Патерни проектування

2005-2012

Що таке патерн (*pattern*)?

За Кристофером Олександером, 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 Олександер 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., Влиссидес Дж. Приемы объектно-ориентированного проектирования. Паттерны проектирования. СПб: Питер, 2001. — 368 с.

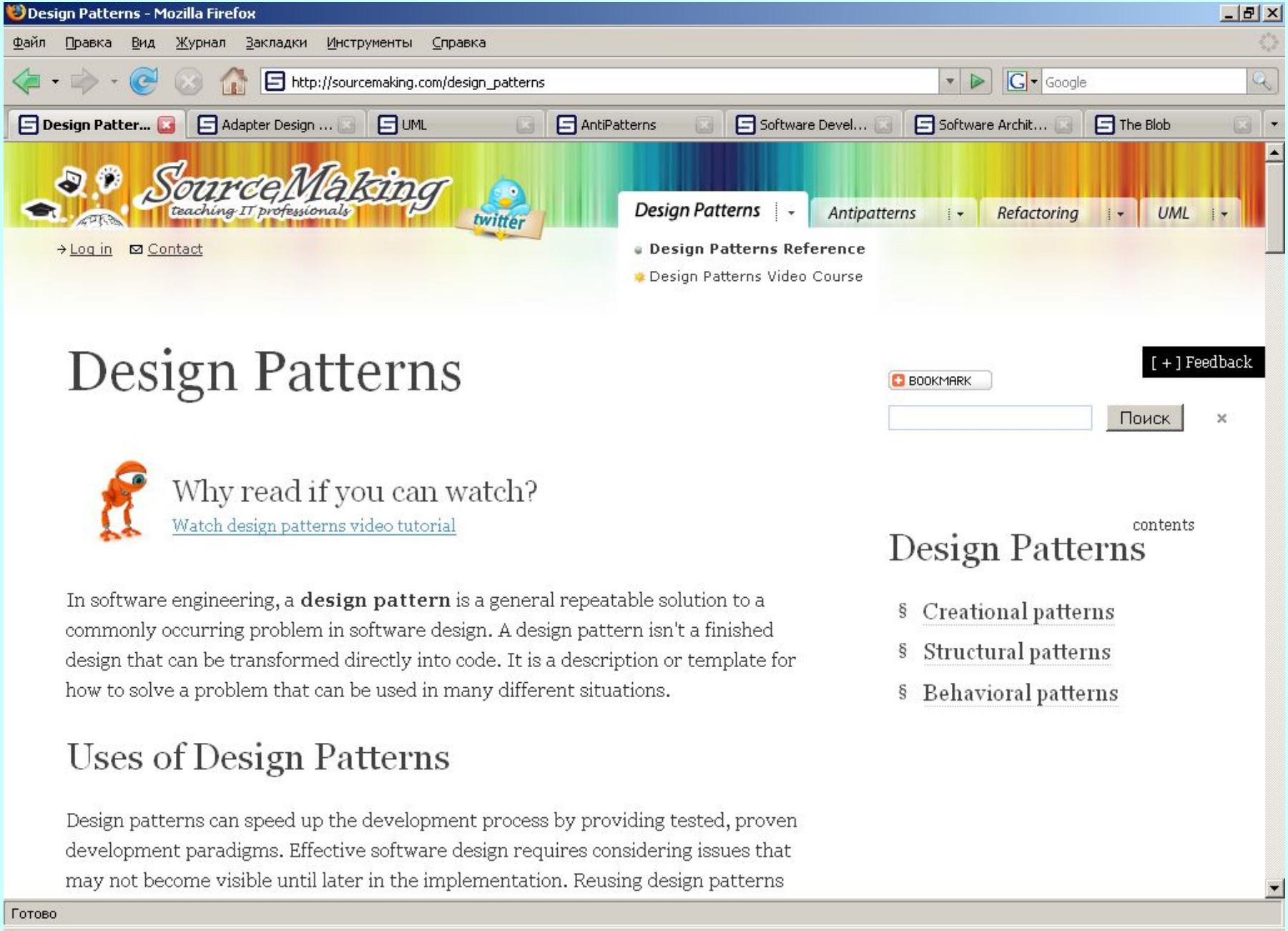
- Шаллоуей А., Тротт Дж. Шаблоны проектирования. — ИД “Вильямс”, 2002.

http://www.dofactory.com

Рекомендовано!

Ілюстрації!

http://sourcemaking.com/design_patterns



http://www.citforum.ru/SE/project/pattern/ (1/2)

2005 г.

Обзор паттернов проектирования

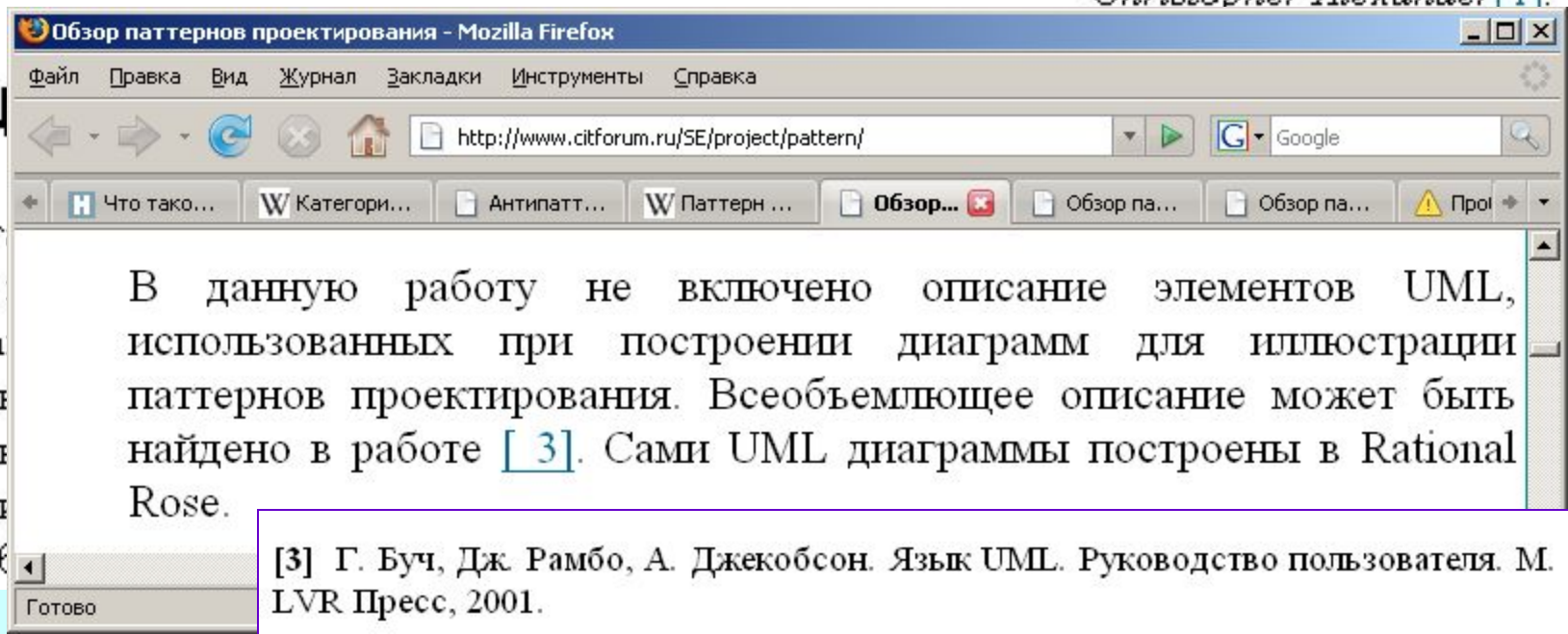
Ольга Дубина

"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].

АННОТАЦ

Данная работ
посвященных
структурирова
информационн
информационн
проектировани
ключевые особ



[3] Г. Буч, Дж. Рамбо, А. Джекобсон. Язык UML. Руководство пользователя. М. LVR Пресс, 2001.

http://www.citforum.ru/SE/project/pattern/ (2/2)

GRASP (General Responsibility Assignment Software Patterns)

Craig Larman
Applying UML and Patterns

GRASP

Материал из Википедии — свободной энциклопедии [править]

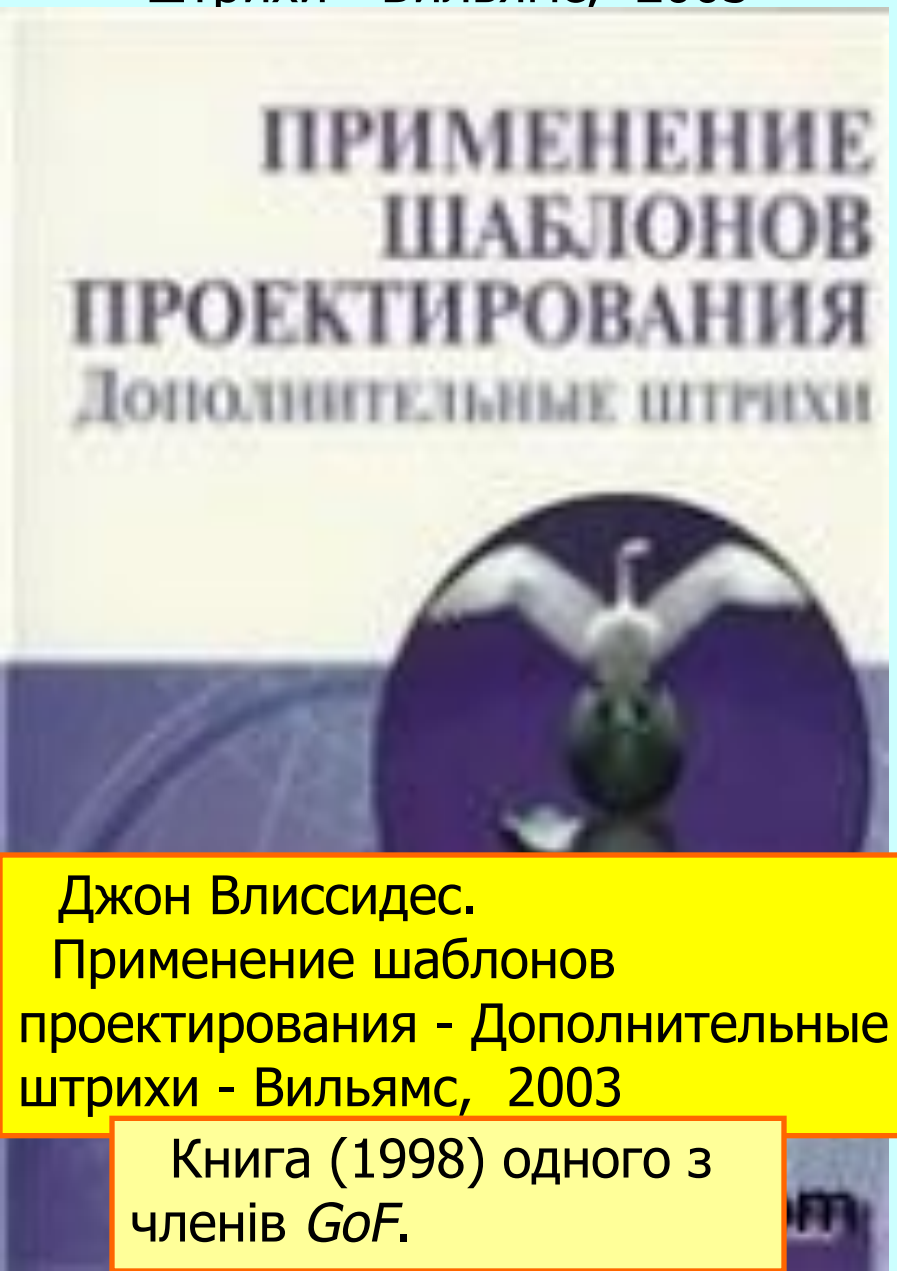
GRASP (*англ.* **General Responsibility Assignment Software Patterns** (общие образцы распределения обязанностей)) - паттерны, используемые в объектно-ориентированном проектировании для решения общих задач по назначению обязанностей классам и объектам.

В книге Крейга Лармана «Применение UML и шаблонов проектиро
Каждый из них помогает решить некоторую проблему, возникающ
анализе, и которая возникает практически в любом проекте по раз
Таким образом, GRASP-паттерны - это хорошо документированны
проверенные временем принципы объектно-ориентированного ана
принципиально новое.

Содержание [убрать]

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

Джон Влиссидес.
Применение шаблонов проектирования - Дополнительные штрихи - Вильямс, 2003



Джон Влиссидес.
Применение шаблонов проектирования - Дополнительные штрихи - Вильямс, 2003

Книга (1998) одного з членів *GoF*.

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]

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.

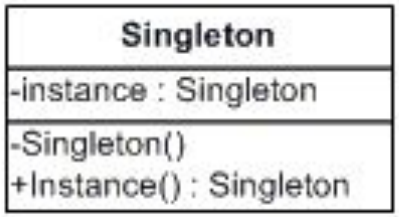
```
class Singleton
{
  // Fields
  private static Singleton instance;
  // Constructor
  protected Singleton() {}
  // Methods
  public static Singleton Instance()
  {
    // Uses "Lazy initialization"
    if( instance == null )
      instance = new Singleton();
    return instance;
  }
}
```

Definition

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

Frequency of use: 1 2 3 4 5 high

UML class diagram

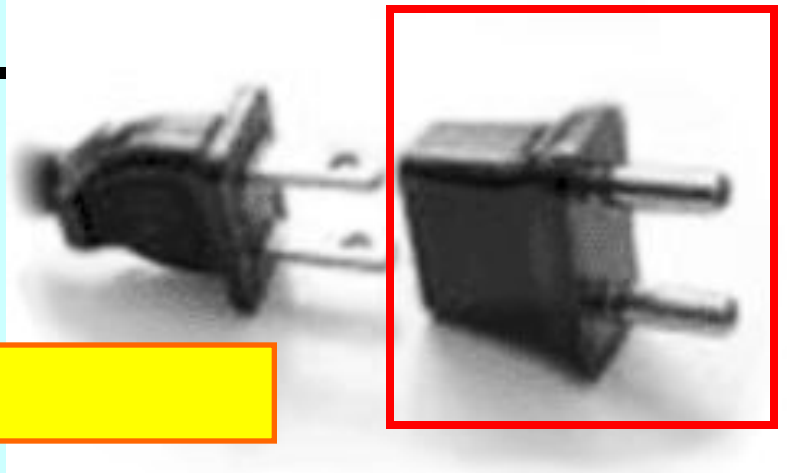


Singleton (data & object factory™, dofactory.com)

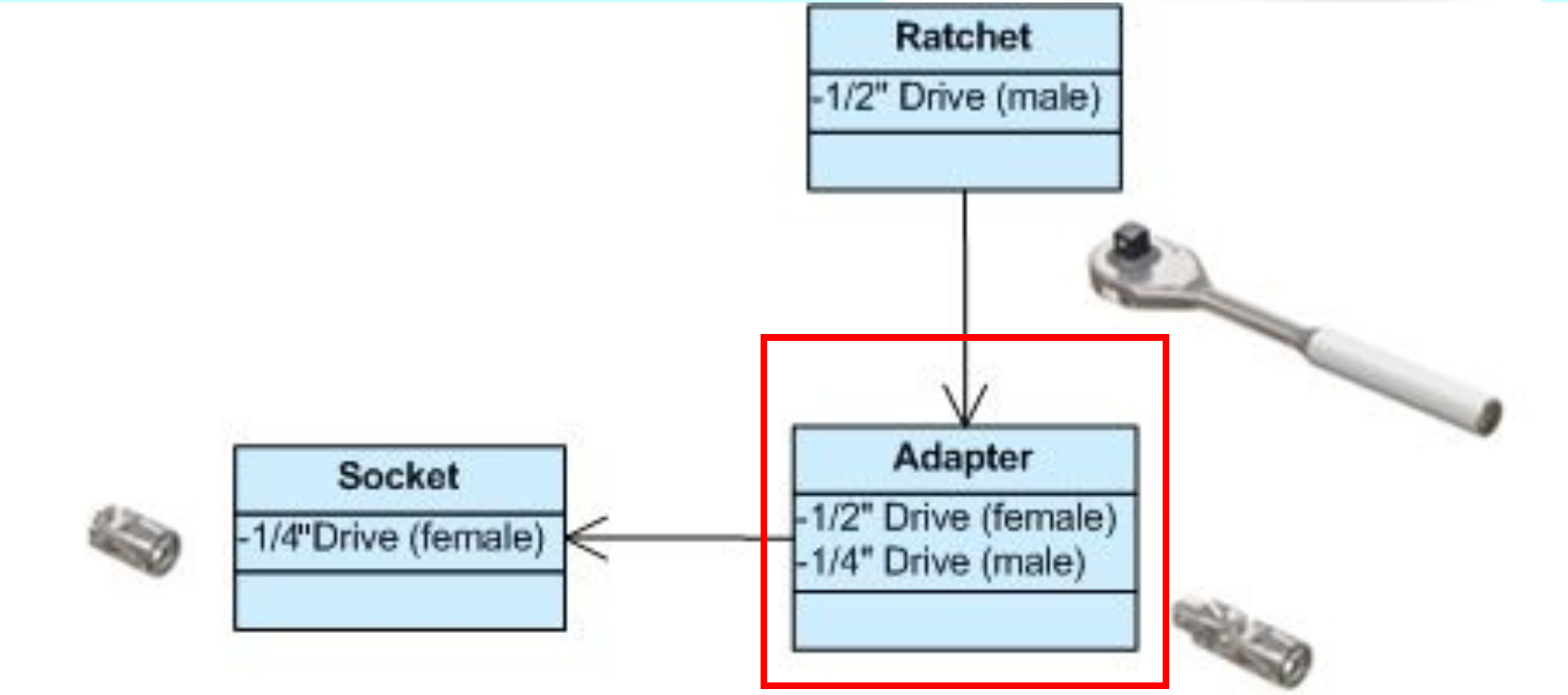
```
using System;
                    // "Singleton"
class Singleton
{
  // Fields
  private static Singleton instance;
  // Constructor
  protected Singleton() {}
  // Methods
  public static Singleton Instance()
  {
    // Uses "Lazy initialization"
    if( instance == null )
      instance = new Singleton();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



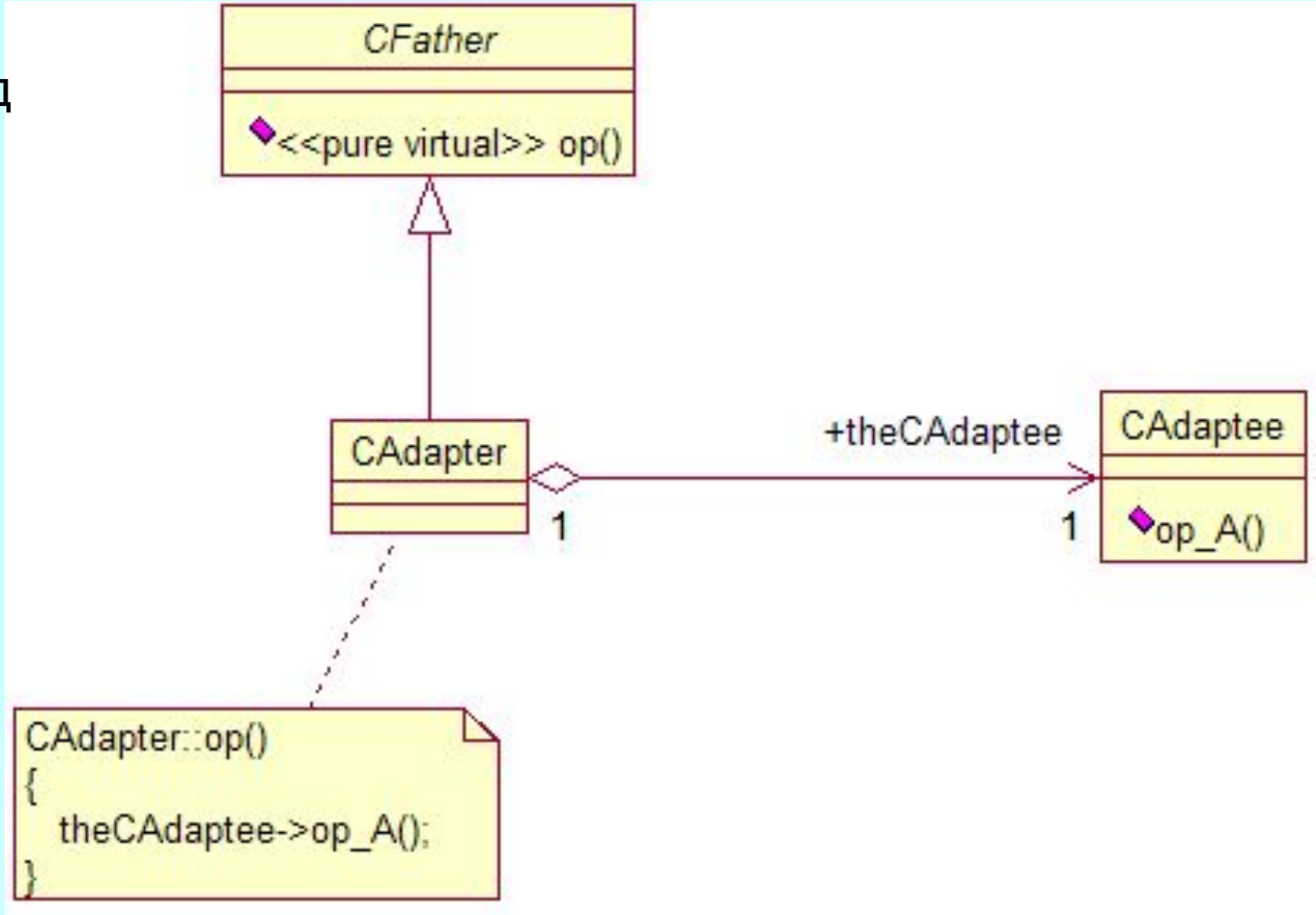
http://sourcemaking.com/design_patterns



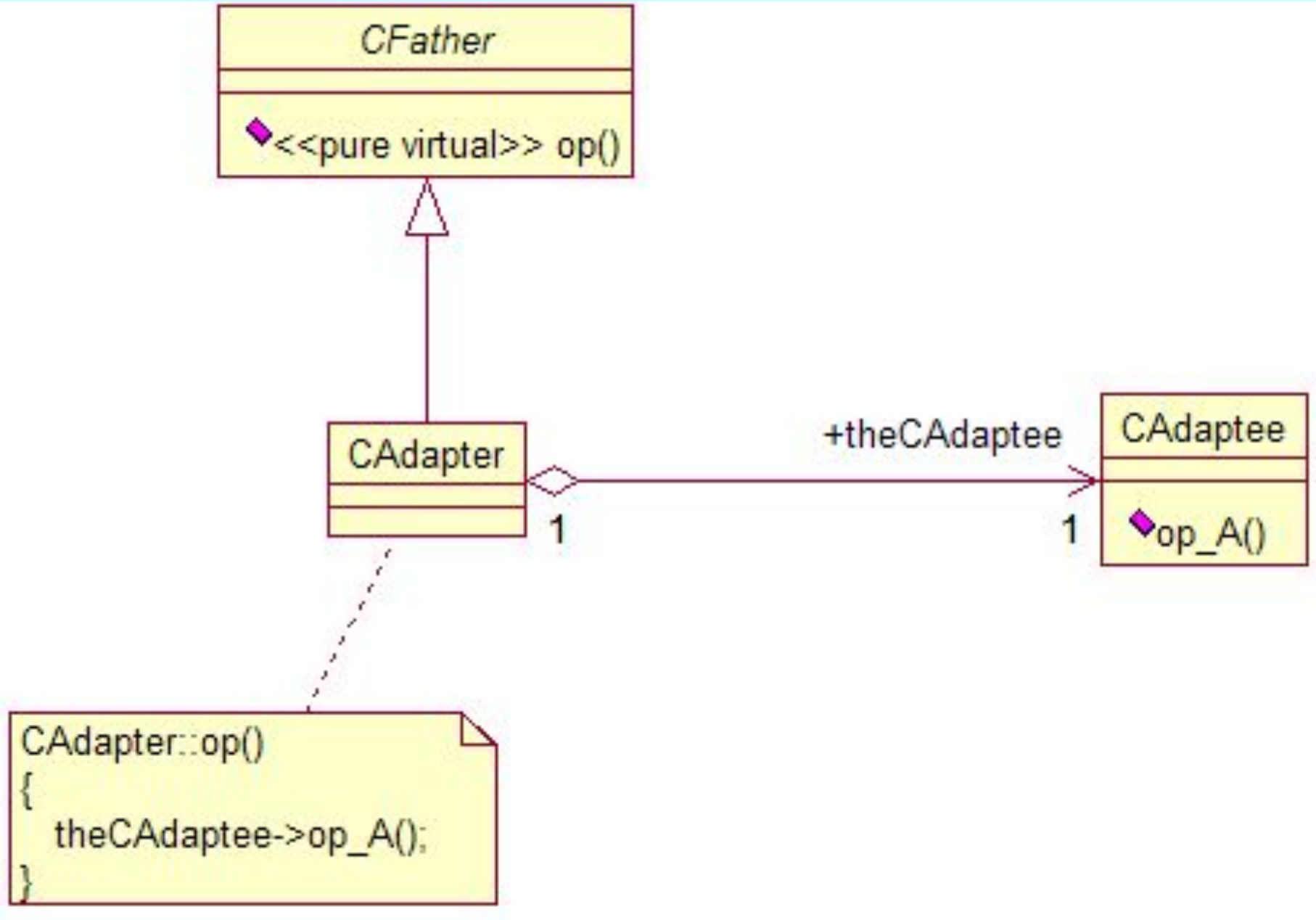
Adapter

- 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***.

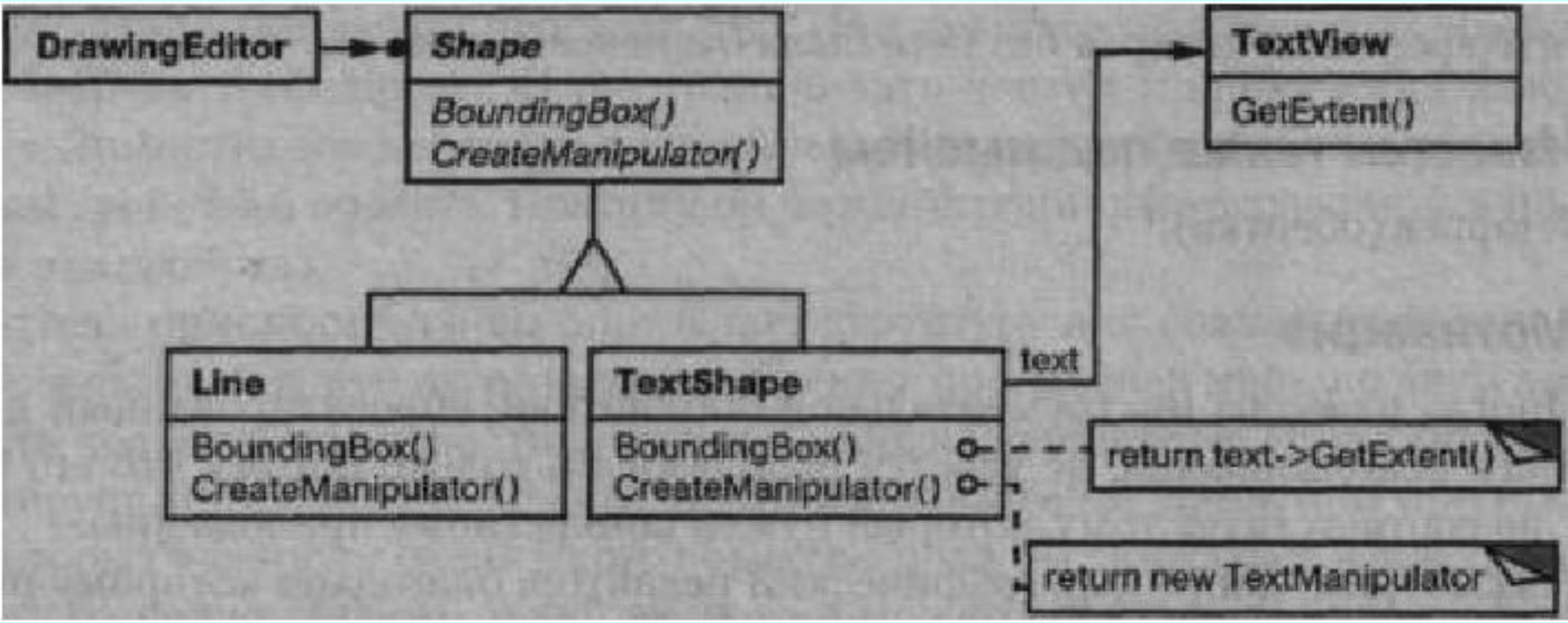
- Відомий також під іменем Wrapper (обгортка).



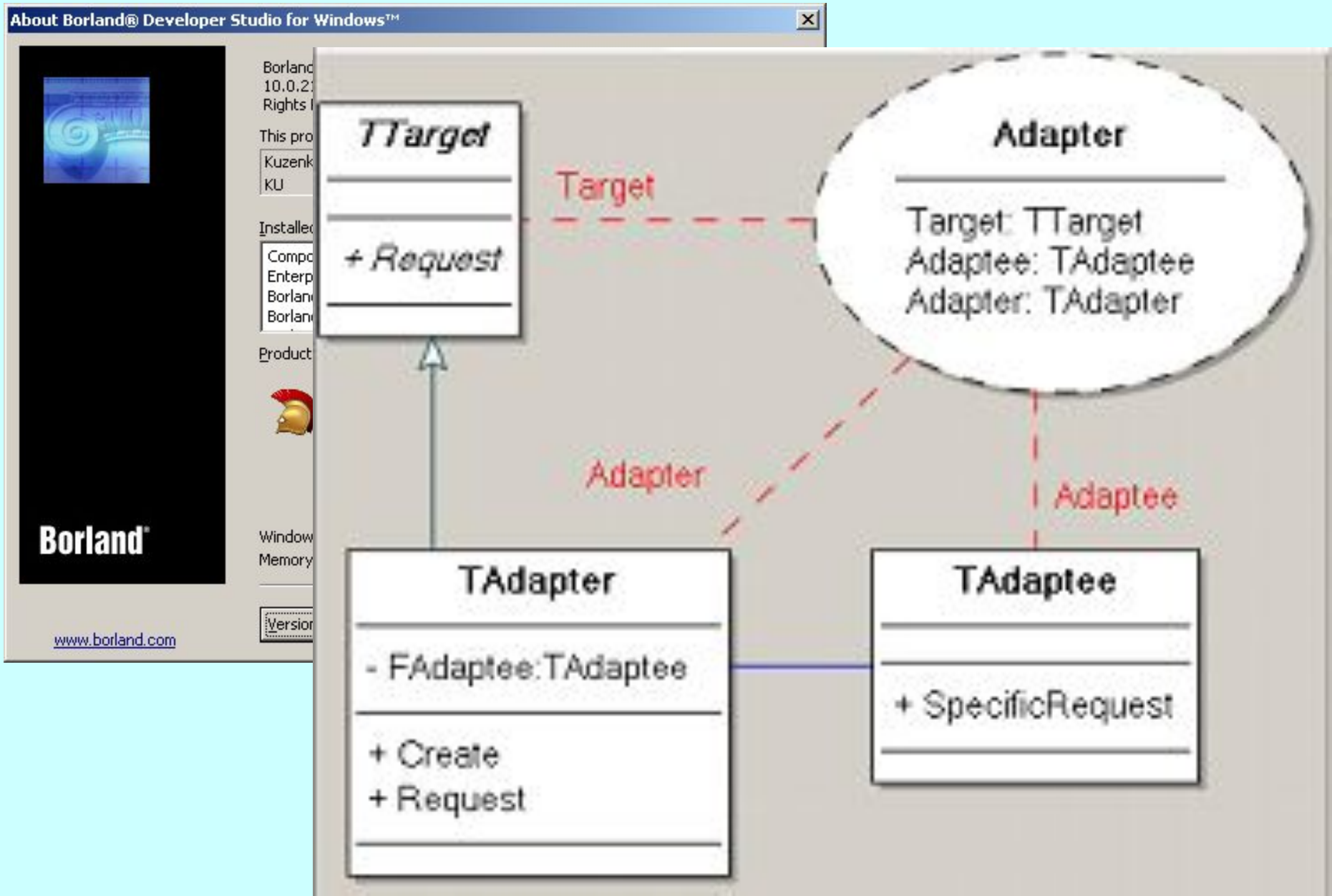
Adapter


CFather
<<pure virtual>> op()
CAdapter
+theCAdaptee
CAdaptee
op_A()
1
1
CAdapter::op()
{
theCAdaptee->op_A();
}

Adapter (GoF)

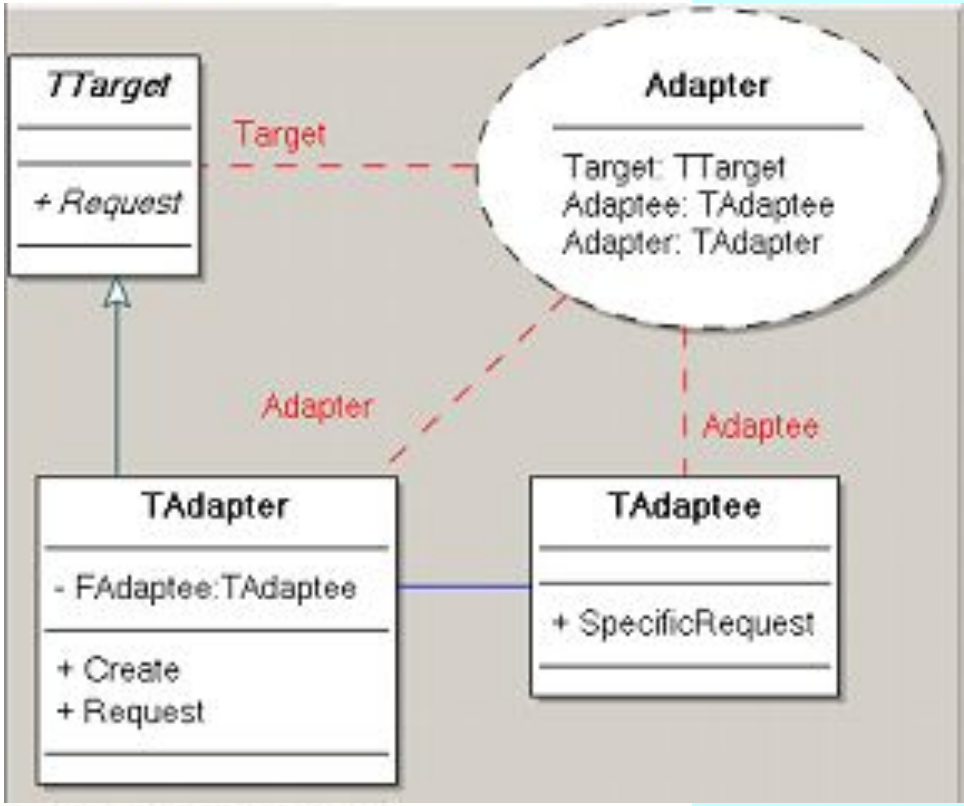

DrawingEditor
Shape
BoundingBox()
CreateManipulator()
TextView
GetExtent()
Line
BoundingBox()
CreateManipulator()
TextShape
BoundingBox()
CreateManipulator()
text
return text->GetExtent()
return new TextManipulator

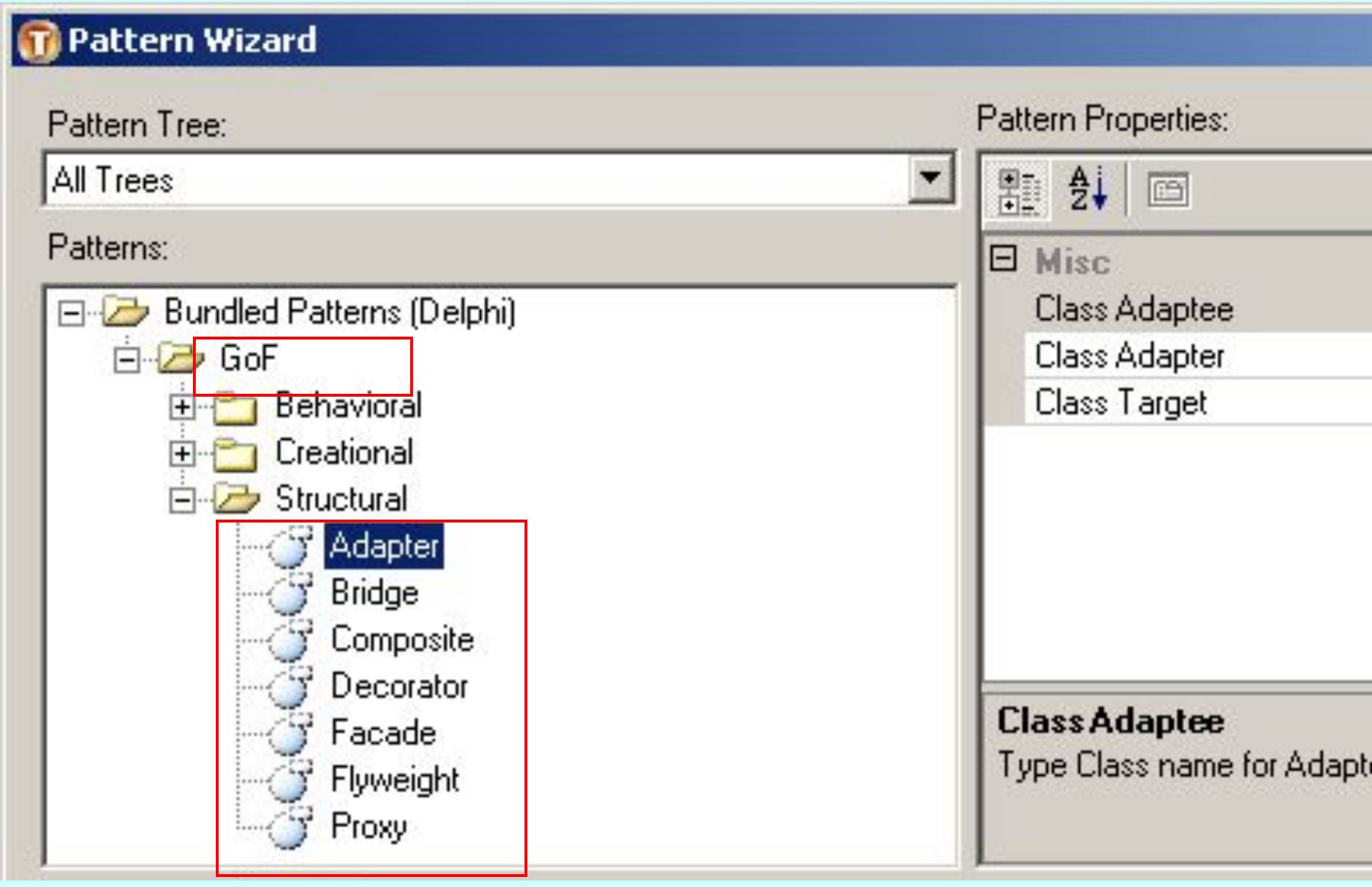
Borland Developer Studio 2006. Adapter



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(AAdaptMe :TAdaptee);
  procedure Request;override;
end;
```




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
ClassAdaptee
Type Class name for Adapt

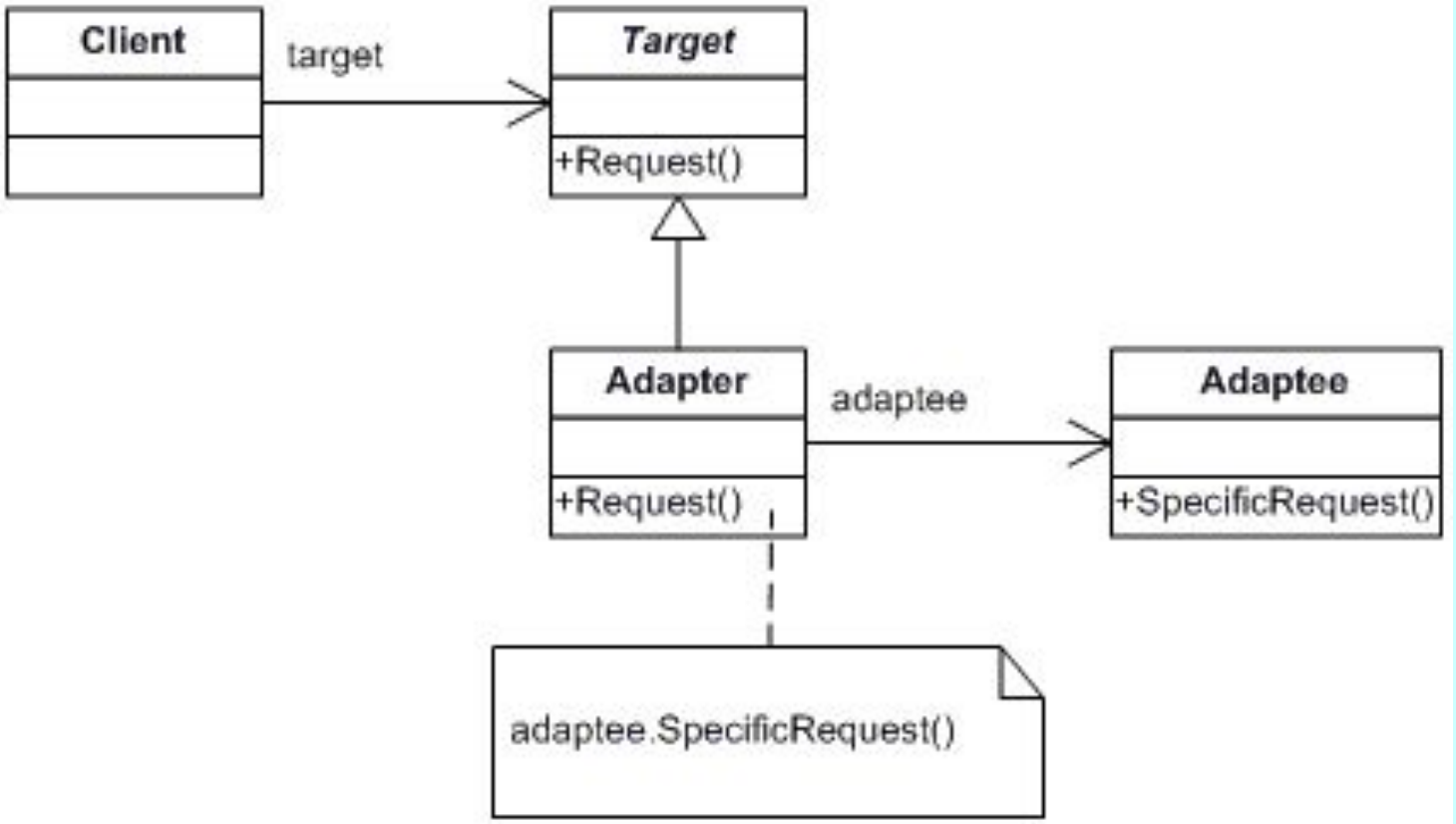
Adapter (data & object factory™, dofactory.com)

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: 1 2 3 4 5 medium high

UML class diagram

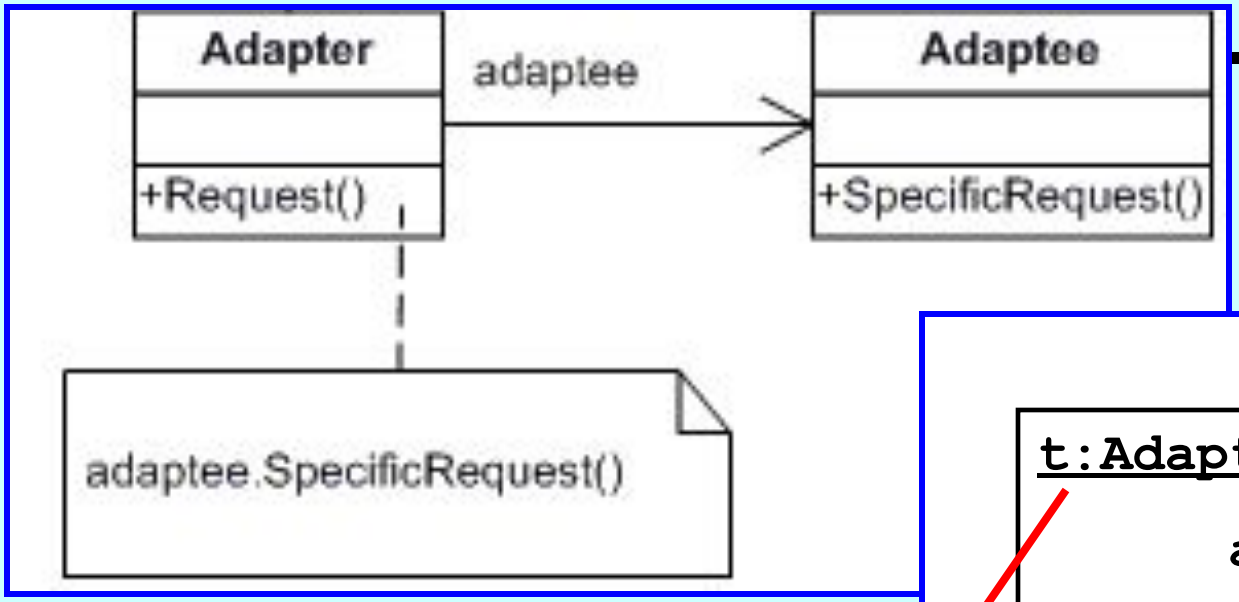


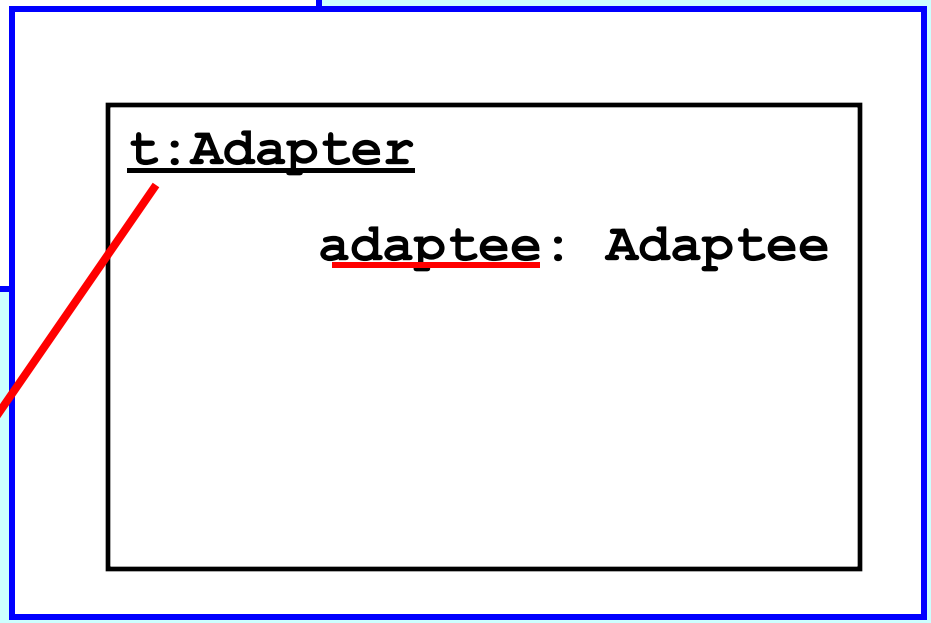
Adapter (data & object factory™, dofactory.com)

```
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();
  }
}
```

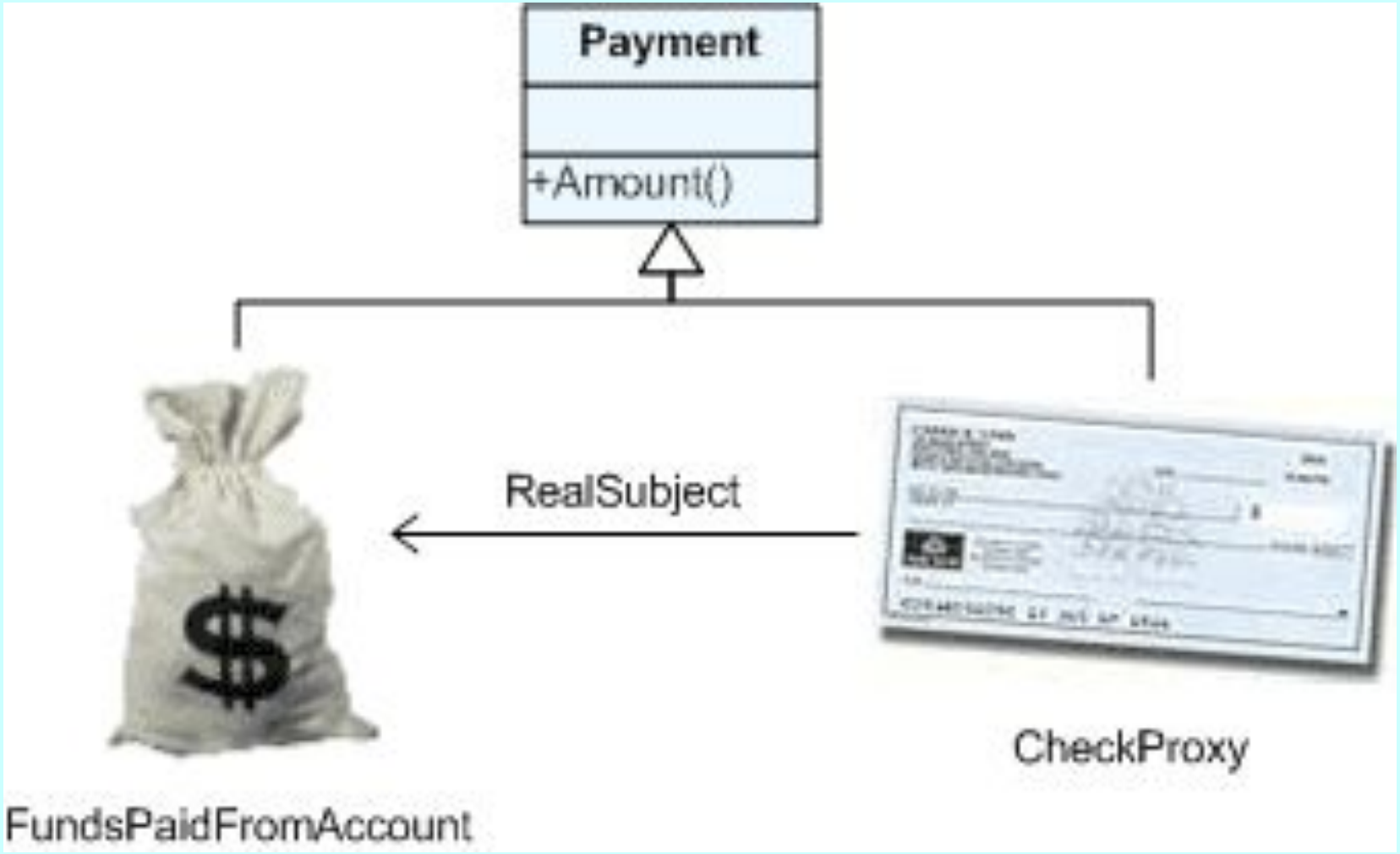
Adapter






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

Proxy



http://sourcemaking.com/design_patterns

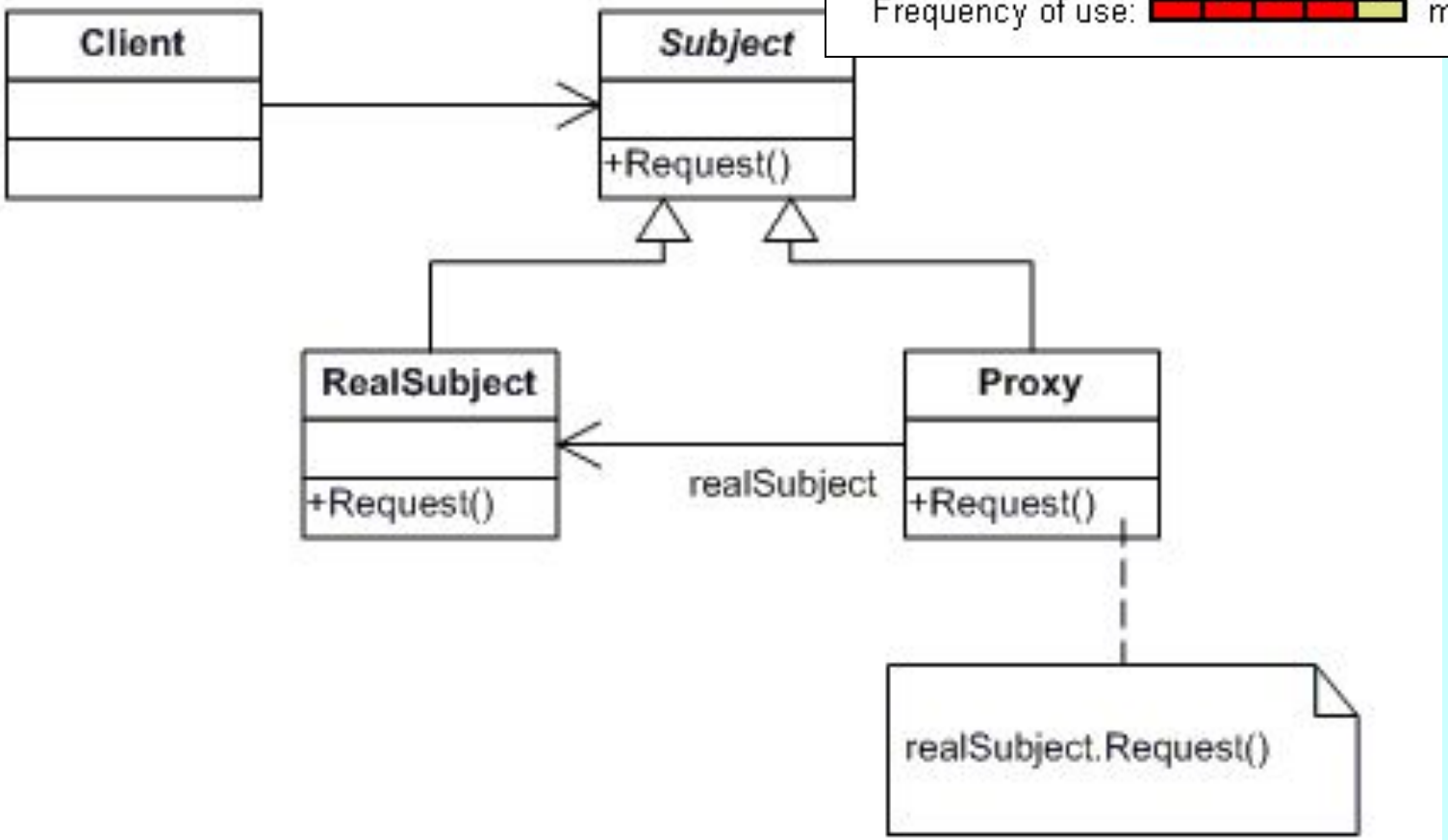
Proxy (data & object factory™, dofactory.com)

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: 1 2 3 4 5 medium high



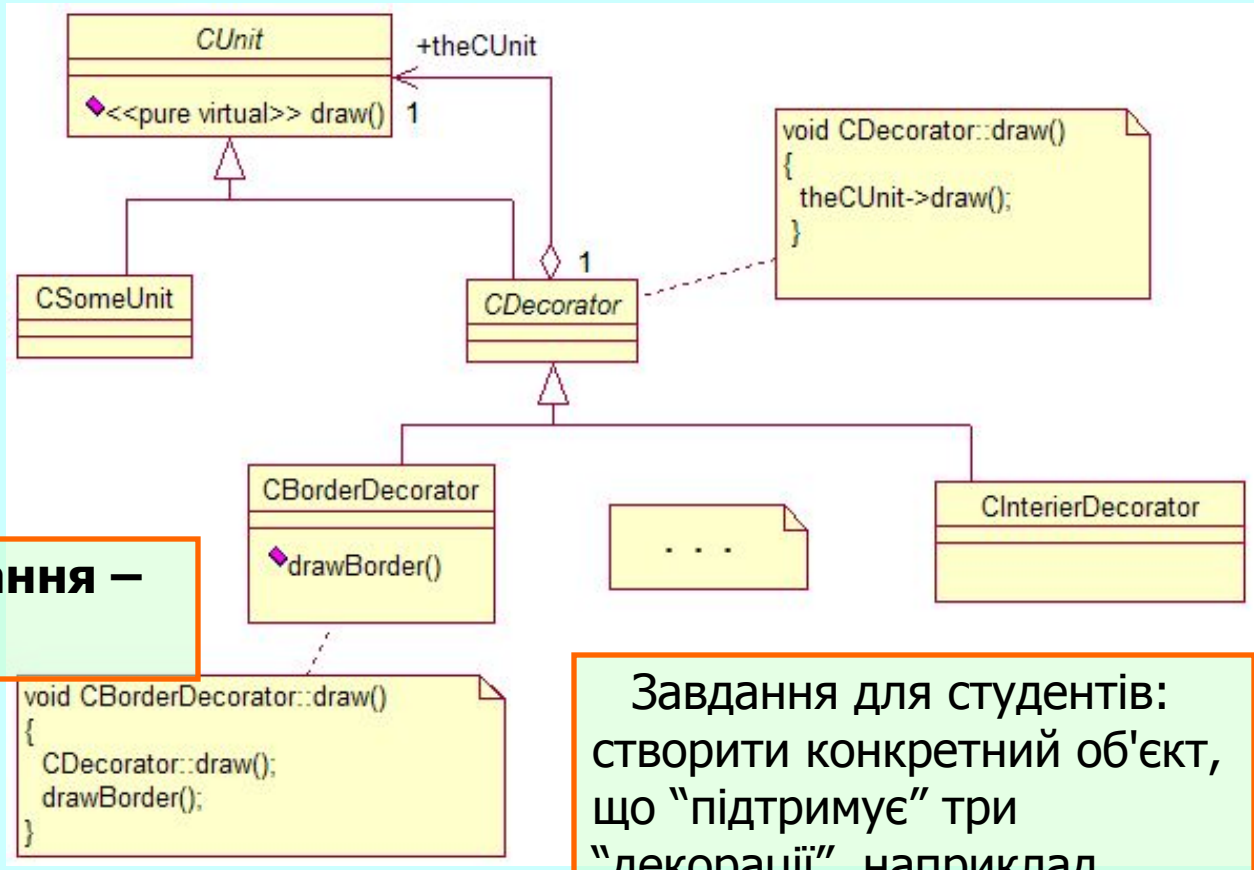
Proxy (data & object factory™, dofactory.com)

```
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 функціональності.



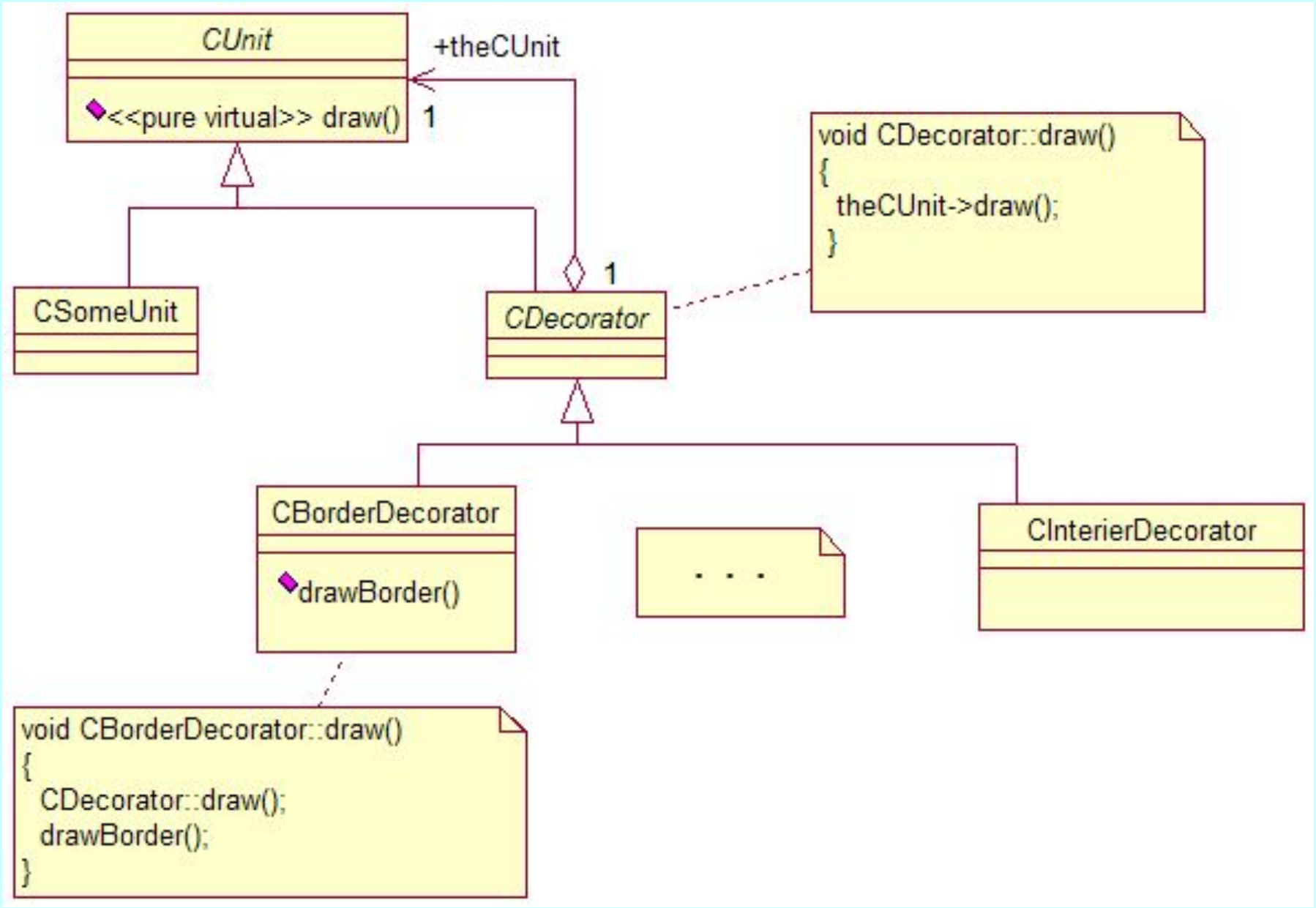
Теза: замість успадкування – композиції

Запитання студентам: як можна отримати розв'язок із використанням успадкування ?

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

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

Decorator


CUnit
+theCUnit
<<pure virtual>> draw()
1
void CDecorator::draw()
{
theCUnit->draw();
}
CSomeUnit
1
CDecorator
CBorderDecorator
drawBorder()
. . .
CInterierDecorator
void CBorderDecorator::draw()
{
CDecorator::draw();
drawBorder();
}

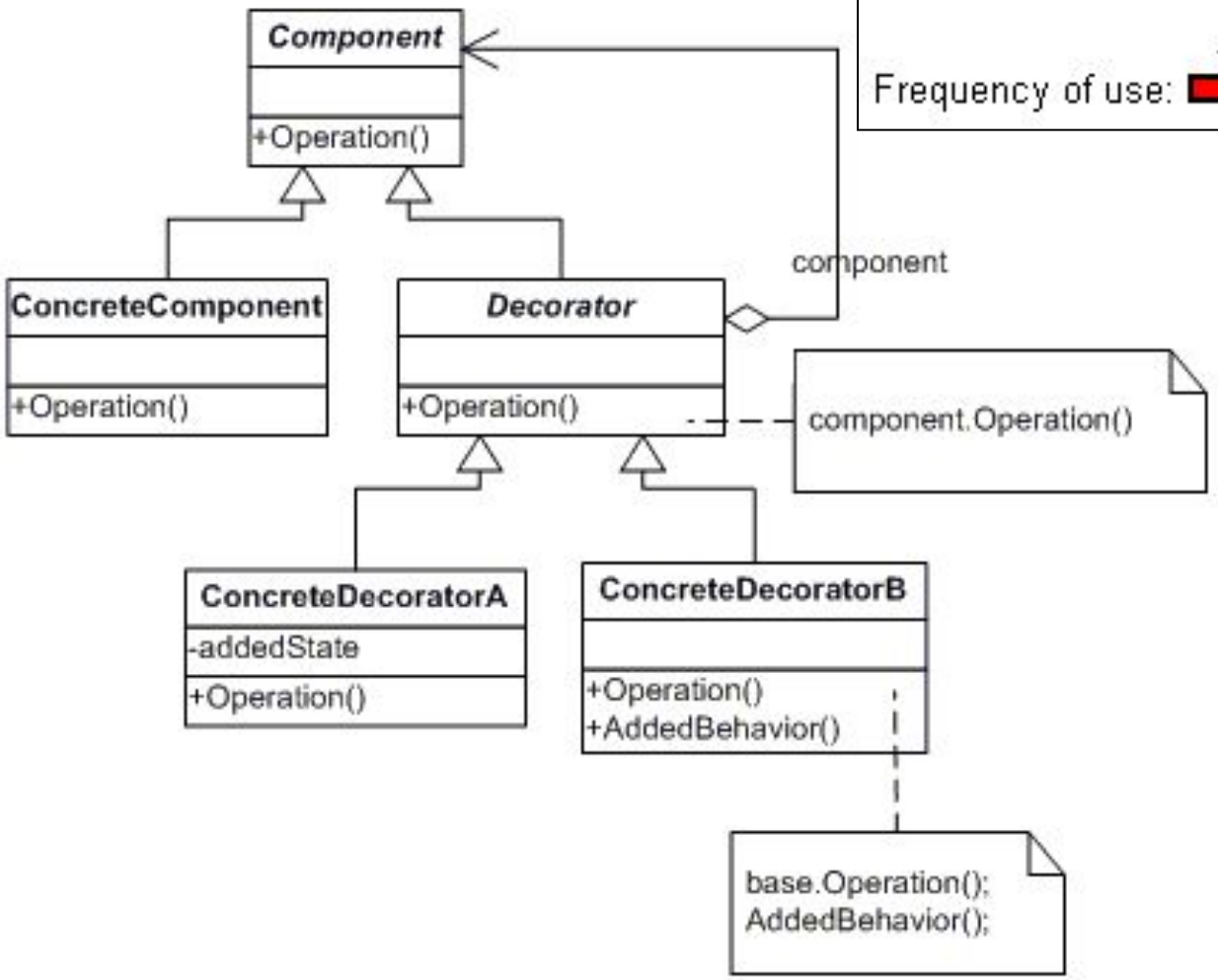
Decorator (data & object factory™, dofactory.com)

Definition

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

Frequency of use: 1 2 3 4 5 medium high

UML class diagram



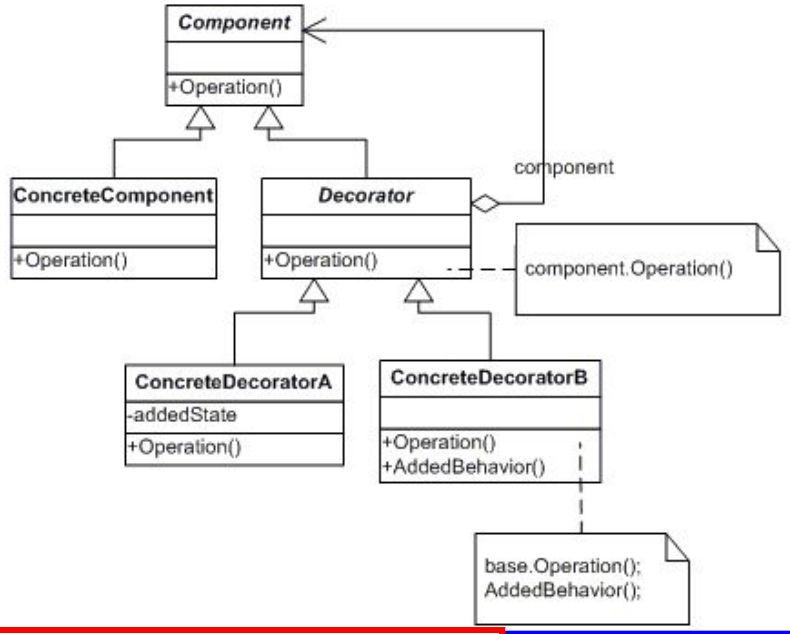
Decorator. Приклад (1/2)
(data & object factory™, dofactory.com)

```
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

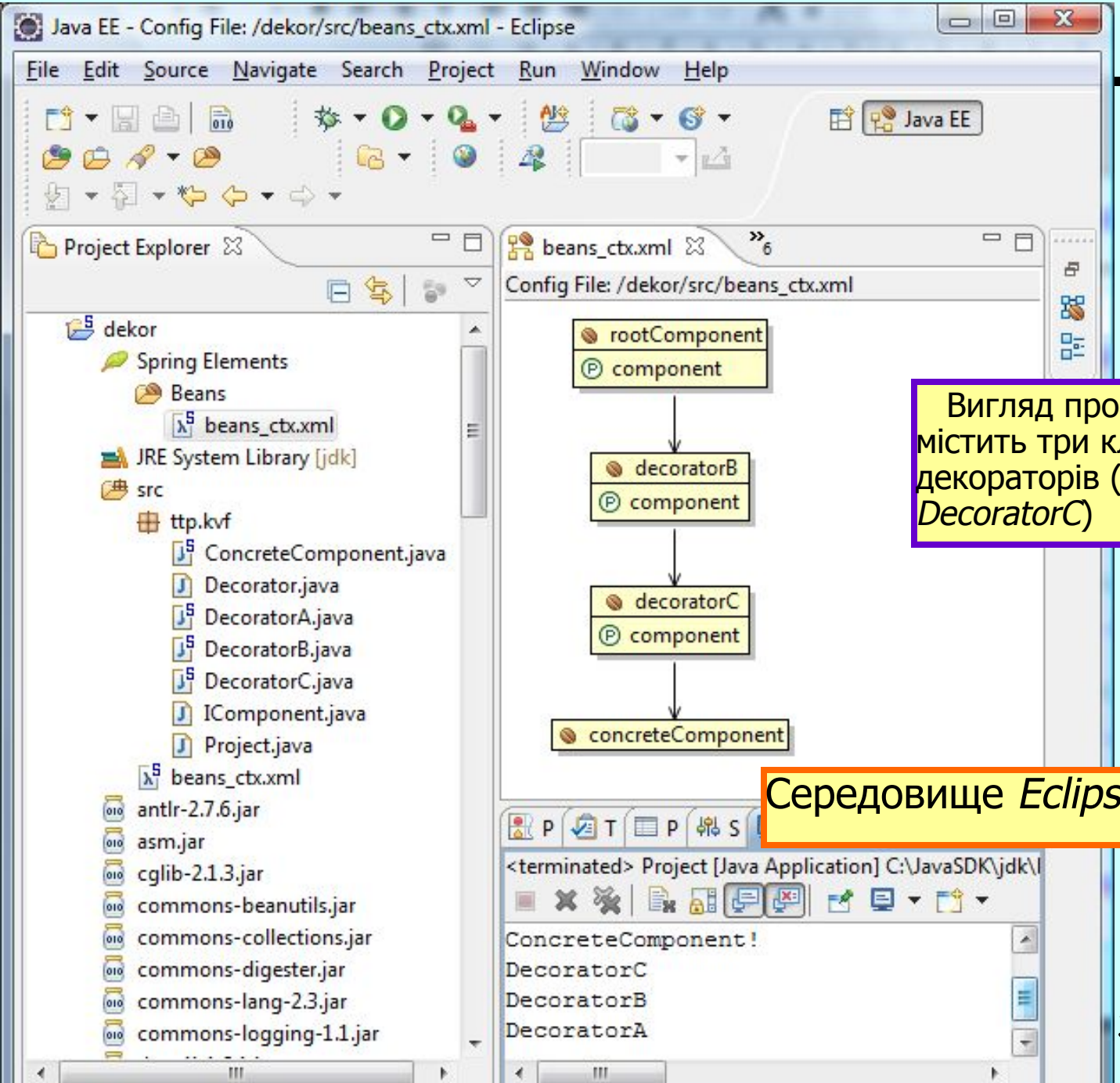
```
// Link decorators
d1.SetComponent( c );
d2.SetComponent( d1 );
d2.Operation();
```

"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
  }
}
```

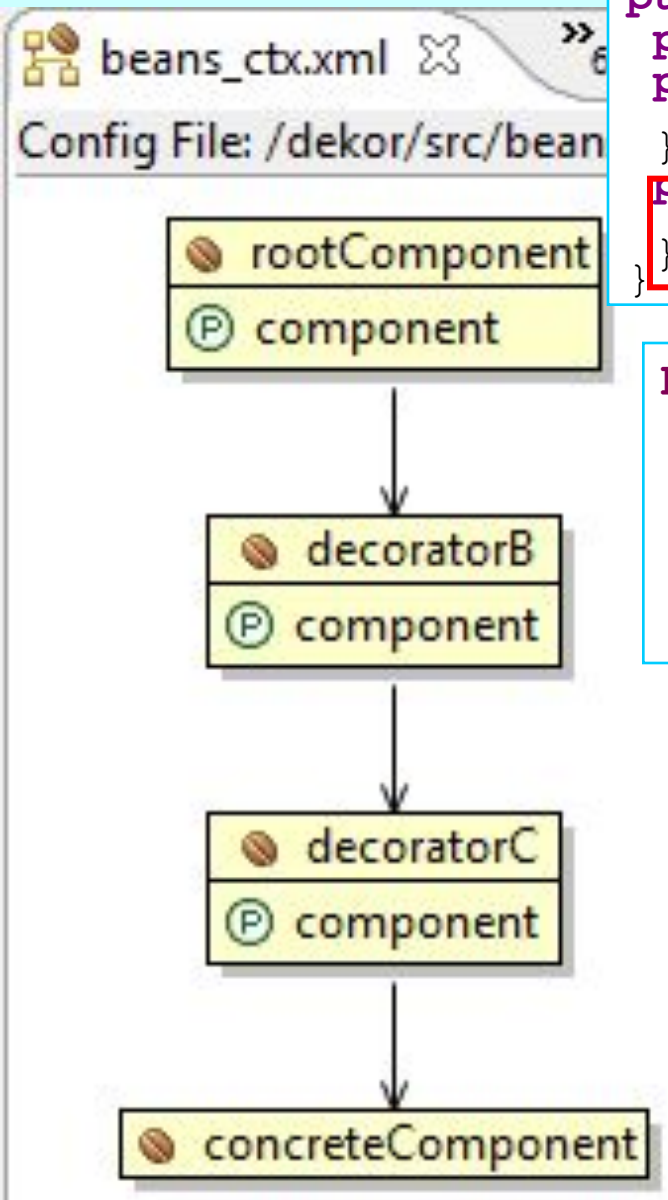
Spring-проект *dekor*



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

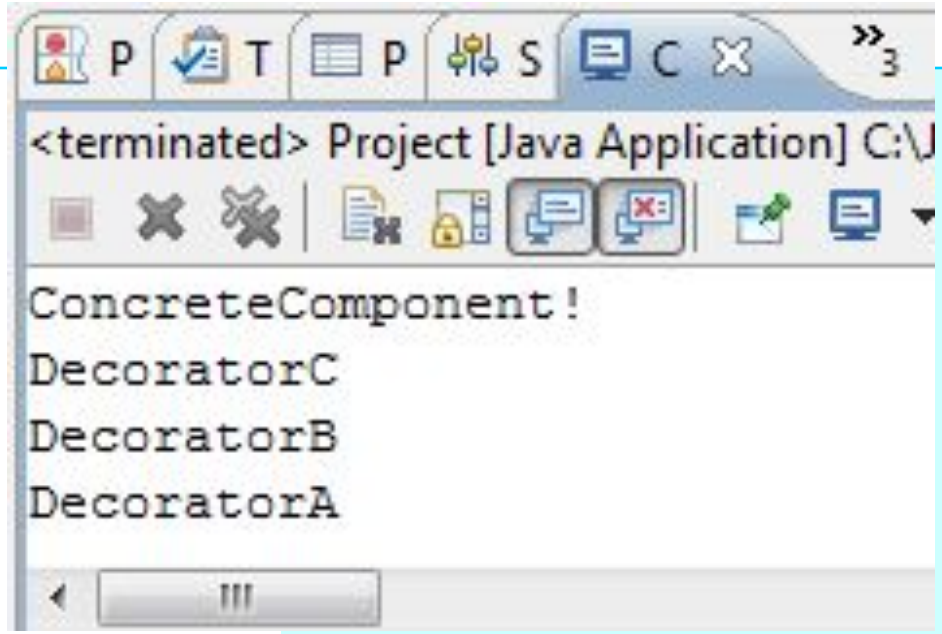
Середовище *Eclipse* +*Spring Plugin*

“Дротяна модель” та виконання проекту



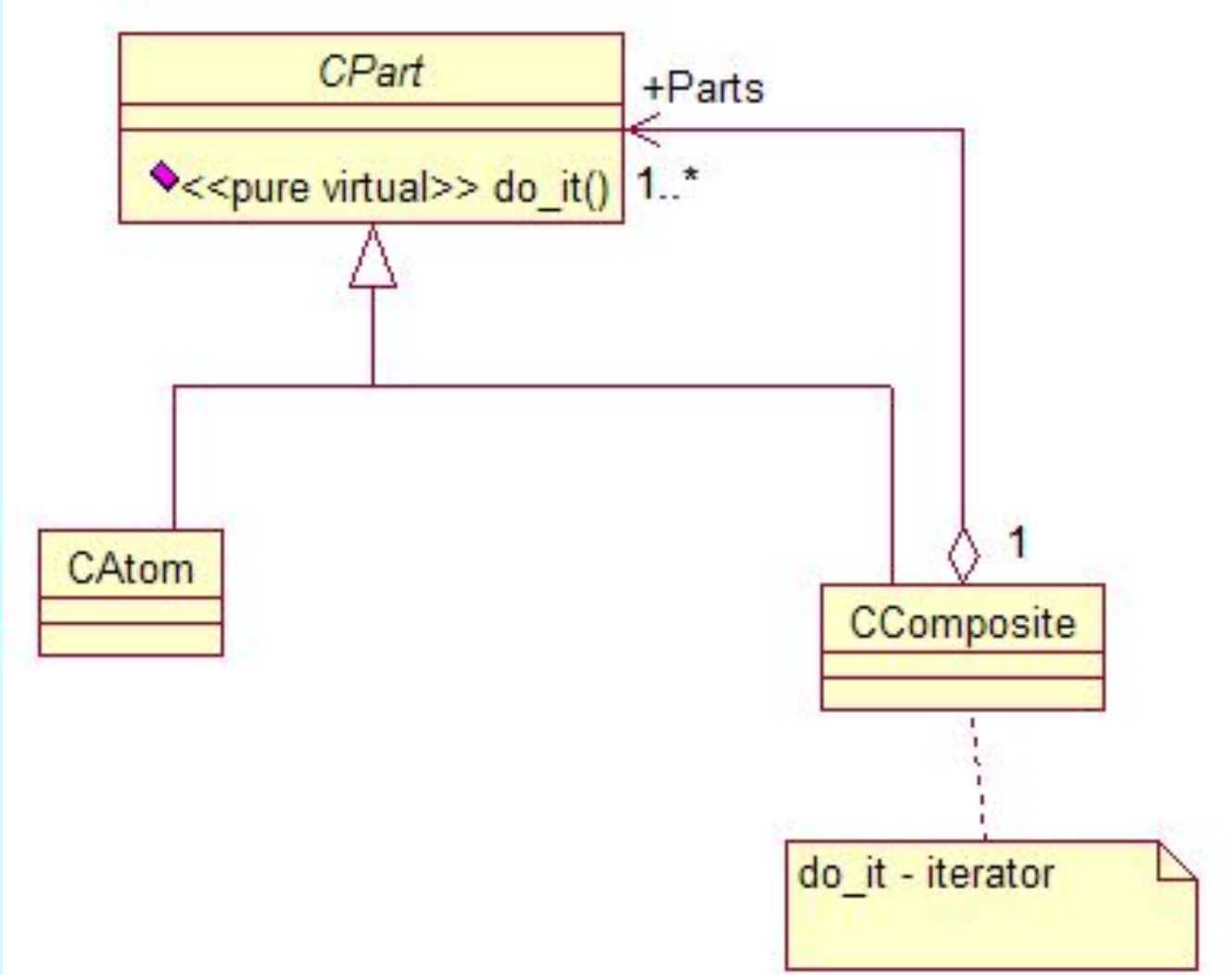

```
public class Decorator implements IComponent{
 private IComponent component;
 public void setComponent(IComponent component){
    this.component = component;
 }
 public void operation(){
    component.operation();
 }
}
```

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



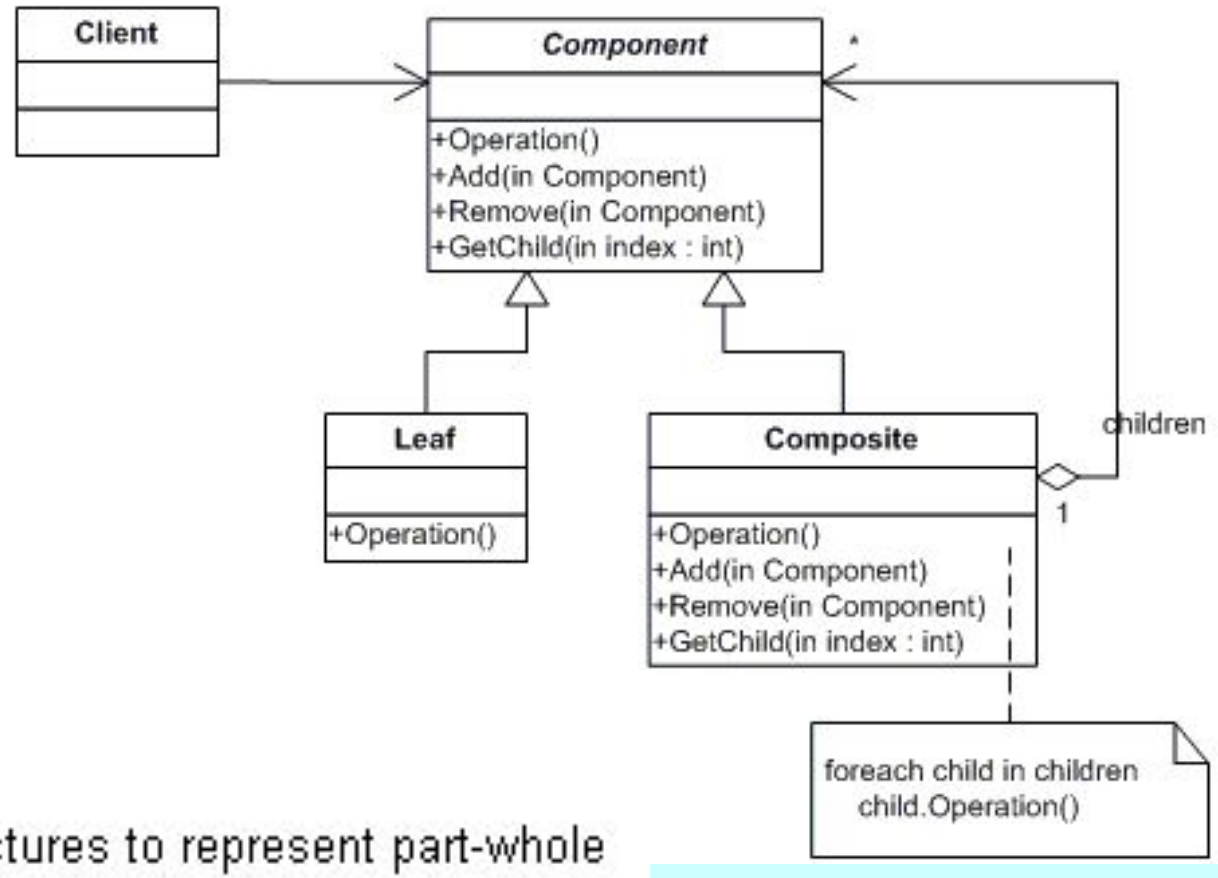
Composite (композитор, компоновник)

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: 1 2 3 4 5 high

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

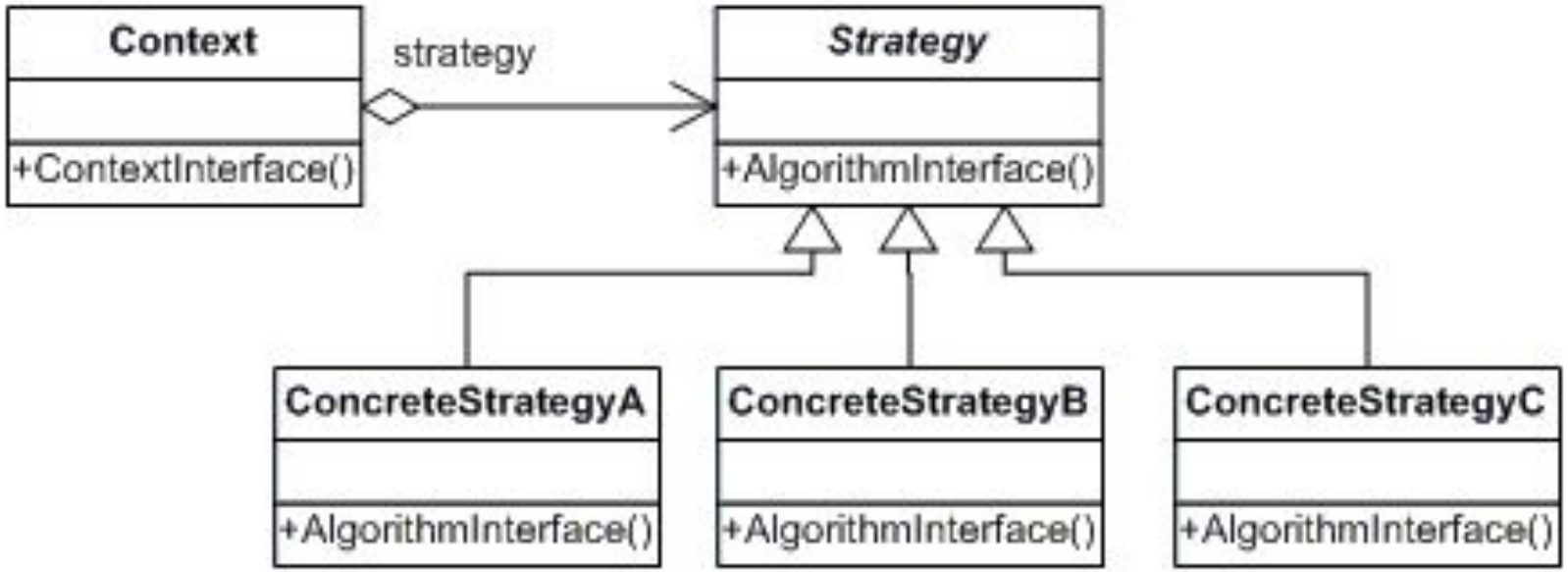
```
using System;
using System.Text;
using System.Collections;
abstract class Component
{  protected string name;
public Component( string name ) // Constructor
  {    this.name = name;
  }
abstract public void Add(Component c);
abstract public void Remove( Component c );
abstract public void Display( 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 nod
    root.Display( 1 );
  }
}
```

Strategy (dofactory.com)

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.

UML class diagram



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: 1 2 3 4 5 medium high

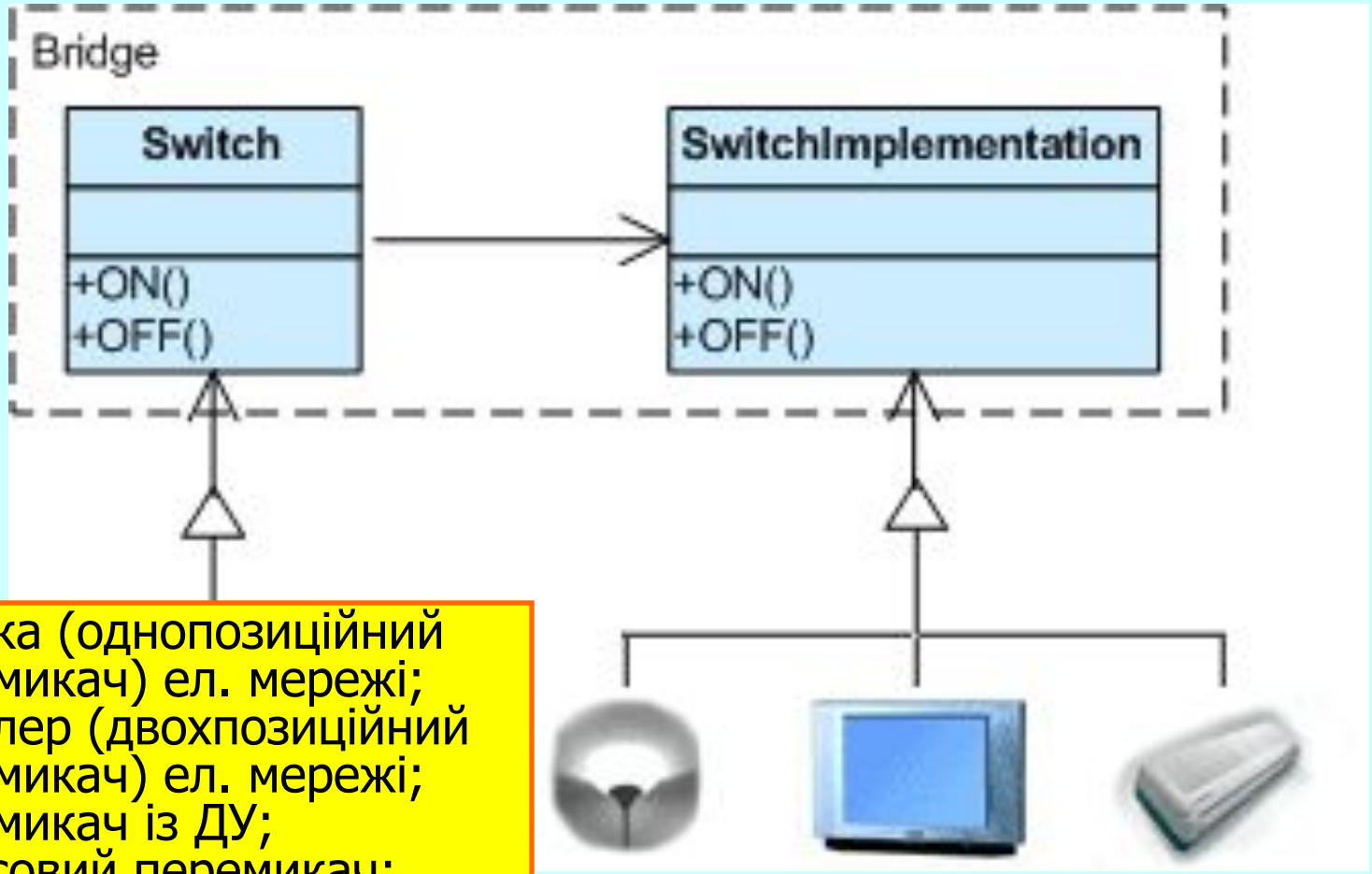
Strategy (dofactory.com)

```
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

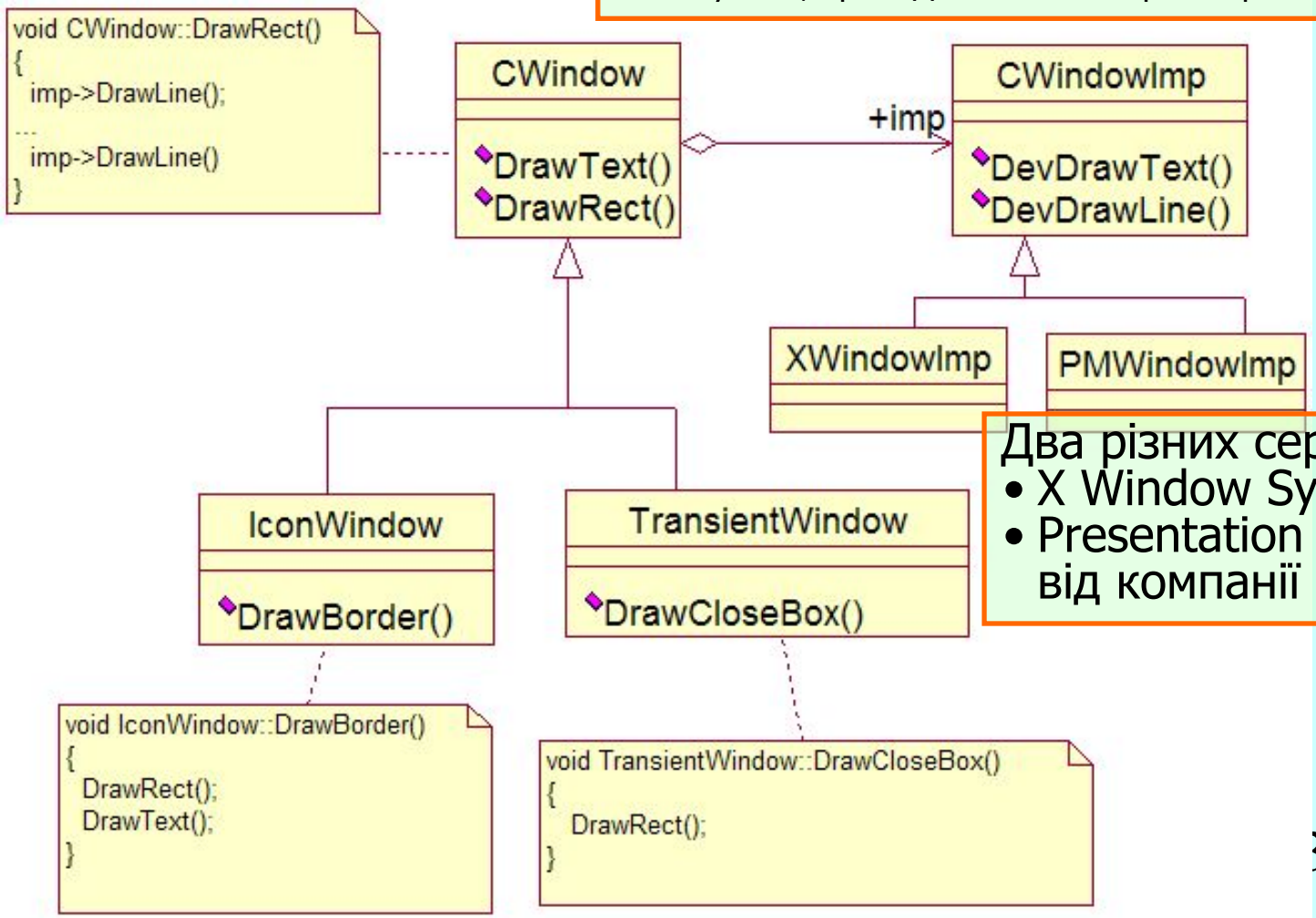


- кнопка (однопозиційний перемикач) ел. мережі;
- тумблер (двохпозиційний перемикач) ел. мережі;
- перемикач із ДУ;
- голосовий перемикач;
- . . .

Bridge

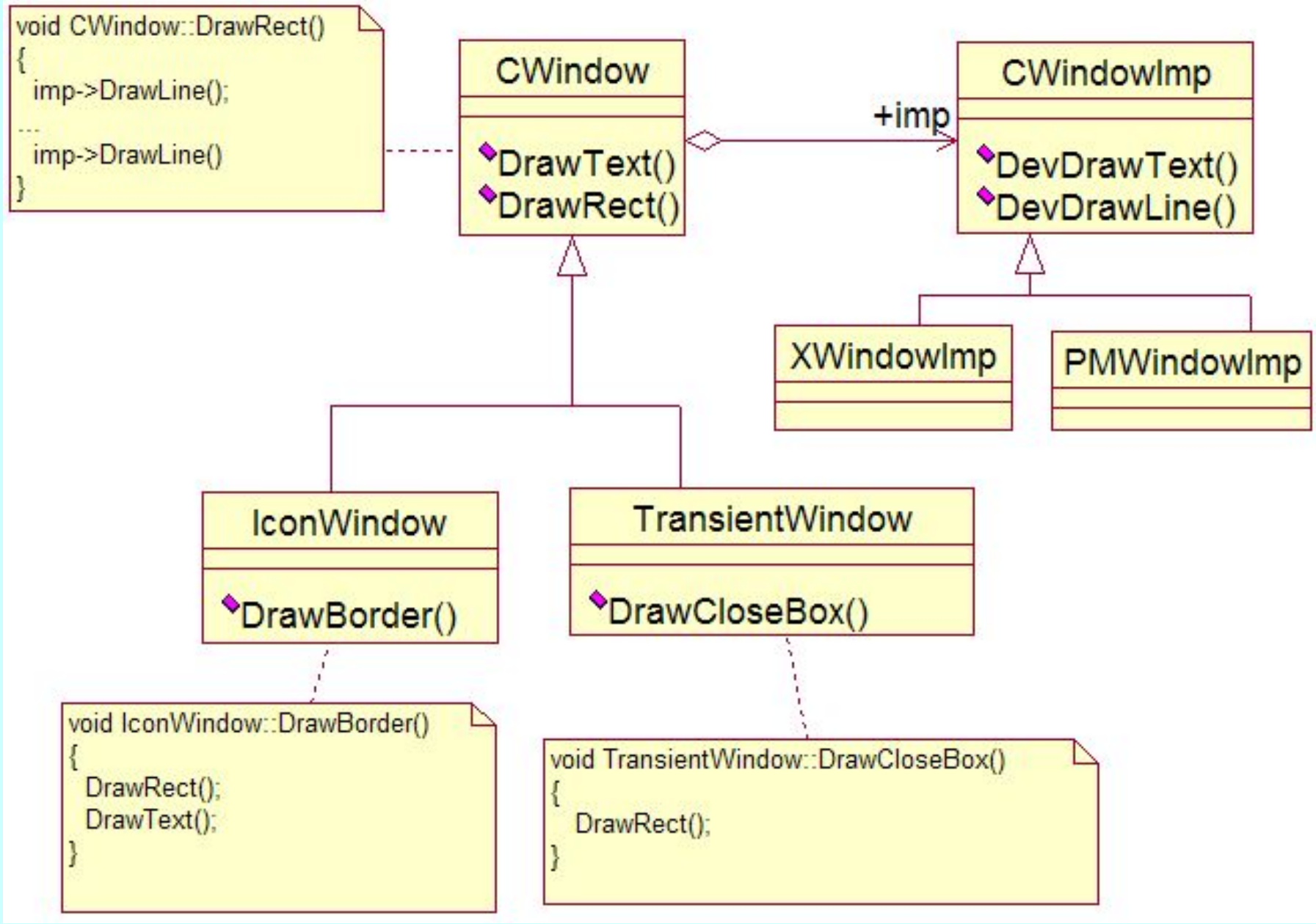
- Відокремлює абстракцію від реалізації, завдяки чому з'являється можливість незалежно змінювати те й інше.
- Відомий також під іменем Handle/Body (описувач/тіло).

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

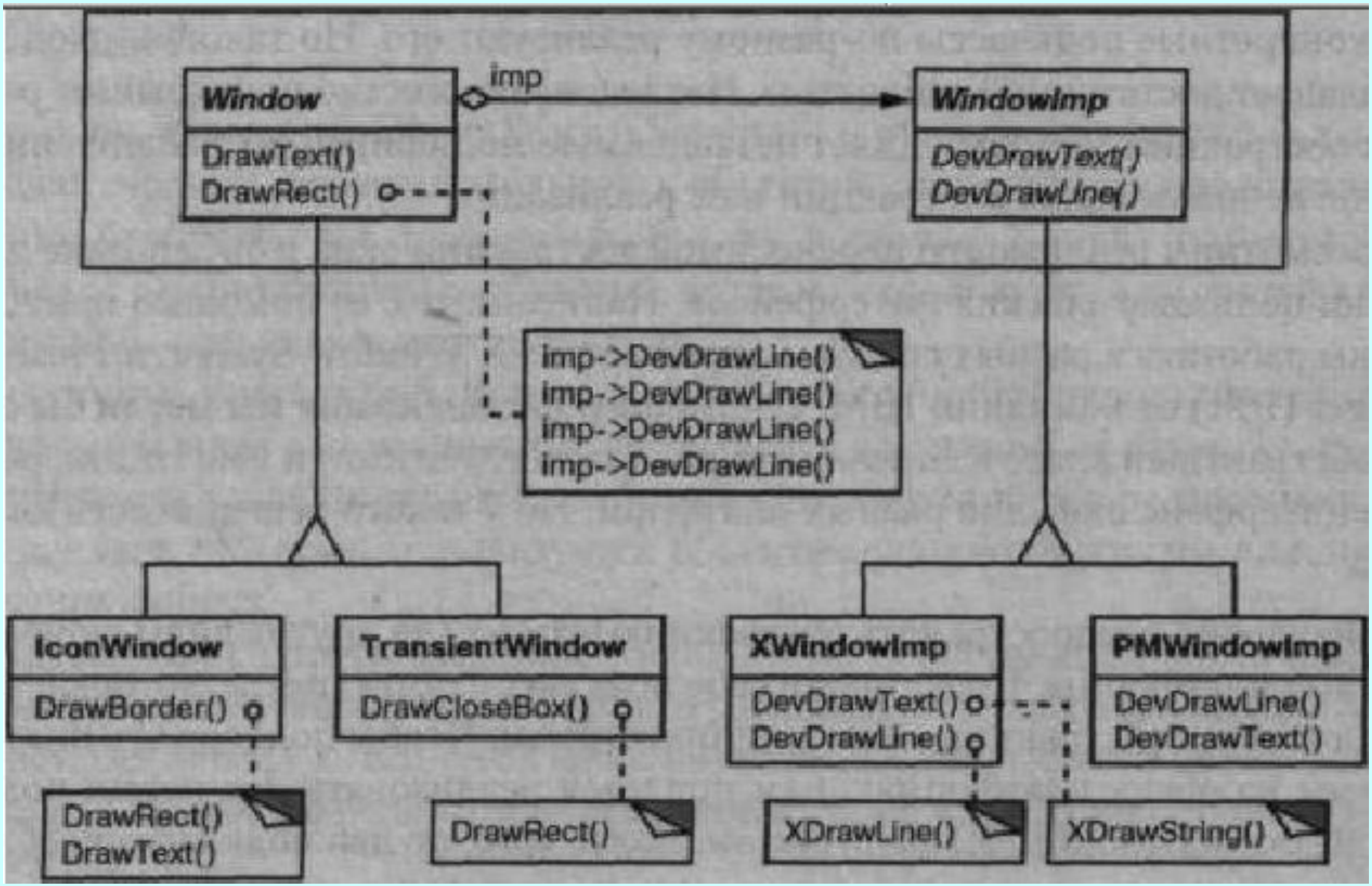


Два різних середовища:
- X Window System;
- Presentation Manager (PM) від компанії IBM.

Bridge

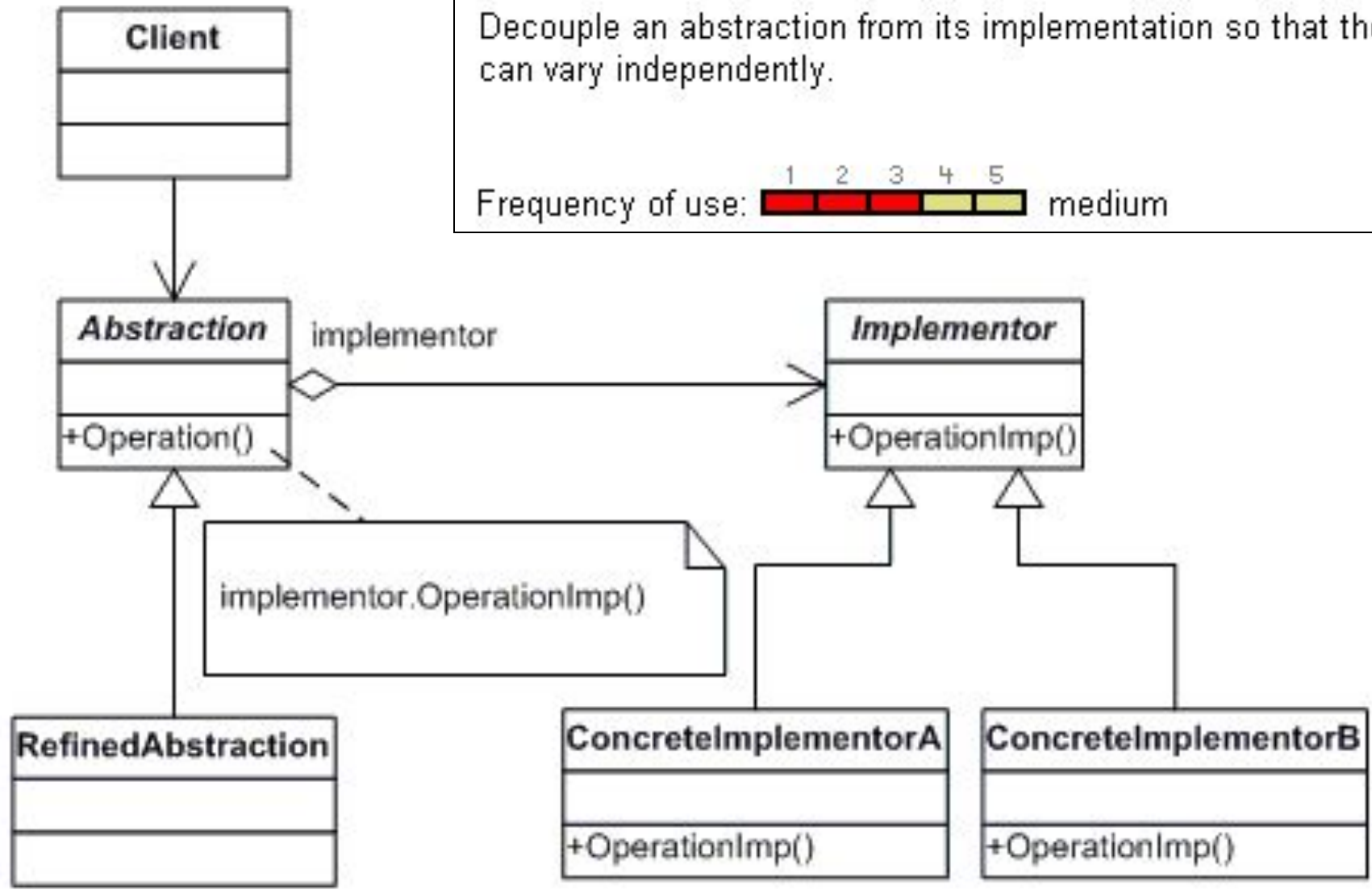

void CWindow::DrawRect()
{
imp->DrawLine();
...
imp->DrawLine()
}
CWindow
DrawText()
DrawRect()
+imp
CWindowImp
DevDrawText()
DevDrawLine()
XWindowImp
PMWindowImp
IconWindow
DrawBorder()
TransientWindow
DrawCloseBox()
void IconWindow::DrawBorder()
{
DrawRect();
DrawText();
}
void TransientWindow::DrawCloseBox()
{
DrawRect();
}

Bridge (GoF)


Window
DrawText()
DrawRect()
imp
WindowImp
DevDrawText()
DevDrawLine()
imp->DevDrawLine()
imp->DevDrawLine()
imp->DevDrawLine()
imp->DevDrawLine()
IconWindow
DrawBorder()
TransientWindow
DrawCloseBox()
XWindowImp
DevDrawText()
DevDrawLine()
PMWindowImp
DevDrawLine()
DevDrawText()
DrawRect()
DrawText()
DrawRect()
XDrawLine()
XDrawString()

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

UML class diagram

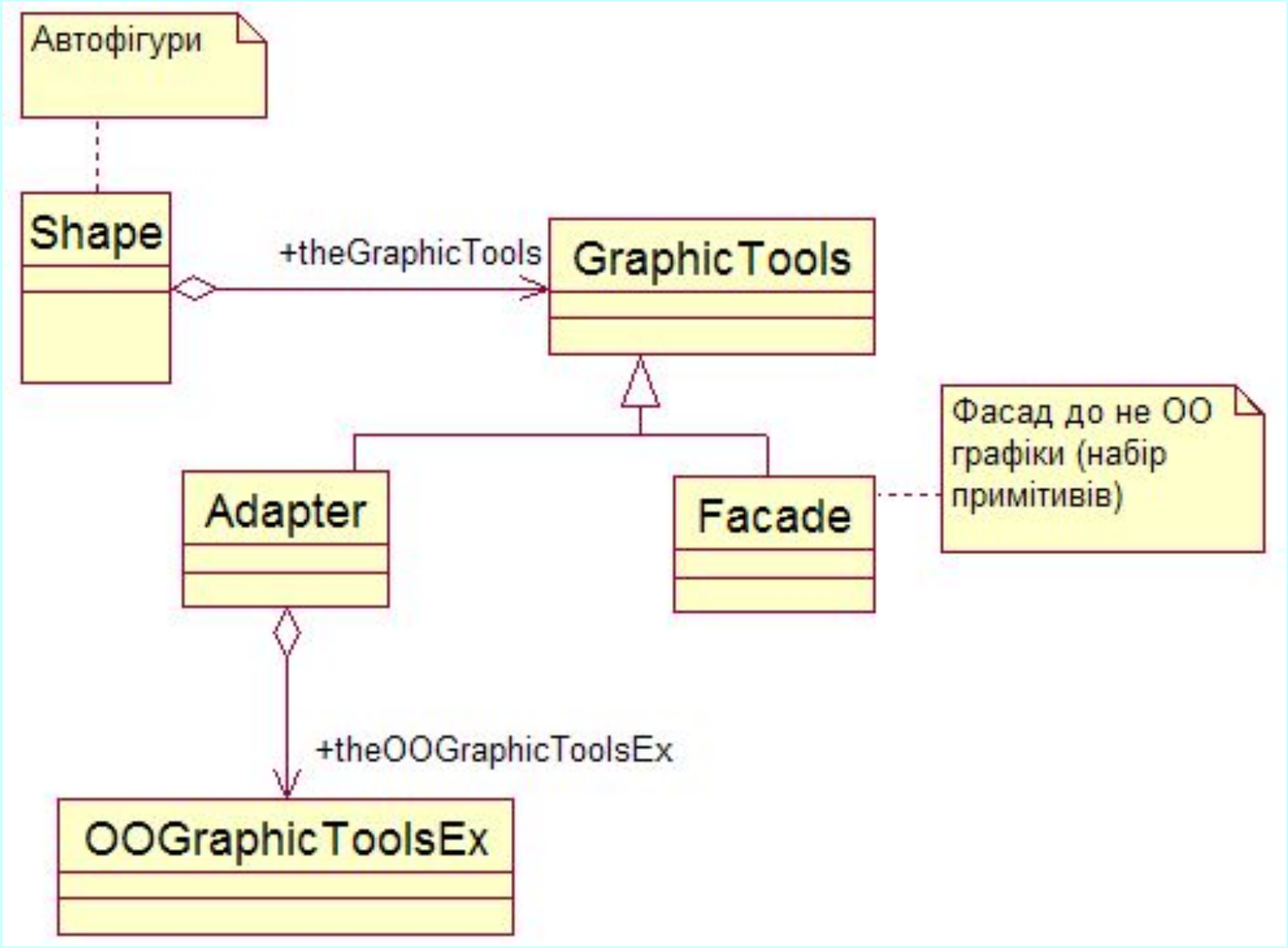


Definition

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.Александер).



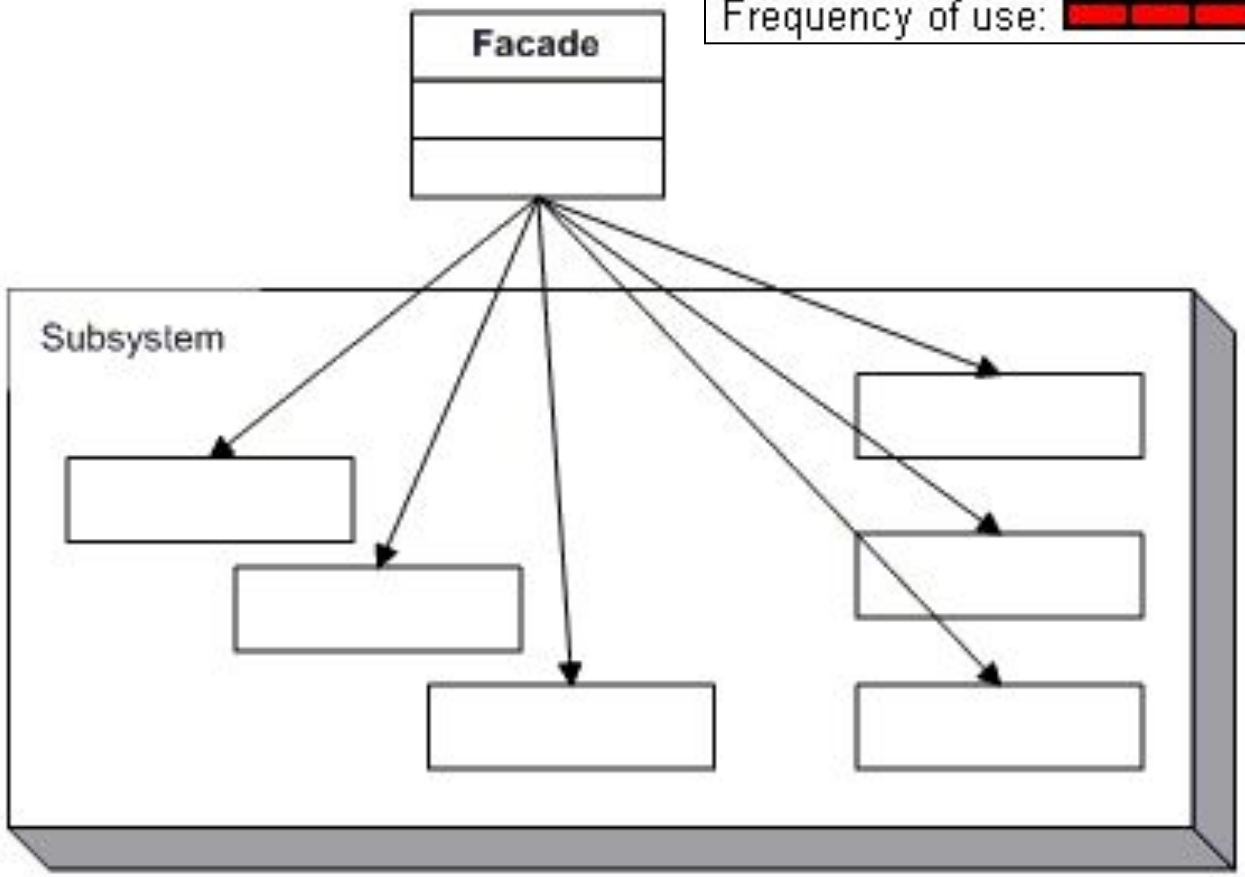
Façade (data & object factory™, dofactory.com)

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)

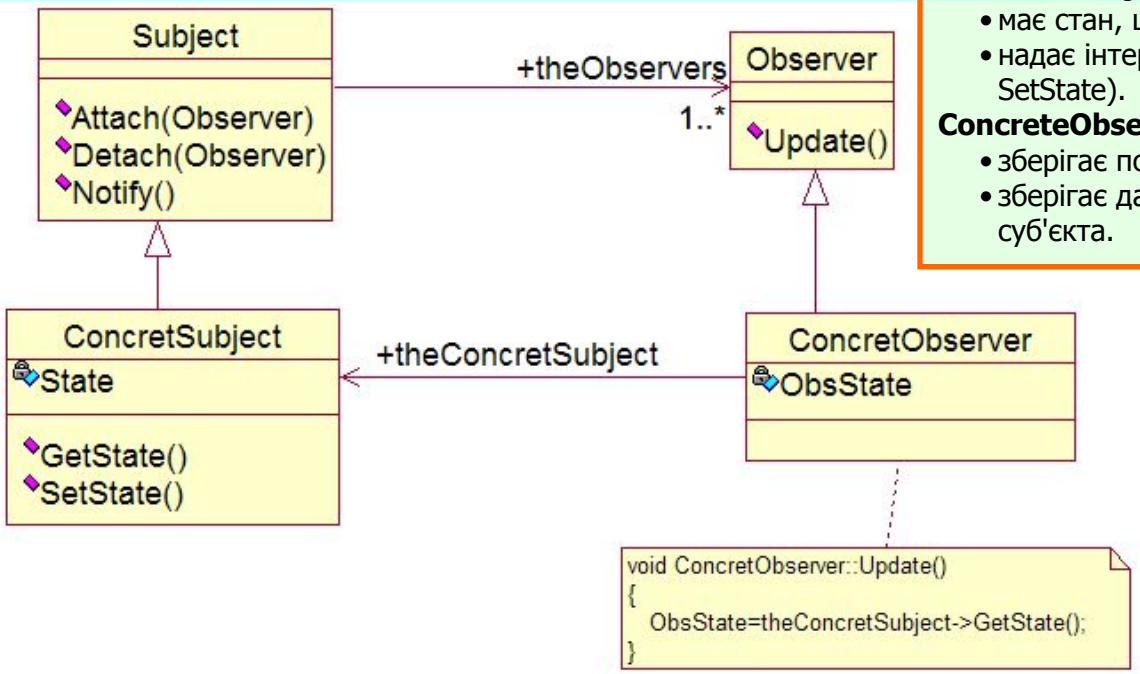
- Визначає між об'єктами відношення типу один (видавець) - до - багатьох (передплатників), так що при зміні стану одного об'єкта (видавця) всі підлеглі (передплатники) одержують повідомлення й автоматично оновлюють дані.
- Відомий також під іменами *Dependents* (підлеглі), *Publish-Subscribe* (видавець-передплатник).

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.

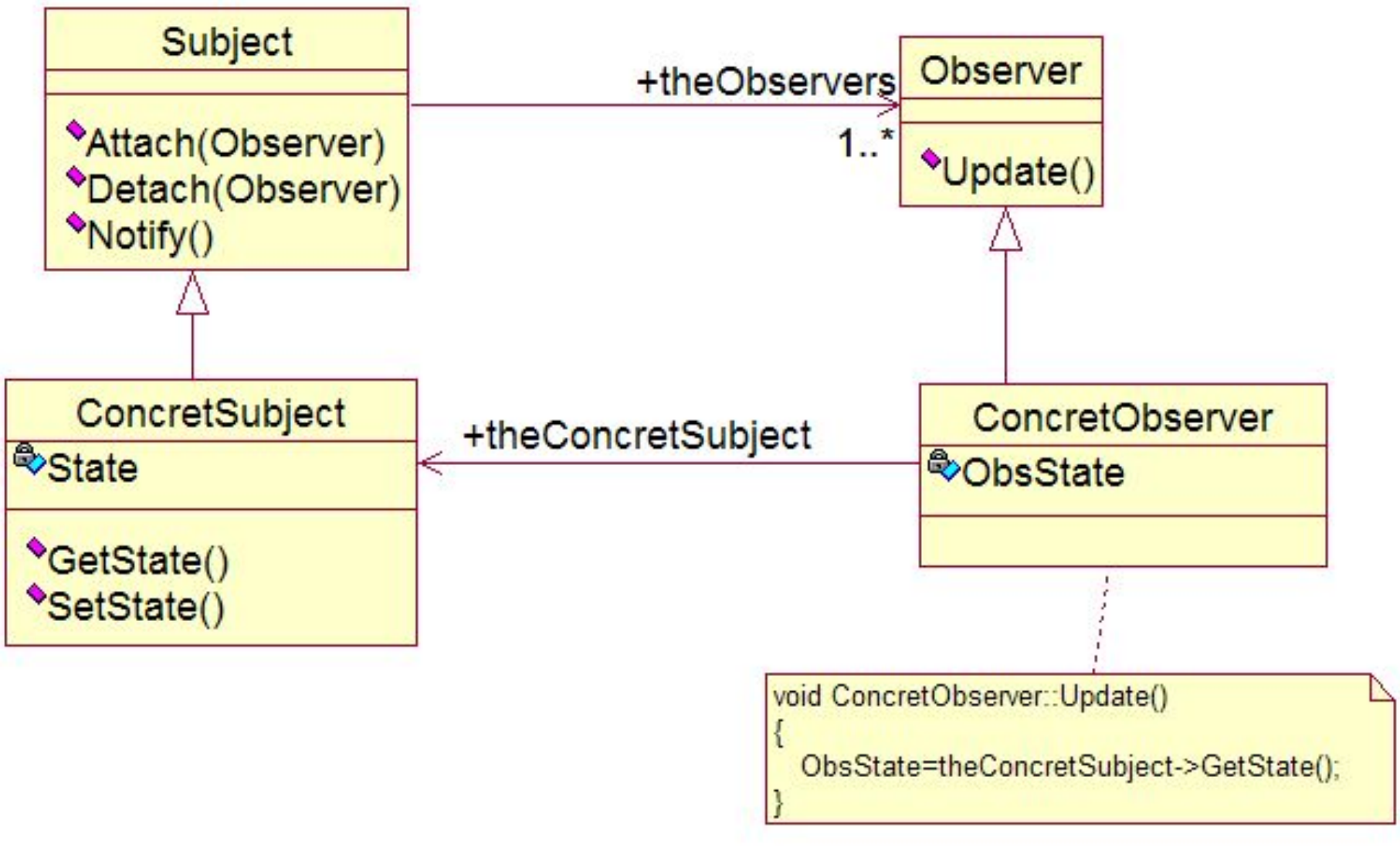
Observer (спостерігач):
- надає інтерфейс для фіксації змін (операція Update).

ConcreteSubject (конкретний суб'єкт):
- має стан, що становить інтерес для ConcreteObserver;
- надає інтерфейс для "читання" та зміни стану (GetState, SetState).

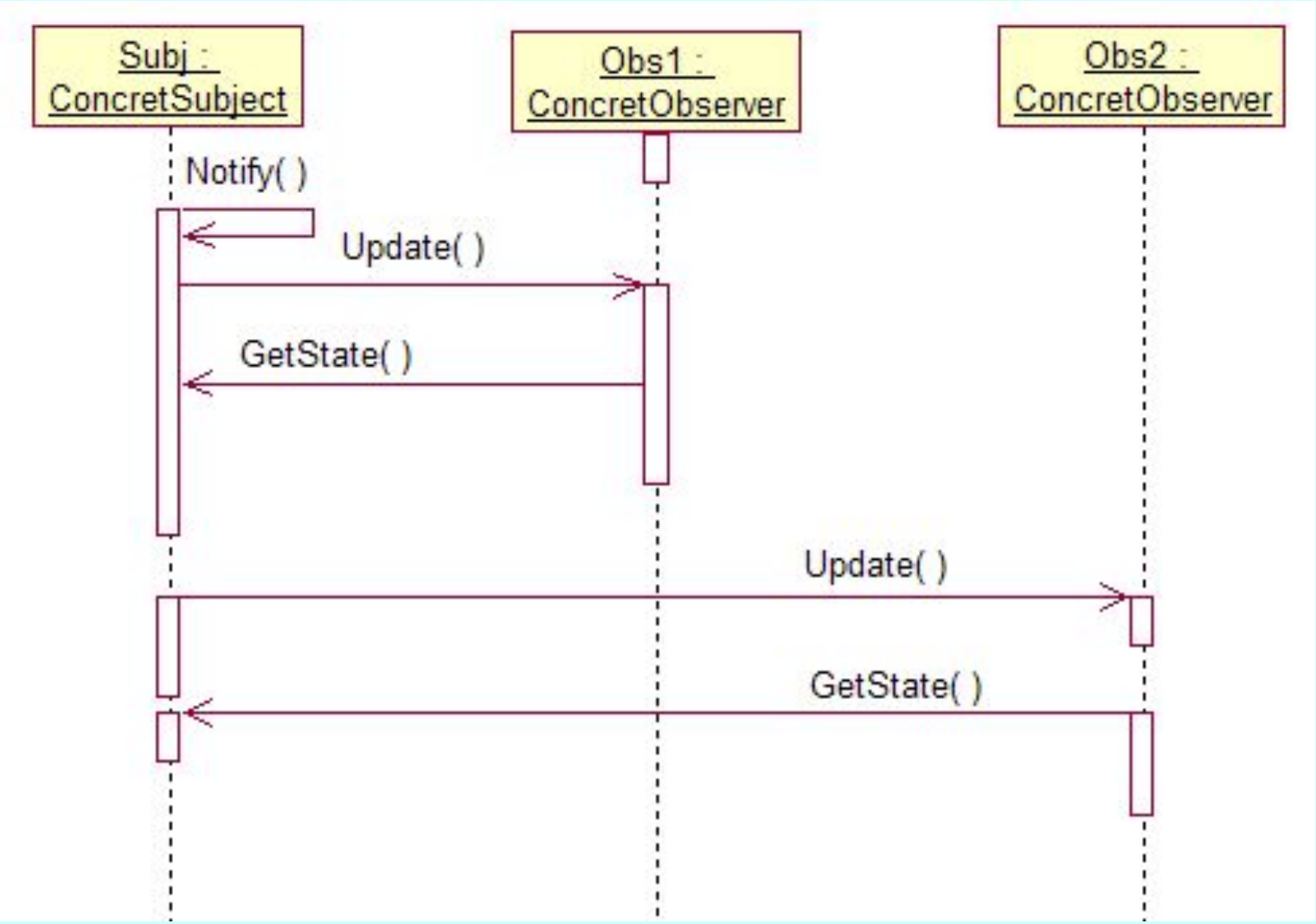
ConcreteObserver (конкретний спостерігач):
- зберігає посилання на об'єкт класу ConcreteSubject;
- зберігає дані про стан, які повинні бути узгоджені із станом суб'єкта.



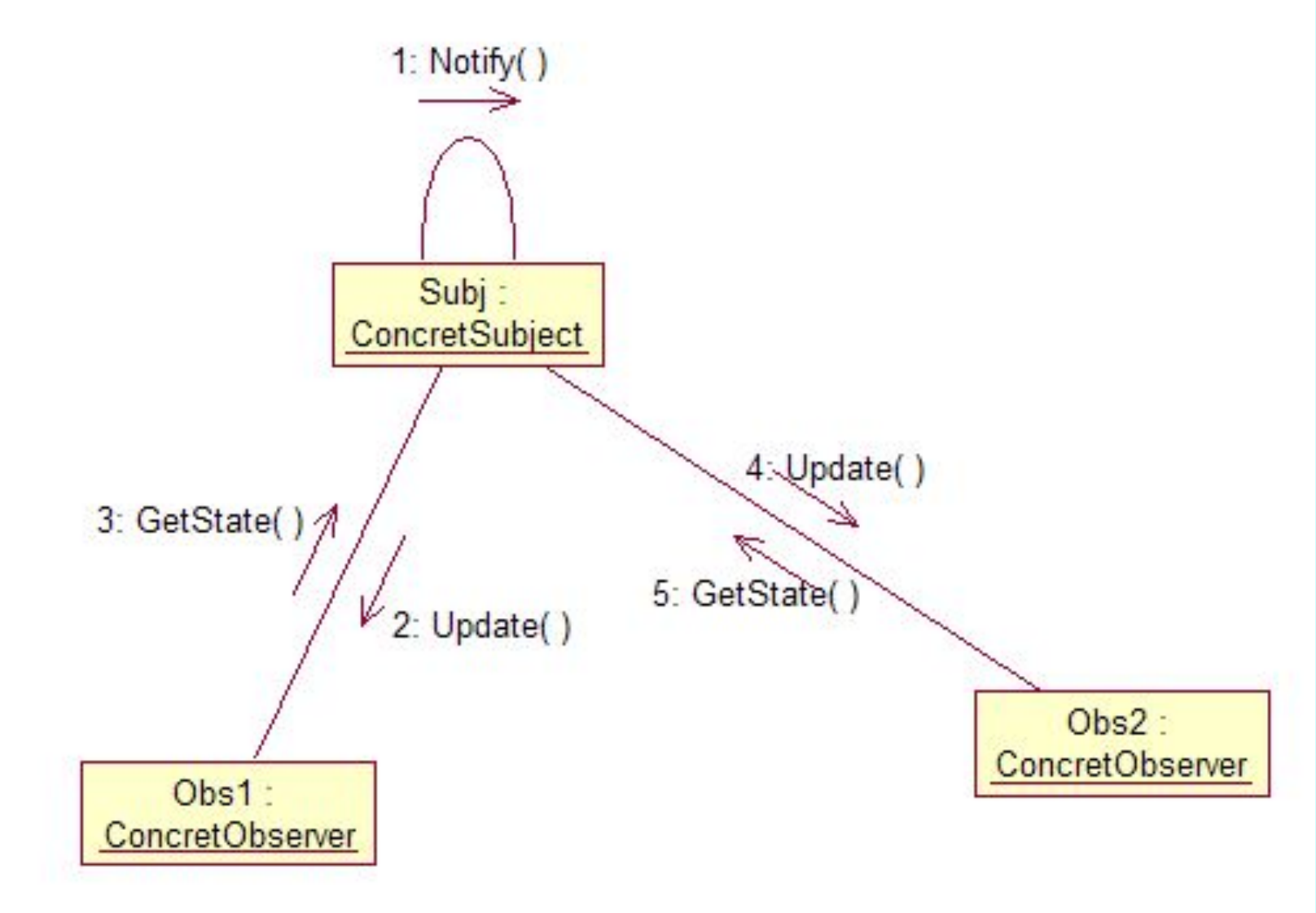
Observer


Subject
Attach(Observer)
Detach(Observer)
Notify()
+theObservers
1..*
Observer
Update()
ConcretSubject
State
GetState()
SetState()
+theConcretSubject
ConcretObserver
ObsState
void ConcretObserver::Update()
{
ObsState=theConcretSubject->GetState();
}

Observer


Subj :
ConcretSubject
Obs1 :
ConcretObserver
Obs2 :
ConcretObserver
Notify( )
Update( )
GetState( )
Update( )
GetState( )

Observer


1: Notify( )
Subj :
ConcretSubject
4: Update( )
3: GetState( )
5: GetState( )
2: Update( )
Obs2 :
ConcretObserver
Obs1 :
ConcretObserver

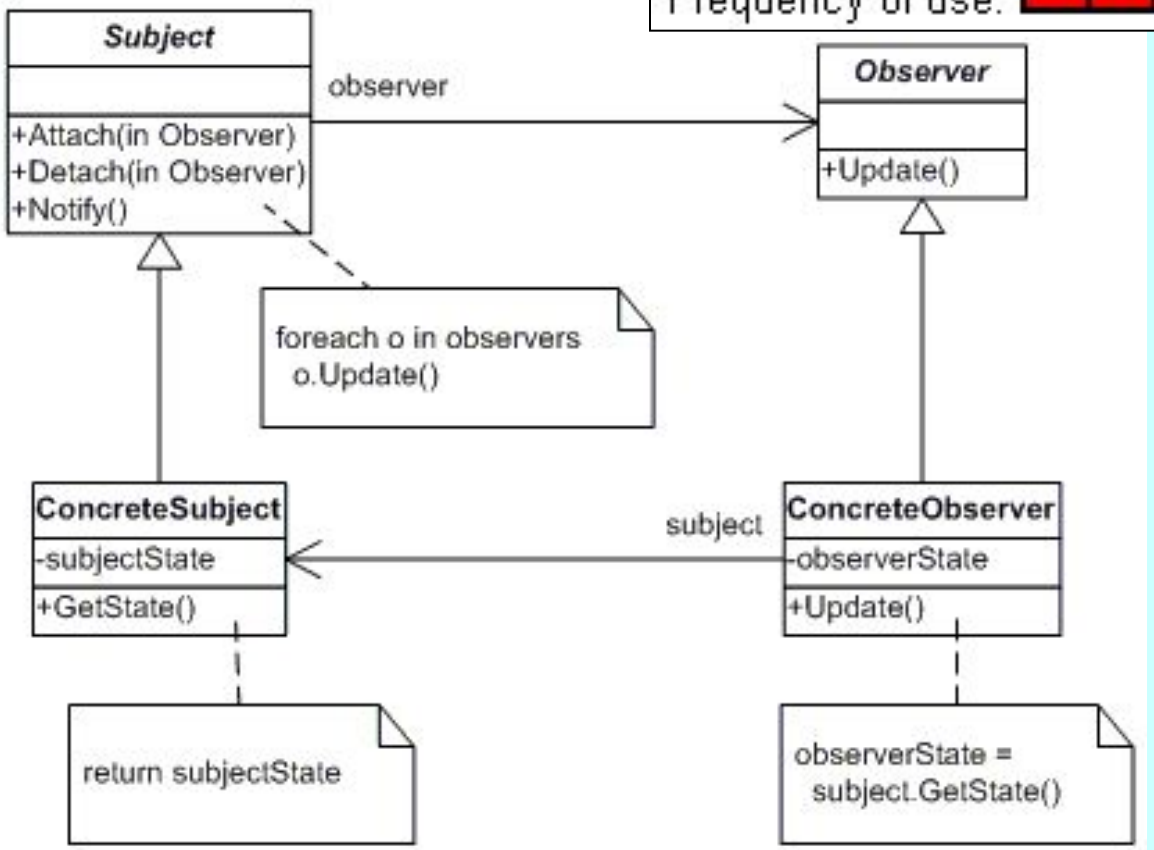
Observer (data & object factory™, dofactory.com)

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: 1 2 3 4 5 high

UML class diagram



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

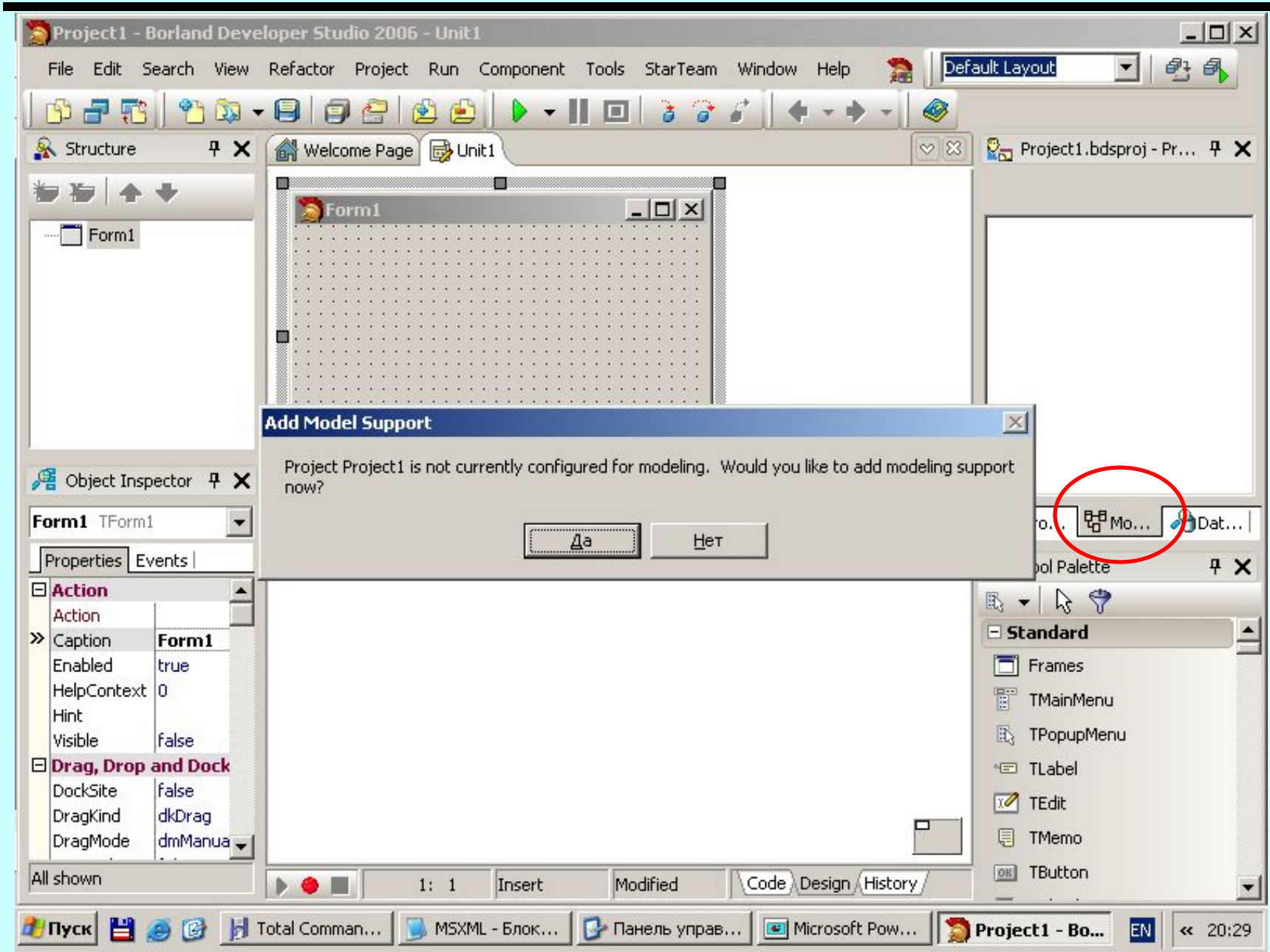
```
using System;
using System.Collections;
abstract class Subject    // "Subject"
{    // Fields
  private ArrayList observers = new
       ArrayList();
  // Methods
  public void Attach( Observer observer )
  {     observers.Add( observer );
  }
  public void Detach( Observer observer )
  {     observers.Remove( 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; }
  }
}
```

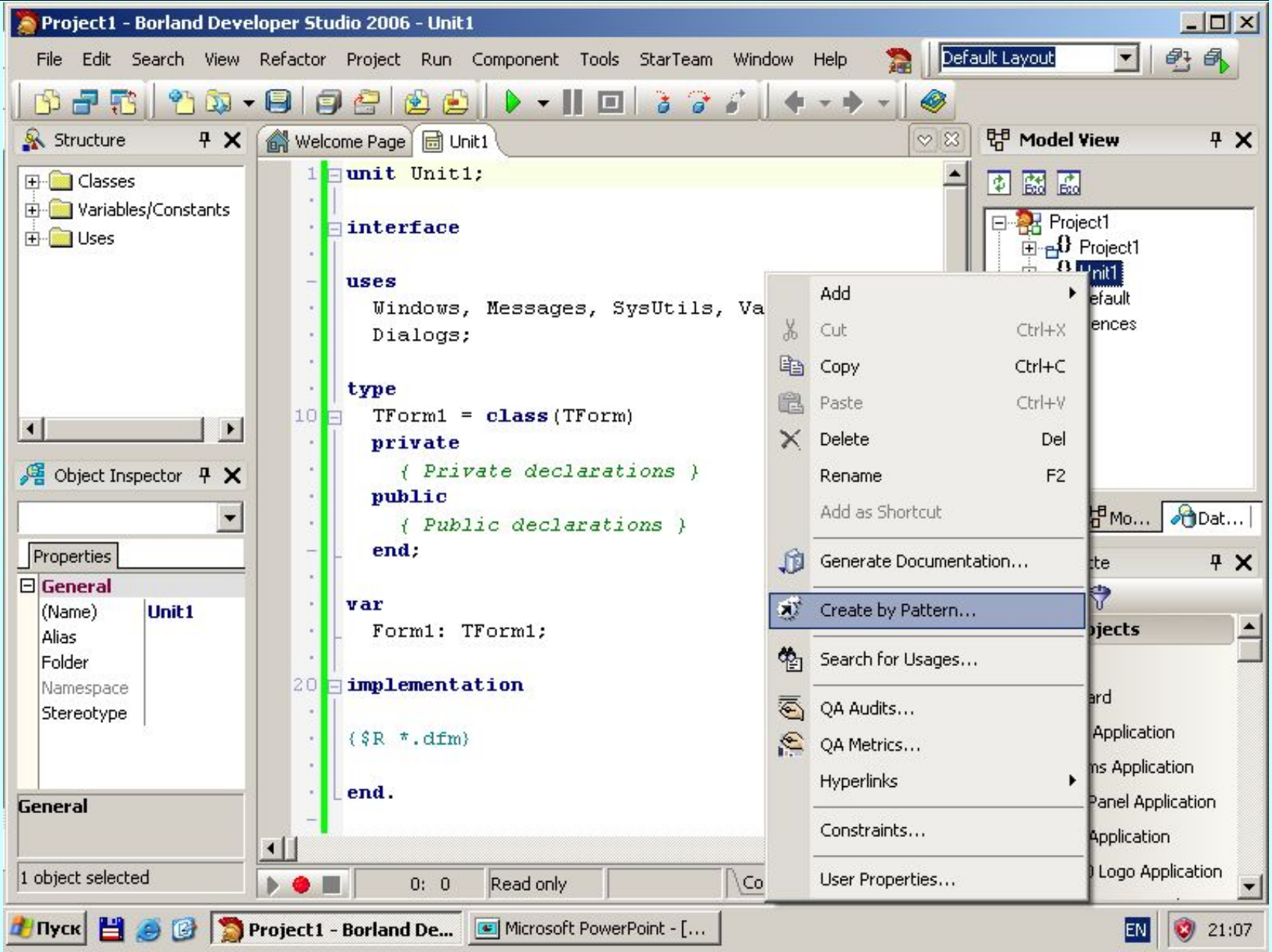
```
///   Client test
public class Client
{
  public static void Main(
         string[] args )
  {
    // Configure Observer
      // structure
    ConcreteSubject s =
      new ConcreteSubject();
    s.Attach(new
      ConcreteObserver( s, "X" ) );
    s.Attach( new
      ConcreteObserver( s, "Y" )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

Додаток

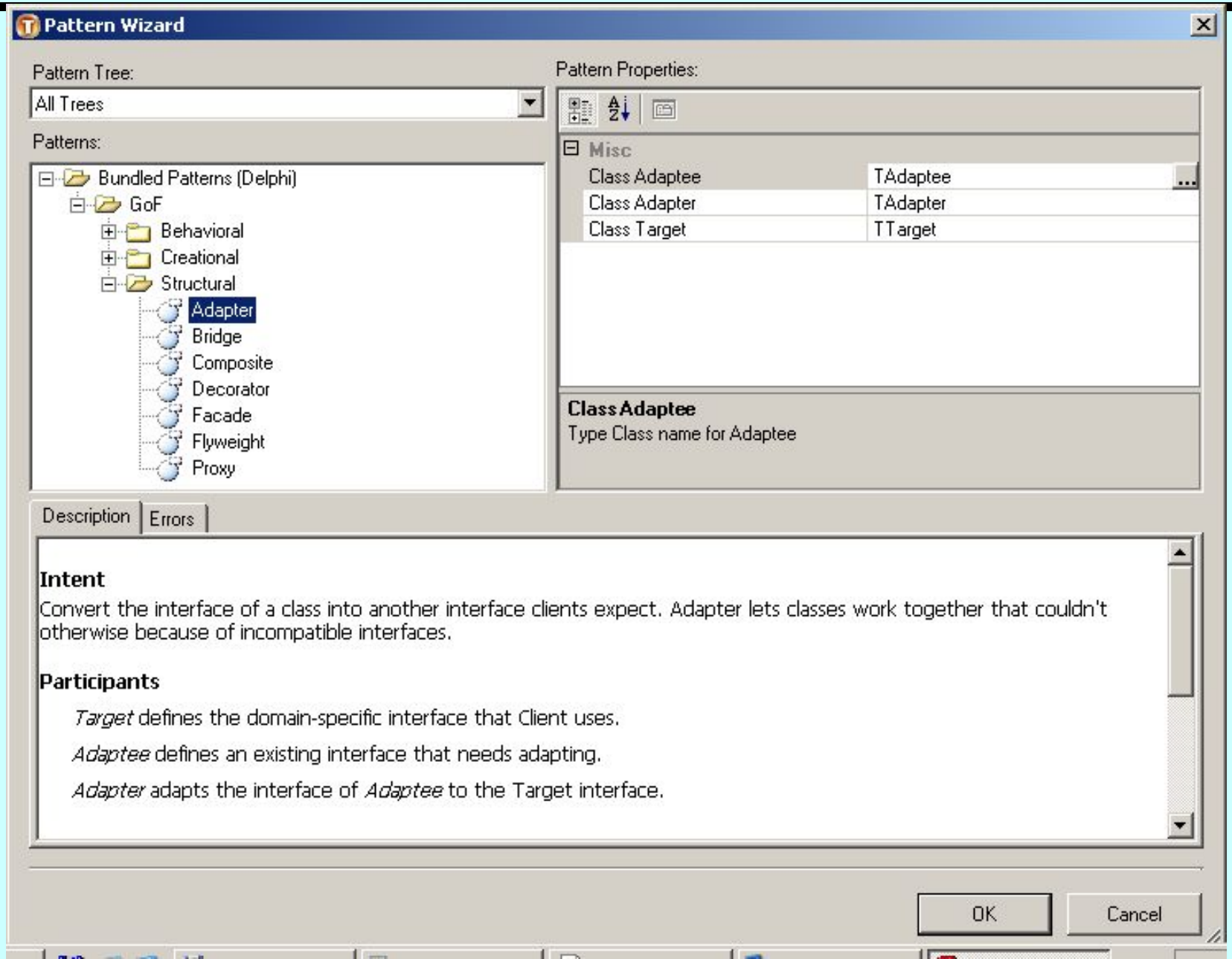
Borland Developer Studio 2006



Borland Developer Studio 2006



Borland Developer Studio 2006



Borland Developer Studio 2006