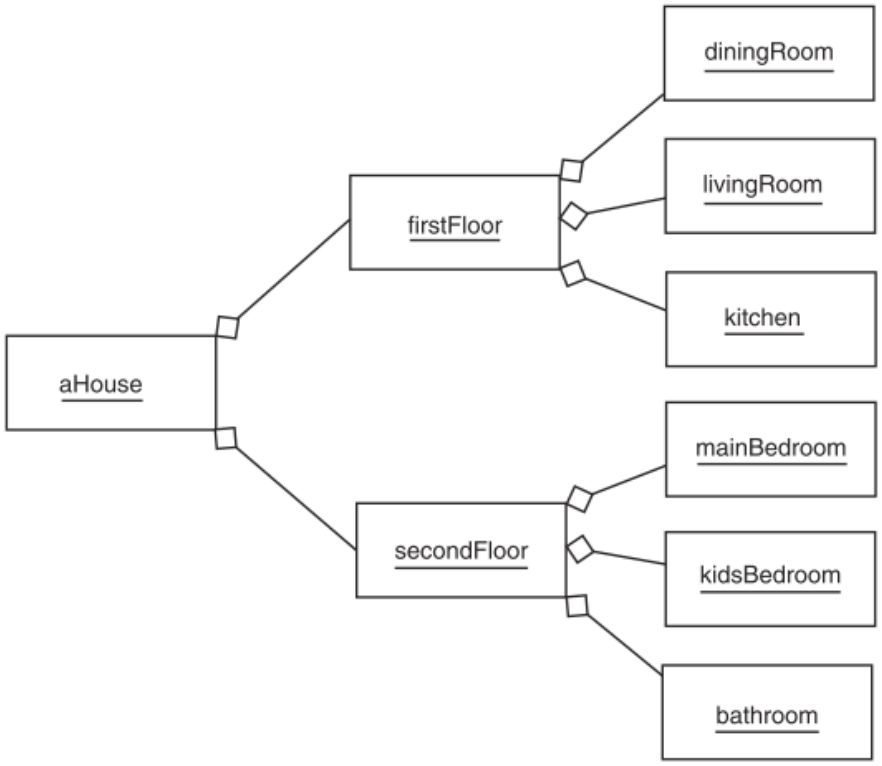
- **Окремі слова мають занадто багато інтерпретацій**.
- **Тому необхідно на самому початку проектування підготувати словник даних**, що містить чіткі і недвозначні визначення всіх
  - **об'єктів** (**класів**),
  - **атрибутів**,
  - **операцій**,
  - **ролей** та
  - **інших сутностей**, що розглядаються в проекті.
- Без такого словника **обговорення проекту з колегами з розробки та замовниками** системи не має сенсу, так як кожен може по-своєму інтерпретувати обговорювані терміни.

Визначення залежностей

- Association

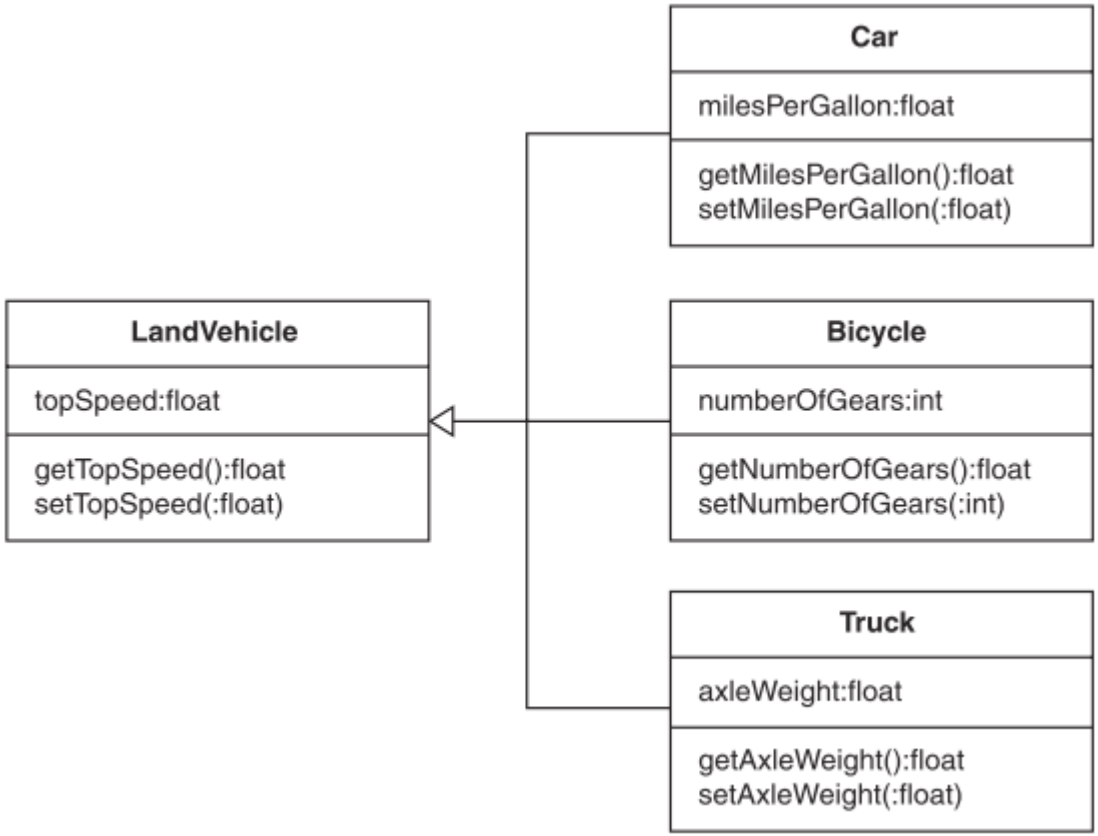


- Aggregation

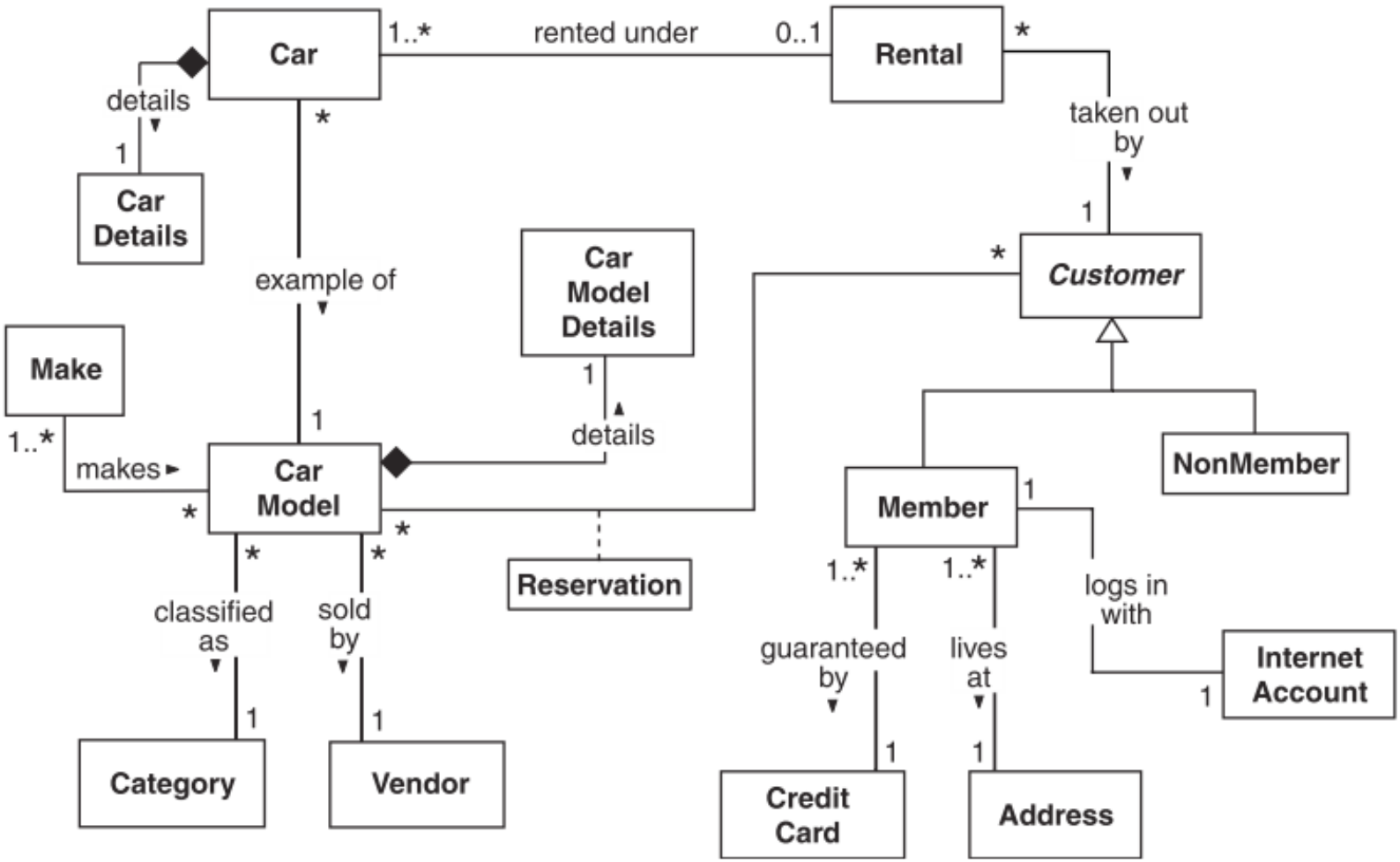


Визначення залежностей

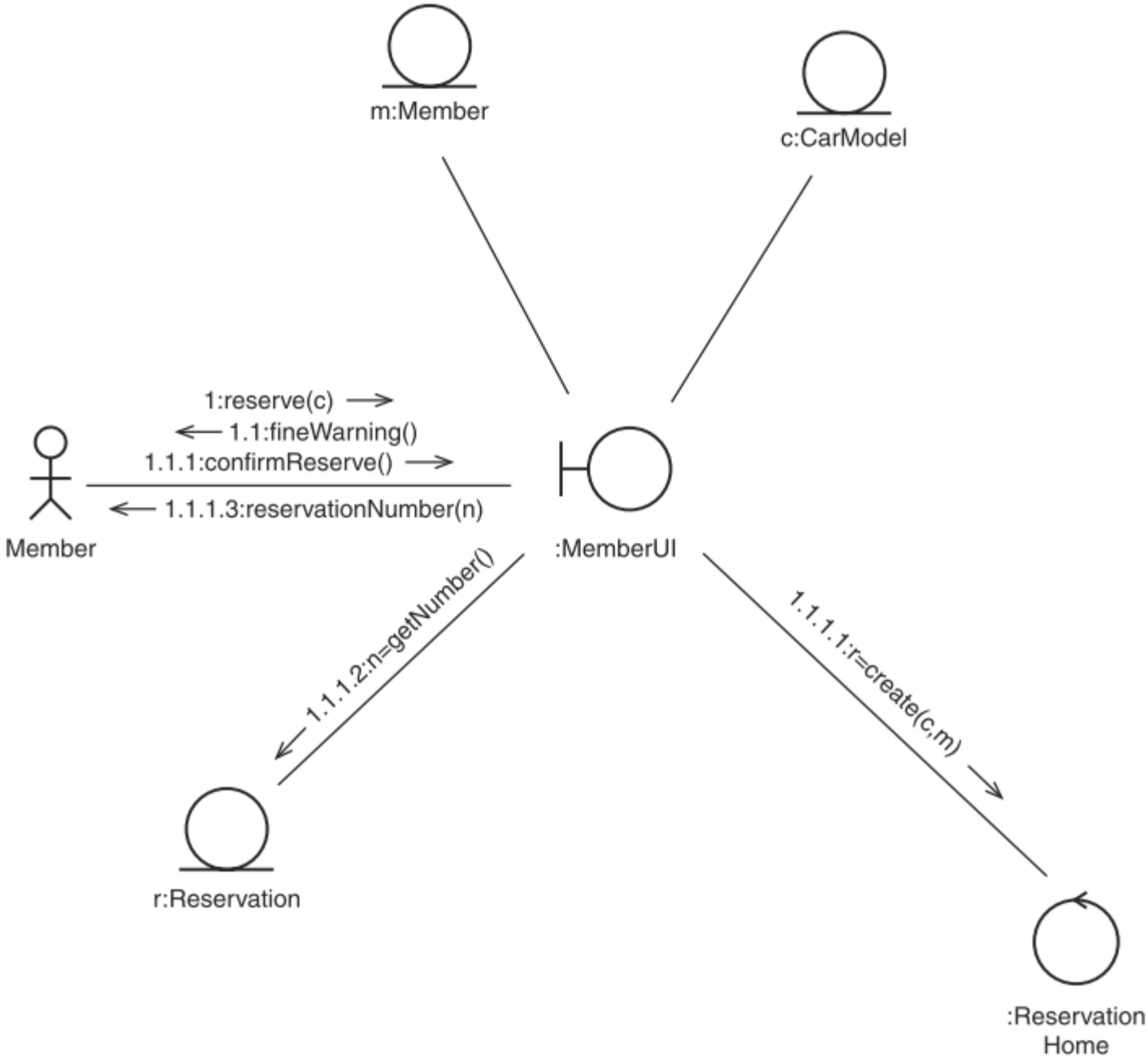
- Generalization



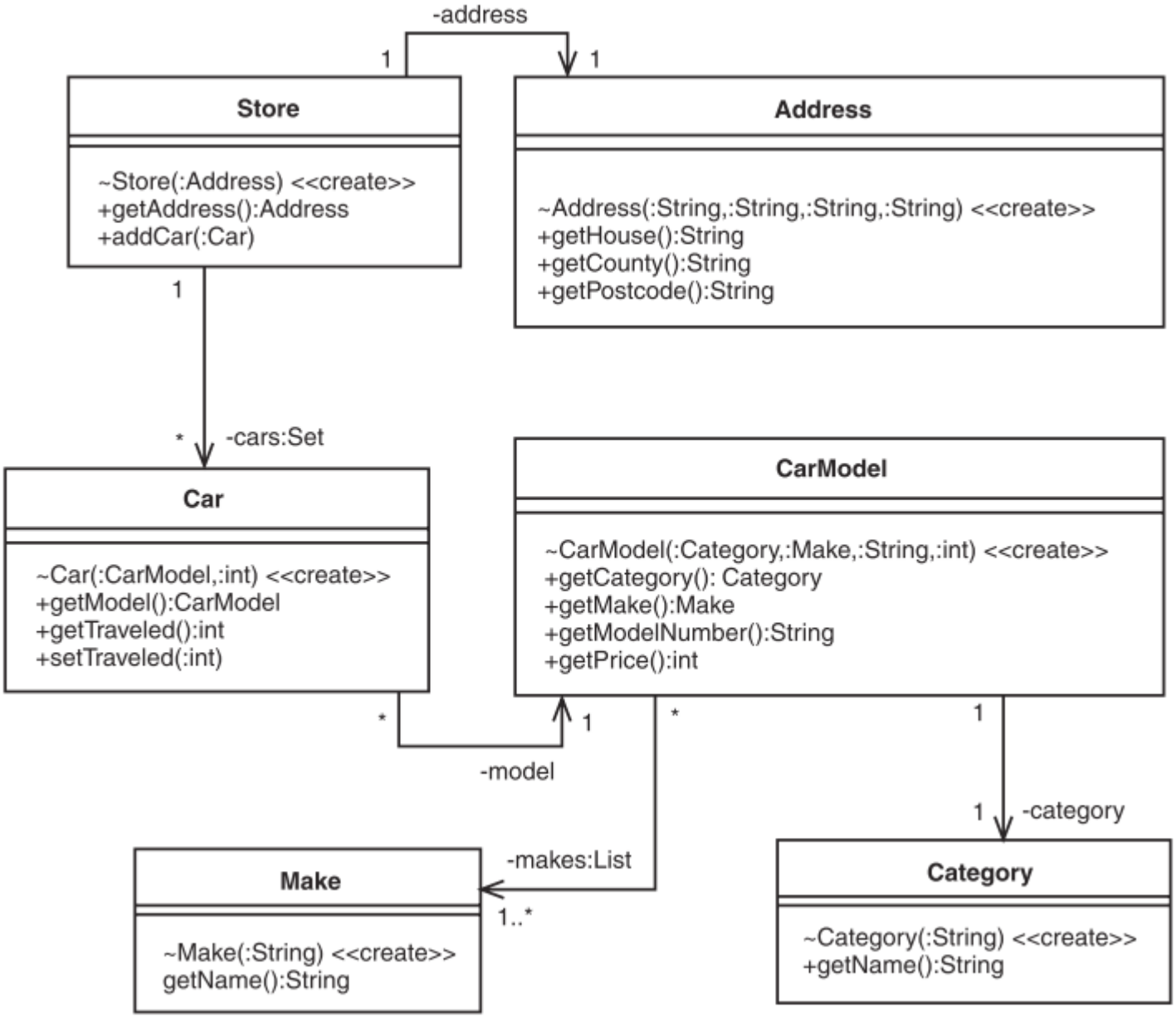
A class diagram at the analysis level



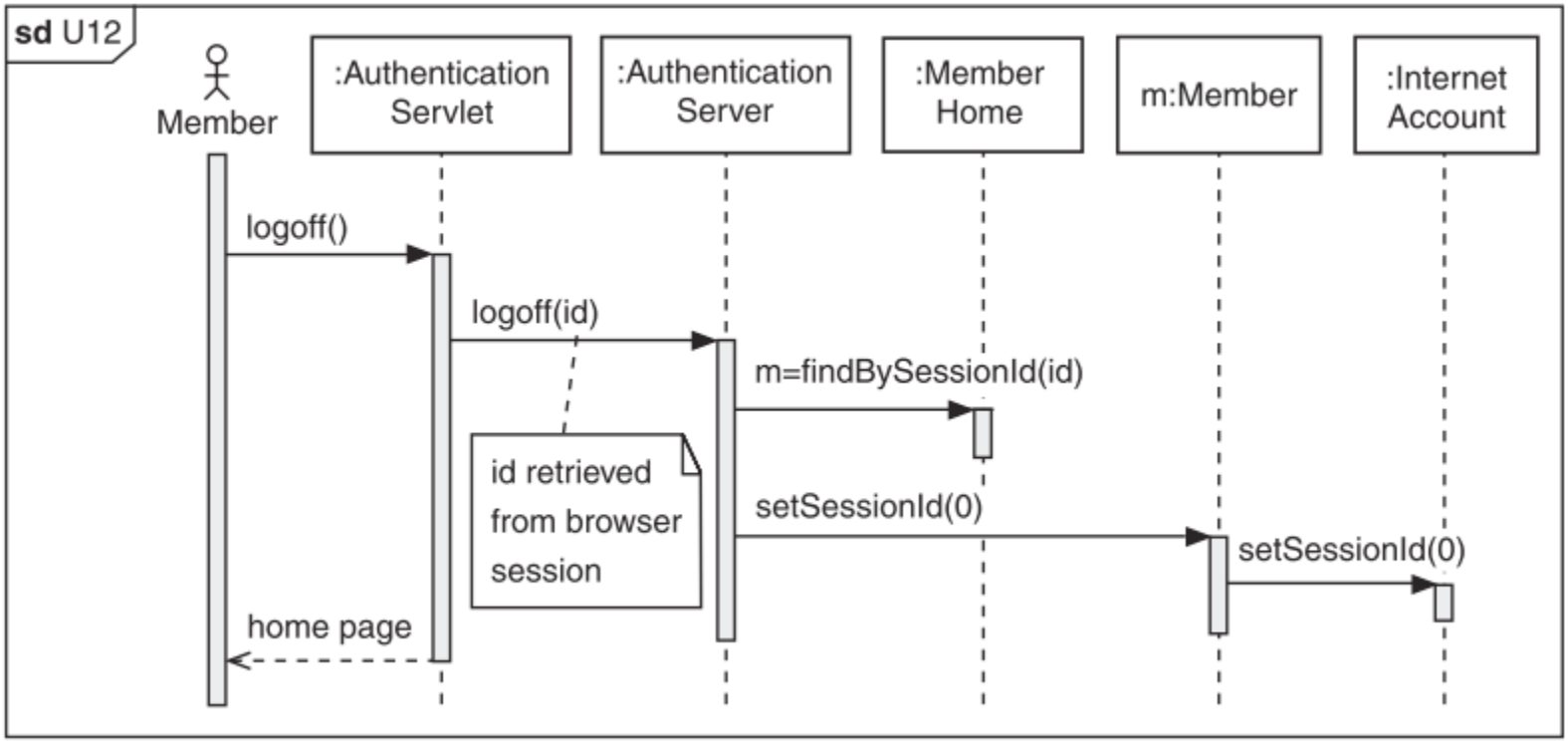
A communication diagram



A design-level class diagram



A sequence diagram from the design phase



Визначення залежностей (по Гайсаряну)

- **Імена можливих залежностей можуть бути отримані з дієслів або дієслівних оборотів**, що зустрічаються в неформальному описі завдання.
- Так зазвичай описуються:
  - **фізичне положення** (слідує_за, є_частиною, міститься_в),
  - **спрямовану дію** (приводить_в_рух),
  - **спілкування** (розмовляє_з),
  - **приналежність** (має, є_частиною) і т.п.

Приклад об’єктної моделі

(Гайсарян) **Система банківського обслуговування**

Список можливих класів:

АТМ (банкомат), касир, програмне забезпечення, банк, касовий термінал, система, банківська мережа, квитанція, перевірка безпеки, дані проводки, клієнт, служба ведення записів, дані рахунку, комп'ютер банку, рахунок, гроші, консорціум, ціна, доступ, користувач, центральний комп'ютер, картка, проводка

Надлишкові класи: користувач

Нерелевантні класи: ціна

Нечітко визначені класи: служба ведення записів, перевірка безпеки, система, банківська мережа

Атрибути: дані проводки, дані рахунку, гроші

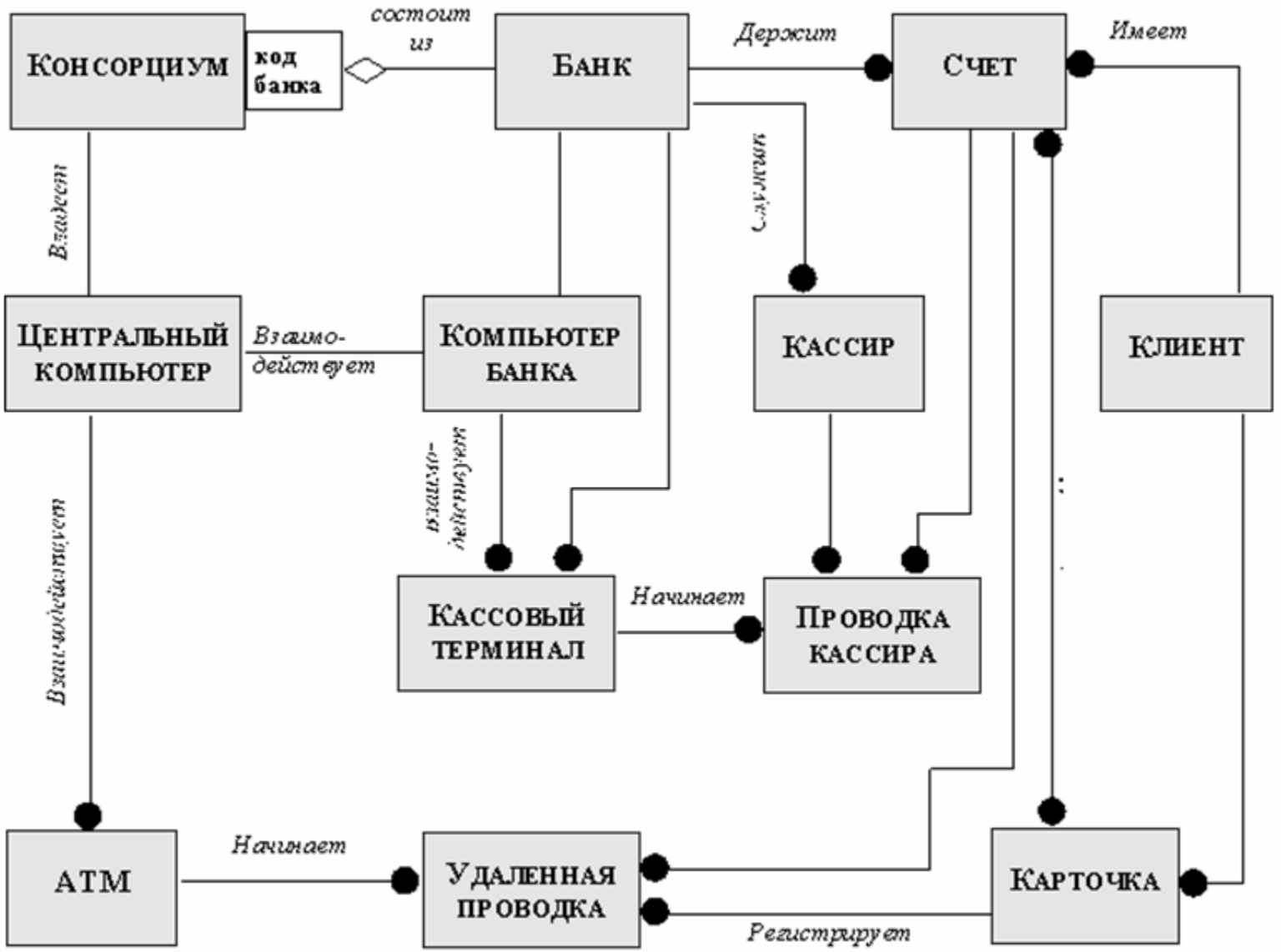
Реалізаційні конструкції: програмне забезпечення, доступ

Система банківського обслуговування

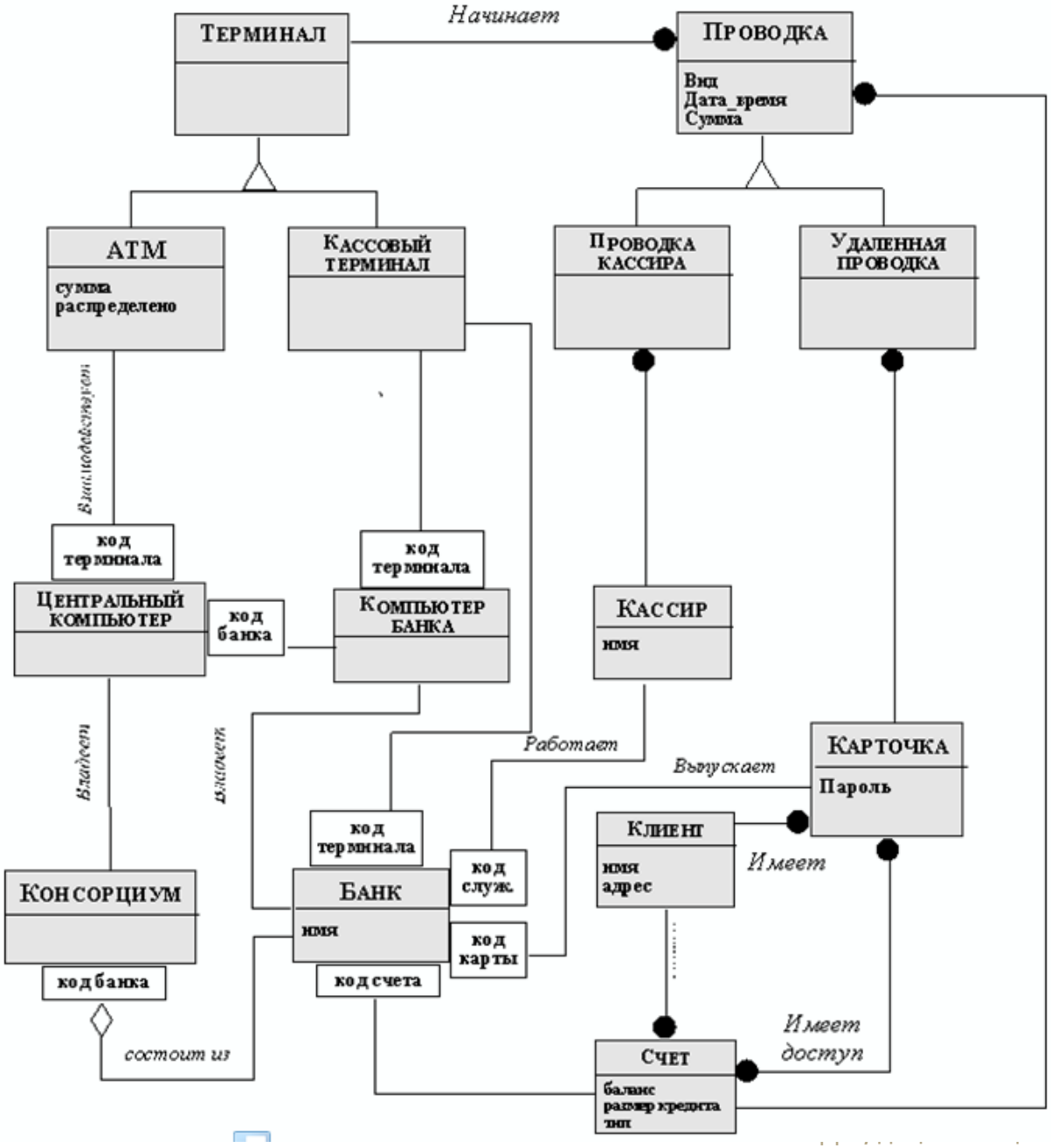
Список залежностей:

- Банк володіє комп'ютером банку
- Комп'ютер банку підтримує рахунки
- Банк володіє касовими терміналами
- Касовий термінал взаємодіє з комп'ютером банку
- Касир вводить проводку
- Проводка відноситься до рахунку
- АТМ'и взаємодіють з центральним комп'ютером
- Проводка починається з АТМ
- Центральний комп'ютер взаємодіє з комп'ютером банку
- Консорціум складається з банків
- Консорціум володіє центральним комп'ютером
- Клієнти мають картки
- Картка забезпечує доступ до рахунку
- У банку служать касири

Перша версія об'єктної діаграми для банківської мережі



Об'єктна діаграма з урахуванням успадкування



Корисні посилання

- Моделювання роботи ліфта
  - A UML Documentation for an Elevator System.pdf
- ATM - ATM Simulation (UML).rar
- links_oop.txt
  - Introduction to Object Oriented Programming Concepts (OOP) and More http://**www.codeproject.com**/Articles/22769/Introduction-to-Object-Oriented-Programming-Concep
  - Identifying Object-Oriented Classes
  - http://www.codeproject.com/Articles/9900/Identifying-Object-Oriented-Classes
  - How I explained OOD to my wife http://www.codeproject.com/Articles/93369/How-I-explained-OOD-to-my-wife
- Object-Oriented Analysis and Design Understanding System Development with UML 2.0.pdf

Принципи проектування класів

П’ять принципів SOLID :

- **Single Responsibility Principle, SRP** (Принцип єдиного обов'язку)
- **Open/closed principle, OCP** (Принцип відкритості / закритості)
- **Liskov substitution principle, LSP** (Принцип підстановки Лісков)
- **Interface segregation principle, ISP** (Принцип поділу інтерфейсів)
- **Dependency inversion principle, DIP** (Принцип інверсії залежностей)

Запропоновані Робертом Мартіном (Robert C. Martin (“Uncle BOB”) (born 1952),) в 2000 році.

Системи, розроблені за цими принципами, легше супроводжувати і розширювати

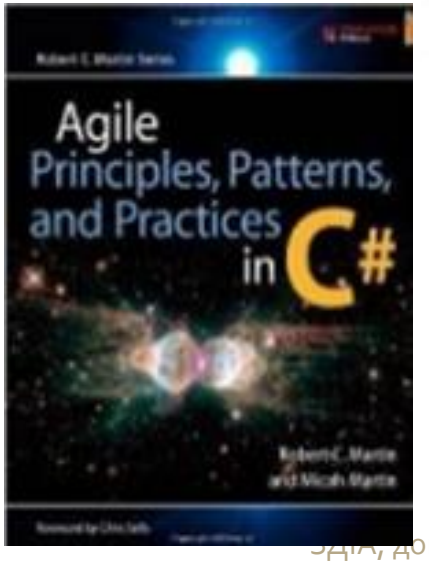

HIGH TECH
Р. С. МАРТИН,
М. МАРТИН
Принципы, паттерны
и методики гибкой
разработки
на языке C#
СИМВОЛ

Single Responsibility Principle

- Означає, що **кожен об'єкт повинен мати один обов'язок** і цей обов'язок повинен бути повністю інкапсулюватися в клас. Всі його сервіси мають бути спрямовані виключно на забезпечення цього обов'язку.
- Мартін визначає обов'язок як причину зміни та робить висновок, що клас або модуль повинні мати одну і тільки одну причину змінитися.
- Наприклад, модуль складає і друкує звіт. Такий модуль може змінитися з двох причин. По-перше, може змінитися сам вміст звіту. По-друге, може змінитися формат звіту.
- Принцип єдиного обов'язку каже, що обидва аспекти цієї проблеми насправді є двома різними обов'язками, і в такому випадку повинні знаходитися в різних класах або модулях. **Об'єднання двох сутностей, що змінюються з різних причин і в різний час, вважається поганим проектним рішенням.**

Single Responsibility Principle

- THERE SHOULD NEVER BE MORE THAN ONE REASON FOR A CLASS TO CHANGE
- Мартін описав SRP на основі принципу зв'язності (*cohesion*), сформульованому Томом ДеМарко



Single Responsibility Principle

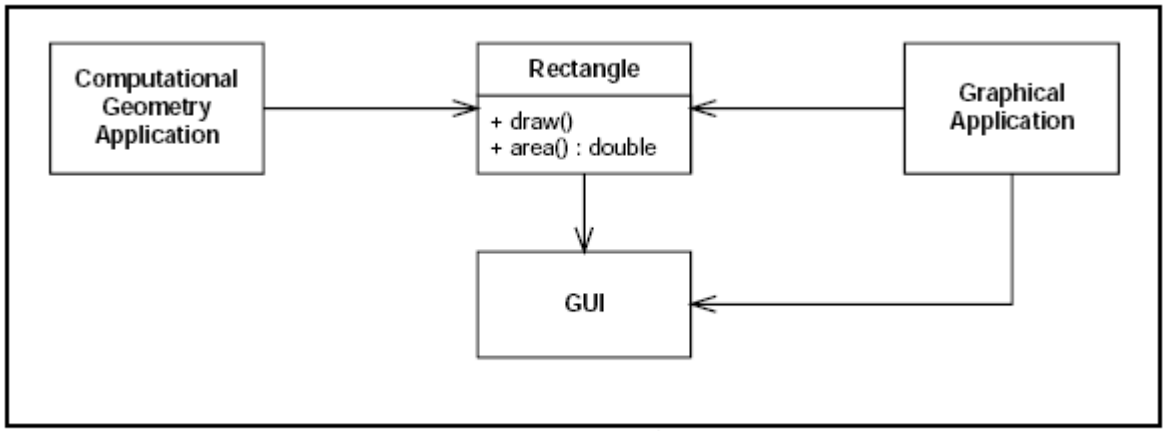


Figure 9-1
More than one responsibility

- The Rectangle class is used by two applications: a Computational Geometry application and a Graphical Application.
  Клас втілює два різних обов'язка.

- Note that the associations shown in the class diagram indicate dependences of one module or class on another. For example, the Computational Geometry Application depends on the Rectangle class, which in turn depends on the GUI module. Dependences are transitive, so indirectly, the Computational Geometry applications depends on teh GUI, even though it doesn't use the GUI!

Single Responsibility Principle

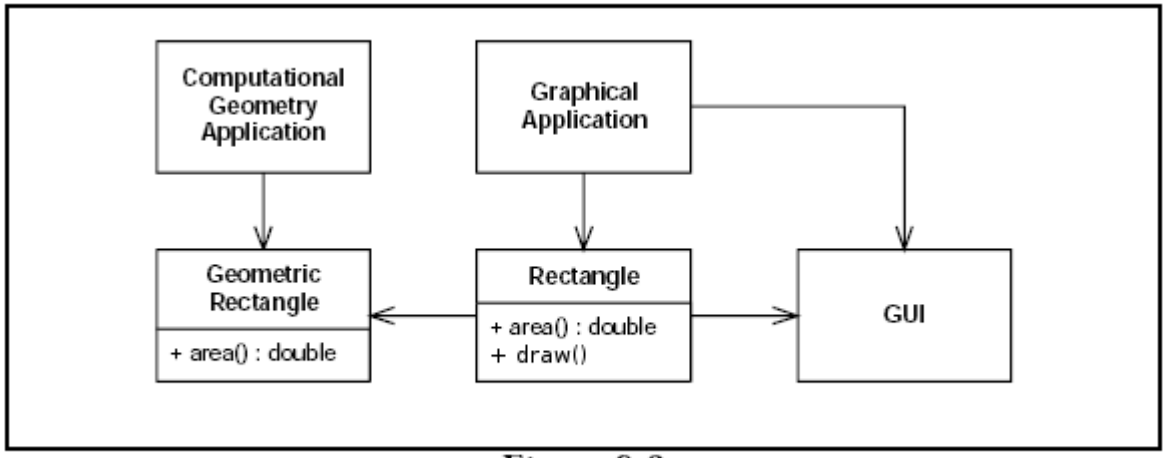


Figure 9-2
Separated Responsibilities

```
public class GeometricRectangle {
private double x, y, width, height;
 ...
public double getX() { return x; }
public double getY() { return y; }
public double getWidth() { return width; }
public double getHeight() { return height; }
public double area() { return width * height; }
}
```

Single Responsibility Principle

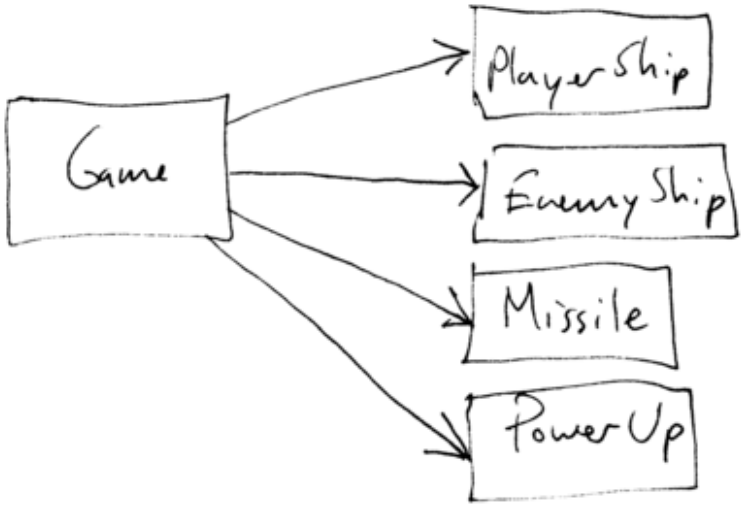
```
public class Rectangle {
    private GeometricRectangle gRect;
    ...
    public double getWidth() { return gRect.getWidth(); }
    public double getHeight() { return gRect.getHeight(); }
    public double area() { return gRect.area(); }
    public void draw(Graphics g) { g.drawRect(gRect.getX(),
            gRect.getY(), gRect.getWidth(), gRect.getHeight());
}
}
```

Now the Rectangle class works as before, but the geometric aspects are delegated to a GeometricRectangle object.

Delegation — one object using another to carry out required tasks — is one of the most important design and implementation techniques in OO software construction.

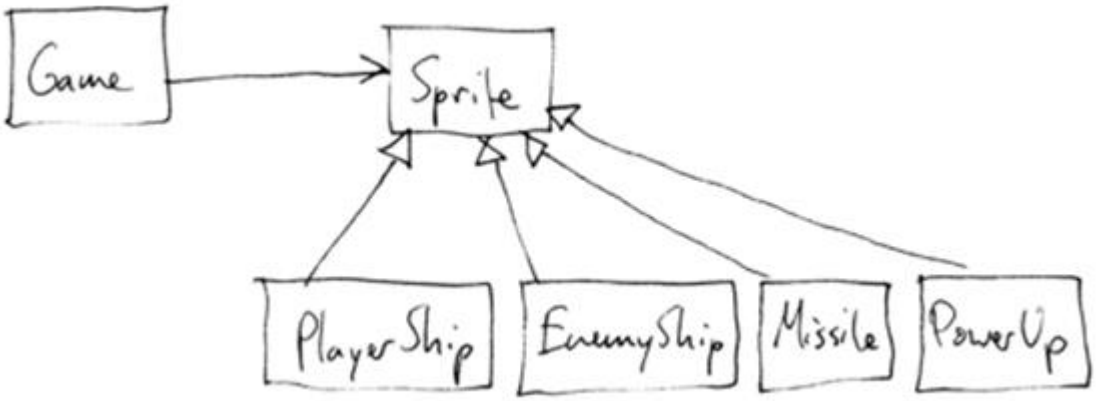
Open/Closed Principle (OCP)

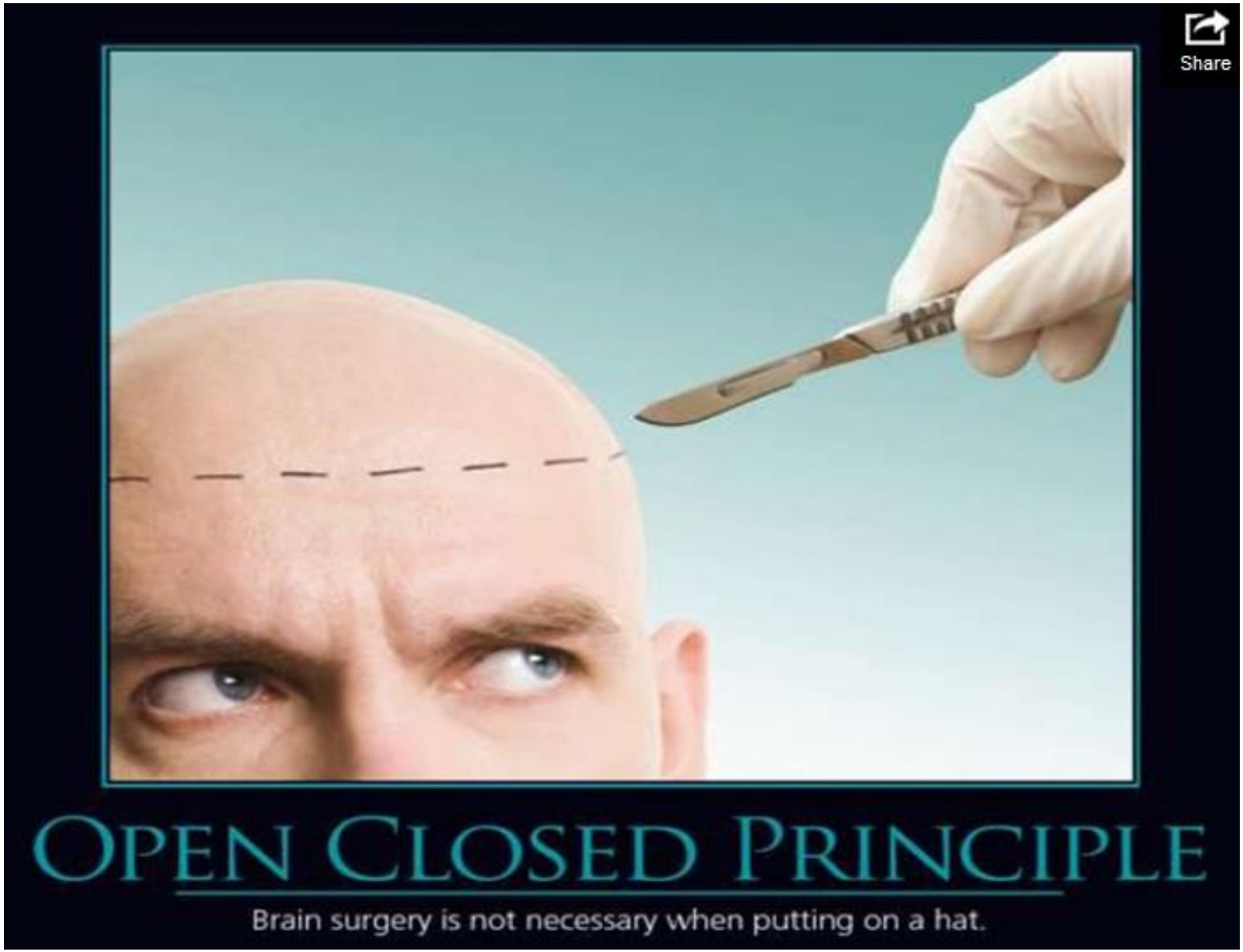
- "A module should be open for extension but closed for modification." (from [Martin])
- It should be possible to extend the functionality of a module (a component of the system) without changing the implementation of that module.
- How is this possible? [Ask.] *Abstract superclasses*.
- Example: A video game. The game supports player ship, enemy ship, missile, and power up objects:



Open/Closed Principle

- If we want to add a new type of visual object to the game, we have to add a new class (not a big problem) and then modify the game module to use it (not so great).
- A better design:




Share
OPEN CLOSED PRINCIPLE
Brain surgery is not necessary when putting on a hat.


OPEN CLOSED PRINCIPLE
Open Chest Surgery Is Not Needed When Putting On A Coat

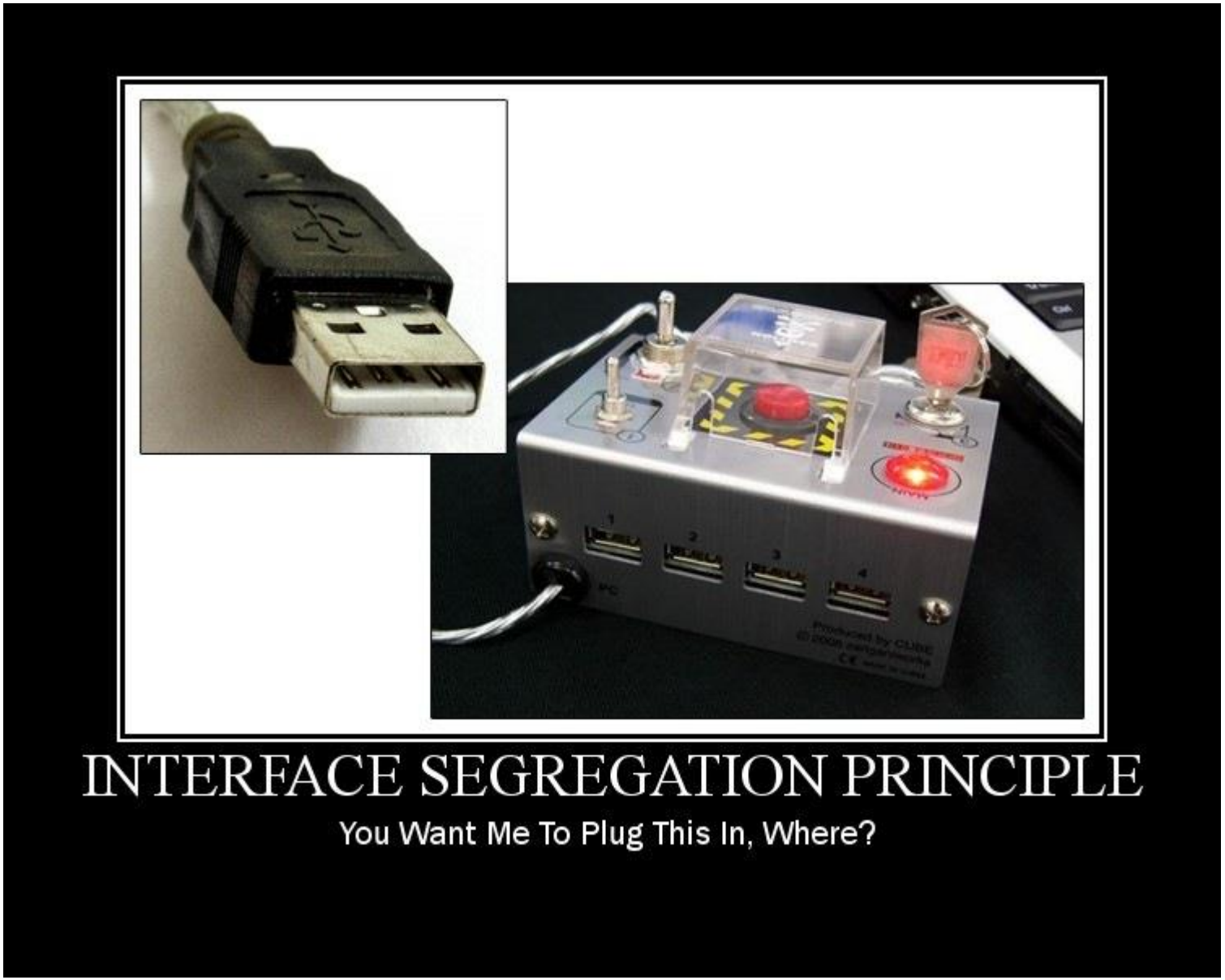
Liskov Substitution Principle

- "Subclasses should be substitutable (взаємозамінними) for their base classes." (from [Martin])
- Another way of stating this principle is that anywhere in a program that an instance of a superclass may be used, an instance of a subclass is allowed.
- This principle is named for Barbara Liskov of MIT
- The principle makes sure that every class follows the contract defined by its parent class. If the class Car has a method called Break it's vital that all subclasses breaks when the Break method is invoked. Imagine the suprise if Break() in a Ferrari only works if the switch ChickenMode is activated.

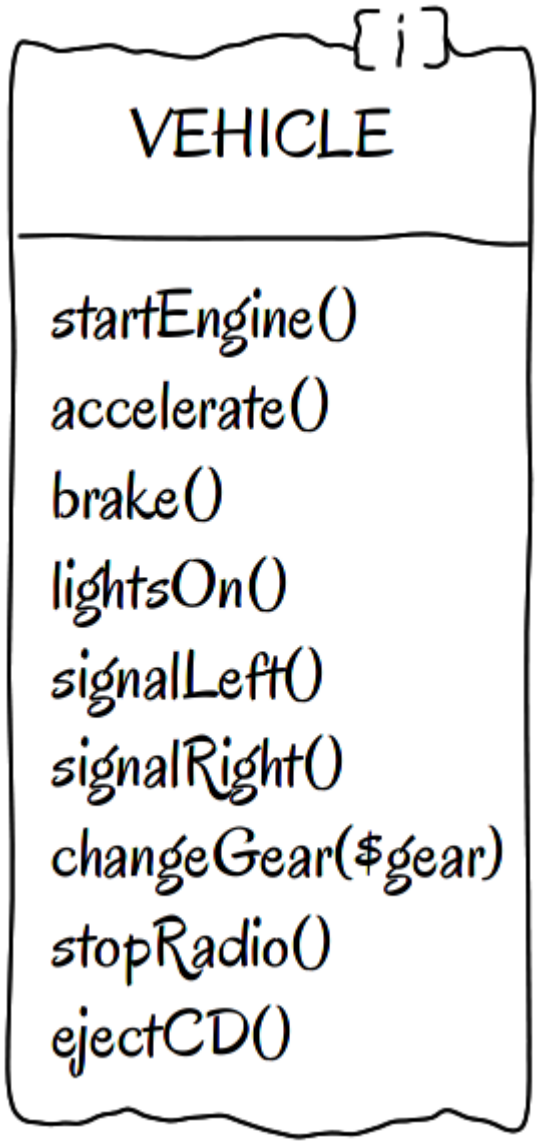

LISKOV SUBSTITUTION PRINCIPLE
If It Looks Like A Duck, Quacks Like A Duck, But Needs Batteries - You Probably Have The Wrong Abstraction

Interface Segregation Principle

- Принцип поділу інтерфейсу
- “many client-specific interfaces are better than one general-purpose interface.”
- **Клієнти не повинні залежати від методів, які вони не використовують**
- Принцип поділу інтерфейсів говорить про те, що **занадто «товсті» інтерфейси необхідно розділяти на більш маленькі і специфічні**, щоб клієнти маленьких інтерфейсів знали тільки про методи, які необхідні їм у роботі

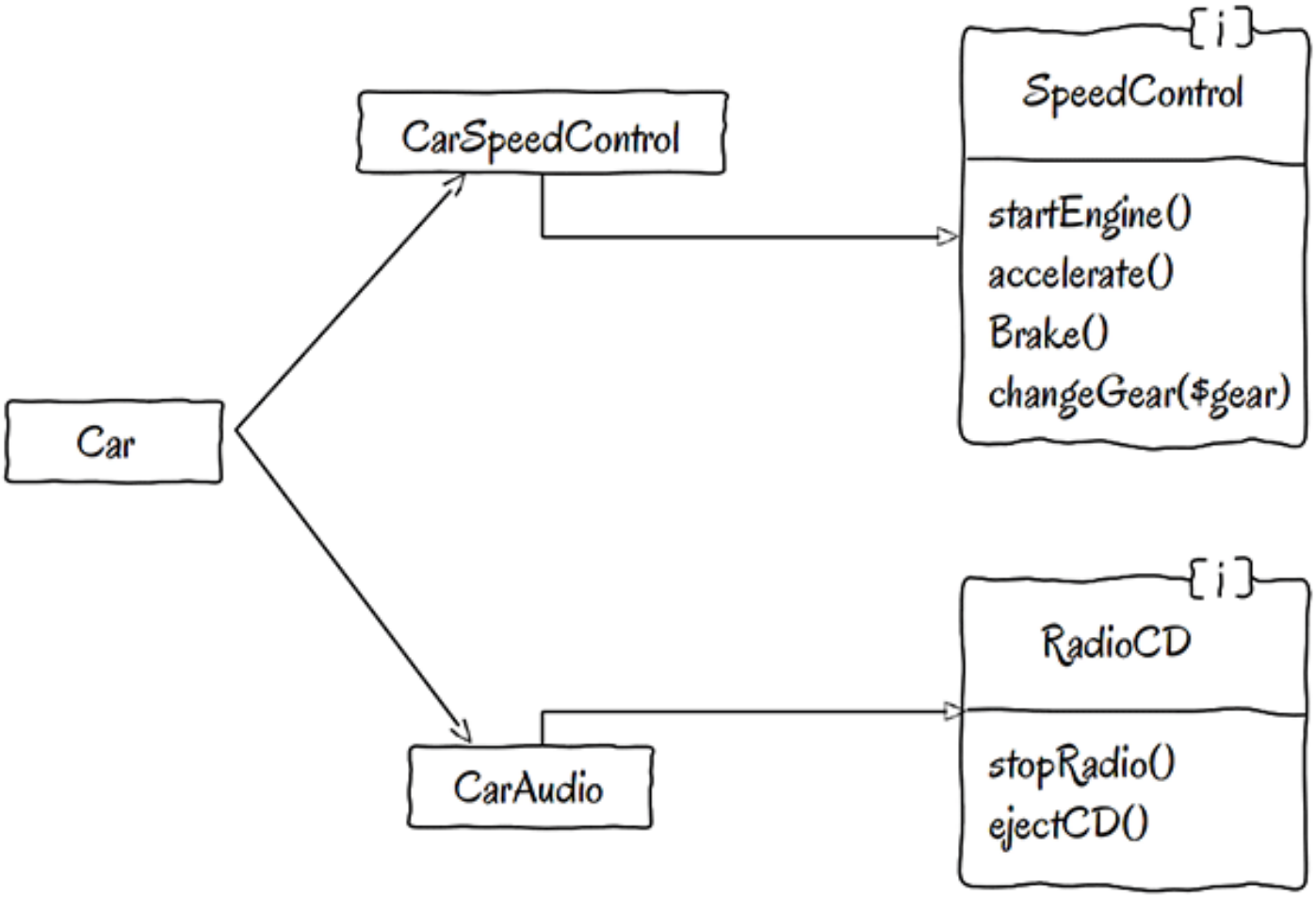

INTERFACE SEGREGATION PRINCIPLE
You Want Me To Plug This In, Where?

Interface Segregation Principle



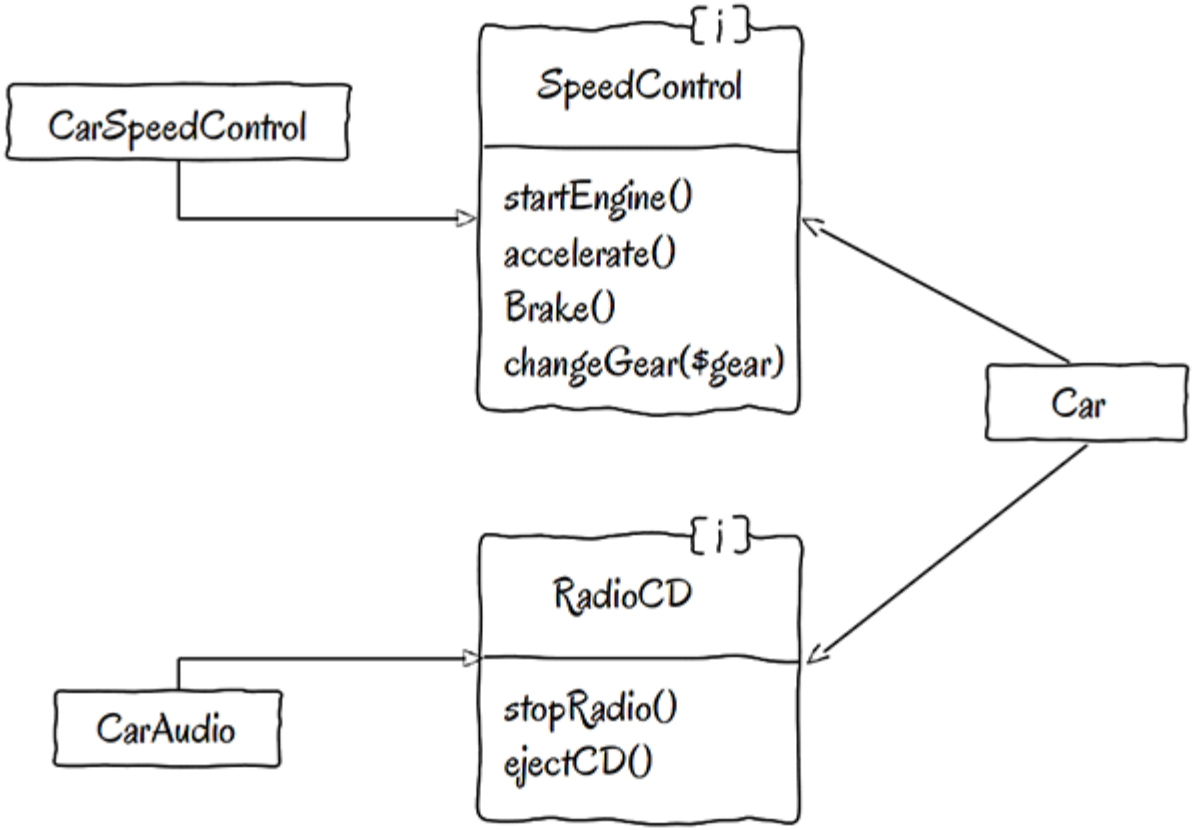
- Приклад "широкого" інтерфейсу
- Є два варіанти реалізації:
  - A huge Car or Bus class implementing all the methods on the Vehicle interface.
  - Or, many small classes like LightsControl, SpeedControl, or RadioCD which are all implementing the whole interface but actually providing something useful only for the parts they implement

Interface Segregation Principle



Interface Segregation Principle

Ми могли б зробити ще один підхід. **Розірвати інтерфейс на шматки, що спеціалізуються в кожній реалізації**. Це допомогло б використовувати невеликі класи, які піклуються про власний інтерфейсі. Об'єкти, що реалізують інтерфейси будуть використовуватися в різних типів транспортних засобів. **Машина буде використовувати реалізації, але буде залежати від інтерфейсів**.



Dependency Inversion Principle

- Принцип інверсії залежностей
- Важливий принцип об'єктно-орієнтованого програмування, що використовується для зменшення зчеплення (з'єднання, coupling) в комп'ютерних програмах
- **Модулі верхніх рівнів не повинні залежати від модулів нижніх рівнів. Обидва типи модулів повинні залежати від абстракцій**.
- **Абстракції не повинні залежати від деталей. Деталі повинні залежати від абстракцій.**

Що таке «Погане проектування»?

- **Система не володіє гнучкістю** (rigit): дуже важко змінити якусь частину системи, так щоб це не торкнулося дуже багато інших її частин.
- **Система ненадійна** (ламка, fragile): при зміні будь-якої окремої частини, інші частини системи перестають коректно працювати.
- **Система або її частина має багато зв'язків** (immobile): дуже важко повторно використовувати частини системи в інших додатки, оскільки вони мають багато зв'язків і залежностей з іншими частинами програми.

Сильна зв'язаність (immobile design)

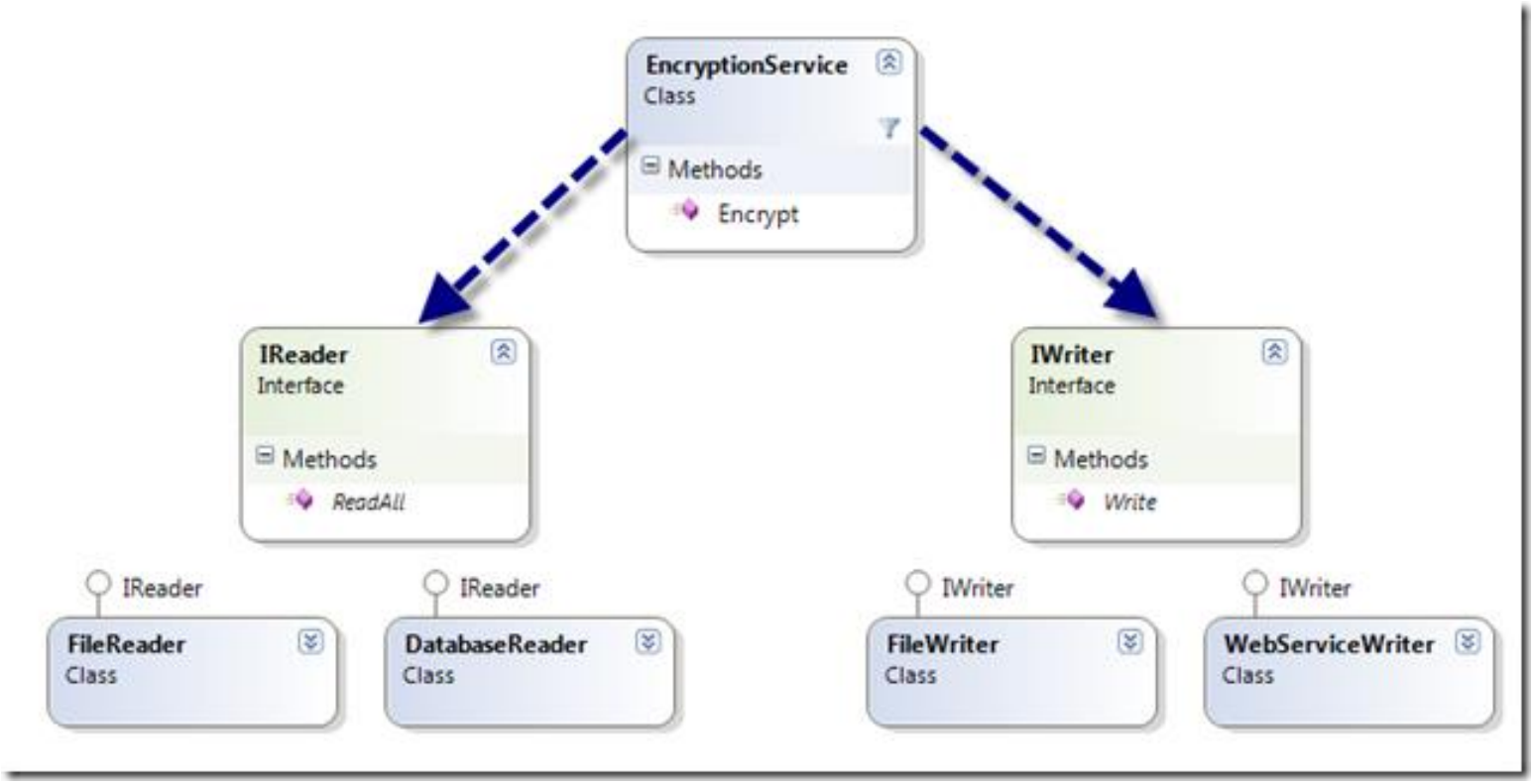
- Частини проекту сильно залежать від інших непотрібних нам деталей
- Приклад. Проектується клас, що реалізує якийсь алгоритм шифрування. Цей клас приймає на вході ім'я файлу-джерела і ім'я файлу-приймача. Потім дані, які повинні бути зашифровані, зчитуються з файлу-джерела, а зашифровані дані записуються в файл-приймач.

```
public class EncryptionService {
    public void Encrypt(string sourceFileName, string
            targetFileName) {
            // Читаємо вміст файлу
            byte[] content;
            using(var fs = new FileStream(sourceFileName,
            FileMode.Open, FileAccess.Read))
            {
```

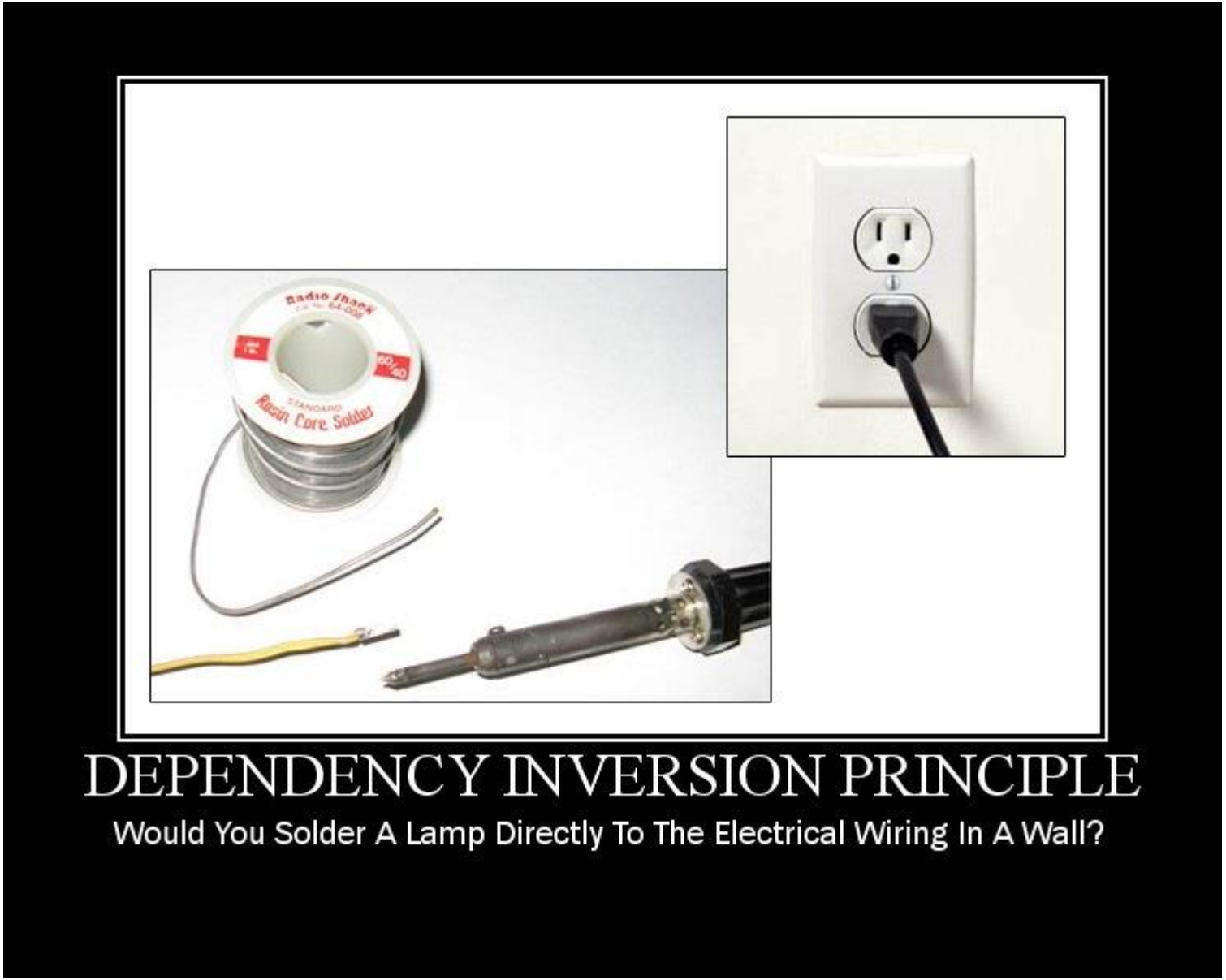
.....

Проблема представленого вище класу в тому, що він сильно прив'язаний до певних засобів вводу / виводу даних

Принцип інверсії залежності



- Тепер метод Encrypt служби шифрування не залежить від особливостей класу Reader або Writer. Залежності були інвертовані; клас EncryptionService залежить від абстракцій, і класи Reader / Writer теж залежать від тих же самих абстракцій.
- Тепер служба шифрування може використовуватися повторно. Ми може придумувати нові види реалізацій «Reader» і «Writer»


Radio Shack
60/40
STANDARD
Rosin Core Solder
DEPENDENCY INVERSION PRINCIPLE
Would You Solder A Lamp Directly To The Electrical Wiring In A Wall?

Підпримка регулярних виразів в мові C#

- **Регулярний вираз – це текстовий шаблон, який задає правила для пошуку в текстових рядках**.

- **Регулярні вирази** (англ. *regular expressions)* спираються на формальну мову пошуку і здійснення маніпуляцій з підрядками в тексті на основі використання метасимволів.

- **По суті це рядок-зразок** (шаблон, *pattern*), що складається з **символів** і **метасимволів** і задає правило пошуку. (Приклад: **a\.?** Відповідність: a. Або a)

- Регулярні вирази справили прорив в електронній обробці текстів в кінці XX століття.

- Багато сучасних мов програмування мають вбудовану підтримку регулярних виразів.

СУБД Access

SELECT * FROM Sumproduct WHERE Product LIKE *'B?k?s'*

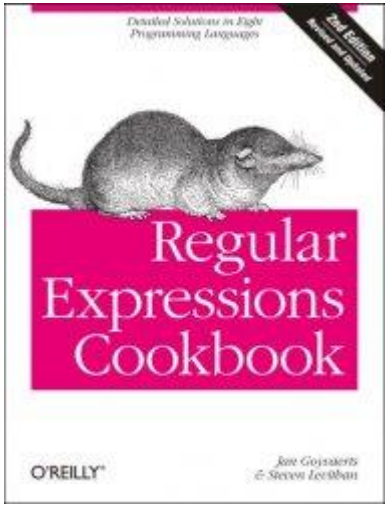
ID	Month	Product	City	Quantity	Amount
2	April	Bikes	Montreal	12	$4 500,00
3	April	Bikes	Montreal	56	$21 000,00
4	April	Bikes	San Francisco	854	$320 250,00
5	April	Bikes	New York	25	$9 375,00
22	February	Bikes	Montreal	663	$248 625,00
23	February	Bikes	San Francisco	21	$7 875,00
24	February	Bikes	San Francisco	54	$20 250,00
25	February	Bikes	New York	658	$246 750,00
42	January	Bikes	Montreal	75	$28 125,00

SELECT * FROM Sumproduct WHERE City LIKE '[TN]*'

ID	Month	Product	City	Quantity	Amount
5	April	Bikes	New York	25	$9 375,00
7	April	Skates	Toronto	854	$84 546,00
9	April	Skates	New York	663	$65 637,00
11	April	Skis Long	Toronto	25	$6 125,00
13	April	Skis Long	New York	21	$5 145,00
15	April	Skis Short	Toronto	4	$836,00
17	April	Skis Short	New York	136	$28 424,00
19	April	Snow Board	Toronto	522	$160 776,00
21	April	Snow Board	New York	663	$204 204,00

Література

- Regular Expressions Cookbook, 2nd Edition, 2012



Елементи мови регулярних виразів

Представлення символів

Звичайні символи і спеціальні символи (метасимволи)

Більшість символів у регулярному виразі представляють самі себе за винятком спеціальних символів (метасимволів)

[ ] \ ^ $. | ? * + ( ) { }

яким може передувати символ \ (зворотна коса риска), котра робить метасимволи «екранованими», «захищеними» для представлення їх самих як символів тексту.

Escape-символи

\a – дзвінок, \t – табуляція, \n – новий рядок і т.д.

Класи символів

Клас	Опис	Шаблон	Відповідність
[character_group]	Один символ з набору	[ae]	“a” в “gray” “a”,”e” в “lane”
[^character group]	Один символ не з набору	[^aei]	"r", "g", "n" в "reign"
[first - last]	Один символ з проміжку	[A-Z]	"A", "B" в "AB123"
.	Будь-який символ	a.e	"ave" в "nave“ "ate" в "water"
\w	Довільний алфавітно-цифровий знак	\w	"I", "D", "A", "1", "3" в "ID A1.3"
\W	Не алфавітно-цифровий знак	\W	" ", "." в "ID A1.3"
\s	Пробільний символ	\w\s	"D " в "ID A1.3"
\S	Не пробільний символ	\s\S	" _" в "int __ctr"
\d	Цифра	\d	"4" в "4 = IV"
\D	Не цифра	\D	" ", "=", " ", "I", "V" в "4 = IV"

Прив'язки

Твердження	Опис	Шаблон	Відповідність
^	Відповідність повинна починатись з початку рядка	^\d{3}	"901" в "901-333-"
$	Відповідність повинна знаходитись в кінці рядка	-\d{3}$	"-333" в "-901-333"
\b	Відповідність повинна знаходитись на границі слова	\b\w+\s\w+\b	"them theme", "them them" в "them theme them them"
\B	Відповідність не повинна знаходитись на границі слова	\Bend\w*\b	"ends", "ender" в "end sends endure lender"

Конструкції групування

Конструкція	Опис	Шаблон	Відповідність
(subexpression)	Захоплює відповідні вирази і присвоює їм нумерацію, починаючи з 1.	(\w)\1	"ee" в "deep"
(?< name > subexpression)	Виділяє відповідну частину виразу в іменовану групу.	(?<double>\w)\k<double>	"ee" в "deep"
(?: subexpression)	Визначає групу, що не виділяється	Write(?:Line)?	"WriteLine" в "Console.WriteLine()" "Write" в "Console.Write(value)"

Квантори

Квантор вказує кількість входжень попереднього елемента

Квантифікатор	Опис	Шаблон	Відповідність
*	Нуль або більше повторів	\d*\.\d	".0", "19.9", "219.9"
+	Один або більше повторів	"be+"	"bee" в "been", "be" в "bent"
?	Нуль або один	"rai?n"	"ran", "rain"
{ n }	Точно *n* повторів	",\d{3}"	",043" в "1,043.6", ",876", ",543" та ",210" в "9,876,543,210"
{ n ,}	Мінімум n повторів	"\d{2,}"	"166", "29", "1930"
{ n , m }	Мінімум *m* та максимум *n* повторів	"\d{3,5}"	"166", "17668" "19302" в "193024"
{,*n*}	Не більше *n*	colou{,3}r	color, colour, colouur, colouuur

Конструкції зворотних посилань

- Одне із застосувань групування — повторне використання раніше знайдених груп символів (*підрядків*, *блоків*).
- **При обробці виразу підрядки, що знайдені за шаблоном усередині групи, зберігаються в окремій області пам'яті й отримують номер, починаючи з одиниці**.
- **Згодом у межах даного регулярного виразу можна використати позначення від \1 до \9 для перевірки на збіг із раніше знайденим підрядком.**
- Наприклад, регулярний вираз **(та|ту)-\1** знайде рядок та-та або ту-ту, але пропустить рядок та-ту.
- Також раніше знайдені підрядки можна використовувати при заміні за регулярним виразом

Конструкції зворотніх посилань

Конструкція	Опис	Шаблон	Відповідність
\ number	Зворотнє посилання. Відповідає значенню нумерованої частини виразу	(\w)\1	"ee" в "seek"
\k< name >	Іменоване зворотнє посилання. Відповідає значенню іменованого виразу.	(?<char>\w)\k<char>	"ee" в "seek"

Конструкції зворотних посилань

```
using System;
using System.Text.RegularExpressions;

public class Example
{
   public static void Main()
   {
      string pattern = @"(\w)\1";
      string input = "trellis llama webbing dresser swagger";
      foreach (Match match in Regex.Matches(input, pattern))
         Console.WriteLine("Found '{0}' at position {1}. ",
                           match.Value, match.Index);
   }
}
```

Конструкції зворотніх посилань

```
// The example displays the following output:
//       Found 'll' at position 3.
//       Found 'll' at position 8.
//       Found 'bb' at position 16.
//       Found 'ss' at position 25.
//       Found 'gg' at position 33.
```

Підпримка регулярних виразів в мові C#

- Основна функціональність регулярних виразів в .NET зосереджена в просторі імен **System.Text.RegularExpressions**.

- Центральним класом при роботі з регулярними виразами є клас **Regex**.

Наприклад, у нас є деякий текст і нам треба знайти в ньому все словоформи якогось слова.

З класом **Regex** це зробити дуже просто:

Метод **Matches** класу **Regex** приймає рядок, до якого треба застосувати регулярний вираз, і повертає колекцію знайдених збігів.

Кожен елемент такої колекції представляє об'єкт **Match**. Його властивість **Value** повертає знайдений збіг.

Приклад 1 (сайт http://metanit.com)

```
string s = "Бык тупогуб, тупогубенький бычок, у быка губа бела
    была тупа";
Regex regex = new Regex(@"туп(\w*)“);
MatchCollection matches = regex.Matches(s);
if (matches.Count > 0)
{
    foreach (Match match in matches)
        Console.WriteLine(match.Value);
}
else
{
    Console.WriteLine("Совпадений не найдено“);
}
```

Приклад 2. Заміна підрядків

```
using System;
using System.Text.RegularExpressions;
public class Example{
  public static void Main()   {
  string pattern = "(Mr\\.? |Mrs\\.? |Miss |Ms\\.? )";
  string[] names = { "Mr. Henry Hunt", "Ms. Sara Samuels",
     "Abraham Adams", "Ms. Nicole Norris" };
   foreach (string name in names)
     Console.WriteLine(Regex.Replace(name, pattern,
     String.Empty));
  }
}
// The example displays the following output:
//     Henry Hunt
//     Sara Samuels
//     Abraham Adams
//    Nicole Norris
```

Приклад 3. Визначення слів, що повторюються

```
using System;
using System.Text.RegularExpressions;

public class Class1 {

public static void Main() {

string pattern = @"\b(\w+?)\s\1\b";

string input = "This this is a nice day. What about this? This tastes
    good. I saw a a dog.";

foreach (Match match in Regex.Matches(input, pattern,
    RegexOptions.IgnoreCase))

Console.WriteLine("{0} (duplicates '{1}') at position {2}", match.Value,
    match.Groups[1].Value, match.Index);

}

}

// The example displays the following output:

// This this (duplicates 'This)' at position 0

// a a (duplicates 'a)' at position 66
```

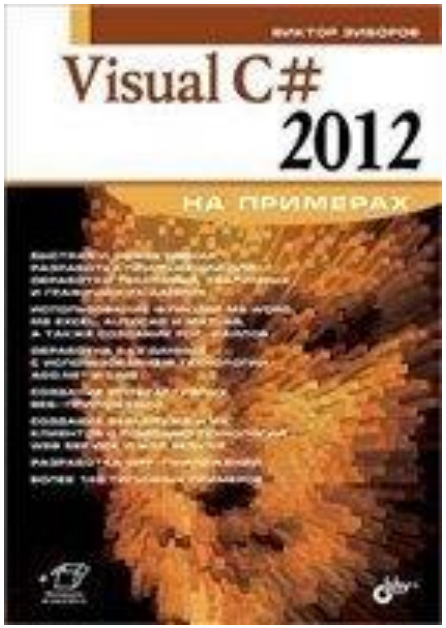
Приклад 4. Метод IsMatch(String)

```
using System;
using System.Text.RegularExpressions;

public class Example {

public static void Main() {

string[] partNumbers= { "1298-673-4192", "A08Z-931-468A", "_A90-123-
    129X", "12345-KKA-1230", "0919-2893-1256" };

Regex rgx = new Regex(@"^[a-zA-Z0-9]\d{2}[a-zA-Z0-9](-\d{3}){2}[A-Za-
    z0-9]$");

foreach (string partNumber in partNumbers)
    Console.WriteLine("{0} {1} a valid part number.", partNumber,
    rgx.IsMatch(partNumber) ? "is" : "is not");
 }
}

// The example displays the following output:
// 1298-673-4192 is a valid part number.
// A08Z-931-468A is a valid part number.
// _A90-123-129X is not a valid part number.
// 12345-KKA-1230 is not a valid part number.
// 0919-2893-1256 is not a valid part number.
```

Інтеграція C# з MS Office

- **Зиборов В.В. Visual C# 2012 на примерах, Глава 9, с.209**
- Пошаговое руководство. Программирование приложений Office (C# и Visual Basic)

  http://msdn.microsoft.com/ru-ru/library/ee342218.aspx
- Джон Скит C# Пр-е для проф-лов, стр.422



To set up an Excel Add-in application

- Start Visual Studio.
- On the **File** menu, point to **New**, and then click **Project**.
- In the Installed **Templates** pane, expand Visual Basic or **Visual C#,** expand **Office**, and then click **2010** (or 2007 if you are using Office 2007).
- In the Templates pane, click **Excel 2010 Add-in** (or Excel 2007 Add-in).
- Look at the top of the Templates pane to make sure that **.NET Framework 4** appears in the Target Framework box.
- **Type a name** for your project in the Name box, if you want to.
- Click **OK**.
- The new project appears in Solution Explorer.

Приклад взаємодії з MS Excel (MSDN)

```
using Excel = Microsoft.Office.Interop.Excel;
using Word = Microsoft.Office.Interop.Word;
```

Додаємо клас

```
class Account {
    public int ID { get; set; }
    public double Balance { get; set; }
}
```

Створюємо список рахунків

```
var bankAccounts = new List<Account> {
    new  Account { ID = 345, Balance = 541.27 },
    new  Account { ID = 123, Balance = -127.44 }
};
```

Приклад взаємодії з MS Excel (MSDN)

To export data to Excel

```
void DisplayInExcel(IEnumerable<Account> accounts,
    Action<Account, Excel.Range> DisplayFunc) {
    var excelApp = this.Application;
    // Add a new Excel workbook.
    excelApp.Workbooks.Add();
    excelApp.Visible = true;
    excelApp.Range["A1"].Value = "ID";
    excelApp.Range["B1"].Value = "Balance";
    excelApp.Range["A2"].Select();
    foreach (var ac in accounts) {
        DisplayFunc(ac, excelApp.ActiveCell);
        excelApp.ActiveCell.Offset[1, 0].Select();
    }
    // Copy the results to the Clipboard.
    excelApp.Range["A1:B3"].Copy();
}
```

Приклад взаємодії з MS Excel (MSDN)

- У методу Add мається **необов'язковий параметр** для зазначення конкретного шаблону.
- Необов'язкові параметри, що вперше з'явилися в Visual C # 2010, дозволяють опускати аргумент для таких параметрів.
- **Властивості Range і Offset об'єкта Range використовують функцію індексованих властивостей**.
- Вона дозволяє використовувати властивості типів COM за допомогою стандартного синтаксису C#.
- Крім того, індексовані властивості дозволяють використовувати властивість **Value** об'єкта **Range**, усуваючи необхідність у використанні властивості Value2.
- Властивість Value є індексованою, але індекс - необов'язковий

Приклад взаємодії з MS Excel (MSDN)

Додайте в кінець методу DisplayInExcel наступний код, щоб ширина стовпця змінювалася відповідно до вмісту:

```
excelApp.Columns[1].AutoFit();
excelApp.Columns[2].AutoFit();
```

Виклик методу DisplayInExcel

```
DisplayInExcel(bankAccounts, (account, cell) =>
{
    cell.Value = account.ID;
    cell.Offset[0, 1].Value = account.Balance;
    if (account.Balance < 0)
    {       cell.Interior.Color = 255;
            cell.Offset[0, 1].Interior.Color = 255;
    }
});
```

Додавання документа Word

Додайте в кінець методу StartUp наступний код, щоб створити документ Word, що містить посилання на книгу Excel.

```
var wordApp = new Word.Application();
wordApp.Visible = true;
wordApp.Documents.Add();
wordApp.Selection.PasteSpecial(Link: true, DisplayAsIcon: true);
```

У цьому коді демонструються деякі нові функції C#:

- можливість опускати ключове слово **ref** при програмуванні в моделі **COM**,
- іменовані аргументи і
- необов'язкові аргументи.

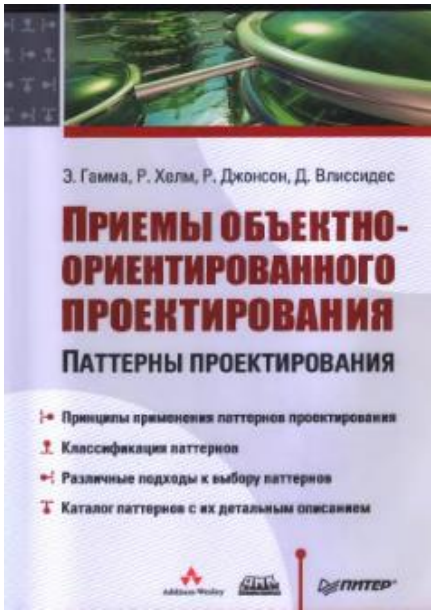
(Метод PasteSpecial приймає 7 параметрів…)

Знайомство з шаблонами проектування

Design Patterns: Elements of Reusable Object-Oriented Software (Прийоми об'єктно-орієнтованого проектування. Паттерни проектування) –

книга 1994 р. про інженерію програмного забезпечення, яка описувала **вирішення деяких частих проблем у проектуванні ПЗ.**

Автори книги: **Еріх Гамма** (Erich Gamma), **Річард Хелм** (Richard Helm), **Ральф Джонсон** (Ralph Johnson), **Джон Вліссідс** (John Vlissides). Колектив авторів також відомий як «Банда чотирьох», Gang of Four, **GoF**. Автор передмови Граді Буч (Grady Booch).



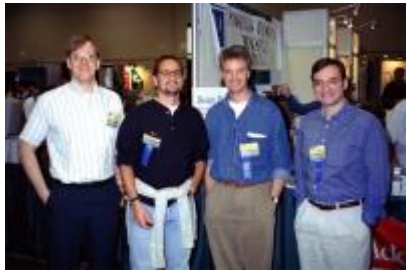
Книга складається з двох частин, в перших двох главах розповідається про ***можливості і недоліки об'єктно-орієнтованого програмування***, а в другій частині описані **23 класичних шаблони проектування**. Приклади в книзі написані на мовах програмування C ++ і Smalltalk.

Еріх Гамма (1961 рік, Цюріх) - програміст зі Швейцарії, один з чотирьох авторів класичної книги Design Patterns про шаблони проектування. Команда авторів книги також відома під назвою «банда чотирьох» (англ. Gang of Four, GoF).

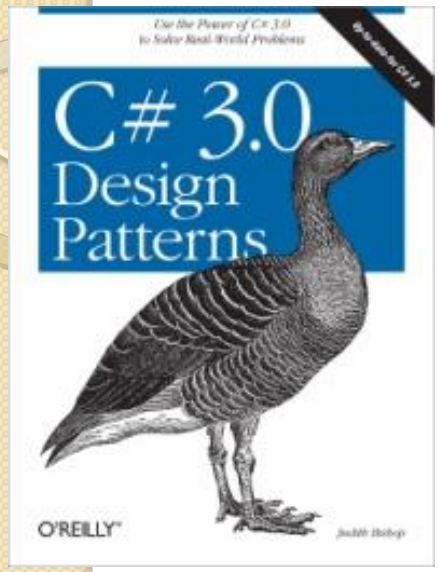
Є провідним розробником **JUnit** (фреймворка для виконання юніт-тестів на Java) і **Eclipse** (крос-платформного інтегрованого середовища розробки програмного забезпечення).

Працював в IBM над проектом Jazz.

З 2011 року керує командою розробки **Microsoft Visual Studio** в Цюріху, Швейцарія



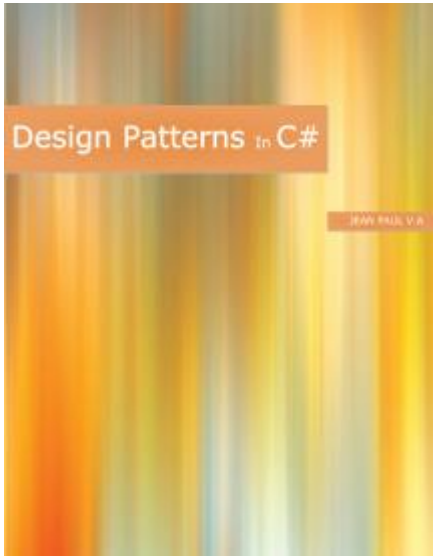
Література


C# 3.0
Design
Patterns
O'REILLY


Элементарные
шаблоны
проектирования
Джейсон Мак-Колм Смит
Предисловие Гради Буча


ПАТТЕРНЫ
проектирования
O'REILLY


МАРК ГРАНД
ШАБЛОНЫ
ПРОЕКТИРОВАНИЯ
В
JAVA
4tivo.com


Design Patterns in C#


Р. С. МАРТИН,
М. МАРТИН
Принципы, паттерны
и методики гибкой
разработки
на языке C#
HIGH TECH

Знайомство з шаблонами проектування

Породжуючі шаблони

- **Abstract Factory** - Абстрактна фабрика
- **Builder** – Будівник
- **Factory Method** - Фабричний метод
- **Prototype** – Прототип
- **Singleton** – Одинак

Структурні шаблони

- **Adapter** – Адаптер
- **Bridge** – Міст
- **Composite** – Компоновщик
- **Decorator** – Декоратор
- **Facade** – Фасад
- **Flyweight** – Пристосуванець
- **Proxy** – Замісник

Поведінкові шаблони

- **Chain of responsibility** - Ланцюжок обов'язків
- **Command** – Команда
- **Interpreter** – Інтерпретатор
- **Iterator** – Ітератор
- **Mediator** – Посередник
- **Memento** – Зберігач
- **Observer** – Спостерігач
- **State** – Стан
- **Strategy** – Стратегія
- **Template method** - Шаблоновий метод
- **Visitor** - Відвідувач

Що таке патерн проектування

Крістофер Александр:

“Будь-який патерн

- **описує задачу, яка знову і знову виникає в нашій роботі,**
- **а також принцип її вирішення, причому таким чином, що це рішення можна потім**
- ***використовувати мільйон разів*, нічого не вигадуючи заново”**

GoF:

“Під патернами проектування розуміється опис взаємодії об'єктів і класів, адаптованих для вирішення загальної задачі проектування в конкретному контексті”

Труднощі проектування

Сама важка задача в об'єктно-орієнтованому проектуванні - розкласти систему на об'єкти.

При вирішенні доводиться враховувати безліч факторів:

- інкапсуляцію,
- глибину деталізації,
- наявність залежностей,
- гнучкість,
- продуктивність,
- розвиток,
- повторне використання і т.д.

Багато об'єктів виникають у проекті з побудованої в ході аналізу моделі.

Але нерідко з'являються і **класи, у яких немає прототипів в реальному світі**.

Патерни проектування допомагають виявити не цілком очевидні абстракції і об'єкти, які можуть їх використовувати.

Патерн Singleton (Одинак)

Одинак – патерн, що породжує об’єкти.

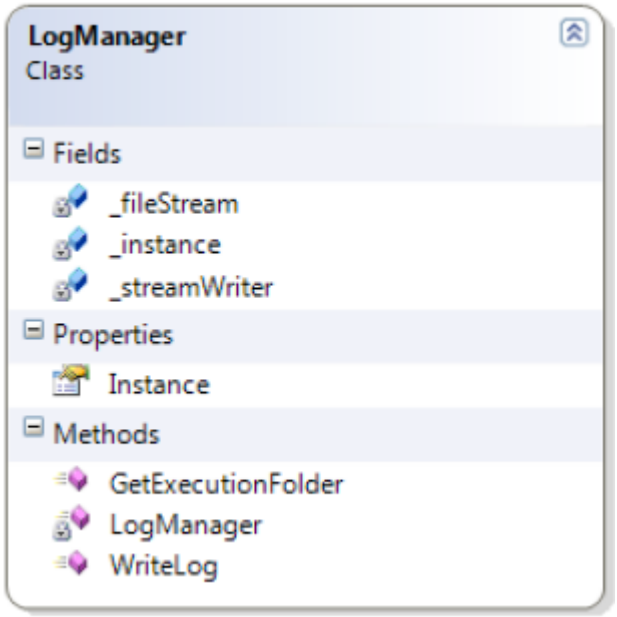
Призначення

Гарантує, що у класа є тільки один екземпляр, і надає до нього точку доступу.

```
public class Singleton
{
   private static Singleton instance;     // The single instance
   static Singleton(){       // Initialize the single instance
         instance = new Singleton();
   }
   public static Singleton Instance{      // The property
         get { return instance; }        // for retrieving the single instance
   }
   private Singleton() { }    // Private constructor:
}                                  //protects against direct instantiation
```

Приклад. Клас LogManager

```
public class LogManager{
    private static LogManager    _instance;
    public static LogManager Instance {
        get{  if (_instance == null)
            _instance = new LogManager();
            return  _instance;
        }
    }
    private LogManager() { // Constructor as Private
        _fileStream = File.OpenWrite(GetExecutionFolder() +
                    "\\Application.log");
        _streamWriter = new StreamWriter(_fileStream);
    }
    private FileStream _fileStream;
    private StreamWriter _streamWriter;
```



Приклад. Клас LogManager

```
public void WriteLog(string message){
    StringBuilder formattedMessage = new StringBuilder();
    formattedMessage.AppendLine("Date: " +
    DateTime.Now.ToString());
    formattedMessage.AppendLine("Message: " + message);
    _streamWriter.WriteLine(formattedMessage.ToString());
    _streamWriter.Flush();
}
public string GetExecutionFolder() {
    return
Path.GetDirectoryName(System.Reflection.Assembly.GetExecutingAs
    sembly().Location);}
}
```

For invoking the write method we can use the following code:

```
LogManager.Instance.WriteLog("Test Writing");
```

Патерн Factory Method

Фабричний метод – патерн, що породжує класи

Призначення

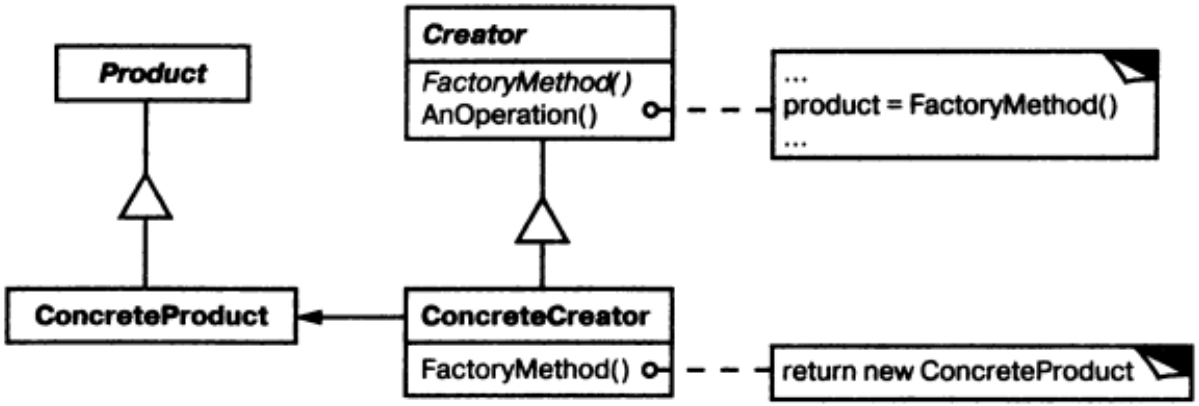
Визначає інтерфейс для створення об'єкта, але залишає підкласам рішення про те, який клас інстанціювати. Фабричний метод дозволяє класу делегувати інстанціювання підкласам.

Застосування

Використовуйте фабричний метод, коли:

- Класу заздалегідь невідомо, об'єкти яких класів йому потрібно створювати

Структура

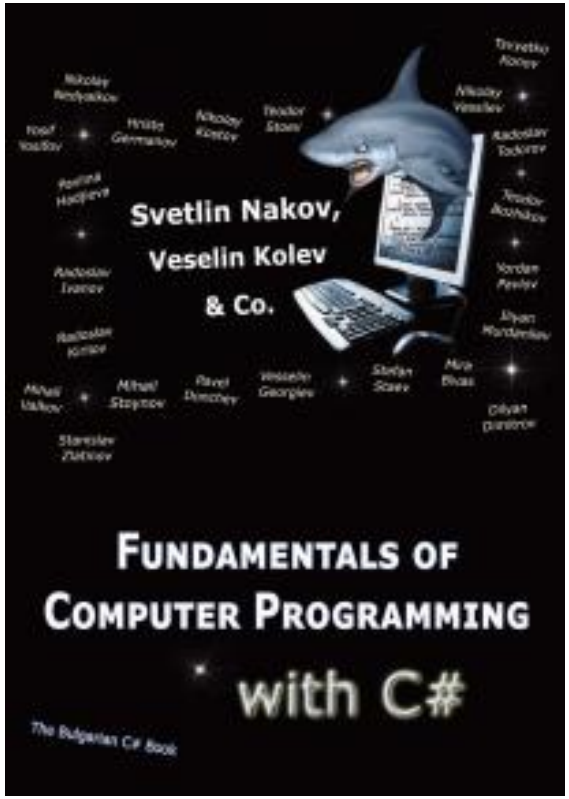


Приклад (Nakov, p.849)

Припустимо, у нас є клас, який містить графічні файли (PNG, JPEG, BMP, і т.д.) і створює їх копії зменшеного розміру (так звані мініатюри, thumbnails).

Підтримуються різні формати, кожен з яких представлений класом.

```
public class Thumbnail
{
// …
}
public interface Image
{
Thumbnail CreateThumbnail();
}
```



Приклад (Nakov, p.849)

```
public class GifImage : Image{
    public Thumbnail CreateThumbnail(){
        // … Create a GIF thumbnail here …
        return gifThumbnail;
    }
}

public class JpegImage : Image{
    public Thumbnail CreateThumbnail(){
        // … Create a JPEG thumbnail here …
        return jpegThumbnail;
    }
}
```

Приклад (Nakov, p.849)

Here is how the class holding an album of images looks like:

```
public class ImageCollection{
    private IList<Image> images;
    public ImageCollection(IList<Image> images){
        this.images = images;
    }
    public IList<Thumbnail> CreateThumbnails(){
        IList<Thumbnail> thumbnails =
                new List<Thumbnail>(images.Count);
        foreach (Image thumb in images){
                thumbnails.Add(thumb.CreateThumbnail());
        }
        return thumbnails;
    }
}
```

Приклад (Nakov, p.849)

The client of the program may require thumbnails of all images in the album:

```
public class Example
{
    static void Main()
    {
        IList<Image> images = new List<Image>();
        images.Add(new JpegImage());
        images.Add(new GifImage());
        ImageCollection imageRepository =
                new ImageCollection(images);
        Console.WriteLine(imageRepository.CreateThumbnails());
    }
}
```

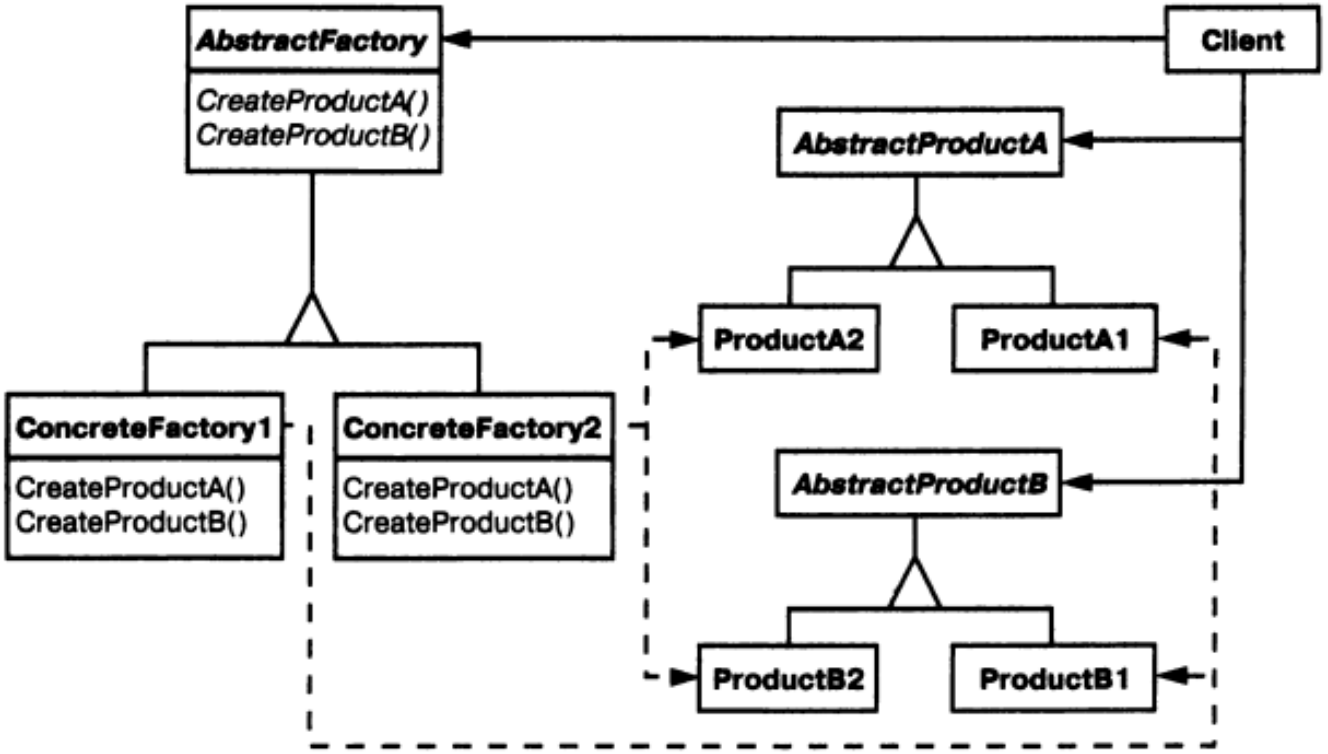
Патерн Abstract Factory

Абстрактна фабрика – патерн, що породжує об’єкти.

Призначення

Надає інтерфейс для створення сімейств взаємопов'язаних або взаємозалежних об'єктів, не специфікуючи їх конкретних класів.

Структура



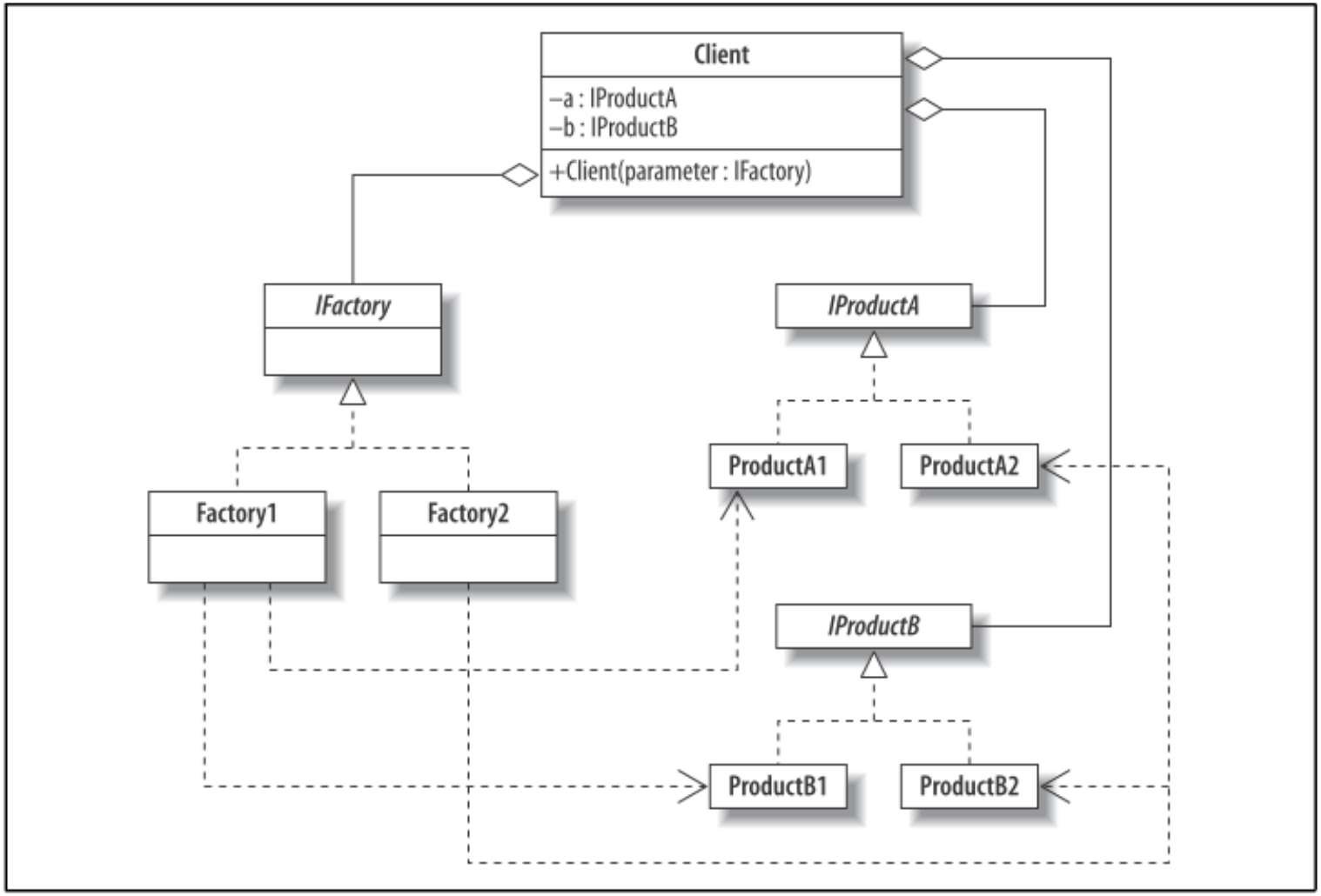
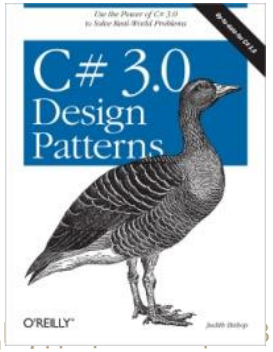


Figure 6-2. Abstract Factory pattern UML diagram



Illustration

Figure 6-1. Abstract Factory pattern illustration—replica Gucci handbags (Poochies)

`Client`	Customer buying a handbag
`AbstractFactory`	Returns a handbag
`FactoryA`	Gucci
`FactoryB`	Poochy
`AbstractProduct`	Handbags
`ProductA1`	Genuine bags
`ProductB1`	Replica bags

Implementation and Example: Gucci and Poochy

```
interface IFactory<Brand>
  where Brand : IBrand {
   IBag CreateBag();
   IShoes CreateShoes();
}
 // Factories (both in the same one)
 class Factory<Brand> : IFactory<Brand>
     where Brand : IBrand, new() {
   public IBag CreateBag() {
     return new Bag<Brand>();
   }
   public IShoes CreateShoes() {
     return new Shoes<Brand>();
   }
 }
```

Implementation and Example: Gucci and Poochy

In our example, the two Products are bags and shoes. Bags indicate what they are made of, and shoes indicate their price.

```
// Product 1
interface IBag {
  string Material { get; }
}

// Concrete Product 1
 class Bag<Brand> : IBag where Brand : IBrand, new() {
   private Brand myBrand;
   public Bag() {
        myBrand = new Brand();
  }
 public string Material { get { return myBrand.Material; } }
}

……..
```

Implementation and Example: Gucci and Poochy

```
// An interface for all Brands
  interface IBrand {
     int Price { get; }
     string Material { get; }
  }

class Gucci : IBrand {
  public int Price { get { return 1000; } }
  public string Material { get { return "Crocodile skin"; } }
}
. . . . . .. . . .
```

Implementation and Example: Gucci and Poochy

```
class Client<Brand> where Brand : IBrand, new() {
  public void ClientMain() {
      IFactory<Brand> factory = new Factory<Brand>();
      IBag bag = factory.CreateBag();
      IShoes shoes = factory.CreateShoes();
      Console.WriteLine("I bought a Bag which is made from " +
                        bag.Material);
       Console.WriteLine("I bought some shoes which cost " +
                        shoes.Price);
        }
  }
. . . . . .
static void Main() {
        new Client<Poochy>().ClientMain();
        new Client<Gucci>().ClientMain();
```

Патерн Command

Команда – патерн поведінки об’єктів.

Призначення

Інкапсулює запит як об’єкт, дозволяючи тим самим задавати параметри клієнтів для обробки відповідних запитів, ставити запити в чергу, а також підтримувати відміну операцій.

Мотивація

Іноді необхідно посилати об’єктам запити, нічого не знаючи про те, виконання якої операції запитано і хто є отримувачем.

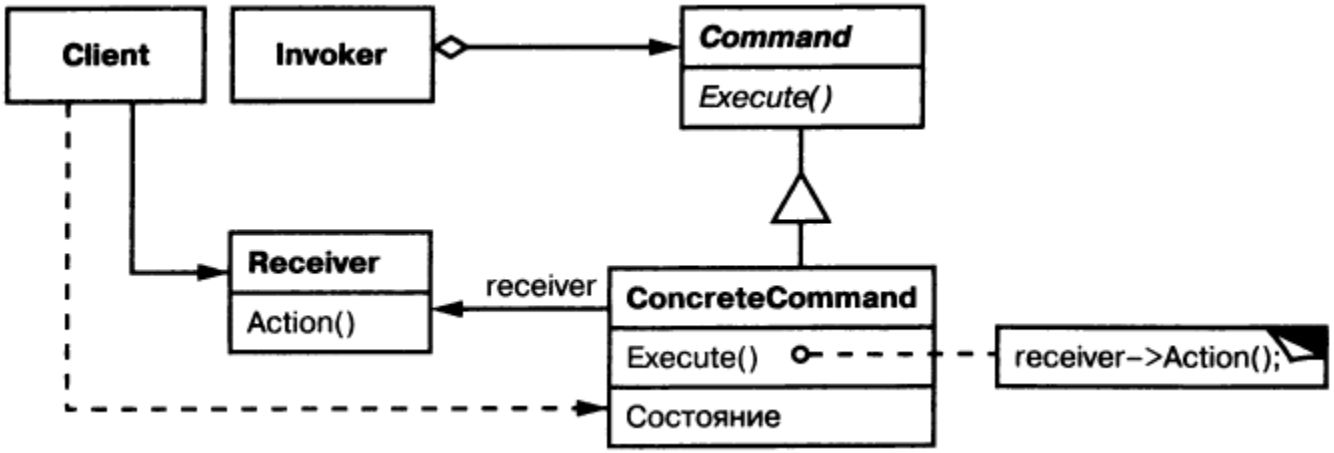
Наприклад, в бібліотеках для побудови користувацьких інтерфейсів, є такі об’єкти, як кнопки та меню, які відсилають запит у відповідь на дію користувача.

Але в саму бібліотеку не закладена можливість обробляти цей запит.

Проектувальник бібліотеки не володіє ніякою інформацією про отримувача запиту і про те, які операції той повинен виконати.

Патерн Command

Структура



Учасники:

Command – команда: оголошує інтерфейс для виконання операції.

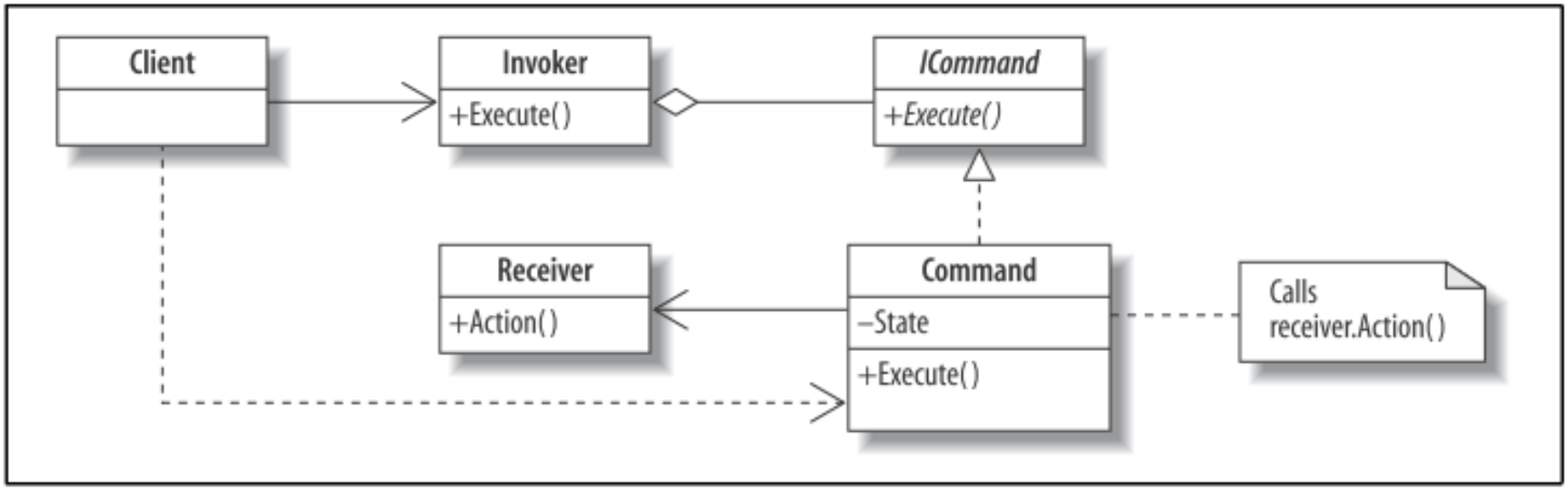
ConcreteCommand – конкретна команда: визначає зв'язок між об'єктом-отримувачем Receiver і дією, реалізує операцію Execute шляхом виклику відповідної операції об'єкта Receiver.

Client – клієнт: створює об'єкт ConcreteCommand і встановлює його отримувача.

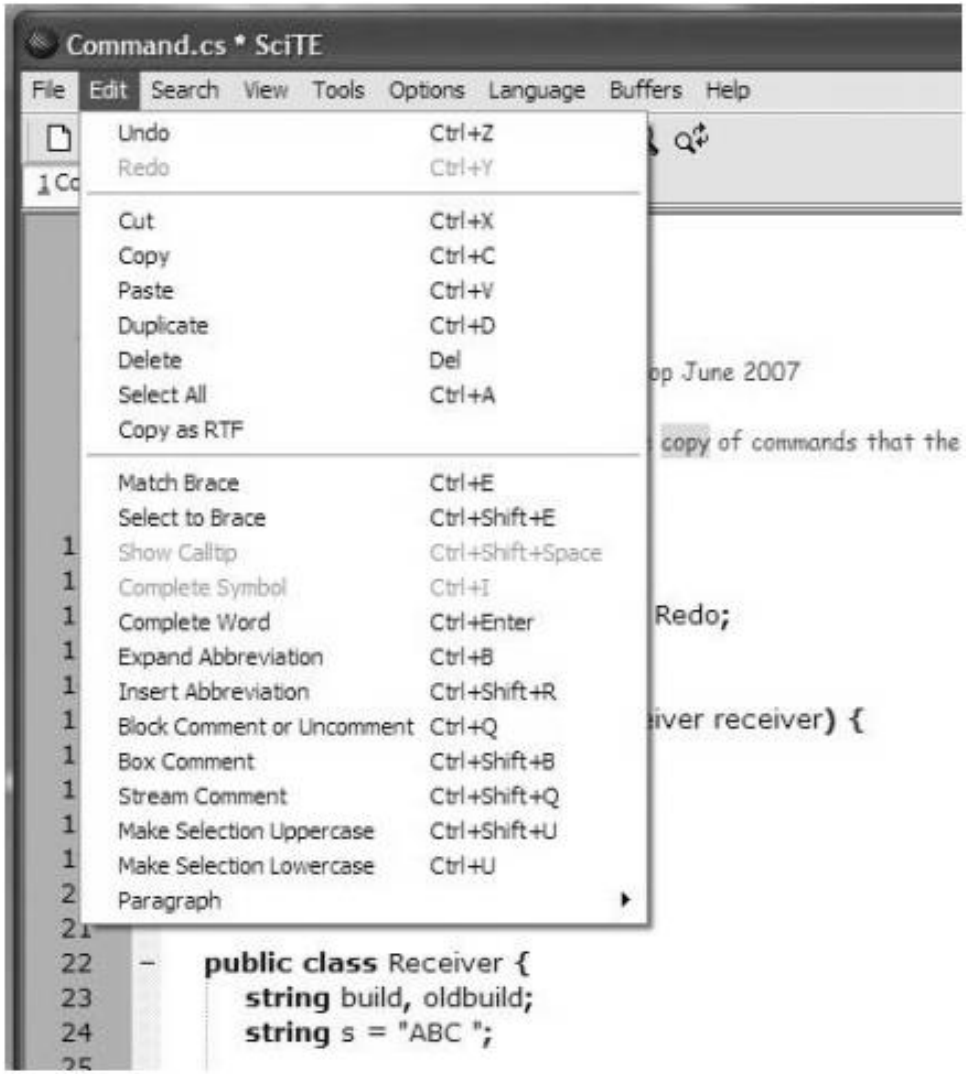
Invoker – ініціатор: звертається до команди для виконання запиту.

Receiver – отримувач: має інф-ю про способи виконання операцій.

Command pattern UML diagram



Command pattern illustration—menus

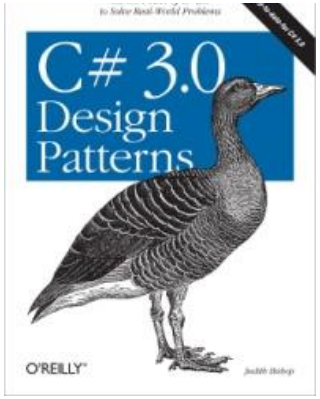


QUIZ

Match the Command Pattern Players with the Cut Request Illustration

To test whether you understand the Command pattern, cover the lefthand column of the table below and see if you can identify its players among the items from the illustrative example (Figure 8-5), as shown in the righthand column. Then check your answers against the lefthand column.

Client	User of the editor, who selects a Cut command on a menu
ICommand	Menu
Command	Holder of a Cut request
Receiver	An object that can perform a Cut
Invoker	Process the selection of a Cut option
Action	Cut



Implementation (p.201)

```
using System;
class CommandPattern {
   delegate void Invoker ();
    static Invoker Execute, Undo, Redo;
     class Command {
        public Command(Receiver receiver) {
                    Execute = receiver.Action;
                    Redo = receiver.Action;
                    Undo = receiver.Reverse;
        }
     }
     public class Receiver {
       string build, oldbuild;
        string s = "some string ";
       public void Action() {
                oldbuild = build;
```

```
            build +=s;
            Console.WriteLine("Receiver is adding "+build);
        }
        public void Reverse() {
            build = oldbuild;
            Console.WriteLine("Receiver is reverting to "+build);
        }
}
static void Main() {
    new Command (new Receiver());
    Execute();
    Redo();
    Undo();
    Execute();
    }
 }
  /* Output
 Receiver is adding some string
 Receiver is adding some string some string
 Receiver is reverting to some string
 Receiver is adding some string some string
```

Ще приклад

Принципы, паттерны и методики гибкой разработки на языке C#, стр.341

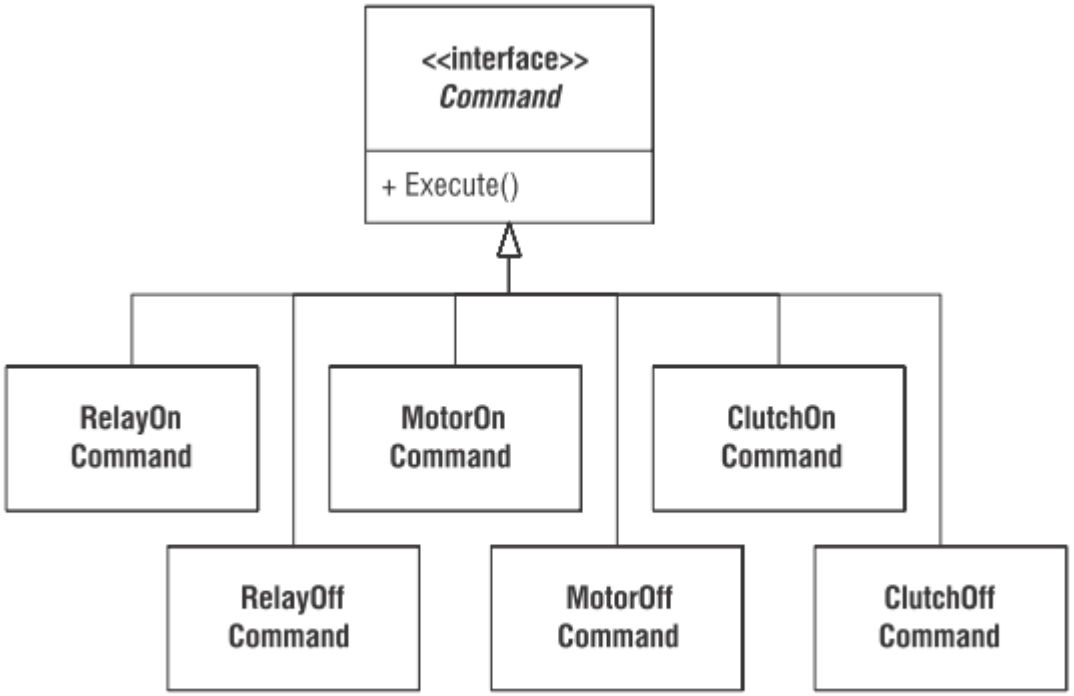


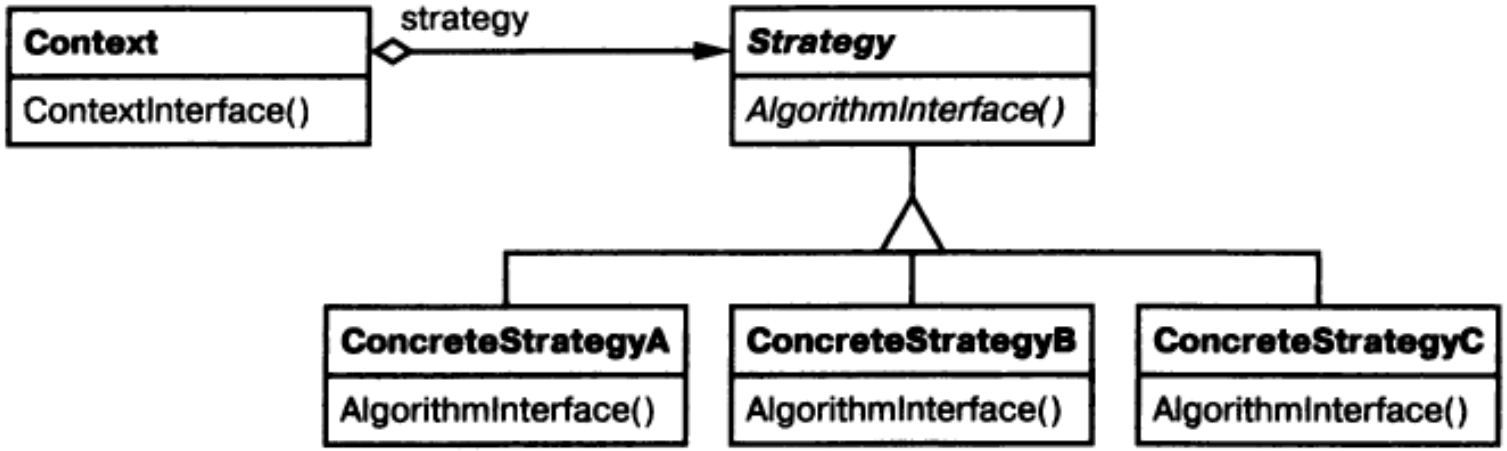
Рис. 21.2. Несколько простых команд для программы управления копировальным устройством

Патерн Strategy

Стратегія – патерн поведінки об'єктів.

Призначення

Визначає сімейство алгоритмів, інкапсулює кожен з них і робить їх взаємозамінними. Стратегія дозволяє змінювати алгоритми незалежно від клієнтів, які ними користуються.



Патерн Strategy

Учасники

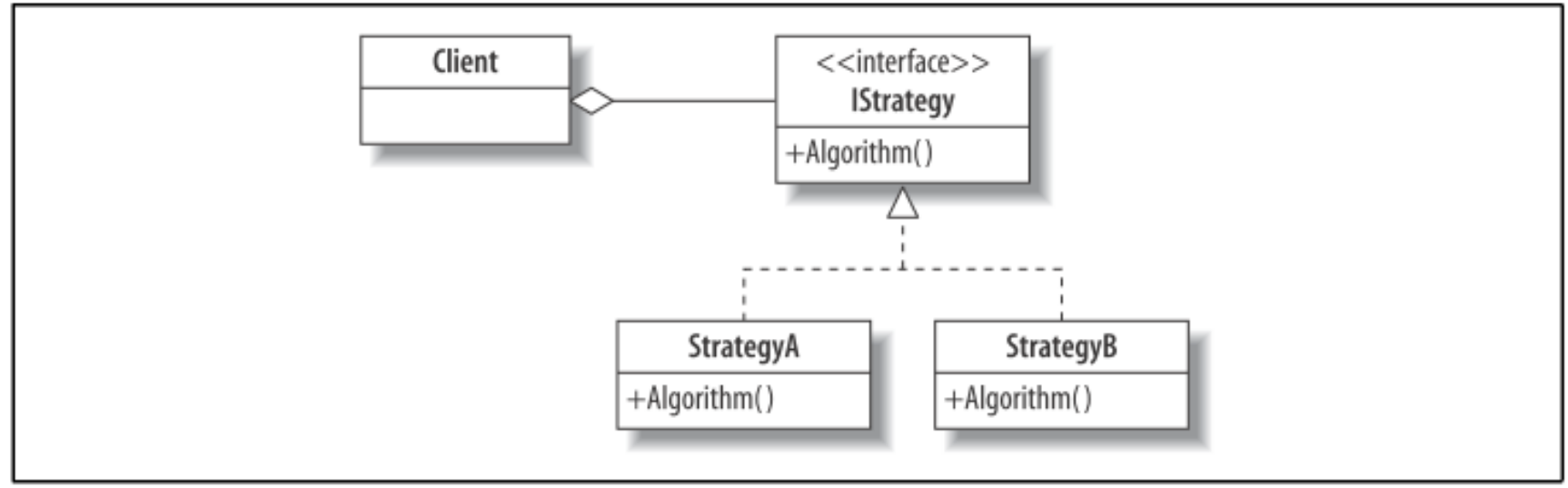
Strategy – стратегія: оголошує спільний для всіх підтримуваних алгоритмів інтерфейс. Клас Context користується цим інтерфейсом для виклику конкретного алгоритма, визначеного в класі ConcreteStrategy.

ConcreteStrategy – конкретна стратегія: реалізує алгоритм, що використовує інтерфейс, визначений в класі Strategy.

Context – контекст:

- конфігурується об'єктом класу ConcreteStrategy,
- зберігає посилання на об'єкт класу Strategy,
- може визначати інтерфейс, який дозволяє об'єкту Strategy отримати доступ до даних контекста.

Strategy pattern UML diagram



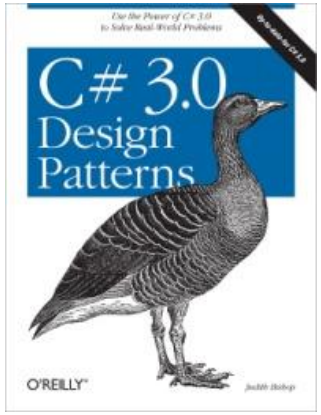
QUIZ

Match the Strategy Pattern Players with the Sorting Animator Illustration

To test whether you understand the Strategy pattern, cover the lefthand column of the table below and see if you can identify its players among the items from the illustrative example (Figure 7-2), as shown in the righthand column. Then check your answers against the lefthand column.

`Context`	The GUI, the list generator, and the selection of the `Strategy`
`Strategy`	Class containing the methods used for a particular sort
`IStrategy`	Sorting interface (e.g., specifying the list and its item type)
`Algorithm`	Mergesort or Quicksort

- Стор. 164



```
using System;
class Context {
  // Context state
  public const int start = 5;
  public int Counter = 5;

  // Strategy aggregation
  IStrategy strategy = new Strategy1();

  // Algorithm invokes a strategy method
  public int Algorithm() {
    return strategy.Move(this);
  }
// Changing strategies
  public void SwitchStrategy() {
     if (strategy is Strategy1)
          strategy = new Strategy2();
     else
          strategy = new Strategy1();
     }
  }
```

```
// Strategy interface
 interface IStrategy {
     int Move (Context c);
 }

// Strategy 1
  class Strategy1 : IStrategy {
      public int Move (Context c) {
         return ++c.Counter;
      }
   }
 // Strategy 2
  class Strategy2 : IStrategy {
     public int Move (Context c) {
       return --c.Counter ;
      }
   }

 // Client
    static class Program {
          static void Main () {
          Context context = new Context();
```

```
context.SwitchStrategy();
Random r = new Random(37);
for (int i=Context.start; i<=Context.start+15; i++) {
        if (r.Next(3) == 2) {
            Console.Write("|| ");
            context.SwitchStrategy();
        }
        Console.Write(context.Algorithm() +"  ");
 }
      Console.WriteLine();
}
}
   /* Output
    4  || 5  6  7  || 6  || 7  8  9  10  || 9  8  7  6  || 7  || 6  5
   */
```

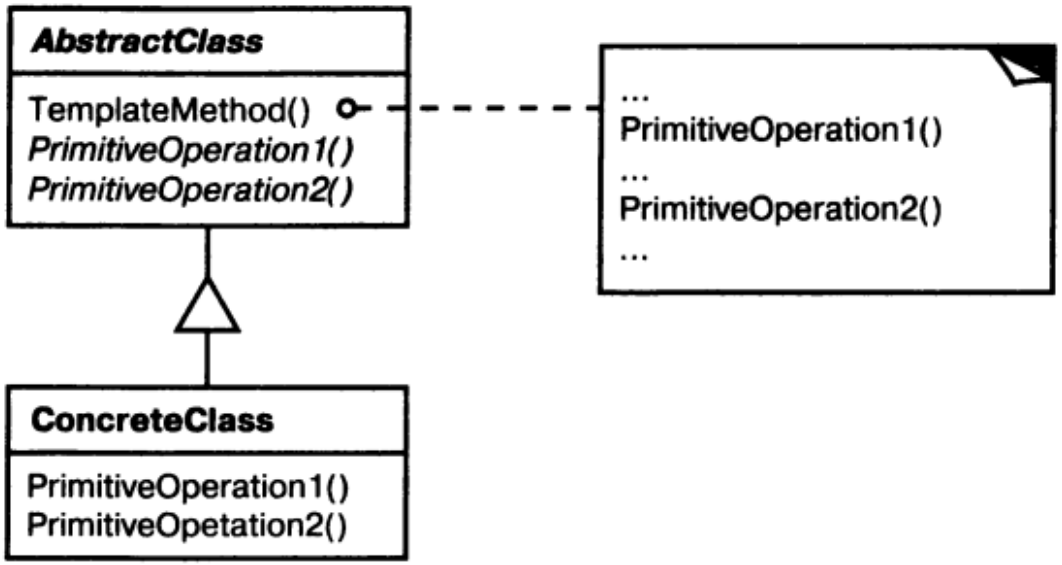
Патерн Template Method

Шаблонний метод – патерн поведінки класів.

Призначення

Шаблонний метод визначає основу алгоритму і дозволяє підкласам перевизначити деякі кроки алгоритму, не змінюючи його структуру в цілому.

Структура

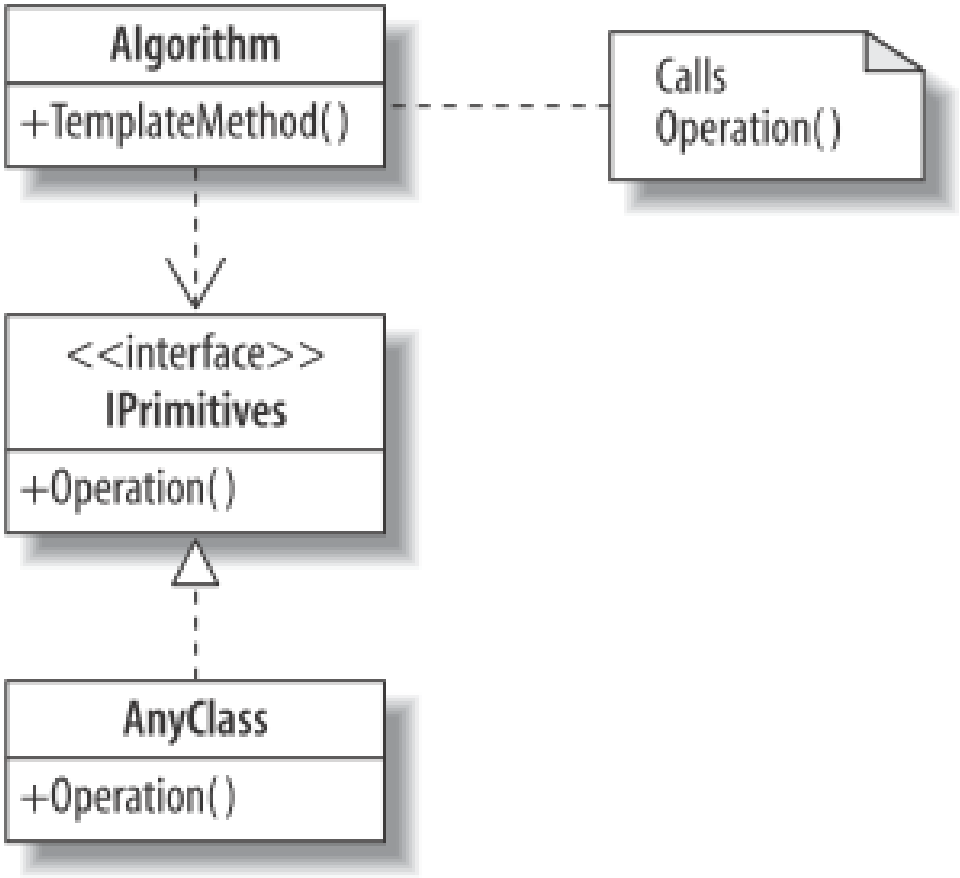


Учасники

AbstractClass – абстрактний клас:

- Визначає абстрактні примітивні операції
- Реалізує шаблонний метод, який викликає ці операції

Template Method pattern UML diagram



Патерн Template Method

Учасники

ConcreteClass – конкретний клас:

- Реалізує примітивні операції, що виконують кроки алгоритму, способом, який залежить від підкласу.

Реалізація (C# 3.0 Design Patterns, p.183 (160))

```
using System;
interface IPrimitives {
    string Operation1();
    string Operation2();
}
class Algorithm {
    public void TemplateMethod(IPrimitives a) {
    string s = a.Operation1() + a.Operation2();
    Console.WriteLine(s);
    }
}
```

```
class ClassA : IPrimitives {
    public string Operation1() {
      return "ClassA:Op1 ";
    }
    public string Operation2() {
       return "ClassA:Op2 ";
    }
}

class ClassB : IPrimitives {
    public string Operation1() {
      return "ClassB:Op1 ";
    }
    public string Operation2() {
      return "ClassB.Op2 ";
    }
}
```

```
class TemplateMethodPattern {
    static void Main() {
            Algorithm m = new Algorithm();
            m.TemplateMethod(new ClassA());
            m.TemplateMethod(new ClassB());
    }
}

 /* Output
 ClassA:Op1 ClassA:Op2
 ClassB:Op1 ClassB.Op2
 */
```

22

Шаблонный метод и Стратегия: наследование или делегирование

Лучшая жизненная стратегия – усердие.
Китайская пословица

Стор.352

- На початку 1990-х років, коли об'єктно-орієнтовані технології тільки зароджувалися, всіх **нас захопила ідея спадкування**. Це відношення обіцяло грандіозні перспективи.
- Але, як більшість прекрасних нових світів, цей на повірку теж виявився не цілком придатним до проживання. До 1995 року стало ясно, що **спадкуванням дуже просто зловжити**, а обходиться таке зловживання вкрай дорого.
- Гамма, Хелм, Джонсон і Вліссідес навіть визнали доречним підкреслити: «**Віддавайте перевагу композиції об'єктів, а не спадкуванню класів**». Тому ми стали рідше застосовувати спадкування, частенько замінюючи його композицією або делегуванням.

- У цій главі ми розповімо про два патерни, які наочно ілюструють відмінності між спадкуванням та делегуванням. Патерни Шаблоновий метод і Стратегія призначені для розв'язання схожих завдань і нерідко взаємозамінні. Але **в Шаблоновому методі застосовується успадкування, а в Стратегії - делегування**.
- **І Шаблоновий метод, і Стратегія вирішують задачу відділення загального алгоритму від конкретного контексту.**
- Така необхідність виникає при проектуванні ПЗ часто-густо. Маємо алгоритм загального вигляду, який можна застосовувати до різних ситуацій. **Відповідно до принципу інверсії залежності ми хотіли б, щоб цей алгоритм не залежав від деталей реалізації. Бажано, щоб як алгоритм, так і конкретна реалізація залежали тільки від абстракцій**.

Патерн Facade

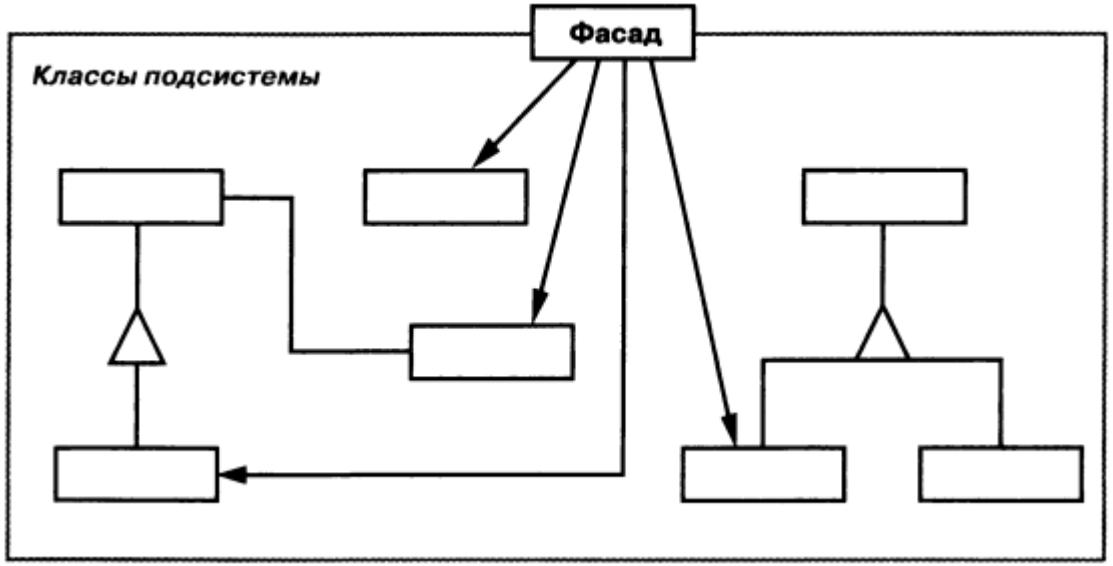
Фасад – патерн, що структурує об’єкти.

Призначення

Надає уніфікований інтерфейс замість набору інтерфейсів деякої підсистеми. Фасад визначає інтерфейс більш високого рівня, який спрощує використання підсистеми.

Патерн Facade

Структура



Учасники

Facade – фасад: "знає", яким класам підсистеми адресувати запит, делегує запити клієнтів підходящим об'єктам всередині підсистеми.

Класи підсистеми:

- Реалізують функціональність підсистеми
- Нічого не знають про існування Фасаду

Фасад и Посредник

Символизм воздвигает респектабельный фасад, чтобы скрыть за ним низменные фантазии.

Мейсон Кули

Стор.366.

Обидва паттерна, що розглядаються в цьому розділі, переслідують одну мету: накласти якусь політику на групу об'єктів. Фасад (Facade) накладає політику зверху, а Посередник (Mediator) - знизу. Фасад видимий і вводить обмеження, Посередник непомітний і ні в чому не обмежує.

Фасад

Патерн Фасад застосовується, коли потрібно надати простий спеціалізований інтерфейс до групи об'єктів, що мають складний загальний інтерфейс.

Розглянемо, наприклад, клас DB. Цей клас накладає дуже простий інтерфейс, специфічний для ProductData, на складні загальні інтерфейси класів з простору імен System.Data.

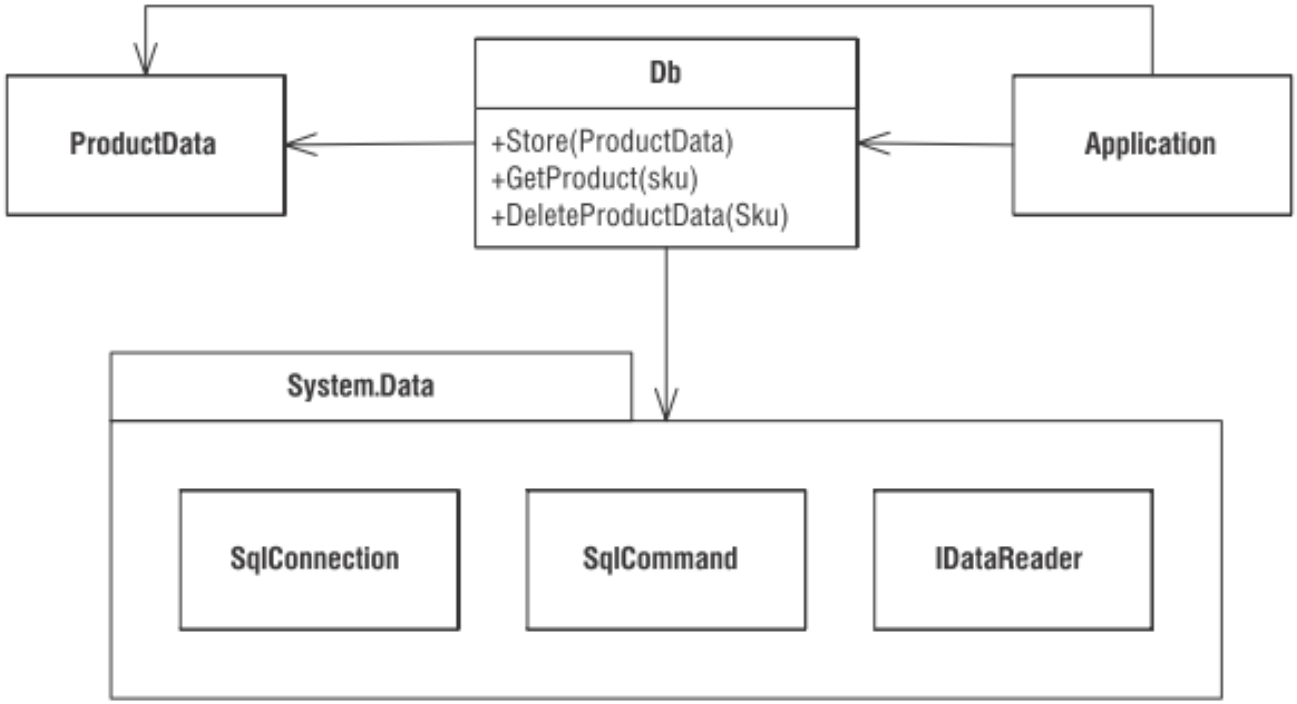


Рис. 23.1. Фасад DB

Клас DB позбавляє Application від необхідності вникати в тонкощі простору імен System.Data. Він приховує спільність і складність System.Data за простим спеціалізованим інтерфейсом.

Клас DB, що є окремим випадком Фасада, визначає політику використання System.Data;

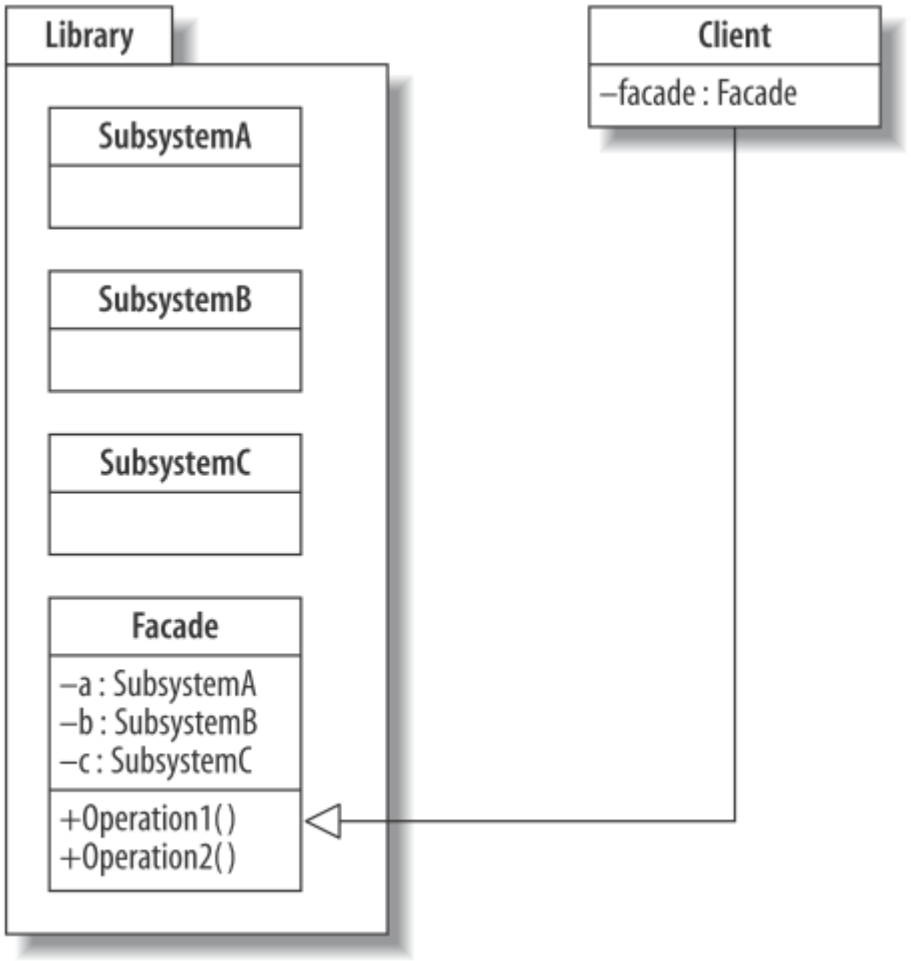
- він знає, як відкрити і закрити з'єднання з базою даних,
- як встановити відповідність між змінними-членами ProductData і полями бази даних,
- як будувати запити для маніпулювання даними.

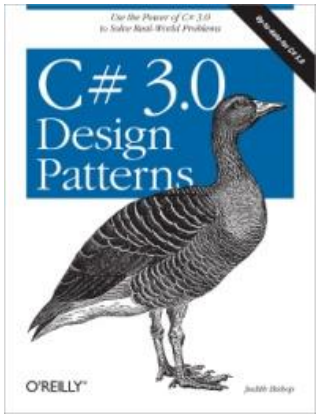
Використання паттерна Фасад має на увазі наступне: **розробники згодні з тим, що всі звернення до бази даних повинні проводитися тільки через клас DB**.

Таким чином, Фасад нав'язує додаткам свою політику.

Фасад

A good illustration is Amazon.com's 1-Click® system, which simplifies the process of ordering items for well-known customers.





Патерн Mediator

Посередник – патерн поведінки об'єктів.

Призначення

Визначає об'єкт, що інкапсулює спосіб взаємодії множини об'єктів. Посередник забезпечує слабку зв'язність системи, позбавляючи об'єкти від необхідності явно посилатися один на одного і дозволяючи тим самим незалежно змінювати взаємодію між ними.

Структура

Учасники

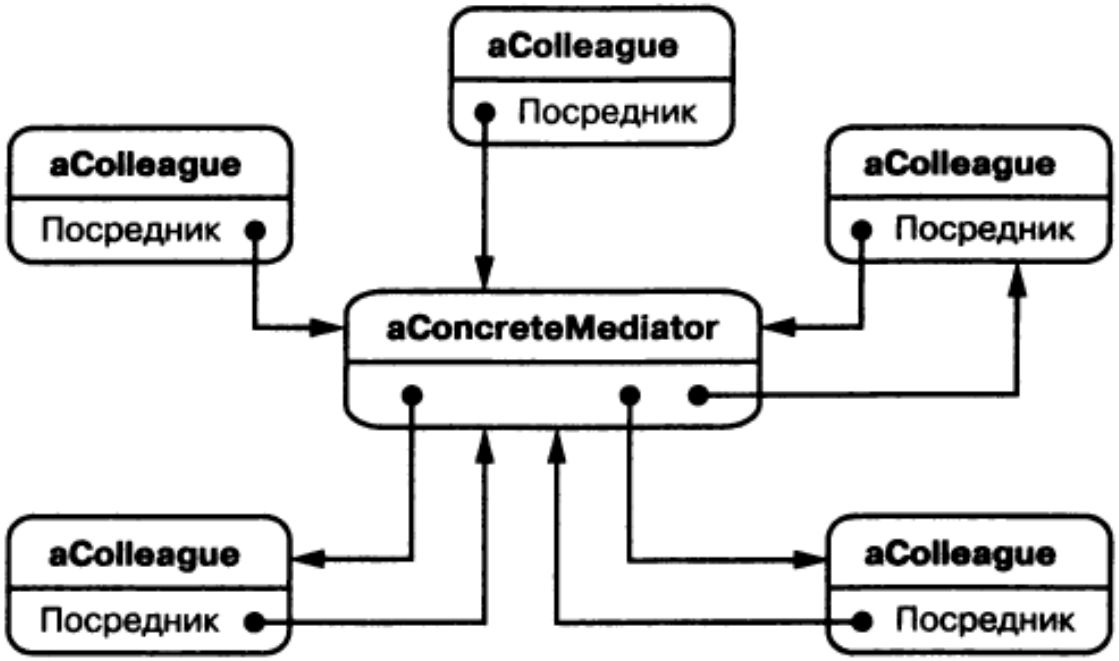
Mediator – посередник: визначає інтерфейс для обміну інформацією з об’єктами Colleague.

ConcreteMediator – конкретний посередник: реалізує кооперативну поведінку, координуючи дію об’єктів Colleague, володіє інформацією про колег.

Colleague – колеги: “знають” про посередника, діють через нього

Патерн Mediator

Типова структура об’єктів:



Приклад (C# 3.0 Design Patterns, p. 224 (201))

From:unscabbarded@talknlink.com

Action to take on all these held messages:

Defer Accept Reject Discard

☐ Preserve messages for the site administrator

☐ Forward messages (individually) to:

polelo-owner@cs.up.ac.za

☐ Add **unscabbarded@talknlink.com** to one of these sender filters:

Accepts Holds Rejects Discards

☐ Ban **unscabbarded@talknlink.com** from ever subscribing to this mailing list

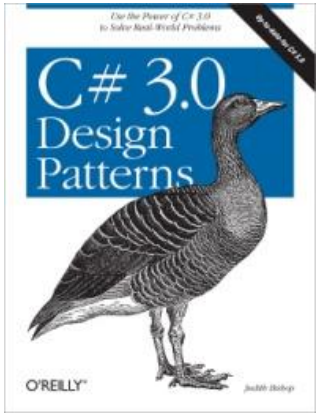
Click on the message number to view the individual message, or you can view all messages from unscabbarded@talknlink.com

[1] **Subject:** *****SPAM***** intellicall
Size: 3580 bytes
Reason: Post by non-member to a members-only list
Received: Mon Sep 24 09:43:44 2007

☐ Discard all messages marked *Defer*

Submit All Data

Figure 9-5. Mediator pattern illustration—mailing list moderation



Design

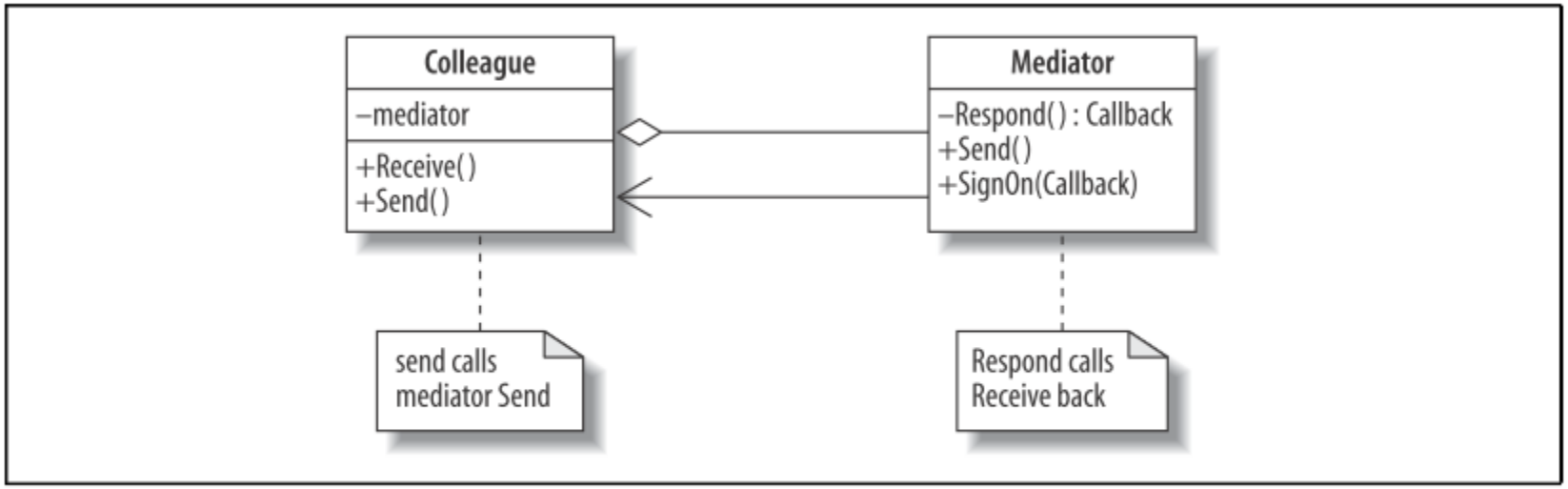


Figure 9-6. Mediator pattern UML diagram

Implementation

```
using System;
using System.Collections.Generic;
class MediatorPattern {
    class Mediator {
        public delegate void Callback (string message, string from);
        Callback respond;
```

```
        public void SignOn (Callback method) {
                respond += method;
        }
        public void Block (Callback method) {
                respond -=method;
        }
        public void Unblock (Callback method) {
                respond +=method;
        }
         // Send is implemented as a broadcast
        public void Send(string message, string from) {
                respond(message, from);
                Console.WriteLine();
        }
}  //End of class Mediator
```

```
class Colleague {
    Mediator mediator;
    protected string name;

    public Colleague(Mediator mediator, string name) {
        this.mediator = mediator;
        mediator.SignOn(Receive);
        this.name = name;
    }

    public virtual void Receive(string message, string from) {
        Console.WriteLine(name +" received from "+from+": " +
                                                    message);
    }

    public void Send(string message) {
     Console.WriteLine("Send (From "+name+ "): "+message);
     mediator.Send(message, name);
    }
}
```

```
class ColleagueB : Colleague {
    public ColleagueB(Mediator mediator, string name)
        : base (mediator, name) {
    }
    // Does not get copies of own messages
    public override void Receive(string message, string from) {
      if (!String.Equals(from,name))
        Console.WriteLine(name +" received from "+from+": " +
                                        message);
    }
}
static void Main () {
    Mediator m = new Mediator();
    // Two from head office and one from a branch office
    Colleague head1 = new Colleague(m,"John");
    ColleagueB branch1 = new ColleagueB(m,"David");
    Colleague head2 = new Colleague(m,"Lucy");
```

```
        head1.Send("Meeting on Tuesday, please all ack");
        branch1.Send("Ack"); // by design does not get a copy
        m.Block(branch1.Receive); // temporarily gets no messages
        head1.Send("Still awaiting some Acks");
        head2.Send("Ack");
        m.Unblock(branch1.Receive); // open again
        head1.Send("Thanks all");
    }
}
/* Output
    Send (From John): Meeting on Tuesday, please all ack
    John received from John: Meeting on Tuesday, please all ack
    David received from John: Meeting on Tuesday, please all ack
    Lucy received from John: Meeting on Tuesday, please all ack

    Send (From David): Ack
    John received from David: Ack
    Lucy received from David: Ack
```

Send (From John): Still awaiting some Acks

John received from John: Still awaiting some Acks

Lucy received from John: Still awaiting some Acks

Send (From Lucy): Ack

John received from Lucy: Ack

Lucy received from Lucy: Ack

Send (From John): Thanks all

John received from John: Thanks all

Lucy received from John: Thanks all

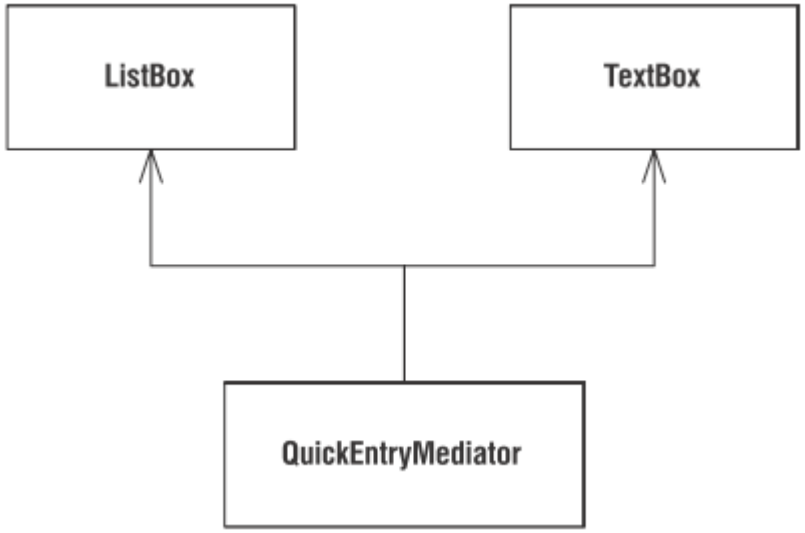
David received from John: Thanks all

*/

Інший приклад патерну Mediator

- Мартін Р. …, стор.368

Клас QuickEntryMediator "тихенько сидить за кулісами" і прив'язує текстове поле вводу до списку. Коли ви вводите текст в поле, перший елемент списку, що починається з введеного рядка, підсвічується.



Висновок (Мартін Р.)

- Накладати політику можна зверху, використовуючи патерн Фасад, якщо ця політика повинна бути явною.
- З іншого боку, якщо необхідні скромність і делікатність, то більше підійде патерн Посередник.
- Фасади зазвичай служать предметом угоди. Всі повинні бути готові використовувати Фасад замість об'єктів, що за ним переховуються.
- Посередник, навпаки, прихований від користувачів. Його політика - це доконаний (“свершившийся”) факт, а не предмет домовленостей.

Патерн Composite

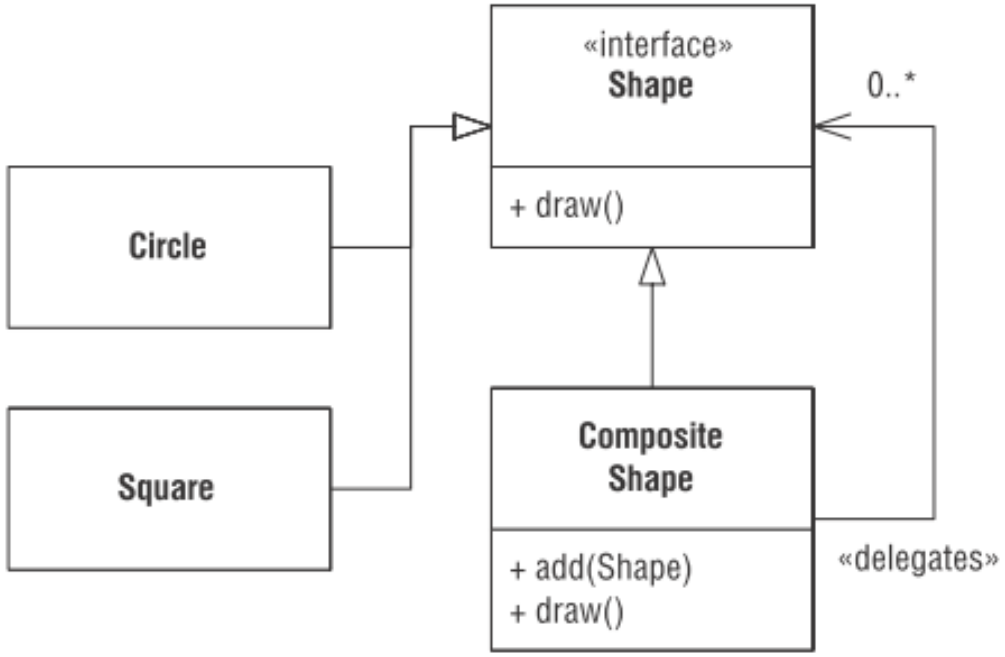
- Компоновщик – патерн, що структурує об’єкти.

Призначення

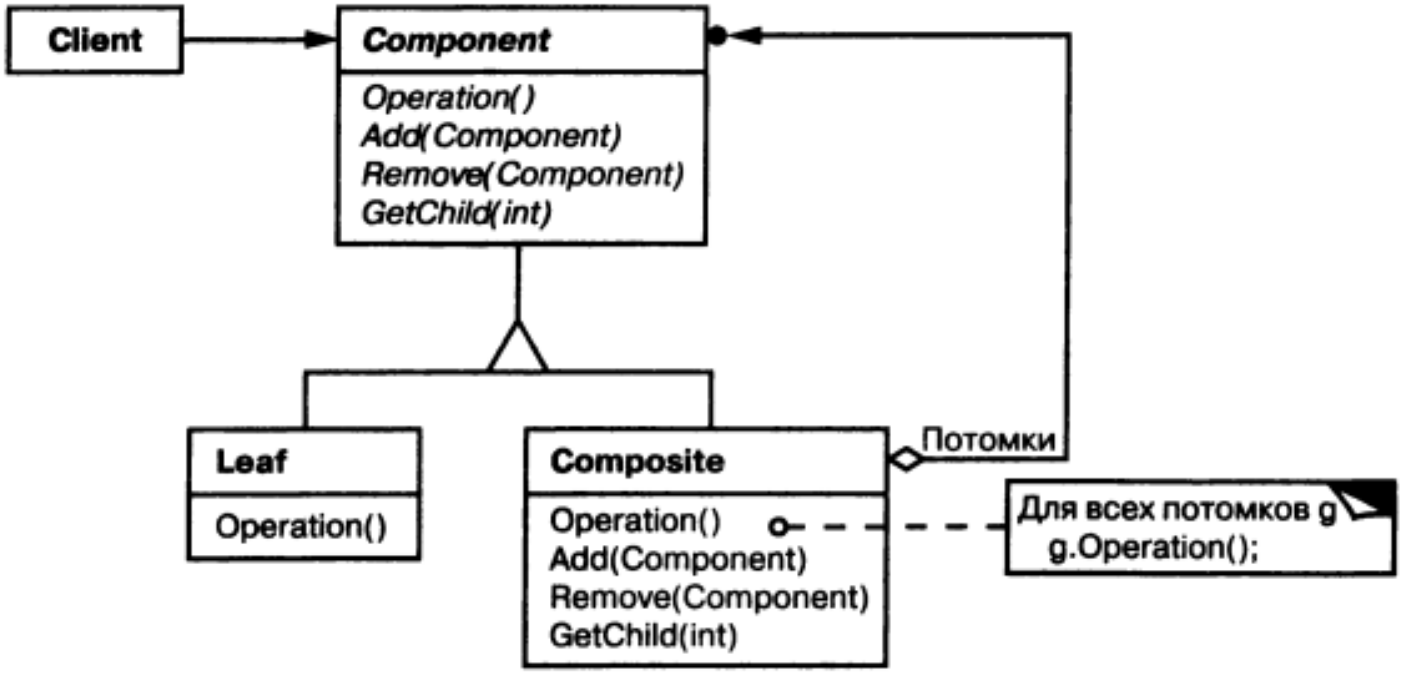
Компонує об’єкти в деревовидні структури для представлення ієрархій “частина-ціле”. Дозволяє клієнтам одноманітно трактувати індивідуальні і складені об’єкти.

Мотивація

Нехай ми маємо ієрархію геометричних фігур (Мартін, с.513)



Структура



Учасники

Component

- оголошує інтерфейс для об’єктів, що компонуються
- надає реалізацію операцій за замовчуванням, спільну для всіх класів
- оголошує інтерфейс для доступу до потомків

Учасники

Leaf – лист

- надає листові вузли композиції і не має потомків
- визначає поведінку примітивних об'єктів в композиції

Composite – складений об'єкт

- визначає поведінку компонентів, у яких є потомки
- зберігає компоненти-потомки
- реалізує операції, які відносяться до управління потомками, в інтерфейсі класу Component

Client – клієнт

- маніпулює об'єктами композиції через інтерфейс Component

Результати

Патерн компоновщик:

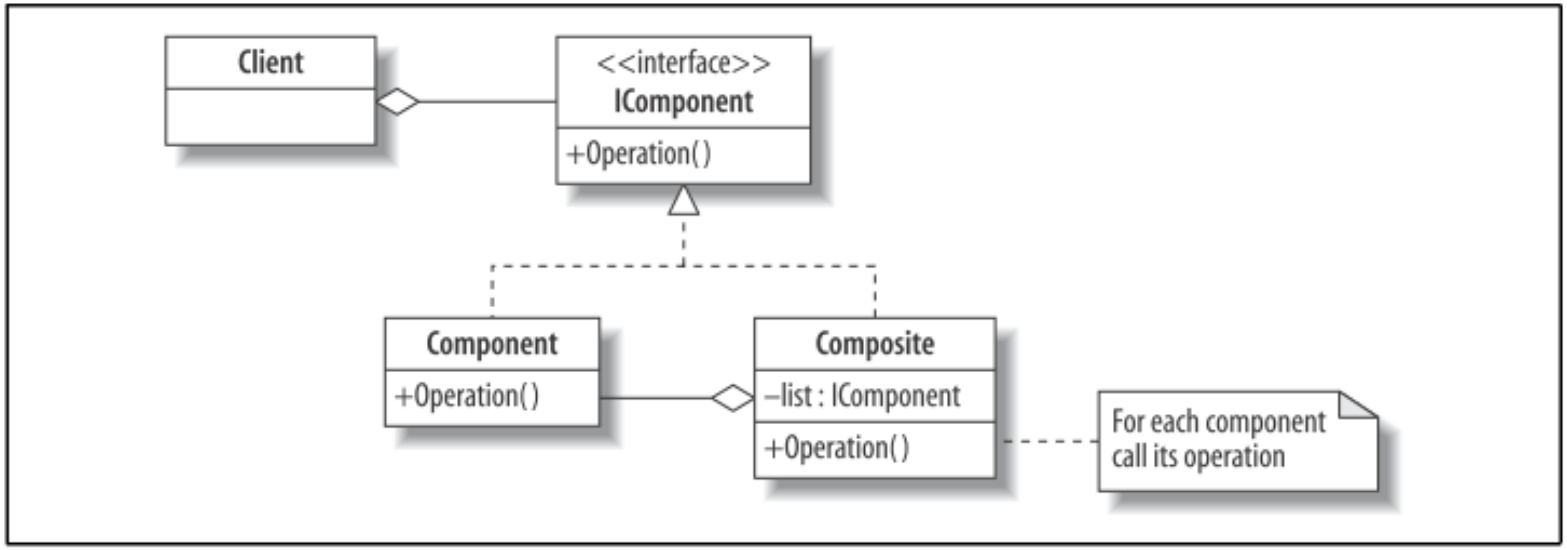
- Визначає ієрархії класів, що складаються з примітивних і складених об’єктів
- Спрощує архітектуру клієнта
- Полегшує додавання нових видів компонентів
- Сприяє створенню спільного дизайну

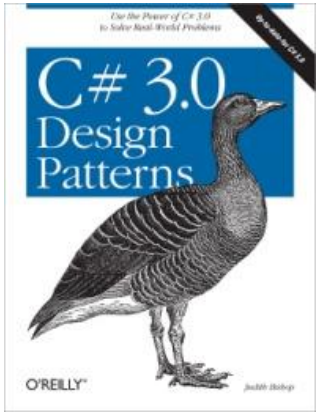
Ілюстрації

- music playlist in iTunes
- digital photo album in Flickr
- iPhoto

Figure 3-1. Composite pattern illustration—iPhoto

Composite pattern UML diagram (p.74)





Приклад (фрагменти)

```
public interface IComponent <T> {
    void Add(IComponent <T> c);
    IComponent <T> Remove(T s);
    string Display(int depth);
    IComponent <T> Find(T s);
    T Name {get; set;}
}

 public class Component <T> : IComponent <T> {
     public T Name {get; set;}
    public Component (T name)  {
            Name = name;
    }
……
}
```

```
public class Composite <T> : IComponent  <T> {
   List  <IComponent <T>> list;
   public T Name {get; set;}
   public Composite (T name)  {
         Name = name;
         list = new List <IComponent <T>> ();
   }
   public void Add(IComponent  <T> c) {list.Add(c);}
   public IComponent <T>  Find (T s) {
         holder = this;
         if (Name.Equals(s)) return this;
         IComponent <T> found=null;
         foreach (IComponent <T> c in list)  {
                  found = c.Find(s);
                  if (found!=null) break;
         }
         return found;
   }
.......
}
```

```
// The Client
 class CompositePatternExample {
  static void Main () {
    IComponent <string> album = new Composite<string> ("Album");
    IComponent <string> point = album;
    string [] s;
    string command, parameter;
// Create and manipulate a structure
    StreamReader instream = new StreamReader("Composite.dat");
    do {
      string t = instream.ReadLine();
      Console.WriteLine("\t\t\t\t"+t);
      s = t.Split();
     command = s[0];
      if (s.Length>1) parameter = s[1]; else parameter = null;
```

```
switch (command) {
        case "AddSet" :
          IComponent <string> c = new Composite <string> (parameter);
          point.Add(c);
          point = c;
          break;
        case "AddPhoto" :
          point.Add(new Component <string> (parameter));
          break;
        case "Remove" :
          point = point.Remove(parameter);
          break;
        case "Find" : …….
}
} while (!command.Equals("Quit")); …….
```

Example 3-2. Composite pattern—Photo Library sample run

AddSet Home
AddPhoto Dinner.jpg
AddSet Pets Going down another level
AddPhoto Dog.jpg
AddPhoto Cat.jpg
Find Album Ensures Garden is at same level as Home
AddSet Garden
AddPhoto Spring.jpg
AddPhoto Summer.jpg
AddPhoto Flowers.jpg
AddPhoto Trees.jpg
Display Returns to start of Album

Set Album length :2
--Set Home length :2
----Dinner.jpg
----Set Pets length :2
------Dog.jpg
------Cat.jpg …….

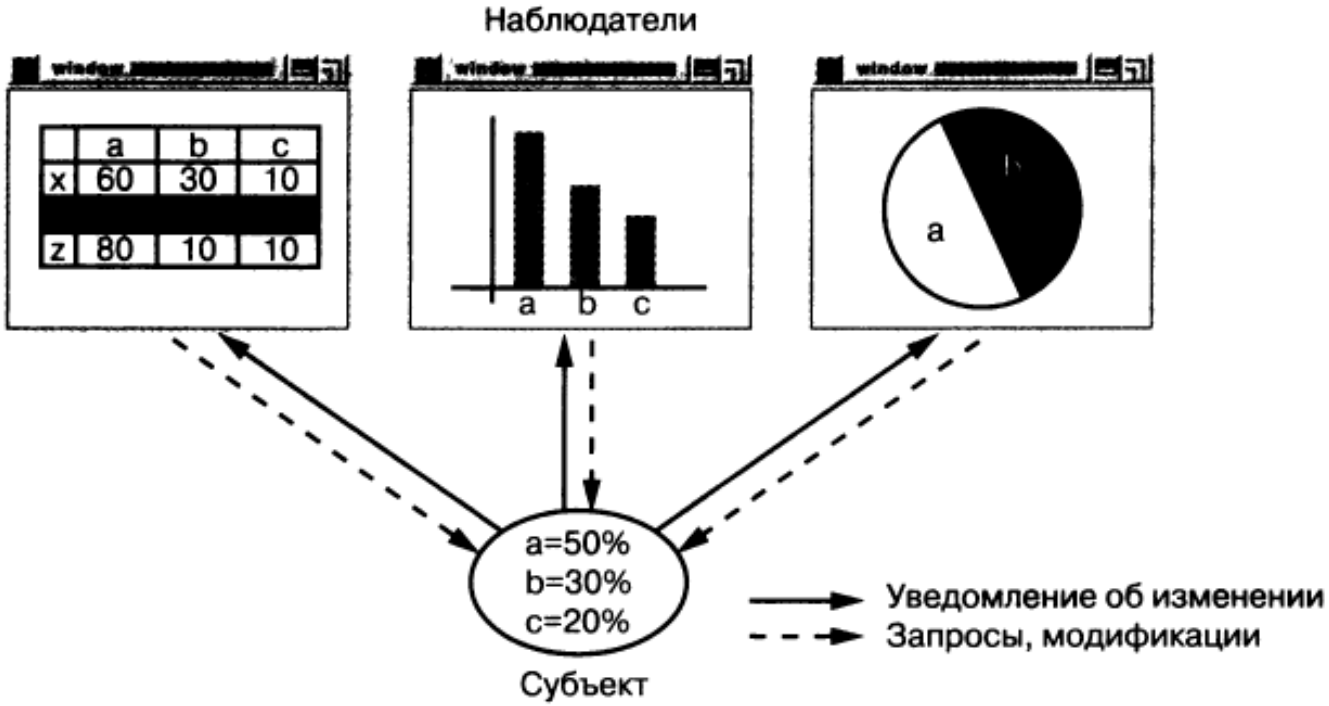
Патерн Observer

- Спостерігач – патерн поведінки об’єктів.

Призначення

Визначає залежність типу “один-до-багатьох” між об’єктами таким чином, що при зміні стану одного об’єкту всі залежні від нього оповіщуються про це і автоматично оновлюються.

Мотивація



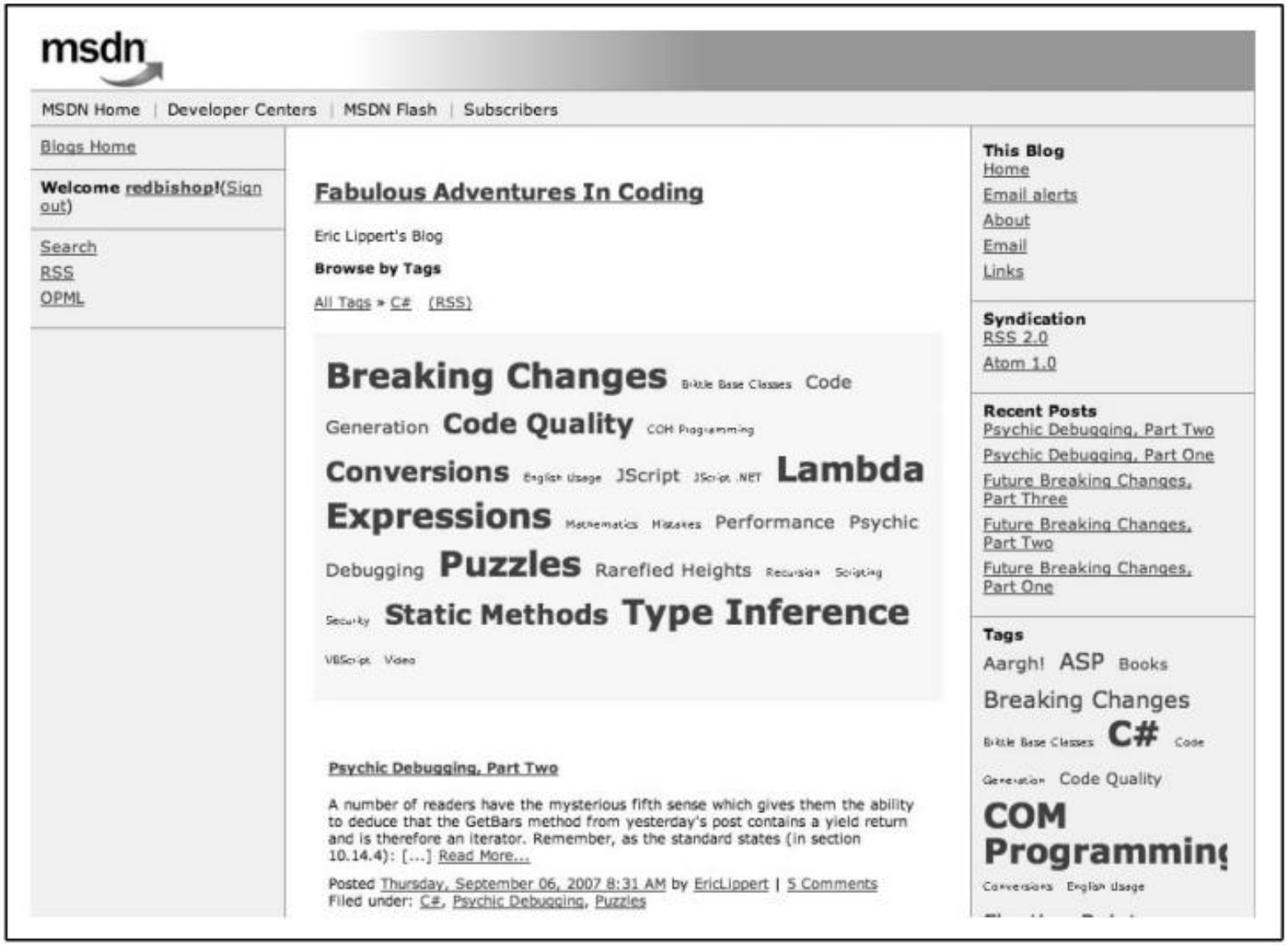


Figure 9-8. Observer pattern illustration—a blog site

Design

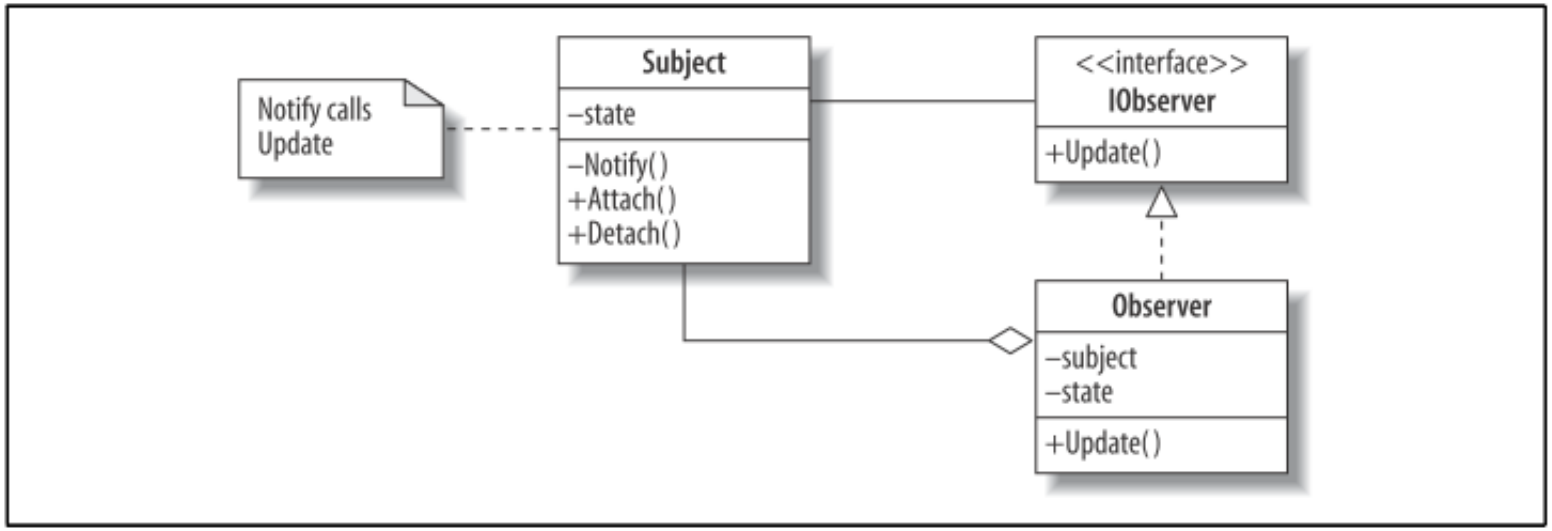
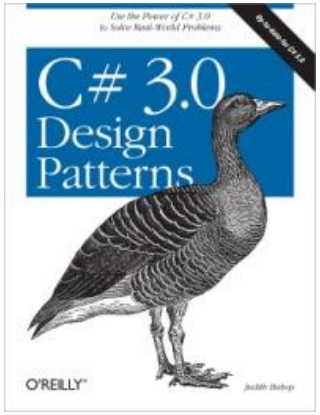


Figure 9-9. Observer pattern UML diagram



Приклад (стор.236)

Існує один Суб'єкт, який виробляє цифри.

Два спостерігача - Center і Right отримують і роздруковують цю інформацію.

Для імітації незалежного характеру роботи Суб'єкта, він реалізований у вигляді Потоку, і його дані надходять з ітератора (клас Simulator).

```
using System;
using System.Collections;
using System.Threading;
class ObserverPattern {
    class Subject {
            public delegate void Callback (string s);
            public event Callback Notify;
```

```
Simulator simulator = new Simulator();
const int speed = 200;
 public string SubjectState {get; set;}
 public void Go() {
    new Thread(new ThreadStart(Run)).Start();
}
void Run () {
    foreach (string s in simulator) {
        Console.WriteLine("Subject: "+s);
        SubjectState = s;
        Notify(s);
        Thread.Sleep(speed); // milliseconds
    }
  }
}
```

```
interface IObserver {
    void Update(string state);
}
class Observer : IObserver {
    string name;
    Subject subject;
    string state;
    string gap;
    public Observer (Subject subject, string name,string gap) {
        this.subject = subject;
        this.name = name;
        this.gap = gap;
        subject.Notify += Update;
    }
    public void Update(string subjectState) {
        state = subjectState;
        Console.WriteLine(gap+name+": "+state);
    }
}
```

```
    static void Main () {
        Subject subject = new Subject();
        Observer Observer =  new Observer(subject,"Center",“\t\t");
        Observer observer2 = new Observer(subject,"Right", "\t\t\t\t");
        subject.Go();
    }
    class Simulator : IEnumerable {
        string [] moves = {"5","3","1","6","7"};
         public IEnumerator GetEnumerator () {
                foreach( string element in moves )
                        yield return element;
        }
    }
}
```

```
/* Output
Subject: 5
        Center: 5
                Right: 5
Subject: 3
        Center: 3
                Right: 3
Subject: 1
        Center: 1
                Right: 1
Subject: 6
        Center: 6
                Right: 6
*/
```

Патерн Decorator

Декоратор – патерн, що структурує об'єкти.

Призначення

Динамічно додає об'єкту нові обов'язки. Є гнучкою альтернативою породження підкласів з метою розширення функціональності.

Відомий також під іменем Wrapper (Обгортка)

Структура

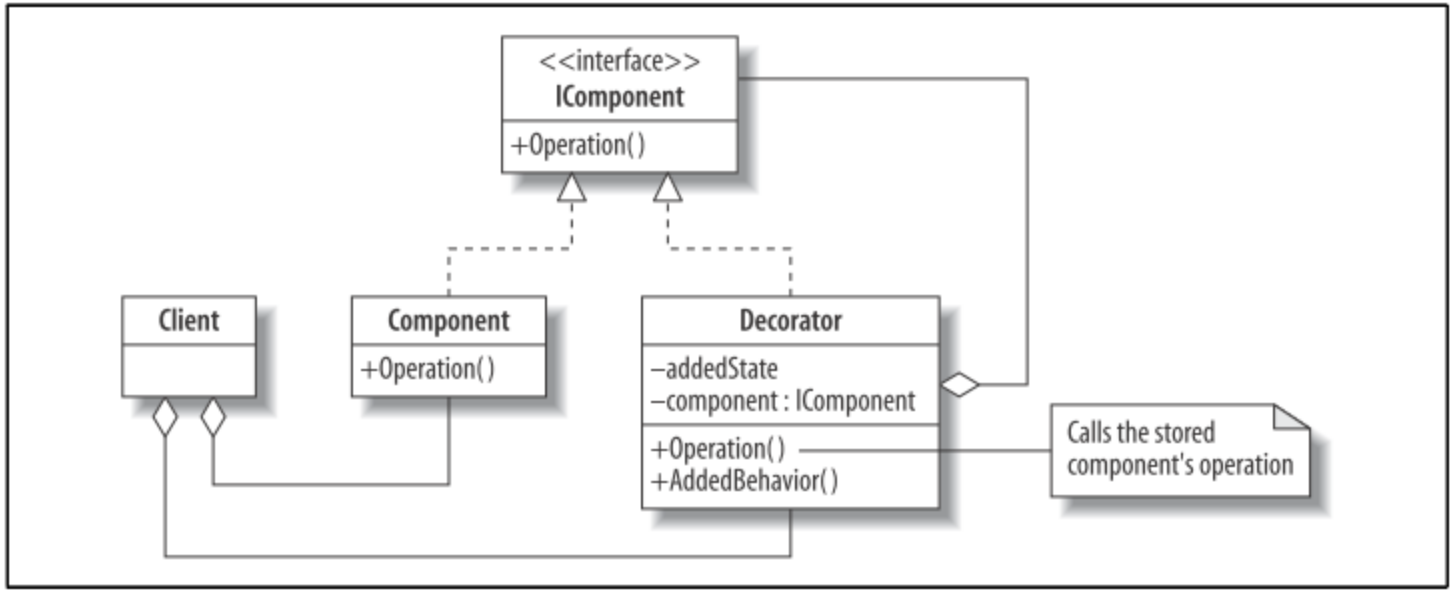
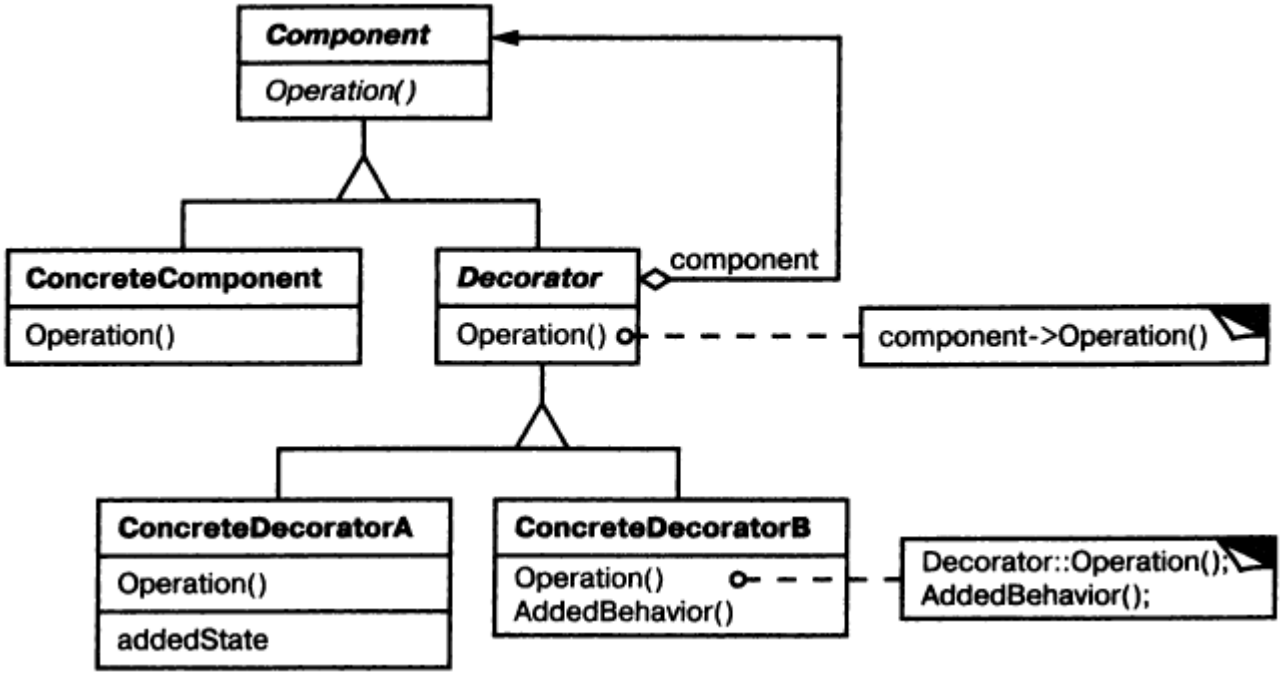


Figure 2-2. Decorator pattern UML diagram

Структура



Учасники

Component – компонент: визначає інтерфейс для об’єктів

ConcreteComponent – конкретний компонент

Decorator – декоратор

ConcreteDecorator – конкретний декоратор

Implementation (C# 3.0 Design Patterns, p.37)

```
using System;
class DecoratorPattern {
    interface IComponent {
        string Operation();
    }
    class Component : IComponent {
        public string Operation () {
            return "I am walking ";
        }
    }
    class DecoratorA : IComponent {
        IComponent component;
        public DecoratorA (IComponent c) {
            component = c;
        }
```

```
        public string Operation() {
                string s = component.Operation();
                s += "and listening to Classic FM ";
                return s;
        }
}
class DecoratorB : IComponent {
        IComponent component;
        public string addedState = "past the Coffee Shop ";
        public DecoratorB (IComponent c) {
                component = c;
        }
        public string Operation () {
                string s = component.Operation ();
                s += "to school ";
                return s;
        }
```

```
    public string AddedBehavior() {
            return "and I bought a cappuccino ";
    }
}
class Client {
    static void Display(string s, IComponent c) {
            Console.WriteLine(s+ c.Operation());
    }
    static void Main() {
        Console.WriteLine("Decorator Pattern\n");
        IComponent component = new Component();
        Display("1. Basic component: ", component);
        Display("2. A-decorated : ", new DecoratorA(component));
        Display("3. B-decorated : ", new DecoratorB(component));
        Display("4. B-A-decorated : ", new DecoratorB(
            new DecoratorA(component)));
```

```
        // Explicit DecoratorB
        DecoratorB b = new DecoratorB(new Component());
        Display("5. A-B-decorated : ", new DecoratorA(b));
        // Invoking its added state and added behavior
        Console.WriteLine("\t\t\t"+b.addedState +
    b.AddedBehavior());
    }
  }
}
```

Decorator Pattern

1. Basic component: I am walking
2. A-decorated : I am walking **and listening to Classic FM**
3. B-decorated : I am walking **to school**
4. B-A-decorated : I am walking **and listening to Classic FM to school**
5. A-B-decorated : I am walking **to school and listening to Classic FM** past the Coffee Shop and I bought a cappuccino

Патерн Proxy

Замісник – патерн, що структурує об’єкти.

Призначення

Є сурогатом іншого об’єкта і контролює доступ до нього.

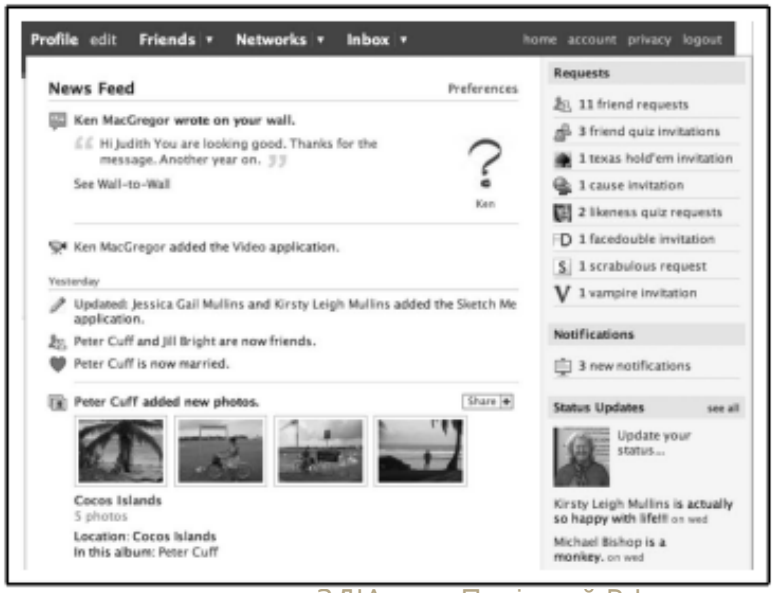
Мотивація

Розумно управляти доступом до об’єкту, оскільки тоді можна відкласти витрати на створення і ініціалізацію до моменту, коли об’єкт дійсно знадобиться.

Приклад: відкриття документу з багатьма растровими зображеннями.

Приклад. Сторінка Facebook

(для початківців – не всі можливості)



Структура

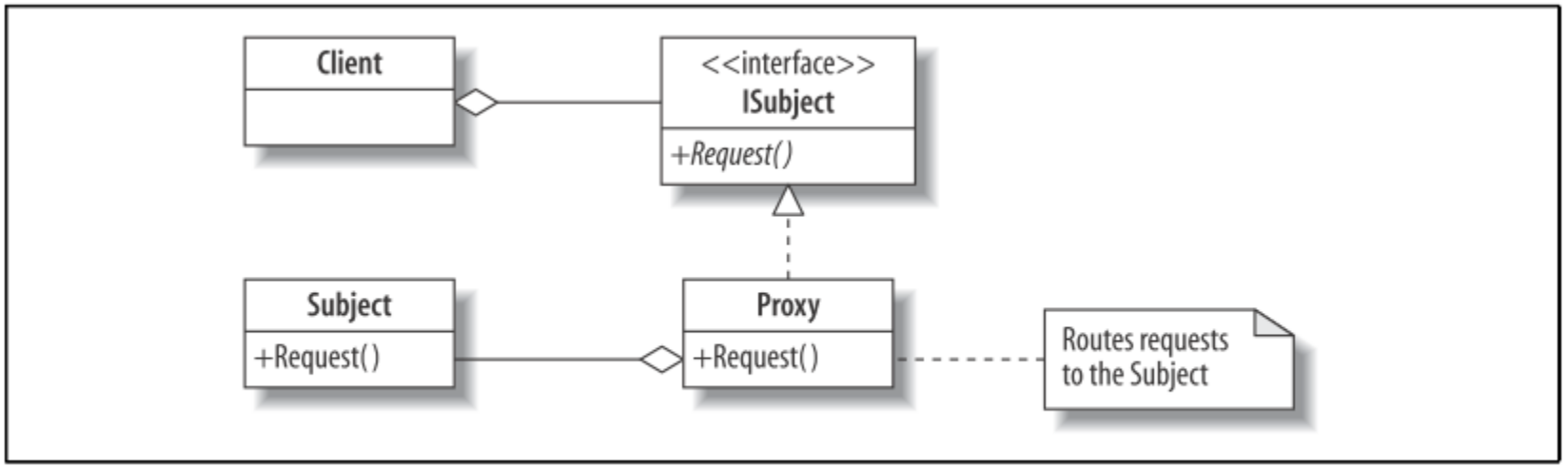


Figure 2-5. Proxy pattern UML diagram

ISubject – спільний інтерфейс для суб'єктів і проксі.

Subject – клас, який буде представляти проксі.

Proxy – клас, який створює і контролює доступ до Subject

Request – операція Subject , яка прямує через Proxy

Види Proxy

Є кілька видів проксі, найбільш поширеним з яких:

- **Virtual proxies** – створює об'єкт за вимогою (корисно, якщо процес створення може бути повільним або може виявитися непотрібним)
- **Authentication proxies** - перевіряє, що права доступу для запиту вірні.
- **Remote proxies** - кодує запити і відправляти їх по мережі
- **Smart proxies** – додає або змінює запити, перш ніж відправити їх.

Приклад (C# 3.0 Design Patterns, p.49)

Два проксі: віртуальний і для авторизації.

```
using System;
class SubjectAccessor {
   public interface ISubject {
         string Request ();
   }
   private class Subject {
         public string Request() {
                  return "Subject Request " + "Choose left door\n";
         }
   }
   public class Proxy : ISubject {
         Subject subject;
         public string Request() {
```

```
// A virtual proxy creates the object only on its first method call
                if (subject == null) {
                   Console.WriteLine("Subject inactive");
                   subject = new Subject();
                 }
                Console.WriteLine("Subject active");
                return "Proxy: Call to " + subject.Request();
        }
}

  public class ProtectionProxy : ISubject {
        // An authentication proxy first asks for a password
        Subject subject;
        string password = "Abracadabra";
        public string Authenticate (string supplied) {
                if (supplied!=password)
                        return "Protection Proxy: No access";
```

```
                else
                    subject = new Subject();
                    return "Protection Proxy: Authenticated";
                }
        public string Request() {
                if (subject==null)
                  return "Protection Proxy: Authenticate first";
                else return "Protection Proxy: Call to "+
                    subject.Request();
        }
    }
}
class Client : SubjectAccessor {
    static void Main() {
        Console.WriteLine("Proxy Pattern\n");
        ISubject subject = new Proxy();
```

```
            Console.WriteLine(subject.Request());
            Console.WriteLine(subject.Request());
            ProtectionProxy subject = new ProtectionProxy();
            Console.WriteLine(subject.Request());
            Console.WriteLine((subject as
    ProtectionProxy).Authenticate("Secret"));
            Console.WriteLine((subject as
    ProtectionProxy).Authenticate("Abracadabra"));
            Console.WriteLine(subject.Request());
        }
}
/* Output
     Proxy Pattern
     Subject inactive
     Subject active
     Proxy: Call to Subject Request Choose left door
```

Subject active

Proxy: Call to Subject Request Choose left door

Protection Proxy: Authenticate first

Protection Proxy: No access

Protection Proxy: Authenticated

Protection Proxy: Call to Subject Request Choose left door

*/

Патерн Adapter

Адаптер – патерн, що структурує класи і об'єкти.

Призначення

Перетворює інтерфейс одного класу в інтерфейс іншого, який чекають клієнти. Адаптер забезпечує спільну роботу класів з несумісними інтерфейсами, яка без нього була б неможливою.

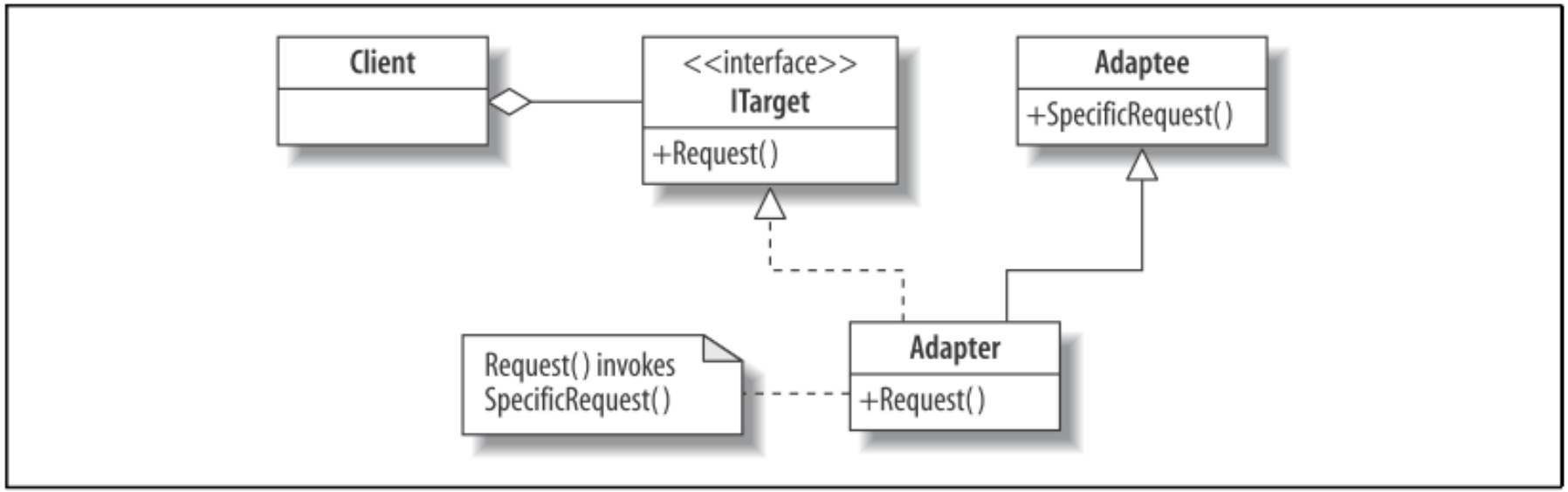


Figure 4-2. Adapter pattern UML diagram

Учасники

ITarget – цільовий. Визначає залежний від предметної області інтерфейс, яким користується Client

Client – клієнт. Вступає у взаємодію з об'єктами, що задовольняють інтерфейсу ITarget

Adaptee – адаптуємий. Визначає існуючий інтерфейс, який потребує адаптації.

Adapter – адаптер. Адаптує інтерфейс Adaptee до інтерфейсу ITarget

Приклад (C# 3.0 Design Patterns, p.101)

```
 using System;
// Existing way requests are implemented
class Adaptee {
    // Provide full precision
    public double SpecificRequest (double a, double b) {
          return a/b;
    }
}
// Required standard for requests
interface ITarget {
    // Rough estimate required
    string Request (int i);
}
```

```
// Implementing the required standard via Adaptee
class Adapter : Adaptee, ITarget {
    public string Request (int i) {
        return "Rough estimate is " + (int)
    Math.Round(SpecificRequest (i,3));
    }
}
class Client {
    static void  Main () {
        // Showing the Adapteee in standalone mode
        Adaptee first = new Adaptee();
        Console.Write("Before the new standard\nPrecise reading: ");
        Console.WriteLine(first.SpecificRequest(5,3));
        // What the client really wants
        ITarget second = new Adapter();
```

```
        Console.WriteLine("\nMoving to the new standard");
        Console.WriteLine(second.Request(5));
    }
}
/* Output
   Before the new standard
   Precise reading: 1.66666666666667

   Moving to the new standard
   Rough estimate is 2
*/
```

Патерн Bridge

Міст – патерн, що структурує об’єкти.

Призначення

Відділити абстракцію від її реалізації так, щоб і ту і іншу можна було змінювати незалежно.

Структура

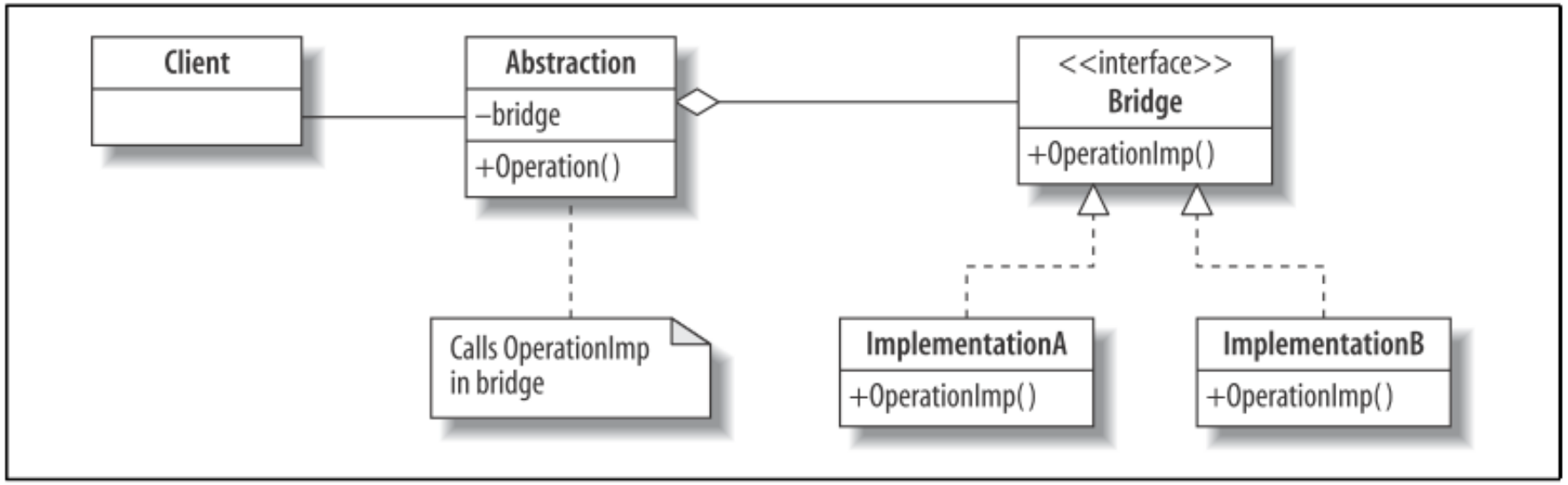


Figure 2-8. Bridge pattern UML diagram

Учасники

Abstraction – визначає інтерфейс абстракції, який бачать клієнти.

Bridge – Інтерфейс визначення тих частин абстракції, які можуть змінюватися

ImplementationA and **ImplementationB** – реалізації інтерфейсу Bridge

Відносини

Об’єкт Abstraction перенаправляє своєму об’єкту Bridge запити клієнта

Результати

Результати застосування патерну міст такі:

- **Відділення реалізації від інтерфейсу**.
  - Реалізація більше не має постійної прив'язки до інтерфейсу
  - Реалізацію абстракції можна конфігурувати під час виконання
- **Підвищення ступеня розширюваності**
- **Приховування деталей реалізації від клієнтів**

Приклад (C# 3.0 Design Patterns, p.61)

```
using System;
class BridgePattern {
    class Abstraction {
        Bridge bridge;
        public Abstraction (Bridge implementation) {
                bridge = implementation;
        }
         public string Operation () {
              return "Abstraction" +" <<< BRIDGE >>>> “
                        +bridge.OperationImp();
         }
    }
    interface Bridge {
        string OperationImp();
    }
```

```
class ImplementationA : Bridge {
        public string  OperationImp () {
            return "ImplementationA";
        }
 }
  class ImplementationB : Bridge {
        public string OperationImp () {
            return "ImplementationB";
        }
  }
   static void Main () {
     Console.WriteLine("Bridge Pattern\n");
     Console.WriteLine(new Abstraction (new
   ImplementationA()).Operation());
        Console.WriteLine(new Abstraction (new
   ImplementationB()).Operation());
      }
}
```

```
/* Output
    Bridge Pattern

    Abstraction <<< BRIDGE >>>> ImplementationA
    Abstraction <<< BRIDGE >>>> ImplementationB
*/
```