

Патерни проектування

2005-2012

Patterns

1

Що таке патерн (*pattern*)?

За Кристофером Олександром, будь-який патерн описує задачу, що постійно постає та розв'язується, застосовуючи один і той же прийом так, що він стає у нагоді тисячі, мільйони разів і не потребує нічого нового.

Хоча Олександр мав на увазі патерни, що виникають при проектуванні архітектурних споруд, його підхід залишається справедливим й для патернів ОО проектування.

Патерн – це **типове** вирішення **типової** проблеми у даному контексті.

- Alexander C., Ishikawa S., Silverstein M. A Pattern Language: Towns/ Buildings/ Construction, NY, 1977.
- Alexander C., Ishikawa S., Silverstein M. The Timeless Way of Buildings, NY, 1979.

GoF (Gang of Four).



- Gamma E., Johnson R., Helm R., Vlissides J. Design Patterns. Elements of Reusable Object-Oriented Software. — Addison-Wesley, 1995.
- Гамма Э., Хелм Р., Джонсон Р., Влссидес Дж. Приемы объектно-ориентированного проектирования. Паттерны проектирования. СПб: Питер, 2001. — 368 с.
- Шаллоуей А., Тротт Дж. Шаблоны проектирования. — ИД "Вильямс", 2002.



data & object factory

#1 in DESIGN PATTERNS

home

C#, ASP.NET, WPF, WCF, LINQ, PATTERNS

- ▶ [home](#)
- [courses](#)
- [schedule](#)
- [registration](#)
- [careers](#)
- [contact us](#)
- [sitemap](#)
- [about us](#)
-
- [login](#)
- [design patterns](#)
- [pattern framework 3.5™](#)
-
- [resources](#)
- [wpf patterns](#)
- [ajax patterns](#)
- [C# reference card](#)
- [connection strings](#)
- [visual studio shortcuts](#)

Design Patterns, WPF, WCF, LINQ, C# and VB.NET Training

.NET courses

- **Complete WPF**
building rich clients with WPF
- **Complete C#**
building applications with C#
- **Complete VB.Net**
building applications with VB.Net
- **Complete ASP.Net**
building web apps with ASP.Net
- **C# Design Patterns**
gang-of-four patterns & beyond

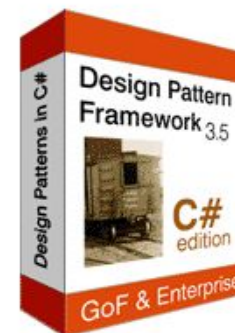


Developer resources

- * **Design Pattern Framework**
Unique .NET design pattern framework in C#, VB.Net [\(more\)](#)
- * **Free GoF Design Patterns**
Gang of four C# source code, UML, frequency of use [\(more\)](#)
- * **AJAX Design Patterns**
Build Web 2.0 applications with Ajax design patterns [\(more\)](#)
- * **Free SQL Connectionstrings**
Database connectionstrings lib, and reference guide [\(more\)](#)
- * **Free C# Reference Card**
Get your personal C# reference card. Just for the asking [\(more\)](#)
- * **Visual Studio shortcut keys**
A reference to Visual Studio's 'must-know' shortcuts [\(more\)](#)

- **Better Code**
- **Better Career**
- **Better Lifestyle**

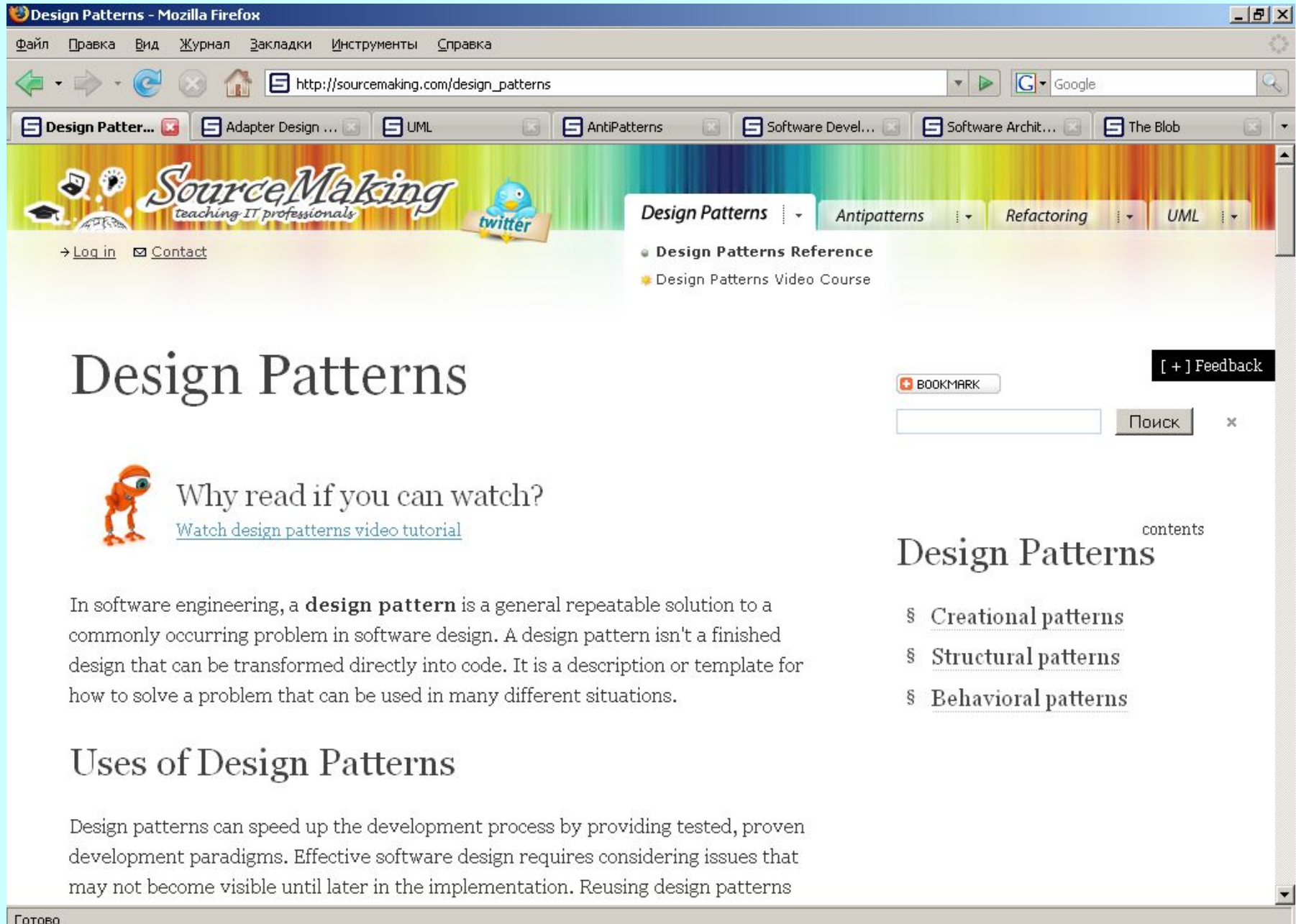
Design Pattern Framework™ 3.5



C# and VB.NET

training and education for

http://sourcemaking.com/design_patterns



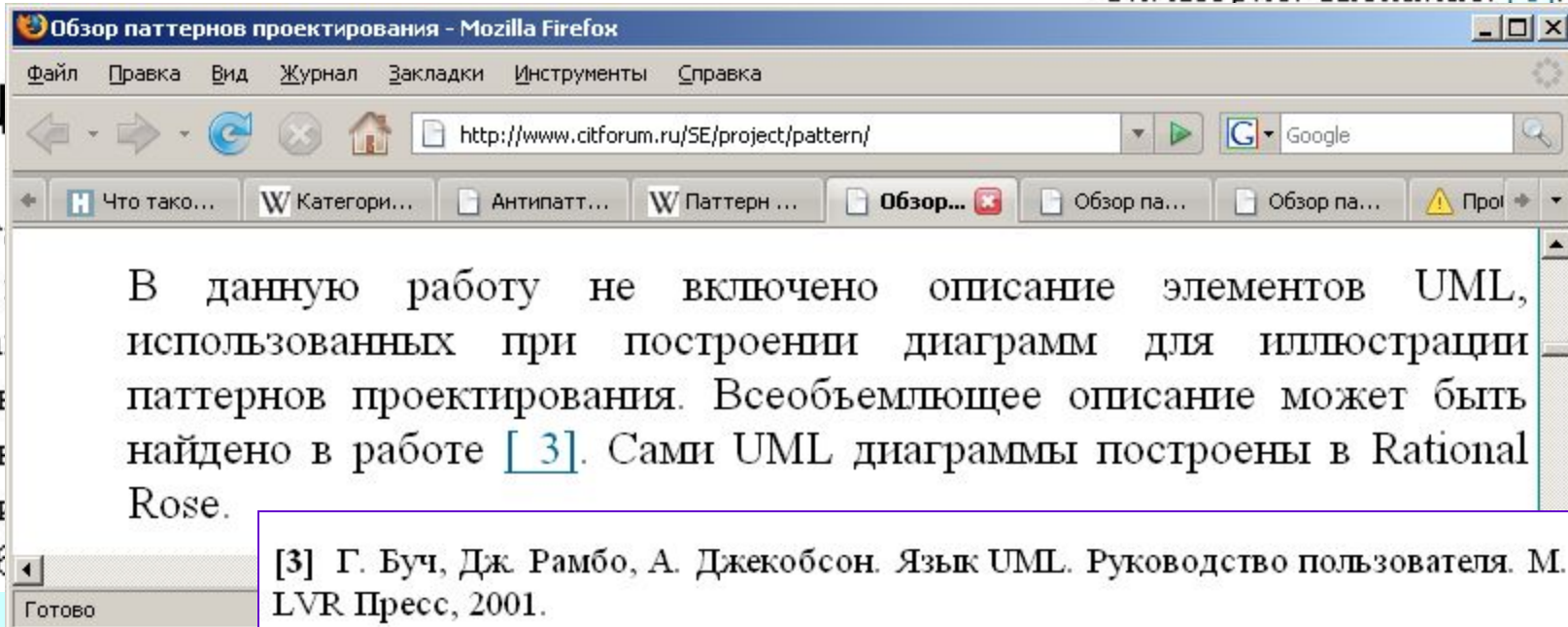
The screenshot shows a Mozilla Firefox browser window displaying the website http://sourcemaking.com/design_patterns. The browser's address bar and tabs are visible at the top. The website header features the SourceMaking logo with the tagline "teaching IT professionals" and a Twitter icon. A navigation menu includes "Design Patterns", "Antipatterns", "Refactoring", and "UML". A dropdown menu for "Design Patterns" is open, showing "Design Patterns Reference" and "Design Patterns Video Course". The main content area has a large heading "Design Patterns" and a sub-heading "Why read if you can watch?" with a link to "Watch design patterns video tutorial". A paragraph explains that a design pattern is a general repeatable solution to a common problem in software design. A "Uses of Design Patterns" section begins with the text: "Design patterns can speed up the development process by providing tested, proven development paradigms. Effective software design requires considering issues that may not become visible until later in the implementation. Reusing design patterns". On the right side, there is a "Feedback" button, a "BOOKMARK" button, a search box with the text "Поиск", and a "contents" link above a list of sections: "§ [Creational patterns](#)", "§ [Structural patterns](#)", and "§ [Behavioral patterns](#)". The browser's status bar at the bottom left shows "Готово".

Обзор паттернов проектирования

Ольга Дубина

"Каждый паттерн описывает некую повторяющуюся проблему и ключ к ее разгадке, причем таким образом, что этим ключом можно пользоваться при решении самых разнообразных задач".

Christopher Alexander[1].



АННОТАЦИЯ

Данная работа посвящена описанию структурированной информации о паттернах проектирования. Ключевые особенности

- ◆ Аннотация
- ◆ 1. Введение
- ◆ 2. Принцип классификации паттернов проектирования
- ◆ 3. Паттерны проектирования классов/объектов
 - 3.1 Структурные паттерны проектирования классов/объектов
 - 3.1.1 Адаптер (Adapter) - GoF
 - 3.1.2 Декоратор (Decorator) или Оболочка (Wrapper) - GoF
 - 3.1.3 Заместитель (Proxy) или Суррогат (Surrogate) - GoF
 - 3.1.4 Информационный эксперт (Information Expert)- GRASP
 - 3.1.5 Компоновщик (Composite) - GoF
 - 3.1.6 Мост (Bridge), Handle (описатель) или Тело (Body) - GoF
 - 3.1.7 Низкая связанность (Low Coupling) - GRASP
 - 3.1.8 Приспособленец (Flyweight) - GoF
 - 3.1.9 Устойчивый к изменениям (Protected Variations) - GRASP
 - 3.1.10 Фасад (Facade) - GoF
 - 3.2 Паттерны проектирования поведения классов/объектов
 - 3.2.1 Интерпретатор (Interpreter) - GoF
 - 3.2.2 Итератор (Iterator) или Курсор (Cursor) - GoF
 - 3.2.3 Команда (Command), Действие (Action) или Транзакция (Транзакция) - GoF
 - 3.2.4 Наблюдатель (Observer), Опубликовать - подписаться (Publish - Subscribe) или Delegation Event Model - GoF
 - 3.2.5 Не разговаривайте с неизвестными (Don't talk to strangers) - GRASP
 - 3.2.6 Посетитель (Visitor) - GoF
 - 3.2.7 Посредник (Mediator) - GoF

GRASP (General Responsibility Assignment Software Patterns)

Craig Larman
Applying UML and Patterns

GRASP

Материал из Википедии — свободной энциклопедии

[править]

GRASP (*англ.* **General Responsibility Assignment Software Patterns** (общие образцы распределения обязанностей)) - паттерны, используемые в объектно-ориентированном проектировании для решения общих задач по назначению обязанностей классам и объектам.

В книге Крейга Лармана «Применение UML и шаблонов проектирования» Каждый из них помогает решить некоторую проблему, возникающую в анализе, и которая возникает практически в любом проекте по разному. Таким образом, GRASP-паттерны - это хорошо документированные и проверенные временем принципы объектно-ориентированного анализа. Принципиально новое.

Содержание [убрать]

- 1 Каталог паттернов
 - 1.1 Information Expert (Информационный эксперт)
 - 1.2 Creator (Производитель)
 - 1.3 Controller (Контроллер)
 - 1.4 Low Coupling (Слабая связанность)
 - 1.5 High Cohesion (Сильное зацепление)
 - 1.6 Polymorphism (Полиморфизм)
 - 1.7 Pure Fabrication (Чистая выдумка)
 - 1.8 Indirection (Посредник)
 - 1.9 Protected Variations (Соккрытие реализации)



К. Ларман. Применение UML и паттернов проектирования. М. , Вильямс, 2002.

Джон Влиссидес.

Применение шаблонов проектирования - Дополнительные штрихи - Вильямс, 2003

ПРИМЕНЕНИЕ
ШАБЛОНОВ
ПРОЕКТИРОВАНИЯ
Дополнительные штрихи



Джон Влиссидес.

Применение шаблонов проектирования - Дополнительные штрихи - Вильямс, 2003

Книга (1998) одного из членів *GoF*.

ШАБЛОНЫ
ПРОЕКТИРОВАНИЯ

В

JAVA

Марк Гранд. Шаблоны проектирования в *JAVA*. Каталог популярных шаблонов проектирования, проиллюстрированных при помощи *UML*, 2002



Анти-патерни

Википедия

Свободная энциклопедия

Анти-паттерны (anti-patterns), также известные как **ловушки** (pitfalls) — это классы наиболее часто внедряемых плохих решений проблем. Они изучаются, как категория, в случае когда их хотят избежать в будущем, и некоторые отдельные случаи их могут быть распознаны при изучении неработающих систем.

Термин происходит из **информатики**, из книги «**Банды четырёх**» *Шаблоны проектирования*, которая заложила примеры практики хорошего программирования. Авторы назвали эти хорошие методы «**шаблонами проектирования**», и противоположными им являются «**анти-паттерны**». Частью хорошей **практики программирования** является избежание анти-паттернов.

Концепция также прекрасно подходит к **машиностроению**. Несмотря на то, что термин нечасто используется вне программной инженерии, концепция является универсальной.

Содержание [убрать]

- 1 Некоторые различаемые анти-паттерны в программировании
 - 1.1 Анти-паттерны в управлении разработкой ПО
 - 1.2 Анти-паттерны в разработке ПО
 - 1.3 Анти-паттерны в объектно-ориентированном программировании
 - 1.4 Анти-паттерны в программировании
 - 1.5 Методологические анти-паттерны
 - 1.6 Анти-паттерны управления конфигурацией
- 2 Некоторые организационные анти-паттерны
- 3 Некоторые социальные анти-паттерны
- 4 Шуточные анти-паттерны

Структура патернів *GoF*

У загальному випадку опис патерну складається з **чотирьох** основних розділів:

1. Ім'я. Пославшись на нього, можна одразу описати як проблему проектування, так і її вирішення (**СЛОВНИК** патернів). Отже, проектування ПС можна проводити більш високому рівні абстракції. **Патерн** — це одне з ключових понять архітектури ПС.

Знаходження виразних імен було однією з найскладніших задач при складанні каталогу **GoF** (*Gang of Four*).

2. Задача. Опис того, коли варто застосовувати патерн. Необхідно сформулювати **задачу** та її **контекст**. Тут може описуватися конкретна проблема проектування, може включатися перелік умов, при виконанні яких має сенс застосовувати даний патерн. (Важливо знати, де і при яких умовах можна скористатись патерном.)

Структура патернів

- 3. Розв'язок .** Абстрактний опис задачі проектування і того, як вона може бути розв'язана за допомогою деякого узагальненого сполучення класів чи об'єктів.
- 4. Результати.** Описуються наслідки застосування патерну, різного роду компроміси, аналізується вибір мови реалізації. Хоча при опису проектних рішень про наслідки часто не згадують, знати про них необхідно, щоб можна було оцінити переваги і недоліки даного патерну у порівнянні з іншими.

Оскільки повторне використання є важливим фактором для ПС, то до результатів варто відносити вплив на гнучкість, масштабування, портабельність розроблюваної системи.

До класифікації патернів

- **Породжуючі** патерни – пов'язані з процесом створення об'єктів.
- **Структурні** патерни – ґрунтуються на композиціях (структурних об'єднаннях) об'єктів чи класів. (Теза: замість успадкування – композиції).
- Патерни **поведінки** – характеризуються взаємодією об'єктів між собою (такі патерни можна розглядати як кооперації *UML*).

Простір патернів проектування

ЦІЛЬ \ Рівень	Породжуючі патерни	Структурні паттерни	Паттерни поведінки
Клас	Factory Method (фабричний метод)	Adapter (адаптер класу)	Interpreter (інтерпретатор) Template Method (шаблонний метод)
Об'єкт	Abstract Factory (абстрактна фабрика) Singleton (одинак) Prototype (прототип) Builder (будівельник)	Adapter (адаптер об'єкту) Decorator (декоратор) Proxy (заступник) Composite (композитор) Bridge (міст) Flyweight (пристосованець) Facade (фасад)	Iterator (ітератор) Command (команда) Observer (спостерігач) Visitor (відвідувач) Mediator (посередник) State (стан) Strategy (стратегія) Memento (хранитель) Chain of Responsibility (ланцюжок обов'язків)

Структурні патерни рівня об'єкта компонують об'єкти для одержання нової функціональності. Додаткова гнучкість пов'язана з можливістю створювати композицію об'єктів під час виконання програми.

Singleton (data & object factory™, dofactory.com)

Гарантує, що є тільки один екземпляр класу, і надає глобальну точку доступу до такого екземпляра.

```
class Singleton
{
// Fields
private static Singleton instance;
// Constructor
protected Singleton() {}
// Methods
public static Singleton Instance()
{
// Uses "Lazy initialization"
if( instance == null )
instance = new Singleton();
return instance;
}
}
```

Definition

Ensure a class has only one instance and provide a global point of access to it.

Frequency of use:  high

UML class diagram



Patt

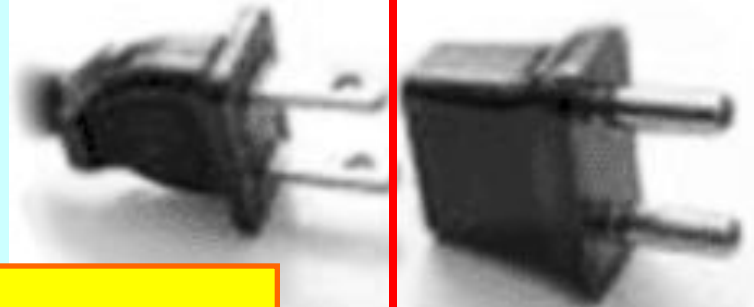
Singleton (data & object factory™, dofactory.com)

```
using System;
// "Singleton"
class Singleton
{
    // Fields
    private static Singleton instance;
    // Constructor
    protected Singleton() {}
    // Methods
    public static Singleton Instance()
    {
        // Uses "Lazy initialization"
        if( instance == null )
            instance = new Singleton();
        return instance;
    }
}
```

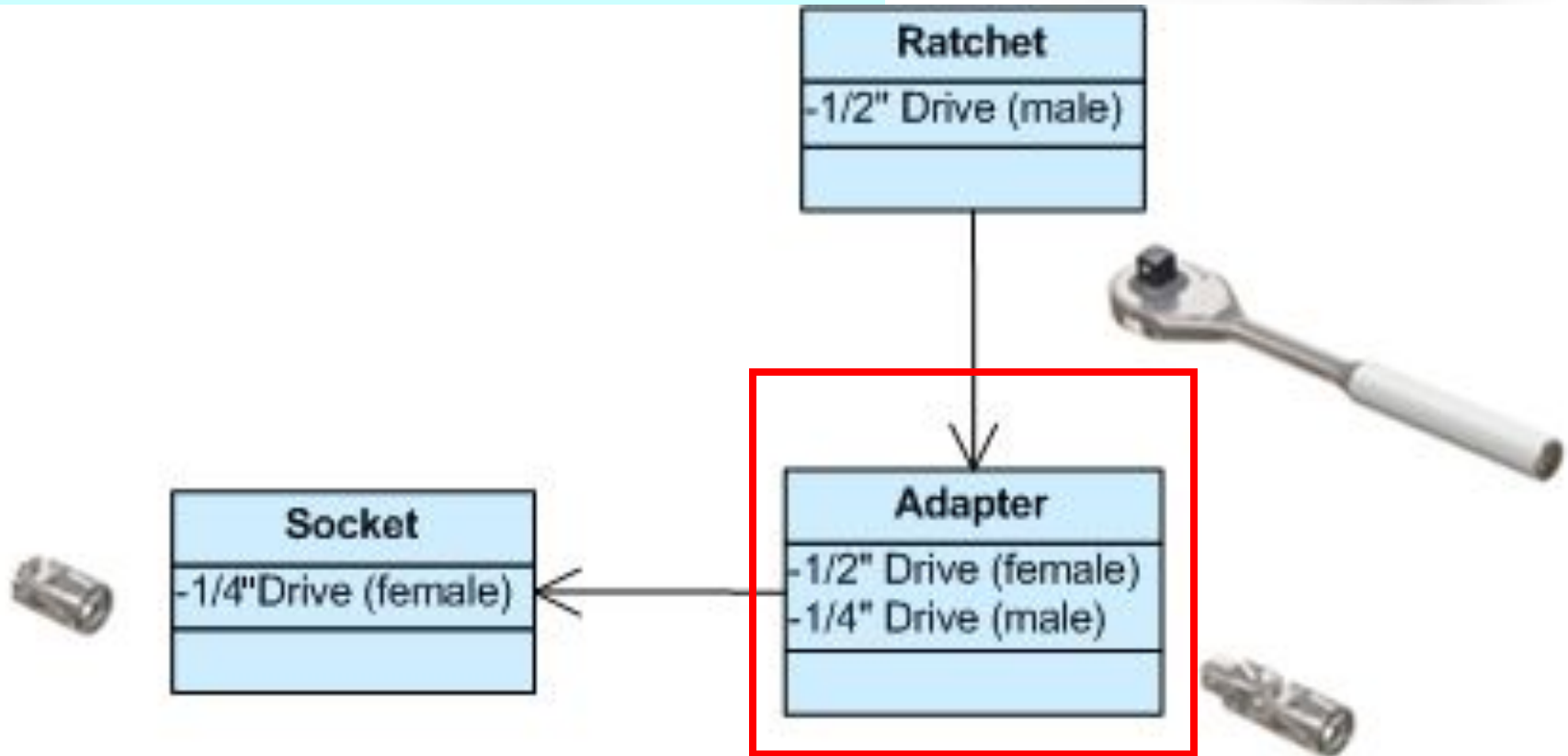
```
/// Client test
public class Client
{
    public static void Main()
    {
        // Constructor is protected - cannot use new
        Singleton s1 = Singleton.Instance();
        Singleton s2 = Singleton.Instance();

        if( s1 == s2 )
            Console.WriteLine( "The same instance"
        );
        Console.Read();
    }
}
```


Adapter



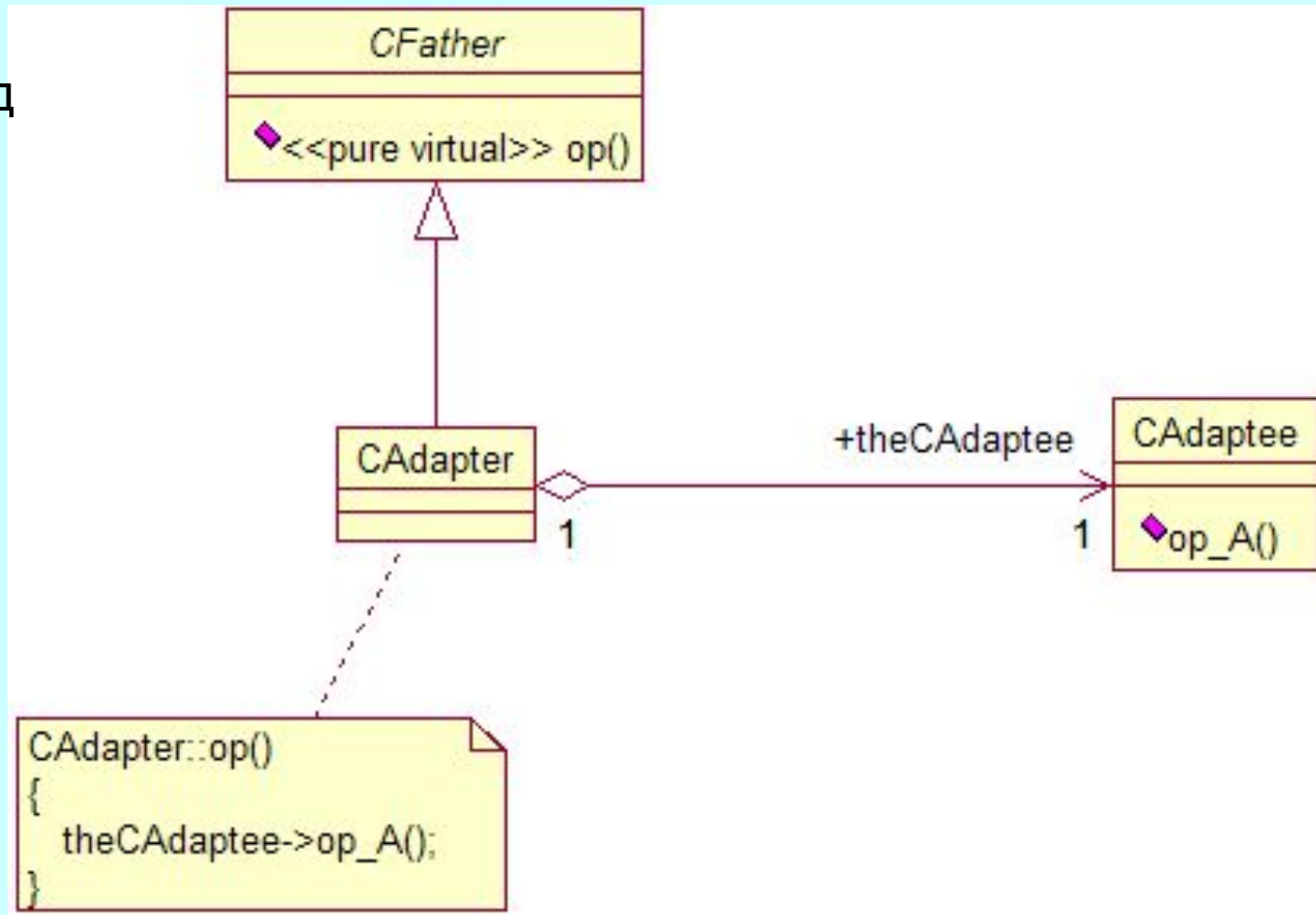
http://sourcemaking.com/design_patterns



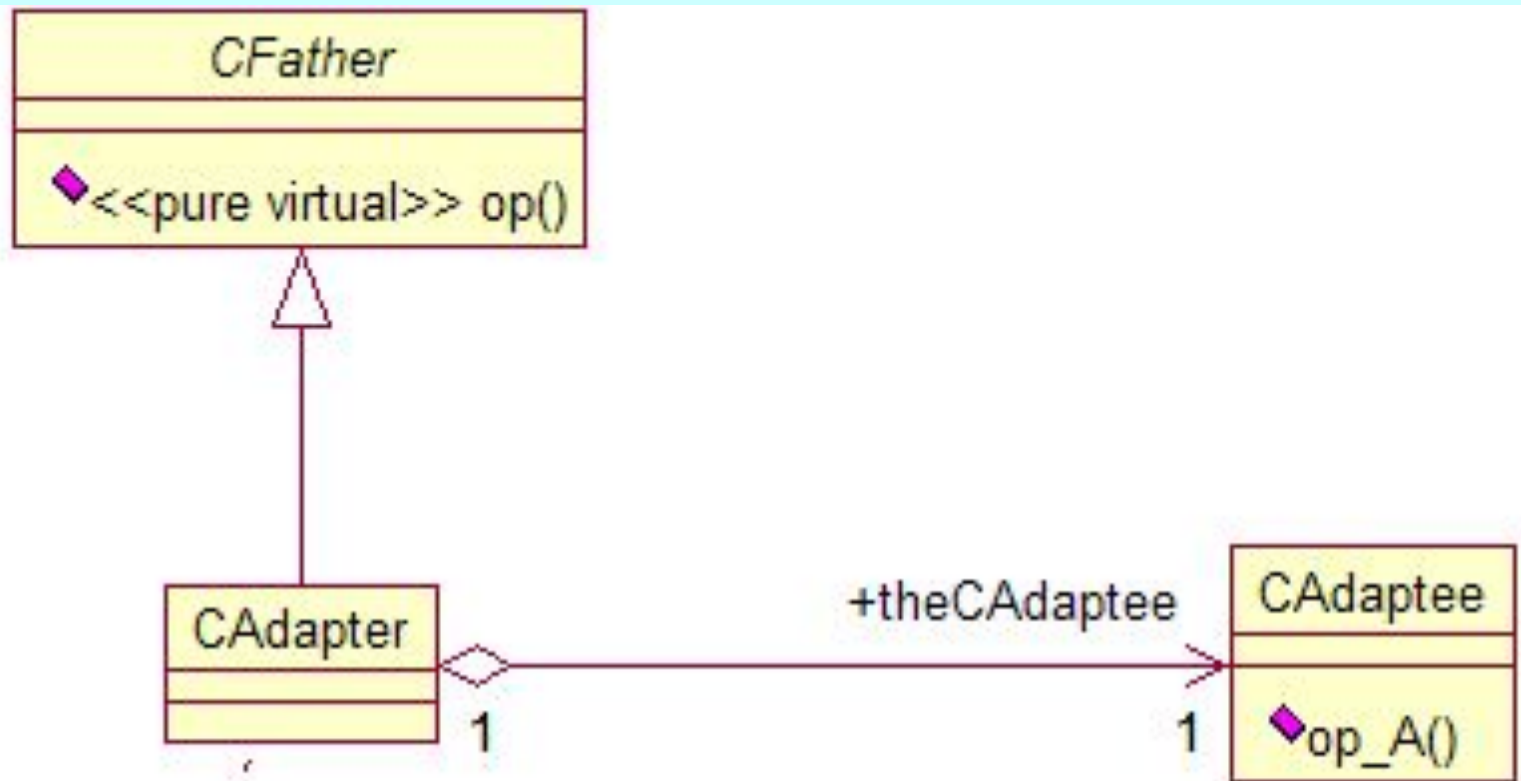
Adapter

- Дозволяє перейти від одного інтерфейсу (у класі чи в об'єкті) до іншого з метою забезпечити спільну роботу (класів чи об'єктів), яка була б неможлива без даного патерна через **несумісність інтерфейсів**.

- Відомий також під іменем Wrapper (обгортка).

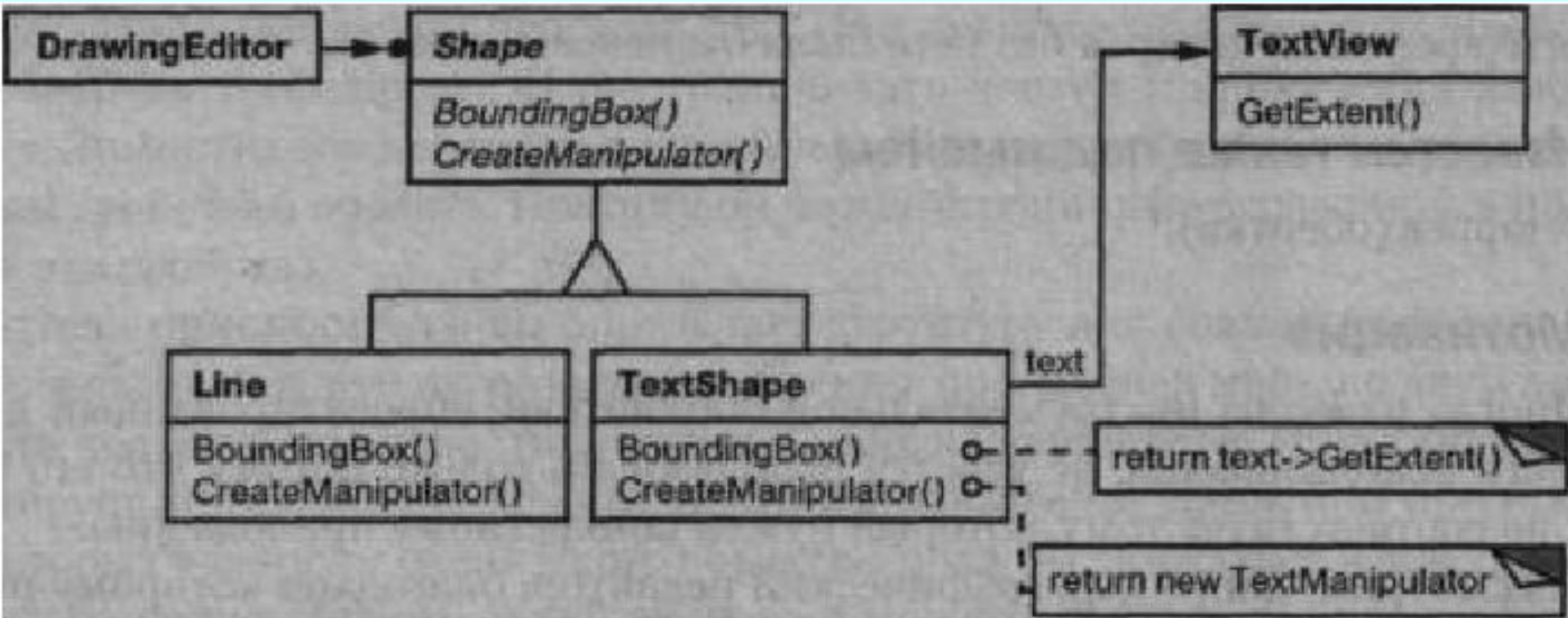


Adapter



```
CAdapter::op()
{
    theCAdaptee->op_A();
}
```



Adapter (GoF)

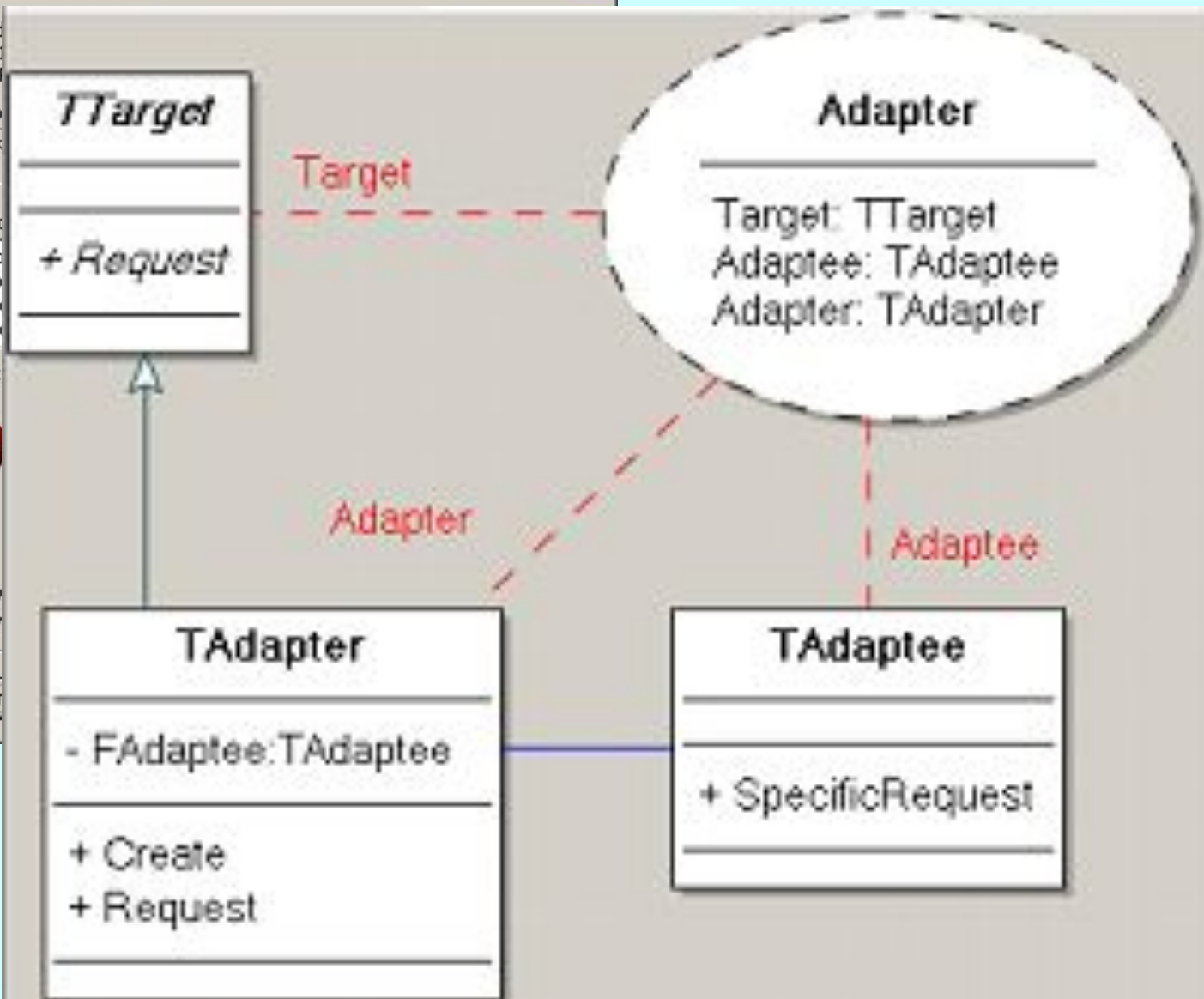




Borland Developer Studio 2006. Adapter

About Borland® Developer Studio for Windows™

Borland
10.0.2
Rights
This pro
Kuzenk
KU
Installed
Compc
Enterp
Borlan
Borlan
Product

Window
Memory
Version

www.borland.com





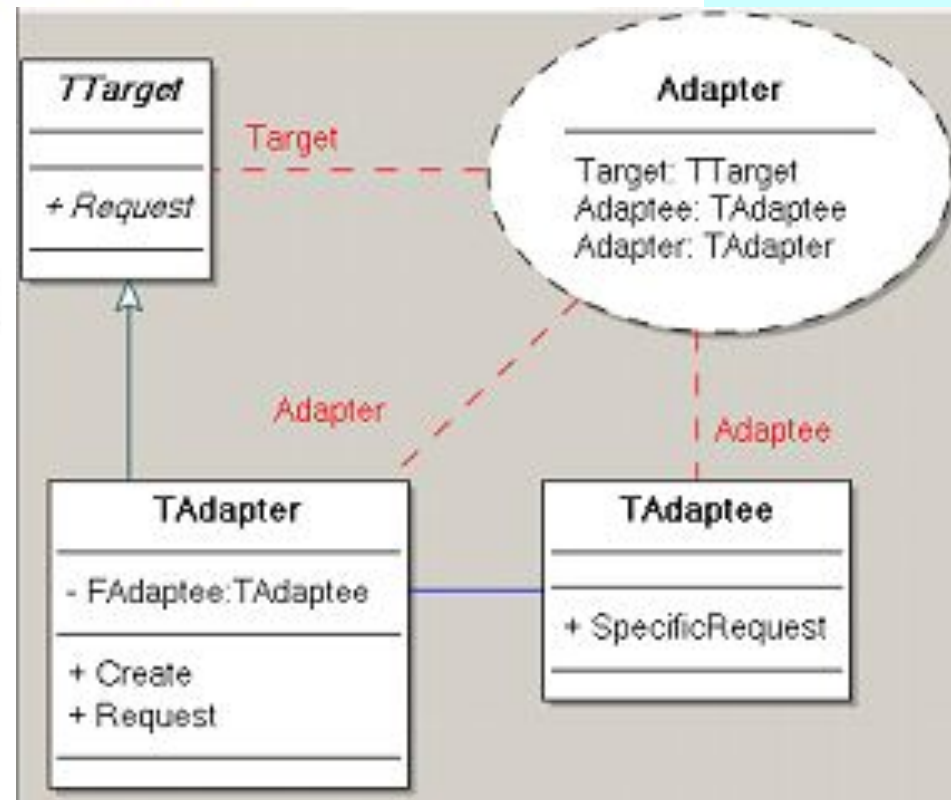
Borland Developer Studio 2006. Adapter

```
TTarget = class abstract
public
  procedure Request;virtual;abstract;
end;
```

```
TAdaptee = class
public
  procedure SpecificRequest;
end;
```

```
TAdapter = class(TTarget)
strict private var
  FAdaptee:TAdaptee;
```

```
public
  constructor Create(AdaptMe :TAdaptee);
  procedure Request;override;
end;
```





Borland Developer Studio 2006

Pattern Wizard

Pattern Tree:
All Trees

Patterns:

- Bundled Patterns (Delphi)
 - GoF
 - Behavioral
 - Creational
 - Structural
 - Adapter
 - Bridge
 - Composite
 - Decorator
 - Facade
 - Flyweight
 - Proxy

Pattern Properties:

Misc


- Class Adaptee
- Class Adapter
- Class Target

Class Adaptee
Type Class name for Adaptee

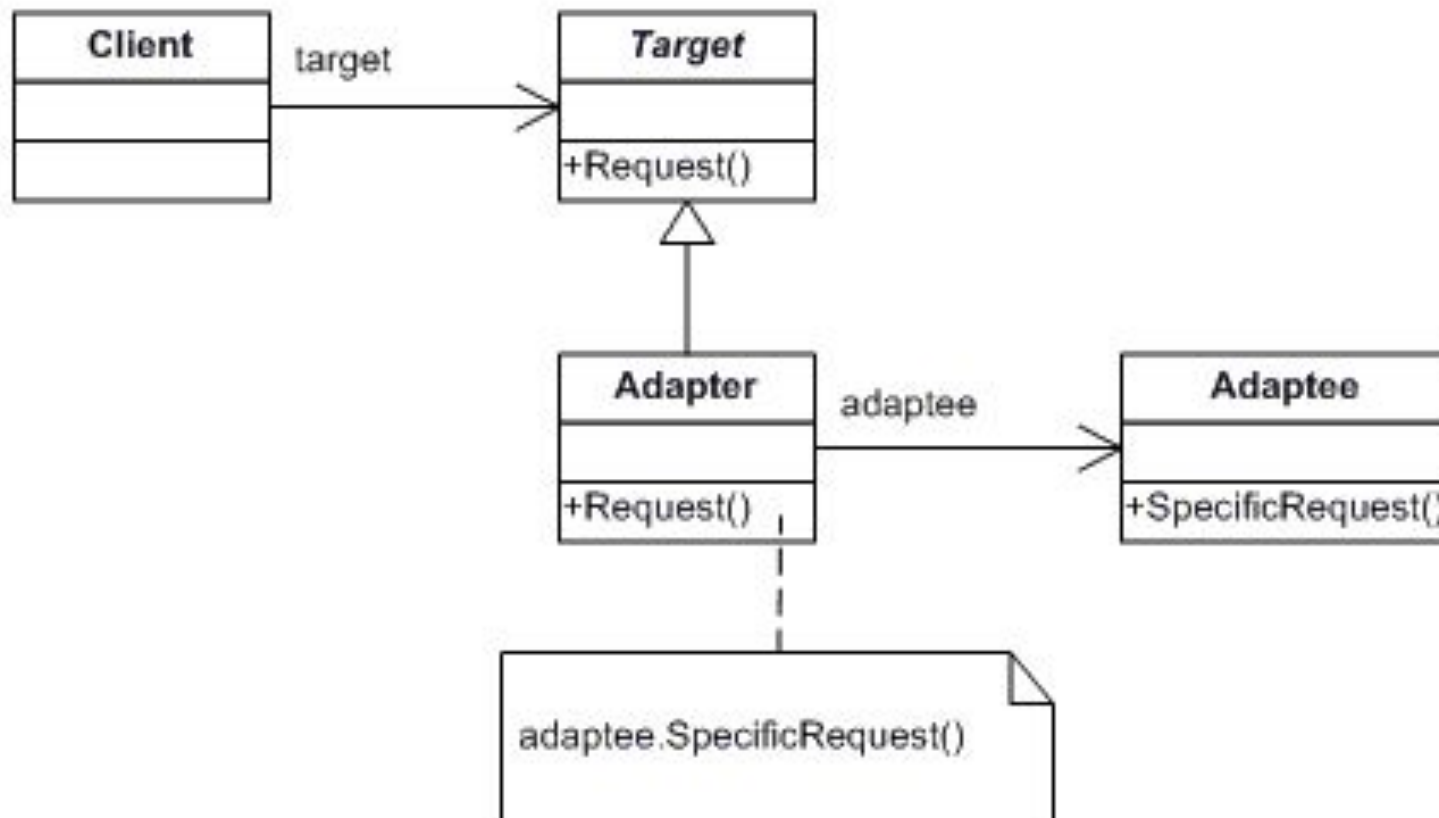
Adapter (data & object factory™, dofactory.com)

Definition

Convert the interface of a class into another interface clients expect. Adapter lets classes work together that couldn't otherwise because of incompatible interfaces.

Frequency of use:  medium high

UML class diagram



Adapter (data & object factory™, dofactory.com)

```
using System;

    // "Target"
class Target
{
    // Methods
    virtual public void Request()
    {
        // Normal implementation goes here
    }
}

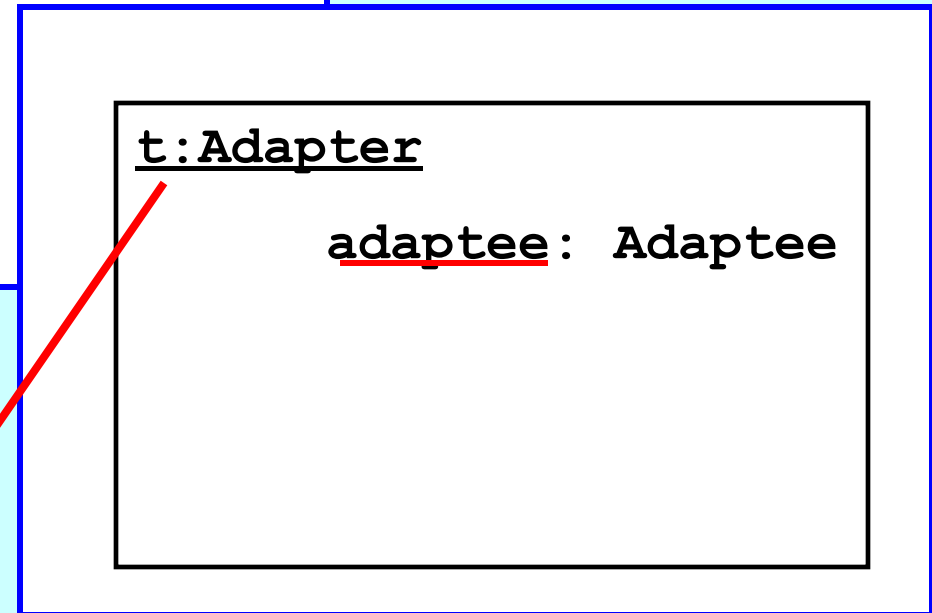
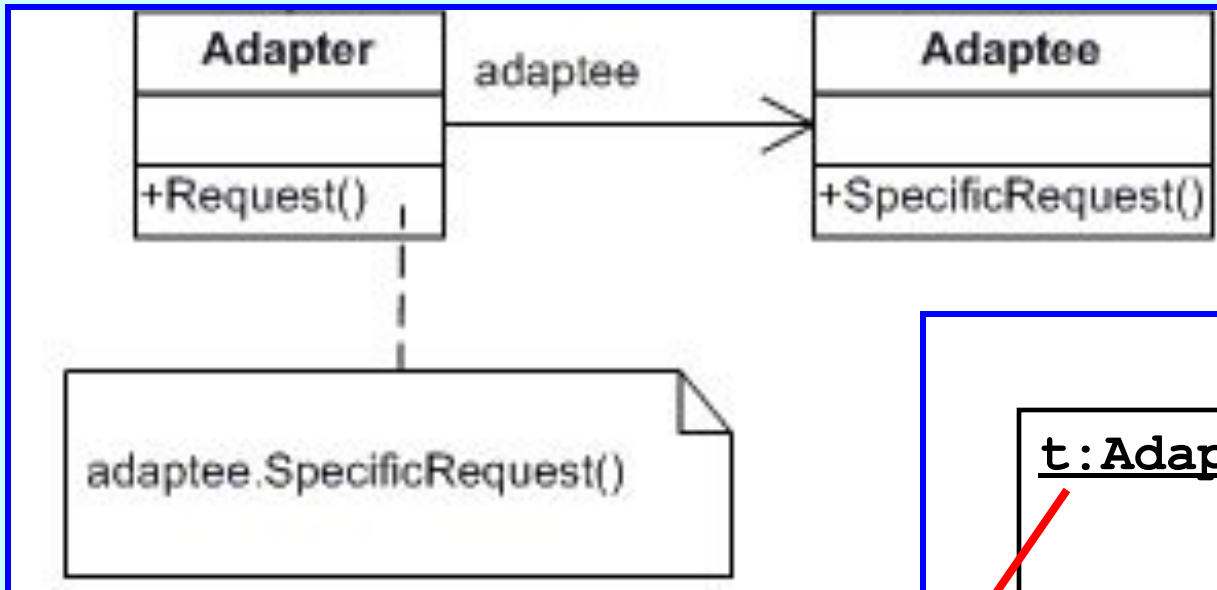
    // "Adapter"
class Adapter : Target
{
    // Fields
    private Adaptee adaptee =
        new Adaptee();

    // Methods
    override public void Request()
    {
        // Possibly do some data manipulation
        // and then call SpecificRequest
        adaptee.SpecificRequest();
    }
}
```

```
    // "Adaptee"
class Adaptee
{
    // Methods
    public void SpecificRequest()
    {
        Console.WriteLine("Called
        SpecificRequest()");
    }
}

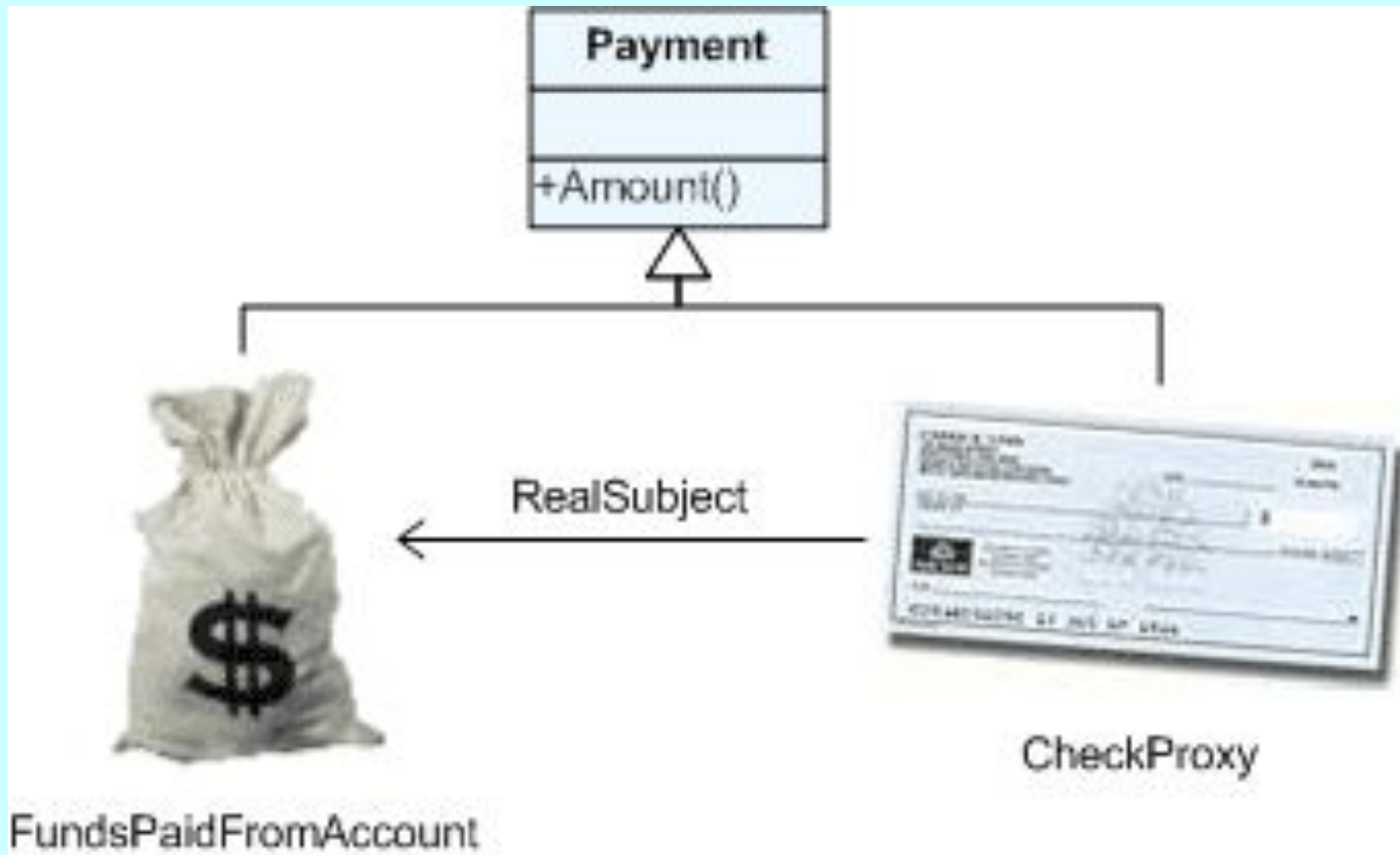
    // Client test
public class Client
{
    public static void Main(string[] args)
    {
        // Create adapter and place a request
        Target t = new Adapter();
        t.Request();
    }
}
```

Adapter



```
override public void Request()
{
    // Possibly do some data manipulation
    // and then call SpecificRequest
    adaptee.SpecificRequest();
}
```

Proxy



http://sourcemaking.com/design_patterns

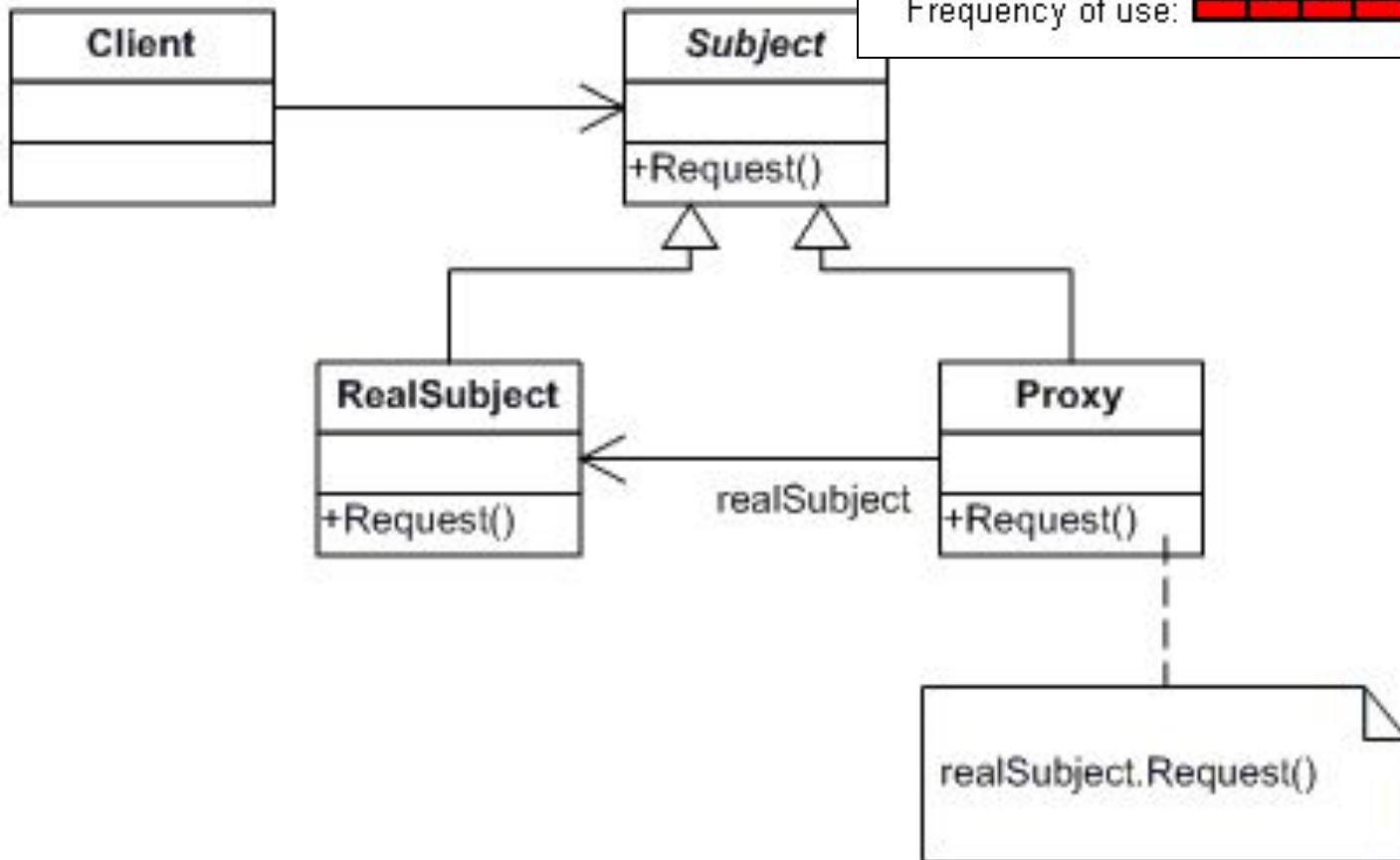
Proxy (data & object factory™, dofactory.com)

Виступає сурогатом (заступником) іншого об'єкта, "контролюючи" доступ до нього.

definition

Provide a surrogate or placeholder for another object to control access to it.

Frequency of use:  medium high



Proxy (data & object factory™, dofactory.com)

```
class MainApp
{
    static void Main()
    {
        // Create proxy and request a service
        Proxy proxy = new Proxy();
        proxy.Request();
        Console.Read(); // Wait for user
    }
}

// "Subject"
abstract class Subject
{
    public abstract void Request();
}

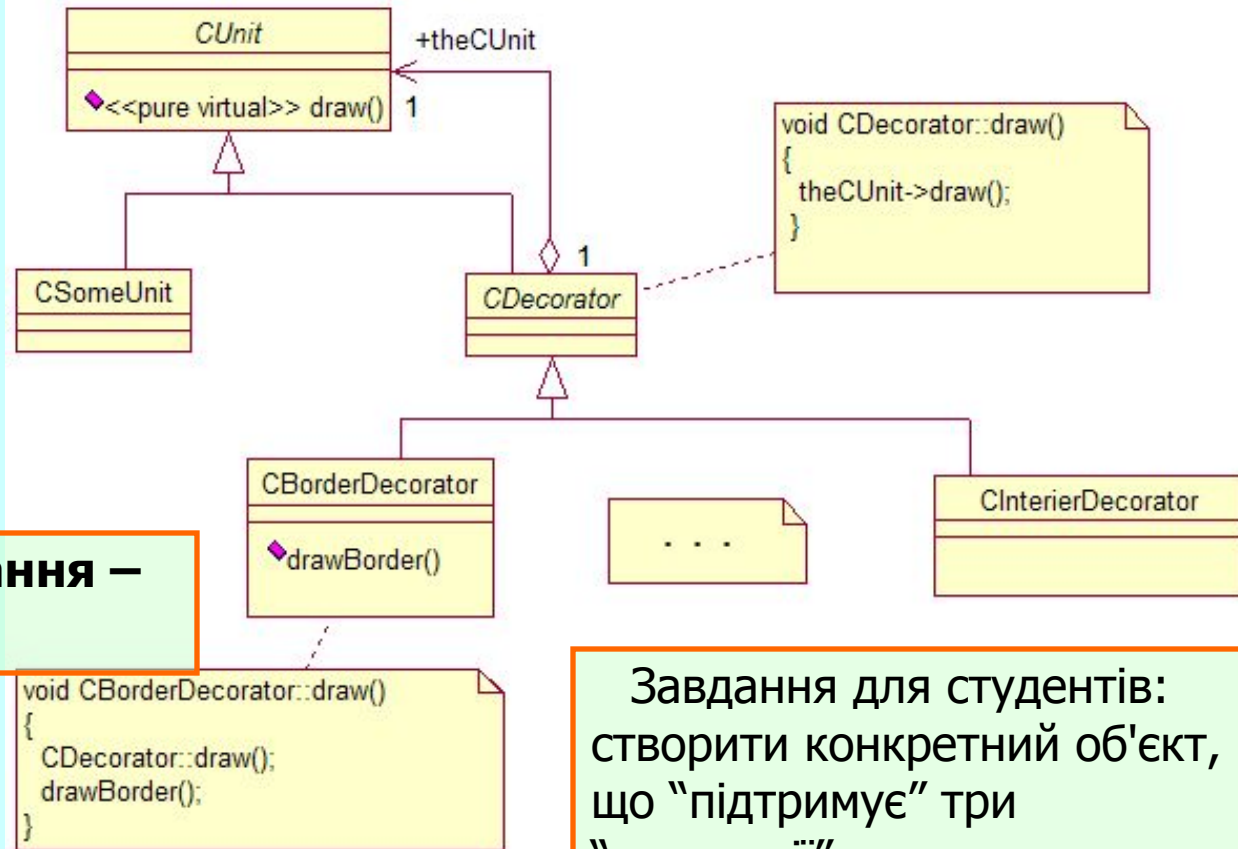
// "RealSubject"
class RealSubject : Subject
{
    public override void Request()
    {
        Console.WriteLine("Called
        RealSubject.Request()");
    }
}
```

```
// "Proxy"
class Proxy : Subject
{
    RealSubject realSubject;
    public override void Request()
    {
        // Use 'lazy initialization'
        if (realSubject == null)
        {
            realSubject = new RealSubject();
        }
        realSubject.Request();
    }
}
```


Decorator

- Декоратор – патерн, що структурує об'єкти. Динамічно додаються об'єкту нові обов'язки. Є гнучкою альтернативою породженню підкласів з метою розширення

Теза замість успадкування – композиції

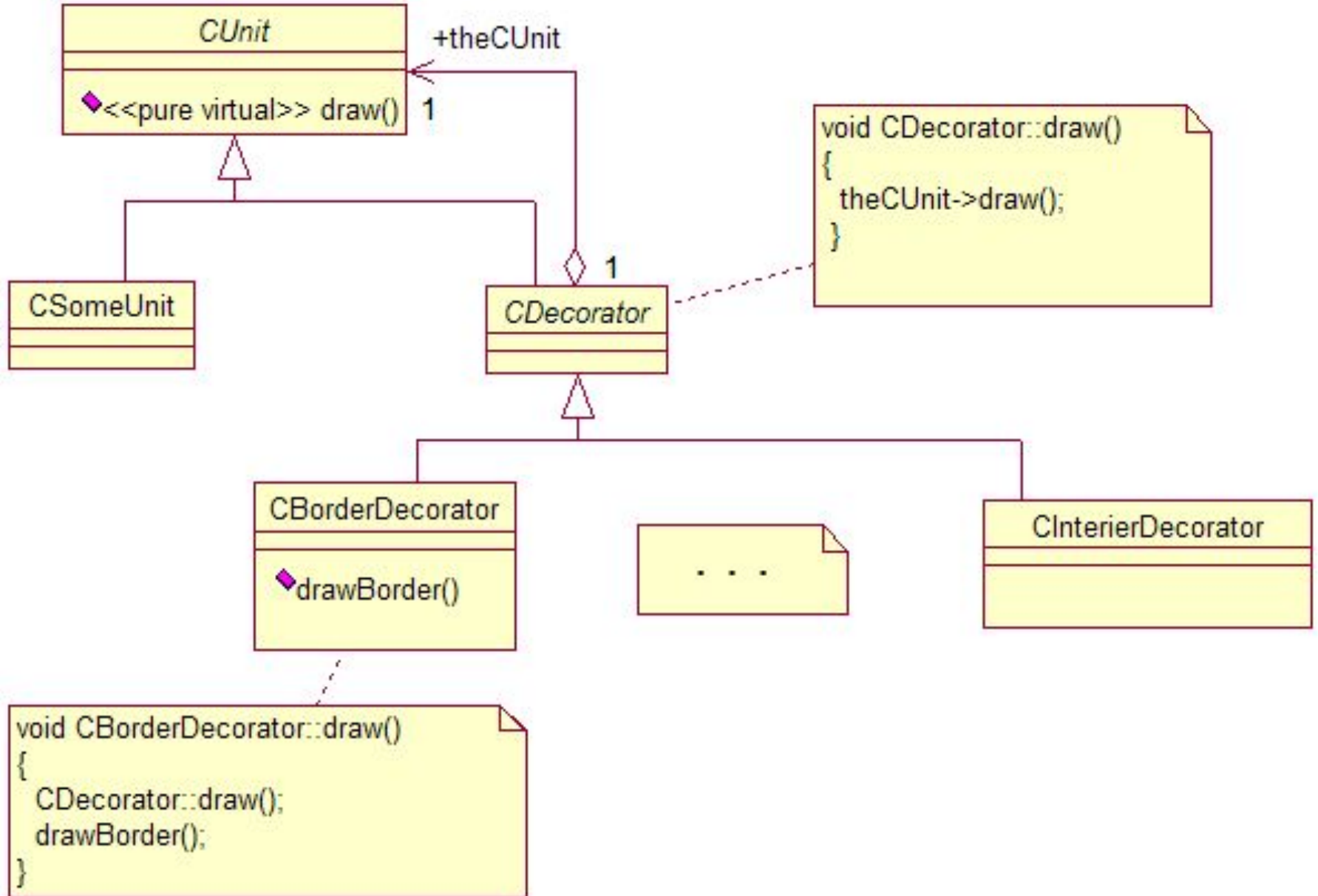


Запитання студентам:
як можна отримати
розв'язок із використан-
ням успадкування ?

Пригадаємо... "Додаткова гнучкість пов'язана з
можливістю задавати композицію об'єктів під час
виконання програми".

Завдання для студентів:
створити конкретний об'єкт,
що "підтримує" три
"декорації", наприклад,
CBorderDecorator,
CInteriorDecorator,
CMyDecorator.


Decorator



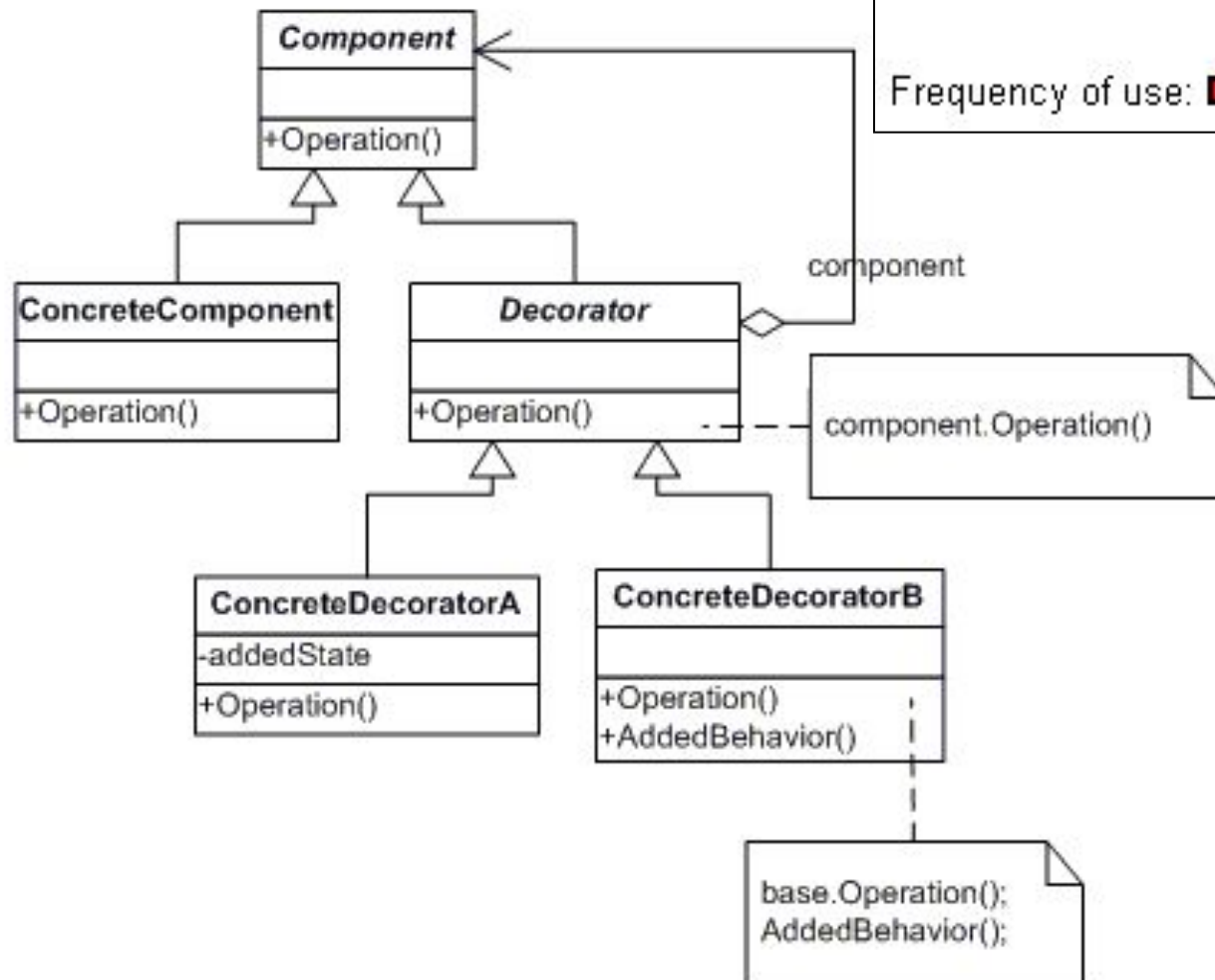
Decorator (data & object factory™, dofactory.com)

Definition

Attach additional responsibilities to an object dynamically. Decorators provide a flexible alternative to subclassing for extending functionality.

Frequency of use:  medium high

UML class diagram



Decorator. Приклад (1/2)

(data & object factory™, dofactory.com)

```
using System;
abstract class Component
{abstract public void Operation();
}
class ConcreteComponent :
Component
{override public void Operation()
{ Console.WriteLine
("ConcreteComponent.Operation()")
};
}
abstract class Decorator :
Component
{protected Component component;
public void SetComponent(
Component component )
{ this.component =
component;
}
override public void Operation()
{ if( component != null )
component.Operation();
}
}
```

```
class ConcreteDecoratorA :
Decorator
{private string addedState;
override public void Operation()
{ base.Operation();
addedState = "new state";
Console.WriteLine
("ConcDecoratorA.Operation()");
}
}
class ConcreteDecoratorB :
Decorator
{override public void Operation()
{ base.Operation();
AddedBehavior();
Console.WriteLine
("ConcDecoratorB.Operation()");
}
void AddedBehavior() { }
}
```

```
public class Client
{ public static void Main
( string[] args )
{
ConcreteComponent
c = new
ConcreteComponent();

ConcreteDecoratorA
d1 = new
ConcreteDecoratorA();

ConcreteDecoratorB
d2 = new
ConcreteDecoratorB();
// Link decorators
d1.SetComponent( c );
d2.SetComponent( d1 );

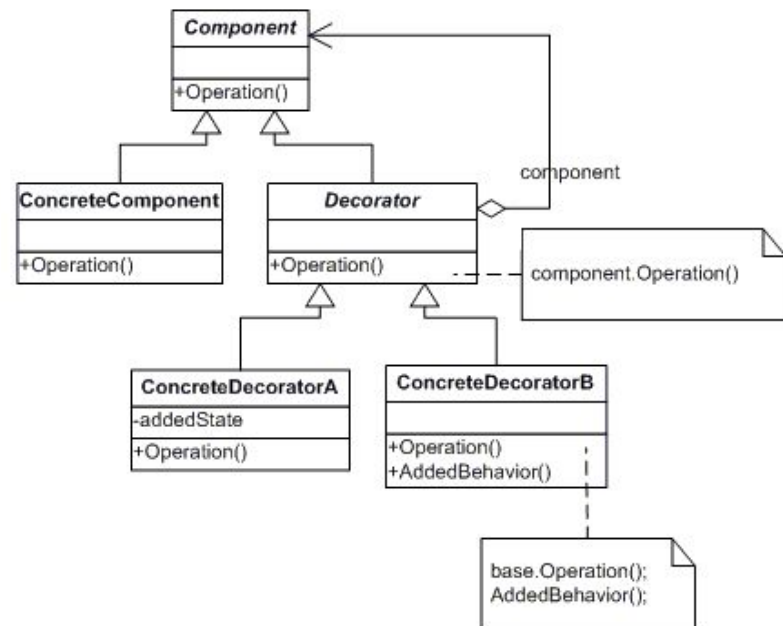
d2.Operation();
}
}
```

Decorator. Приклад (2/2)

d2 : CDecB

component = d1 : CDecA

component = c : CComponent



// Link decorators

```
d1.SetComponent( c );
d2.SetComponent( d1 );
d2.Operation();
```

“Додаткова гнучкість пов'язана з можливістю створювати композицію об'єктів під час виконання програми”.

```
abstract class Decorator
{protected Component component;
    .
    .
    .
    override public void Operation()
    {
        if( component != null )
            component.Operation();
    }
}
```

```
class ConcreteDecoratorB :
    Decorator
{override public
    void Operation()
    { base.Operation();
      // Decor B
    }
}
```

The screenshot shows the Eclipse IDE interface. The title bar reads "Java EE - Config File: /dekor/src/beans_ctx.xml - Eclipse". The menu bar includes File, Edit, Source, Navigate, Search, Project, Run, Window, and Help. The toolbar contains various icons for file operations and development. The Project Explorer on the left shows the project structure:

- dekor
 - Spring Elements
 - Beans
 - beans_ctx.xml
 - JRE System Library [jdk]
 - src
 - ttp.kvf
 - ConcreteComponent.java
 - Decorator.java
 - DecoratorA.java
 - DecoratorB.java
 - DecoratorC.java
 - IComponent.java
 - Project.java
 - beans_ctx.xml
 - antlr-2.7.6.jar
 - asm.jar
 - cglib-2.1.3.jar
 - commons-beanutils.jar
 - commons-collections.jar
 - commons-digester.jar
 - commons-lang-2.3.jar
 - commons-logging-1.1.jar

The main editor displays the "beans_ctx.xml" configuration file with the following structure:

```
graph TD; rootComponent[rootComponent] --> component1[component]; component1 --> decoratorB[decoratorB]; decoratorB --> component2[component]; component2 --> decoratorC[decoratorC]; decoratorC --> component3[component]; component3 --> concreteComponent[concreteComponent];
```

The bottom console shows the output of the application:

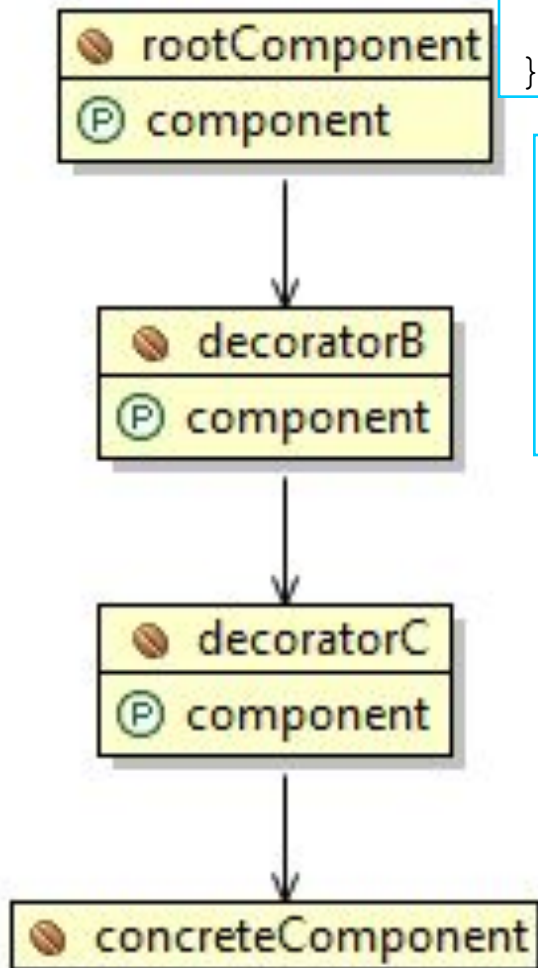
```
<terminated> Project [Java Application] C:\JavaSDK\jdk\l  
ConcreteComponent !  
DecoratorC  
DecoratorB  
DecoratorA
```

Вигляд проекту *dekor* (проект містить три класи конкретних декораторів (*DecoratorA*, *DecoratorB*, *DecoratorC*))

Середовище *Eclipse + Spring Plugin*

“Дротяна модель” та виконання проекту

beans_ctx.xml
Config File: /dekor/src/bean



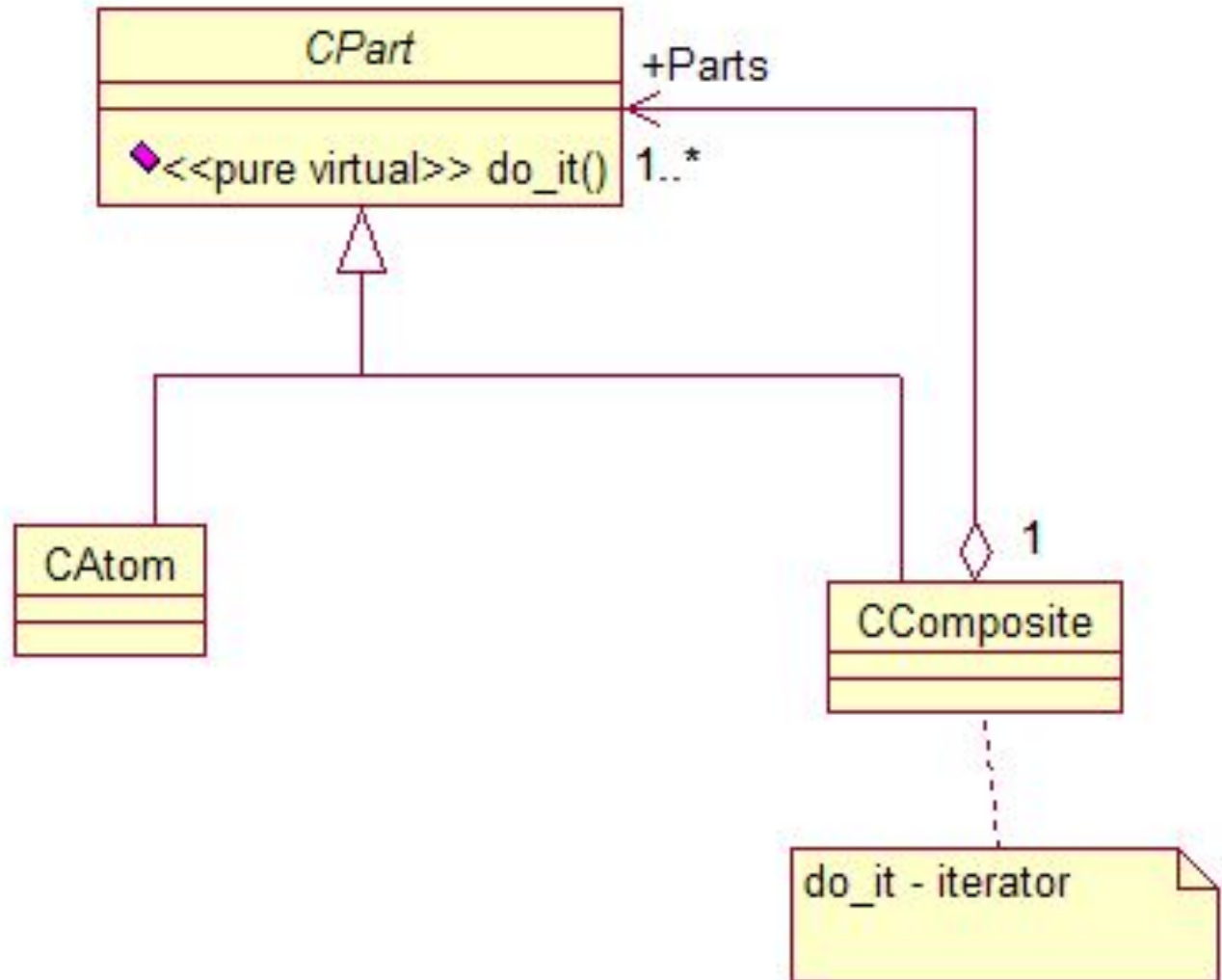
```
public class Decorator implements IComponent {  
    private IComponent component;  
    public void setComponent(IComponent component) {  
        this.component = component;  
    }  
    public void operation() {  
        component.operation();  
    }  
}
```

```
public class DecoratorA extends Decorator {  
    public void operation() {  
        super.operation();  
        System.out.println("DecoratorA");  
    }  
}
```

ConcreteComponent!
DecoratorC
DecoratorB
DecoratorA

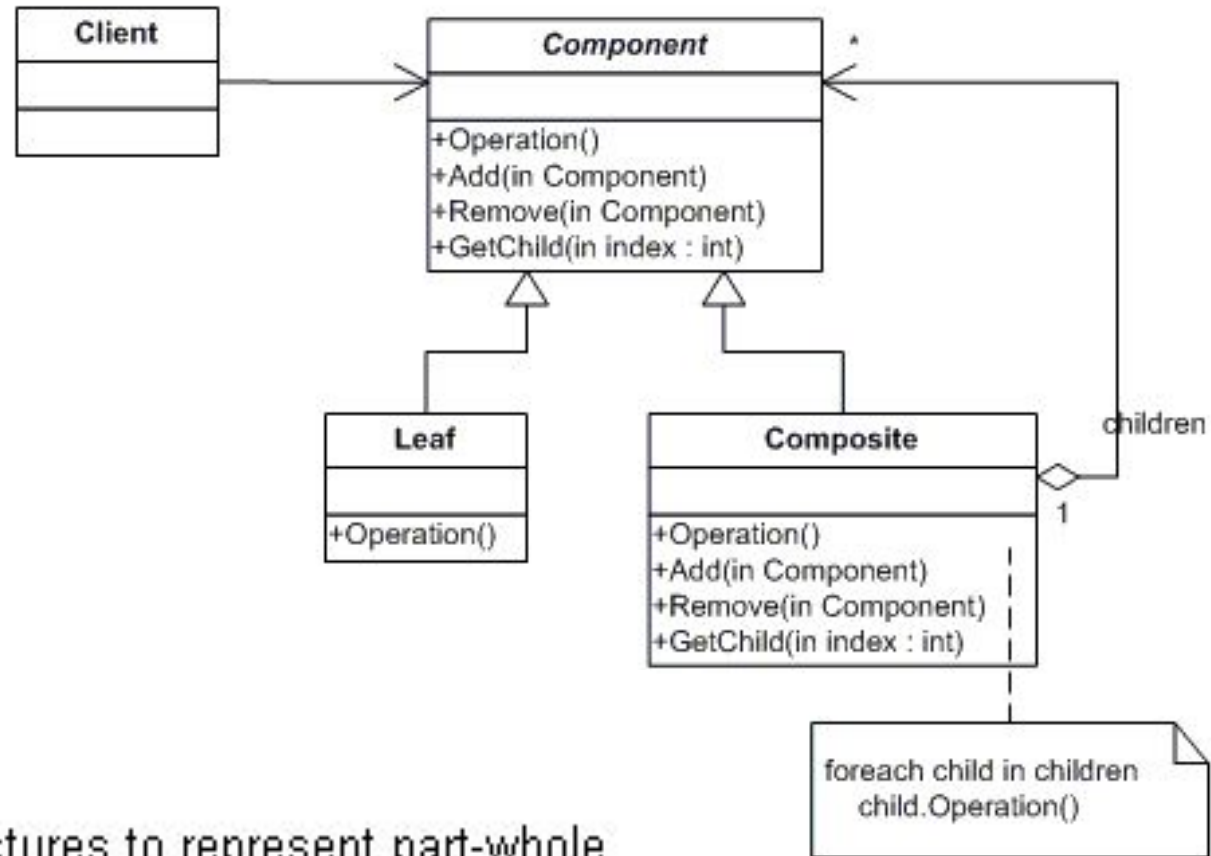
Composite (комполитор, компоновник)

Групує об'єкти в деревоподібні структури для представлення ієрархій типу "частина-ціле". Дозволяє уніфікувати дії як з листками, так і з будь-якими групами піддерев.




Composite (data & object factory™, dofactory.com)

UML class diagram



Definition

Compose objects into tree structures to represent part-whole hierarchies. Composite lets clients treat individual objects and compositions of objects uniformly.

Frequency of use:  high

Composite (data & object factory™, dofactory.com)

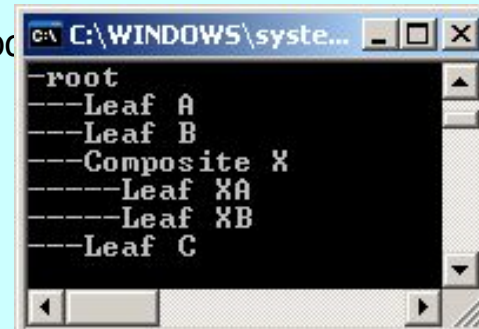
```
using System;
using System.Text;
using System.Collections;
abstract class Component
{ protected string name;
public Component( string name ) // Constructor
{ this.name = name;
}
abstract public void Add(Component c);
abstract public void Remove( Component c );
abstract public void Display( int depth );
}
class Composite : Component
{private ArrayList children = new ArrayList();
// Constructor
public Composite( string name ) : base( name ) {}
public override void Add( Component component )
{ children.Add( component );
}
public override void Remove( Component component
)
{ children.Remove( component );
}
public override void Display( int depth )
{ Console.WriteLine( new String( '-', depth ) + name
);
// Display each of the node's children
foreach( Component component in children )
component.Display( depth + 2 );
}
}
```

```
class Leaf : Component
{public Leaf( string name ) : base( name ) {} //
Constructor
public override void Add( Component c )
{ Console.WriteLine("Cannot add to a leaf");
}
public override void Remove( Component c )
{ Console.WriteLine("Cannot remove from a leaf");
}
public override void Display( int depth )
{ Console.WriteLine( new String( '-', depth ) +
name );
}
}
public class Client
{ public static void Main( string[] args )
{ // Create a tree structure
Composite root = new Composite( "root" );
root.Add( new Leaf( "Leaf A" ));
root.Add( new Leaf( "Leaf B" ));

Composite comp = new Composite( "Composite X"
);
comp.Add( new Leaf( "Leaf XA" ));
comp.Add( new Leaf( "Leaf XB" ));

root.Add( comp );
root.Add( new Leaf( "Leaf C" ));

// Recursively display node
root.Display( 1 );
}
}
```

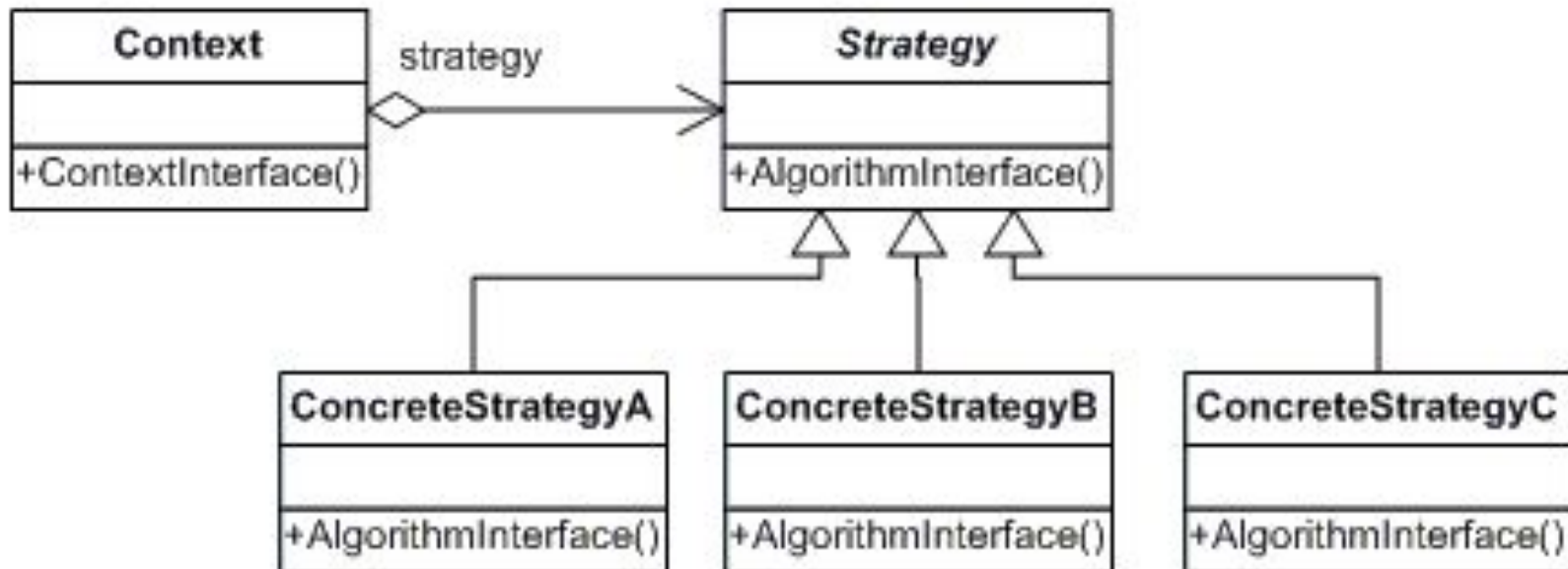


```
C:\WINDOWS\system... _ □ ×
- root
--- Leaf A
--- Leaf B
--- Composite X
----- Leaf XA
----- Leaf XB
--- Leaf C
```

Strategy (dofactory.com)


UML class diagram

Визначає сімейство алгоритмів, в якому інкапсулюється кожен з них і забезпечується їх взаємозаміна. Патерн "Стратегія" дозволяє змінювати алгоритми сімейства незалежно від клієнтів, які використовують ці алгоритми.



definition

Define a family of algorithms, encapsulate each one, and make them interchangeable. Strategy lets the algorithm vary independently from clients that use it.

Frequency of use:  medium high

Strategy (dofactory.com)

class MainApp

```
{
static void Main()
{
    Context context;
    context = new Context(
        new ConcreteStrategyA());
    context.ContextInterface();
    context = new Context(
        new ConcreteStrategyB());
    context.ContextInterface();
}
}

/// The 'Strategy' abstract class
abstract class Strategy
{
    public abstract void AlgorithmInterface();
}

class ConcreteStrategyA : Strategy
{
    public override void AlgorithmInterface()
    {
        Console.WriteLine(
            "CalledConcreteStrategyA."+
            "AlgorithmInterface()");
    }
}
```

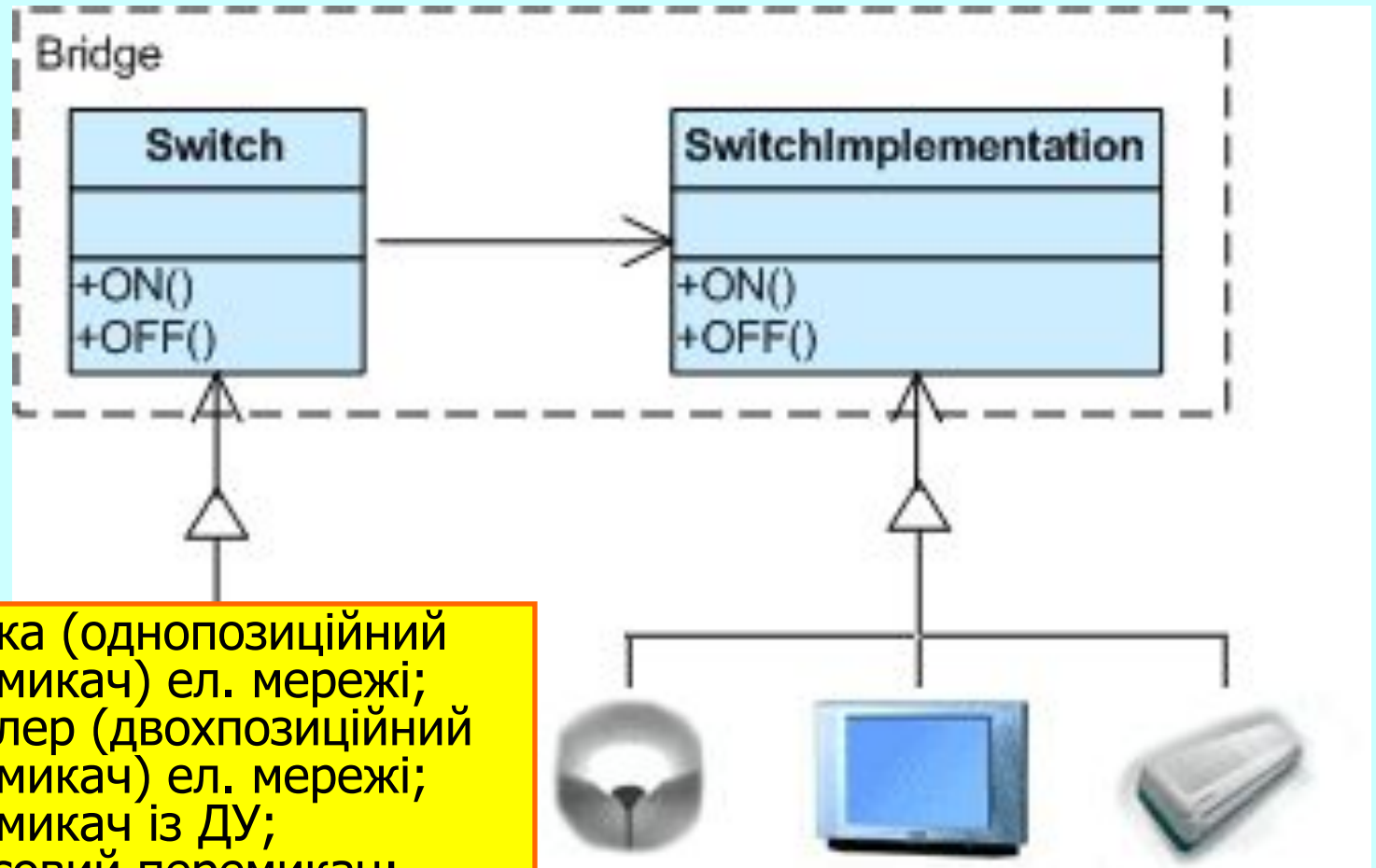
class ConcreteStrategyB : Strategy

```
{
    public override void AlgorithmInterface()
    {
        Console.WriteLine(
            "CalledConcreteStrategyB."+
            "AlgorithmInterface()");
    }
}

class Context
{
    private Strategy _strategy;
    // Constructor
    public Context(Strategy strategy)
    {
        this._strategy = strategy;
    }
    public void ContextInterface()
    {
        _strategy.AlgorithmInterface();
    }
}
```


Bridge

- Відокремлює абстракцію від реалізації, завдяки чому з'являється можливість незалежно змінювати те й інше

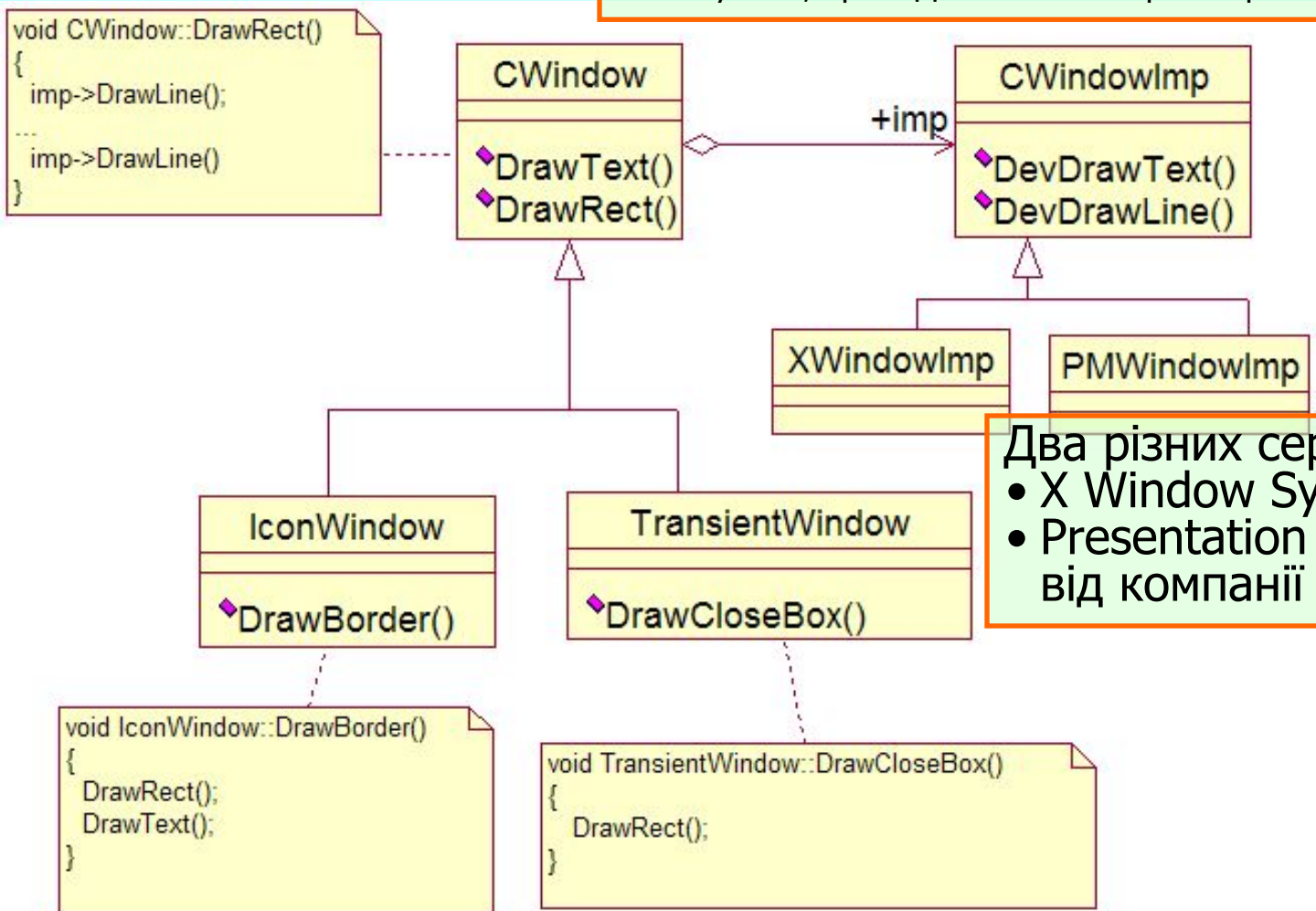


- кнопка (однопозиційний перемикач) ел. мережі;
- тумблер (двохпозиційний перемикач) ел. мережі;
- перемикач із ДУ;
- голосовий перемикач;
- . . .

Bridge

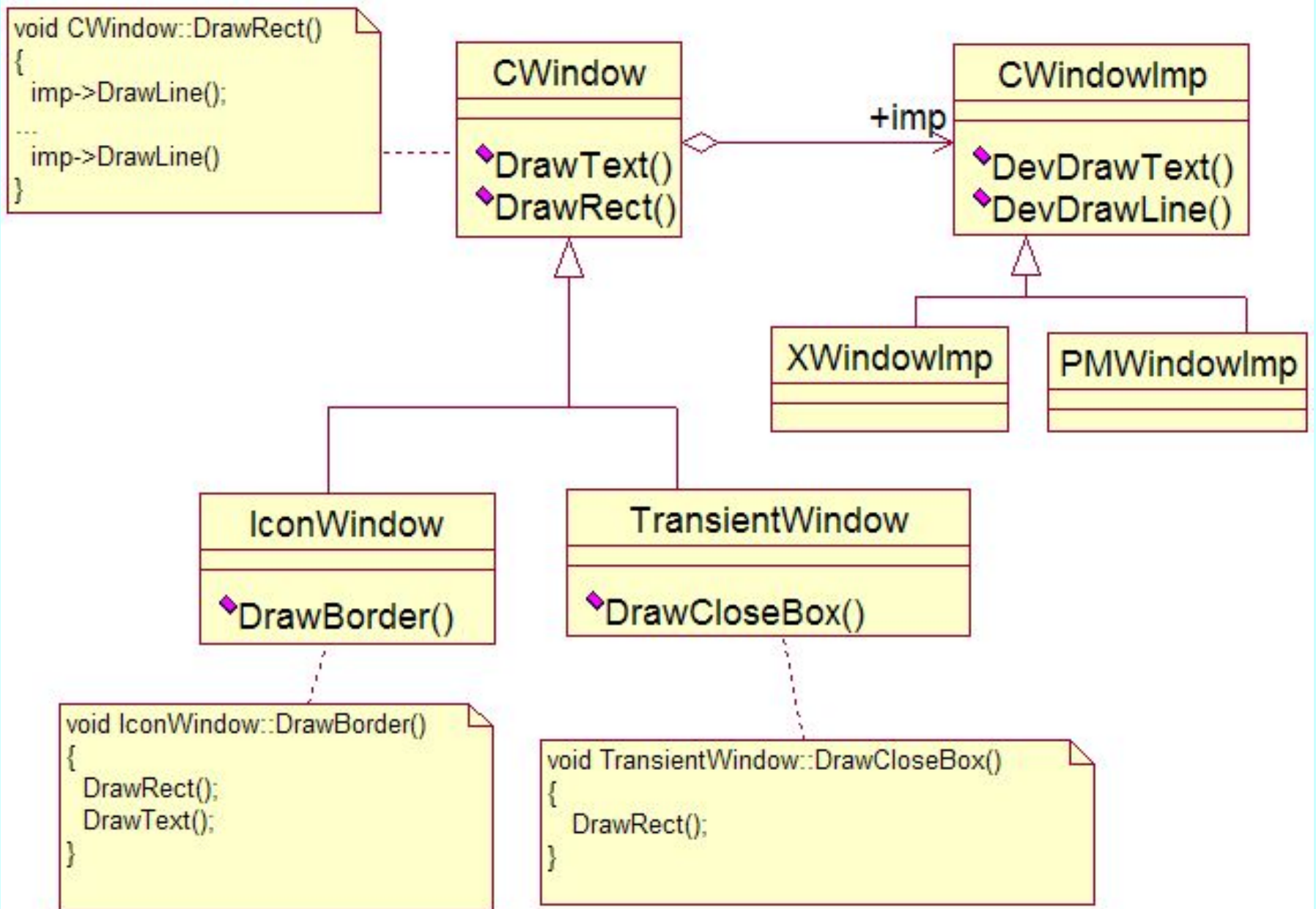
- Відокремлює абстракцію від реалізації, завдяки чому з'являється можливість незалежно змінювати те й інше.
- Відомий також під іменем Handle/Body (описувач/тіло).

Шаллоуей А., Тротт Дж. Шаблоны проектирования. — ИД "Вильямс", 2002.

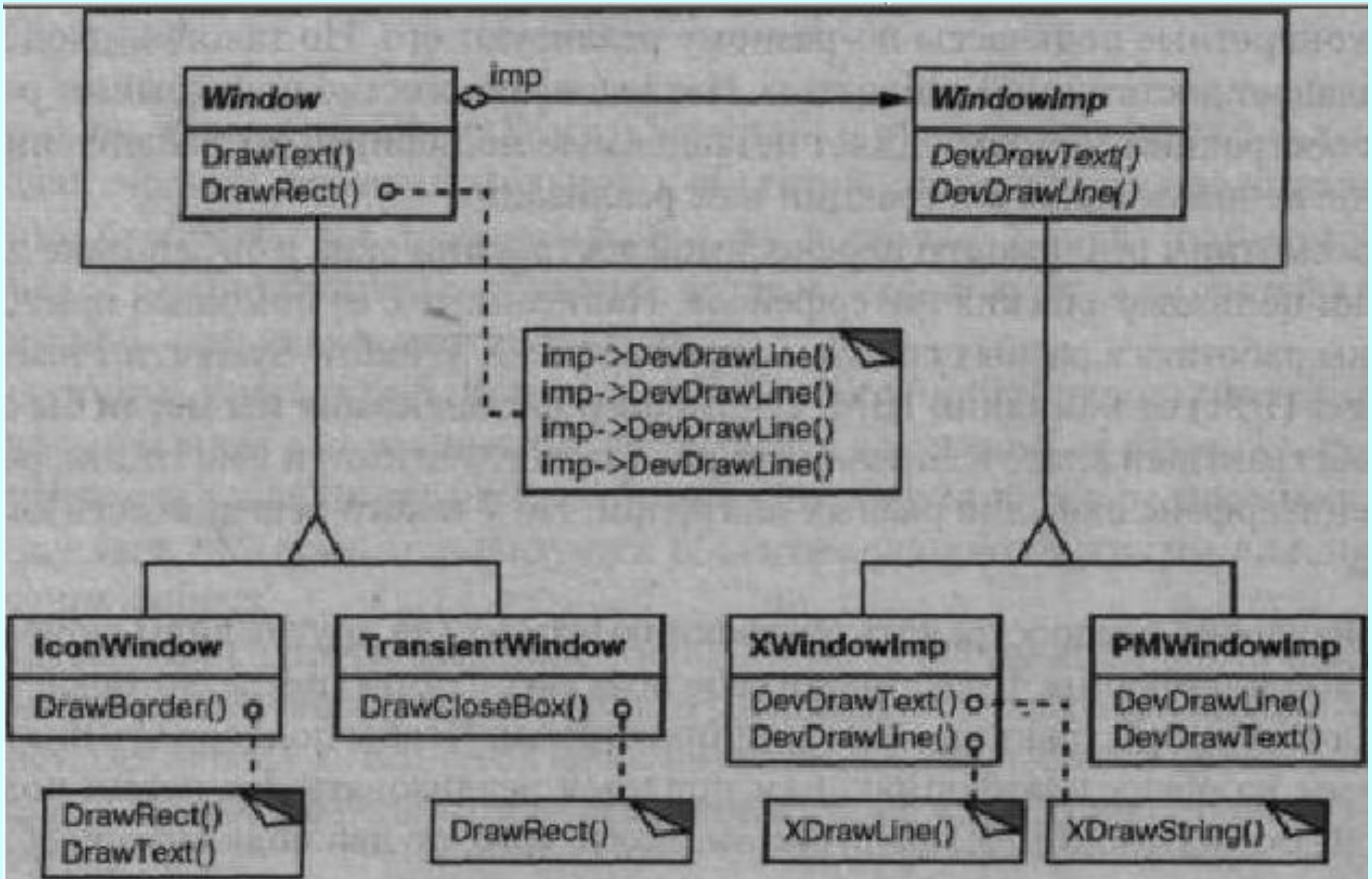


Два різних середовища:
• X Window System;
• Presentation Manager (PM)
від компанії IBM.

Bridge

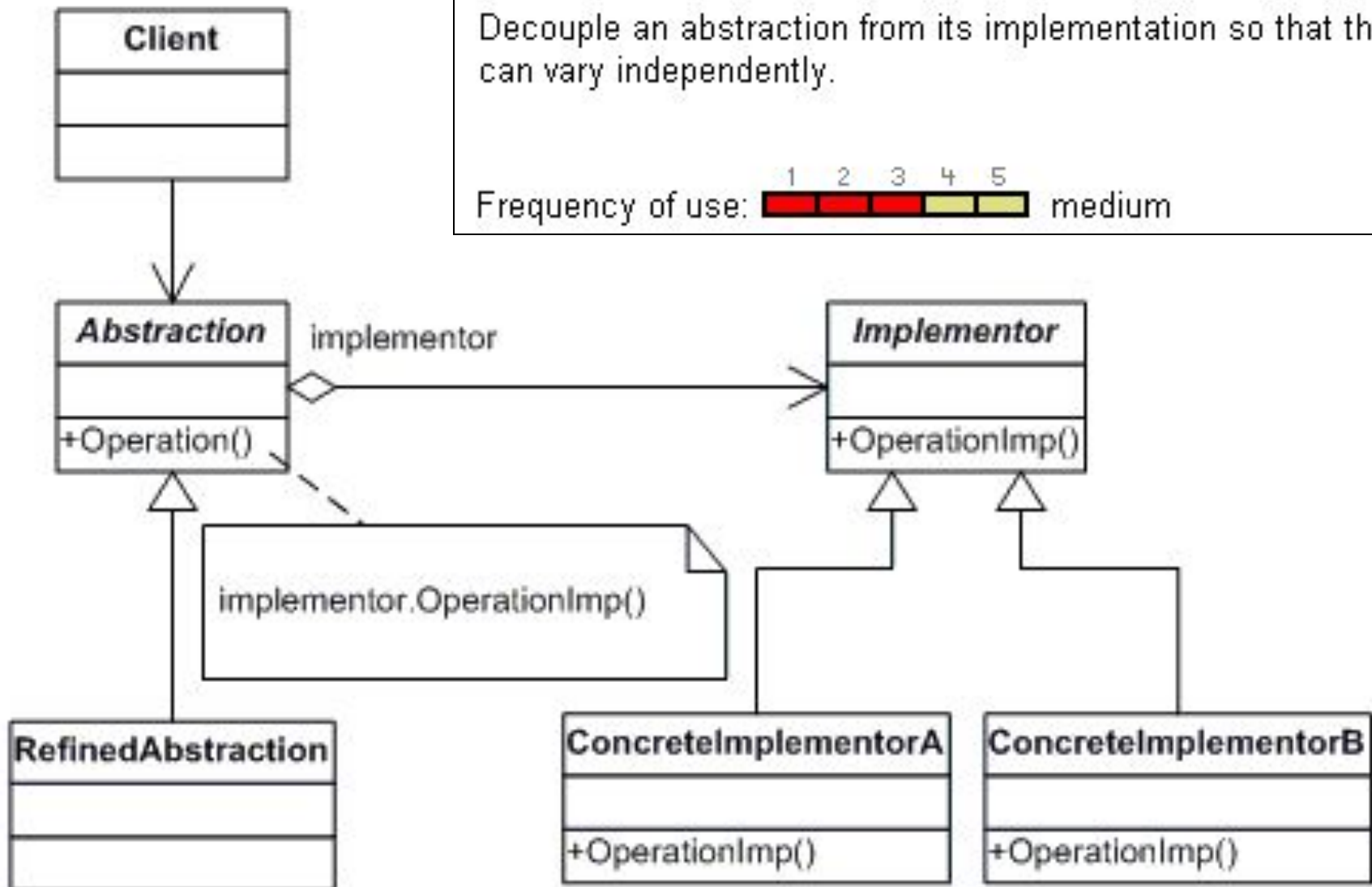


Bridge (GoF)



Bridge (data & object factory™, dofactory.com)

UML class diagram



Definition

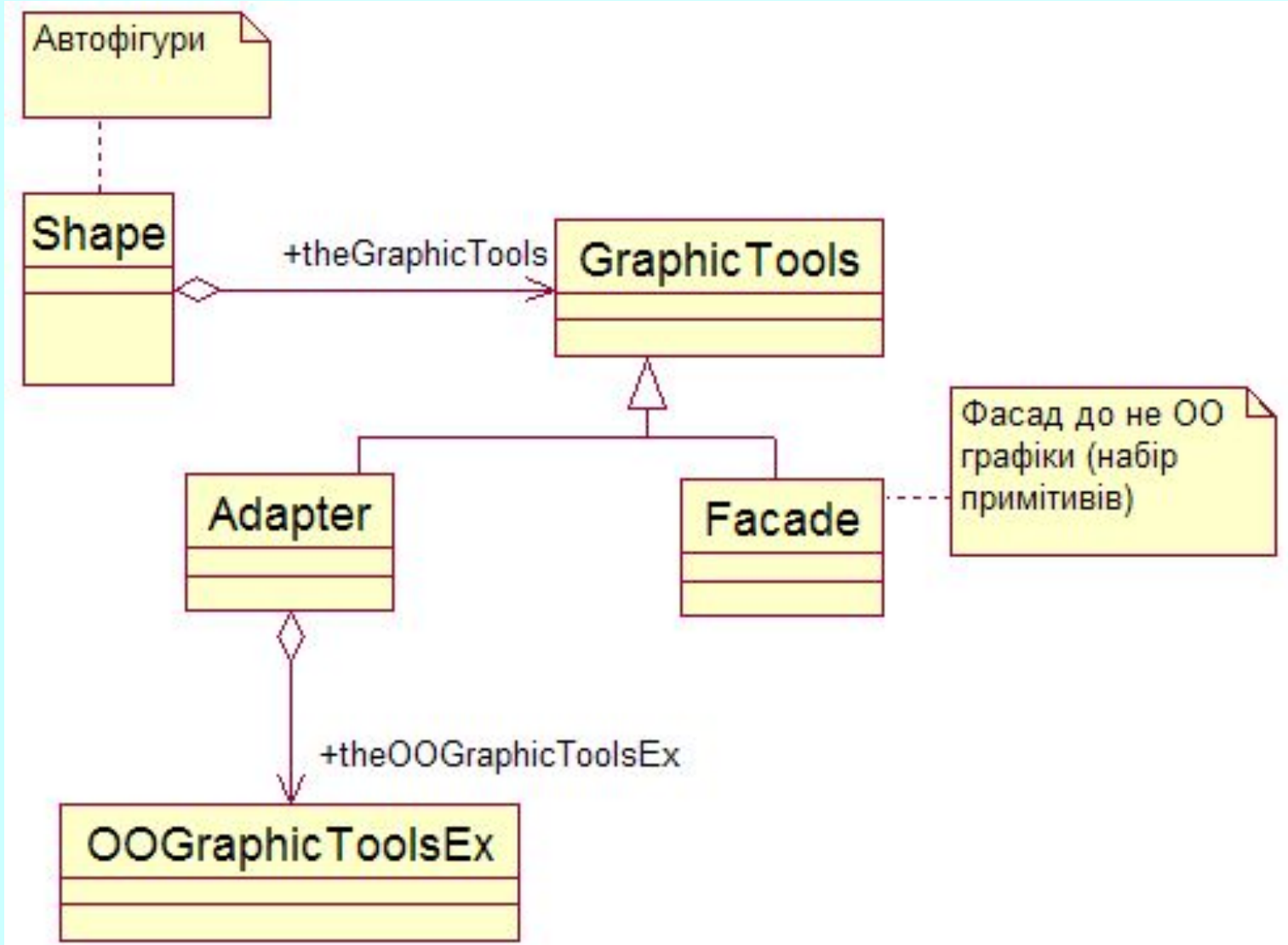
Decouple an abstraction from its implementation so that the two can vary independently.

Frequency of use:

1	2	3	4	5
■	■	■	■	■

 medium

Bridge. "Починати з тих шаблонів, що створюють контекст для інших" (К.Александр).



Façade (data & object factory™, dofactory.com)

Definition

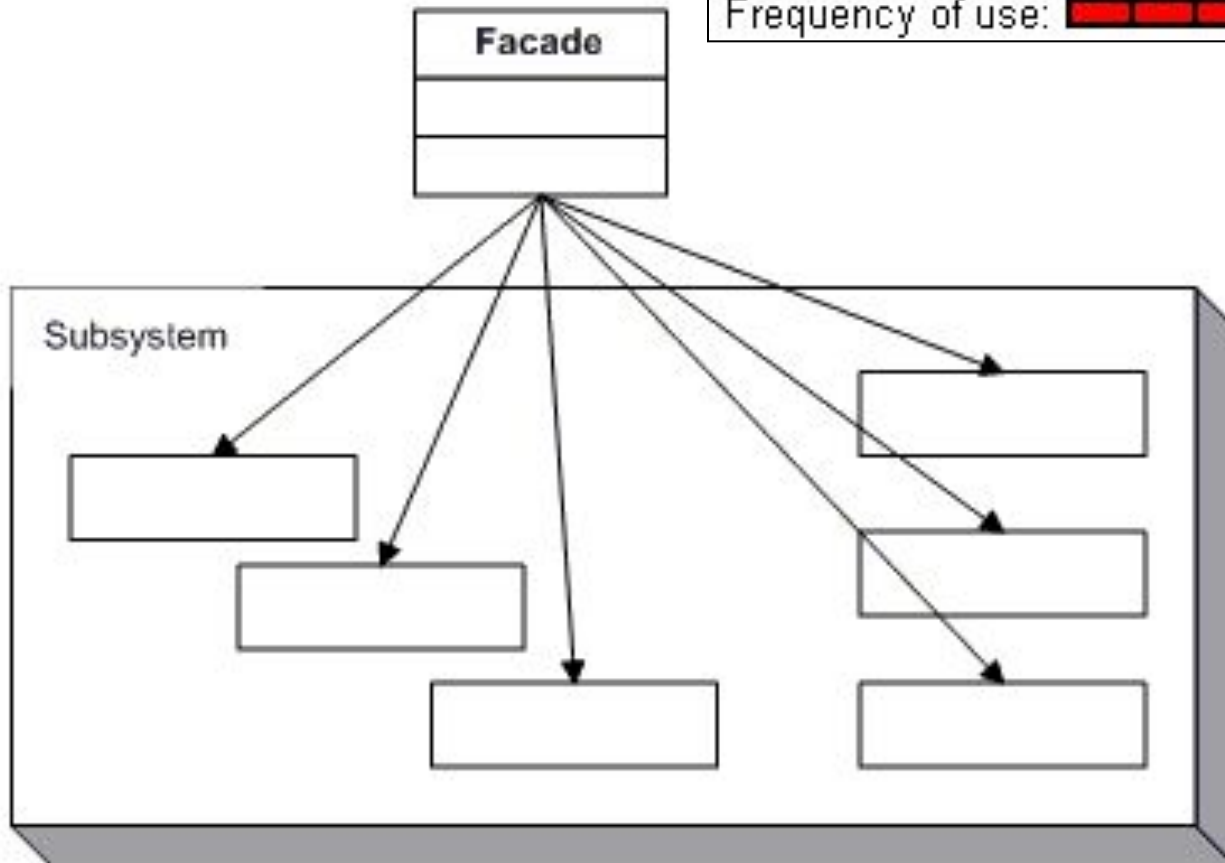
Provide a unified interface to a set of interfaces in a subsystem. Façade defines a higher-level interface that makes the subsystem easier to use.

Frequency of use:

1	2	3	4	5
---	---	---	---	---

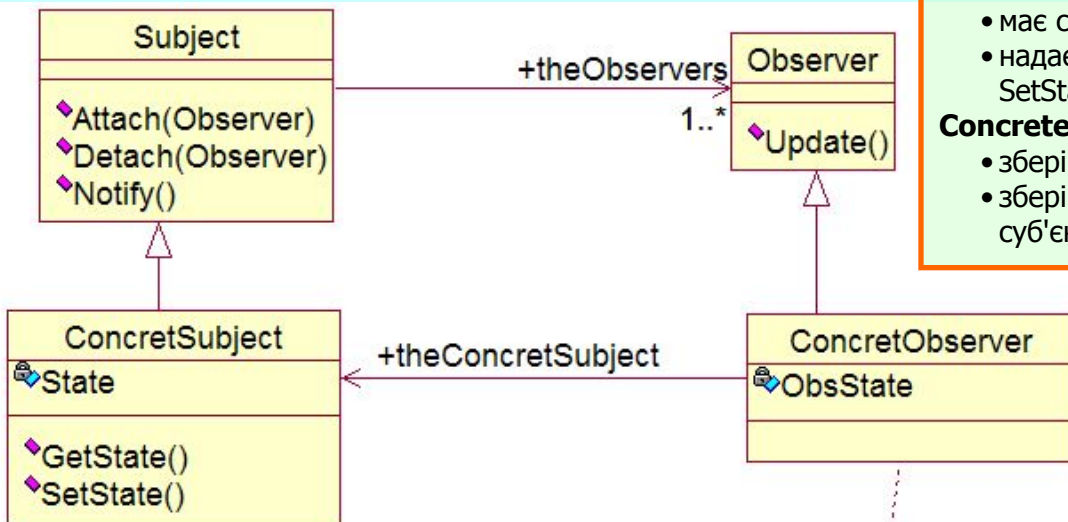
 high

UML class diagram



Observer (Спостерігач)

- Визначає між об'єктами відношення типу один (видавець) - до - багатьох (передплатників), так що при зміні стану одного об'єкта (видавця) всі підлеглі (передплатники) одержують повідомлення й автоматично оновлюють дані.
- Відомий також під іменами *Dependents* (підлеглі), *Publish-Subscribe* (видавець-передплатник).



Subject (суб'єкт):

- надає інтерфейс для приєднання та від'єднання спостерігачів;
- має інформацію про приєднаних спостерігачів (для їх оповіщення); за суб'єктом можуть "стежити" скільки завгодно спостерігачів.

Observer (спостерігач):

- надає інтерфейс для фіксації змін (операція Update).

ConcreteSubject (конкретний суб'єкт):

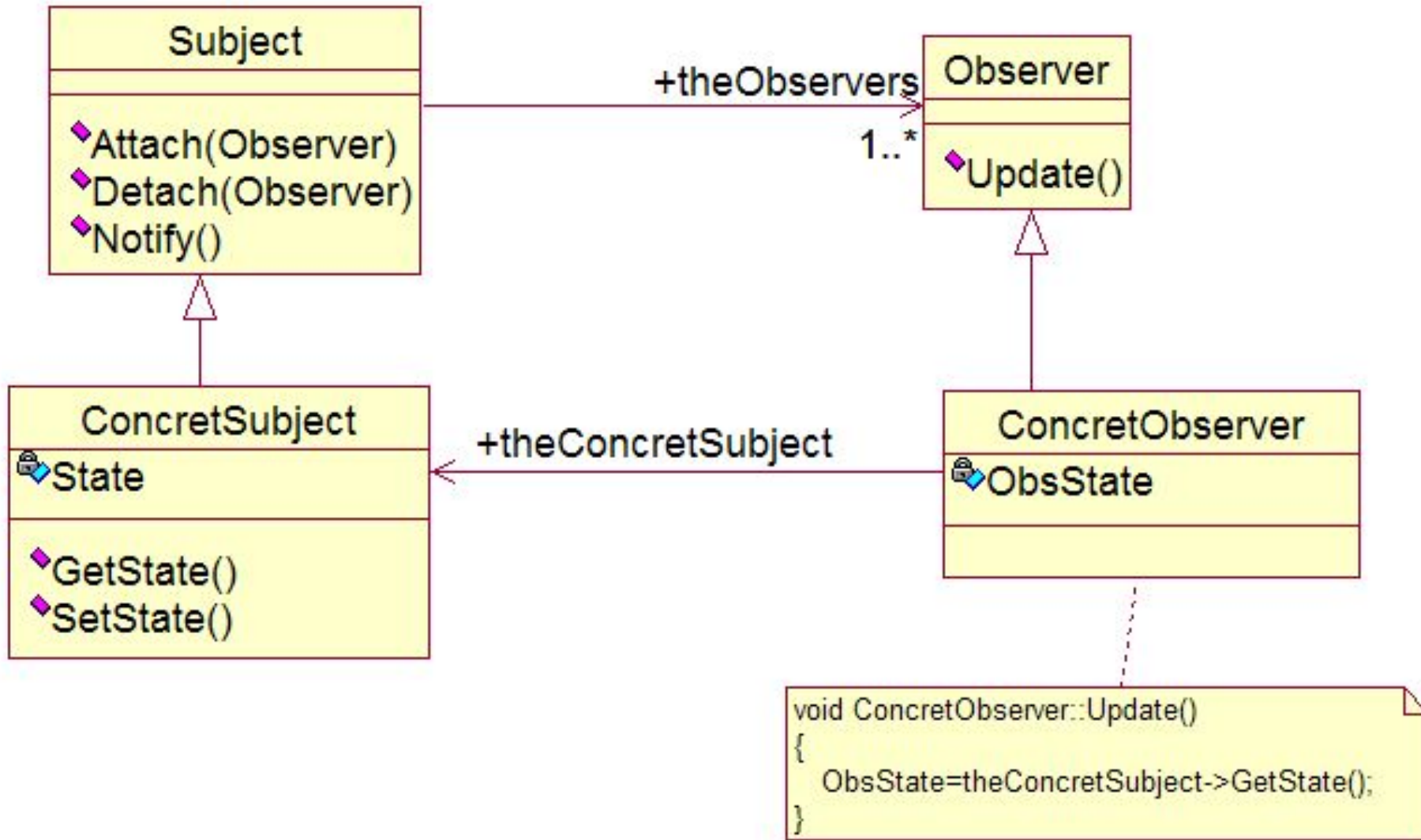
- має стан, що становить інтерес для ConcreteObserver;
- надає інтерфейс для "читання" та зміни стану (GetState, SetState).

ConcreteObserver (конкретний спостерігач):

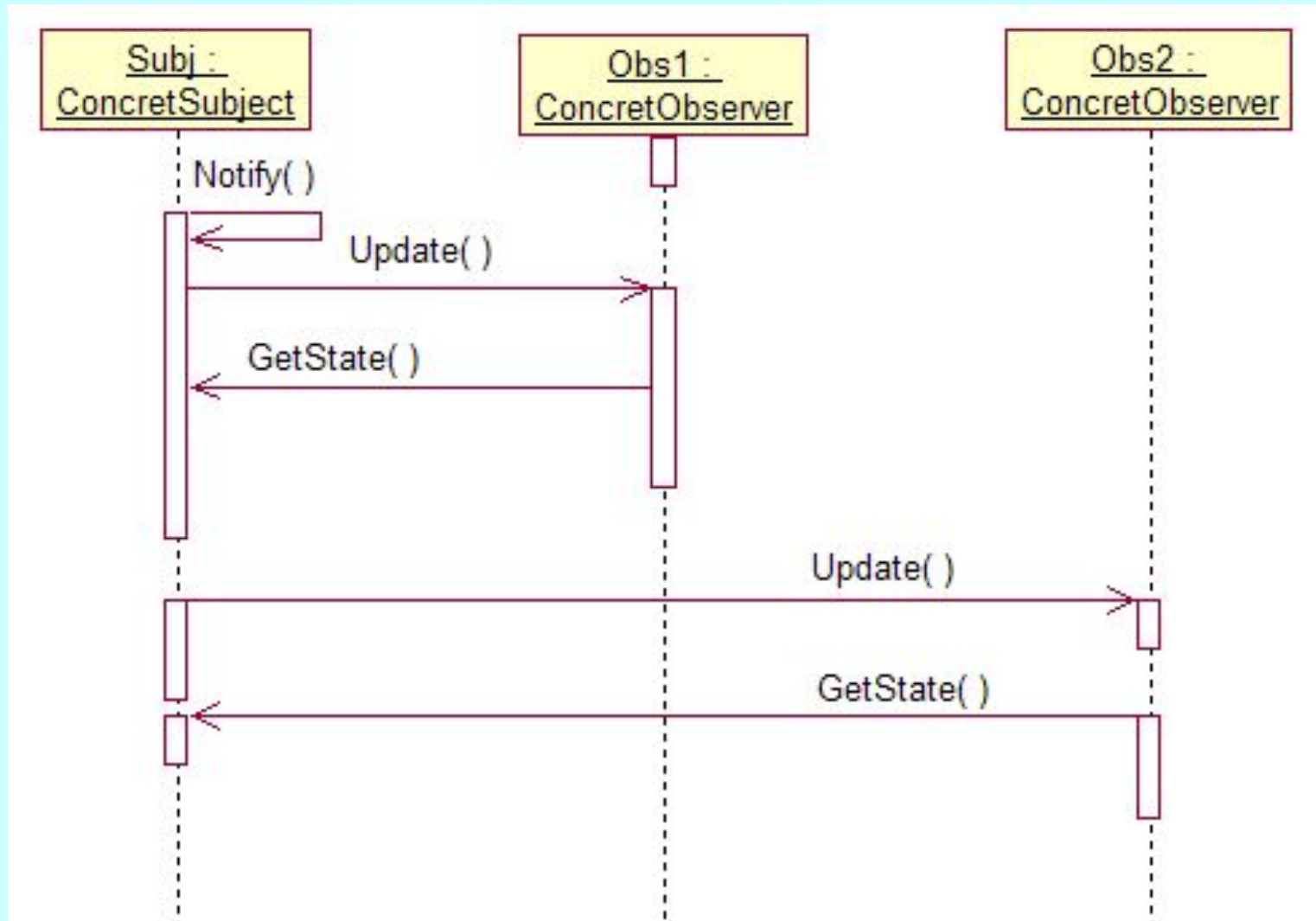
- зберігає посилання на об'єкт класу ConcreteSubject;
- зберігає дані про стан, які повинні бути узгоджені із станом суб'єкта.

```
void ConcretObserver::Update()
{
    ObsState=theConcretSubject->GetState();
}
```

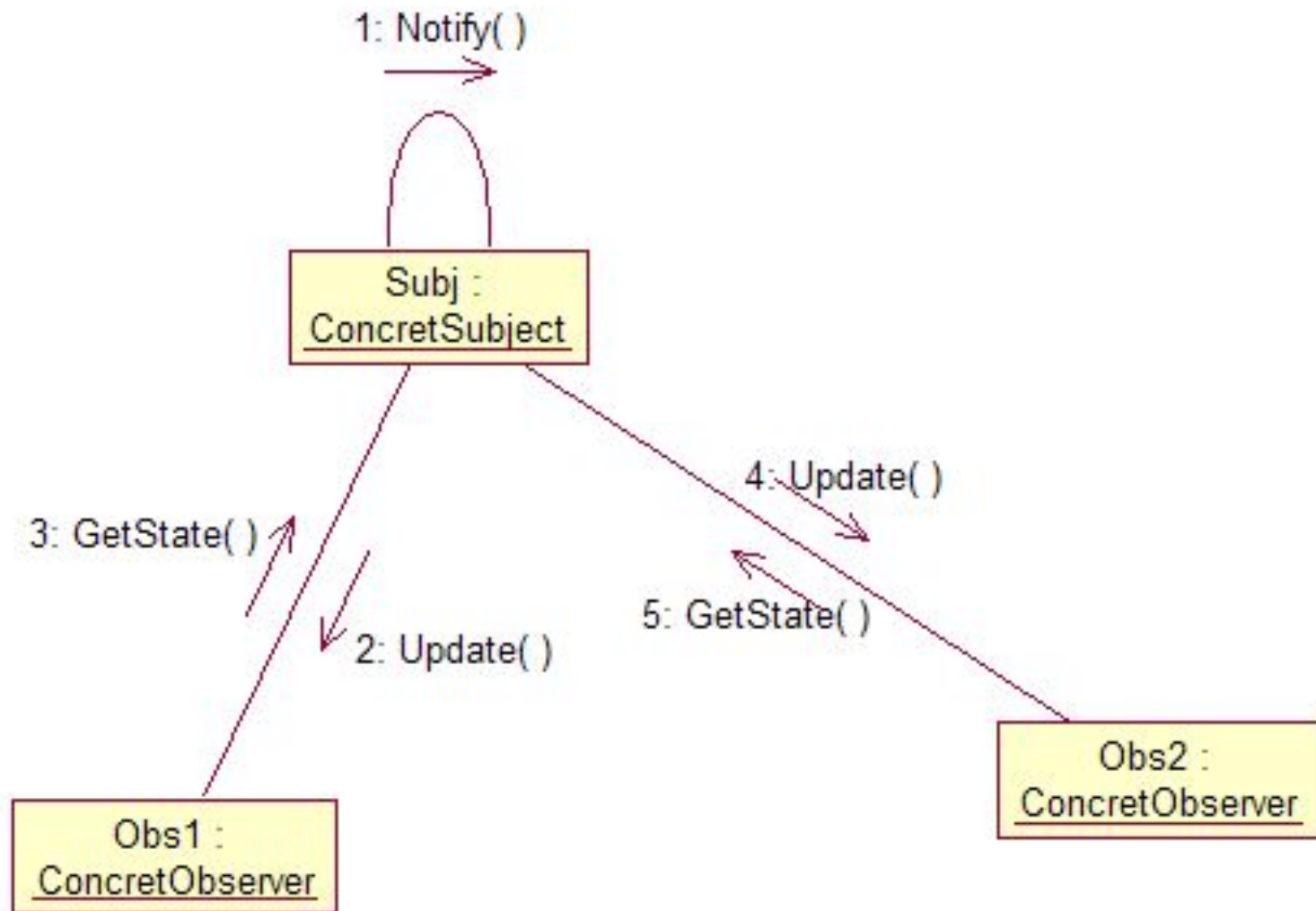
Observer



Observer




Observer



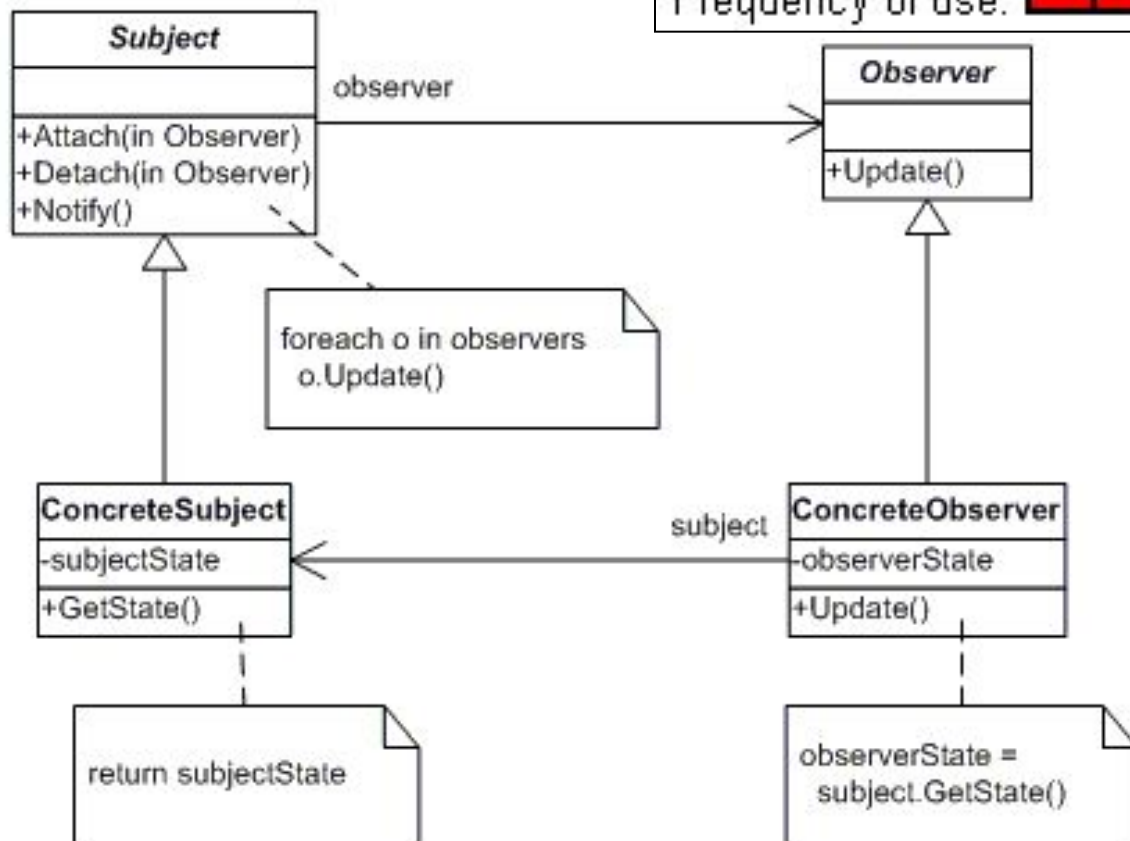
Observer (data & object factory™, dofactory.com)

Definition

Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.

Frequency of use:  high

UML class diagram



Observer (data & object factory™, dofactory.com)

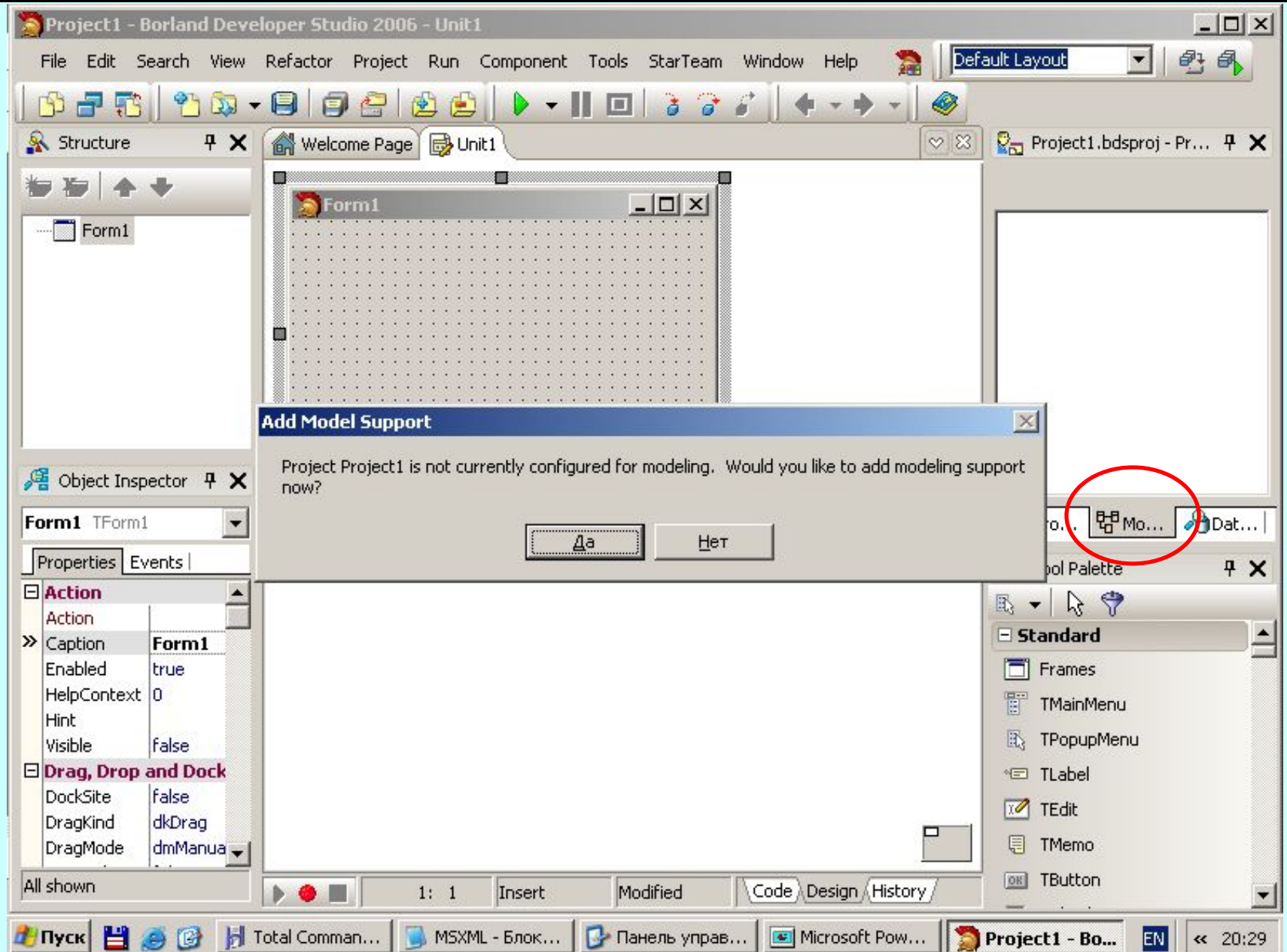
```
using System;
using System.Collections;
abstract class Subject // "Subject"
{ // Fields
    private ArrayList observers = new
        ArrayList();
    // Methods
    public void Attach( Observer observer )
    { observers.Add( observer );
    }
    public void Detach( Observer observer )
    { observers.Remove( observer );
    }
    public void Notify()
    { foreach( Observer o in observers )
        o.Update();
    }
}
class ConcreteSubject : Subject
{ // Fields
    private string subjectState;
    // Properties
    public string SubjectState
    { get{ return subjectState; }
      set{ subjectState = value; }
    }
}
```

```
abstract class Observer // "Observer"
{abstract public void Update();
}
class ConcreteObserver : Observer
{ private string name;
  private string observerState;
  private ConcreteSubject subject;
public ConcreteObserver( ConcreteSubject
    subject, string name ) // Constructor
{ this.subject = subject;
  this.name = name;
}
override public void Update()
{ observerState = subject.SubjectState;
  Console.WriteLine( "Observer {0}'s
    new state is {1}",
    name, observerState );
}
public ConcreteSubject Subject
{ get { return subject; }
  set { subject = value; }
}
}
```

```
/// Client test
public class Client
{
    public static void Main(
        string[] args )
    {
        // Configure Observer
        // structure
        ConcreteSubject s =
            new ConcreteSubject();
        s.Attach(new
            ConcreteObserver( s, "X" ) );
        s.Attach( new
            ConcreteObserver( s, "Y" ) );
        s.Attach( new
            ConcreteObserver( s, "Z" ) );
        // Change subject, notify
        // observers
        s.SubjectState = "ABC";
        s.Notify();
    }
}
```

Додаток

Borland Developer Studio 2006



Borland Developer Studio 2006

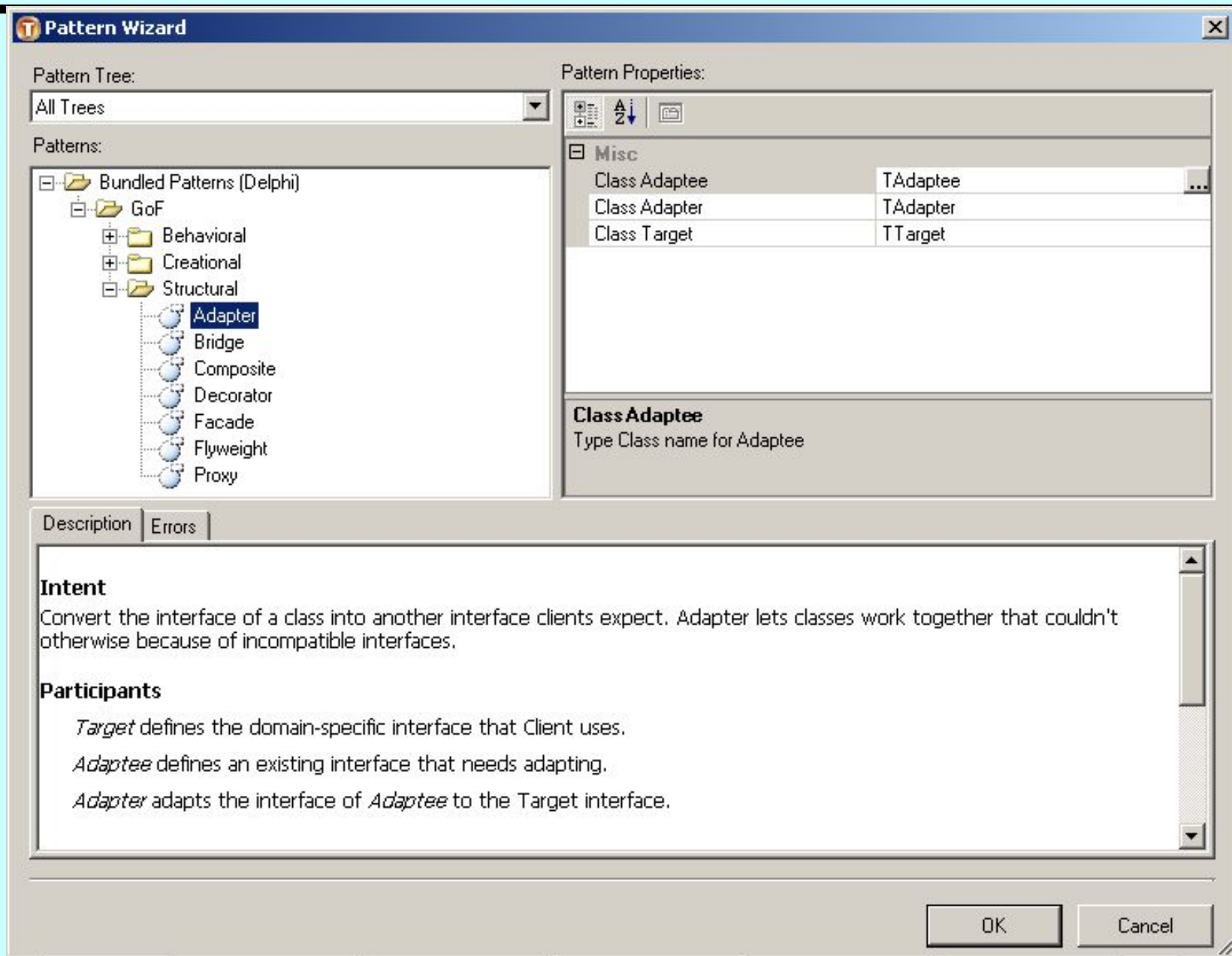
The screenshot displays the Borland Developer Studio 2006 interface. The main window is titled "Project1 - Borland Developer Studio 2006 - Unit1". The menu bar includes File, Edit, Search, View, Refactor, Project, Run, Component, Tools, StarTeam, Window, and Help. The toolbar contains various icons for file operations and development actions. The left sidebar shows the Structure view with folders for Classes, Variables/Constants, and Uses. The Object Inspector shows the General properties for the selected Unit1 object. The central code editor displays the following Pascal code:

```
1 unit Unit1;
.
. interface
.
. uses
  Windows, Messages, SysUtils, Va
  Dialogs;
.
. type
10 TForm1 = class(TForm)
  private
    { Private declarations }
  public
    { Public declarations }
  end;
.
. var
  Form1: TForm1;
20
. implementation
.
  {$R *.dfm}
.
. end.
```

A context menu is open over the code, listing various actions such as Add, Cut, Copy, Paste, Delete, Rename, Add as Shortcut, Generate Documentation..., Create by Pattern... (highlighted), Search for Usages..., QA Audits..., QA Metrics..., Hyperlinks, Constraints..., and User Properties... The right sidebar shows the Model View with a tree structure of the project.

At the bottom of the screen, the Windows taskbar is visible, showing the Start button, several application icons, and the active window "Project1 - Borland De...". The system tray on the right shows the language set to EN and the time as 21:07.

Borland Developer Studio 2006



Borland Developer Studio 2006

