Structural Patterns

Structural Patterns (1)

- تحدد كيف يتم تجميع الأغراض
- الـ Patterns في هذا الصنف متممين لبعضهم البعض.

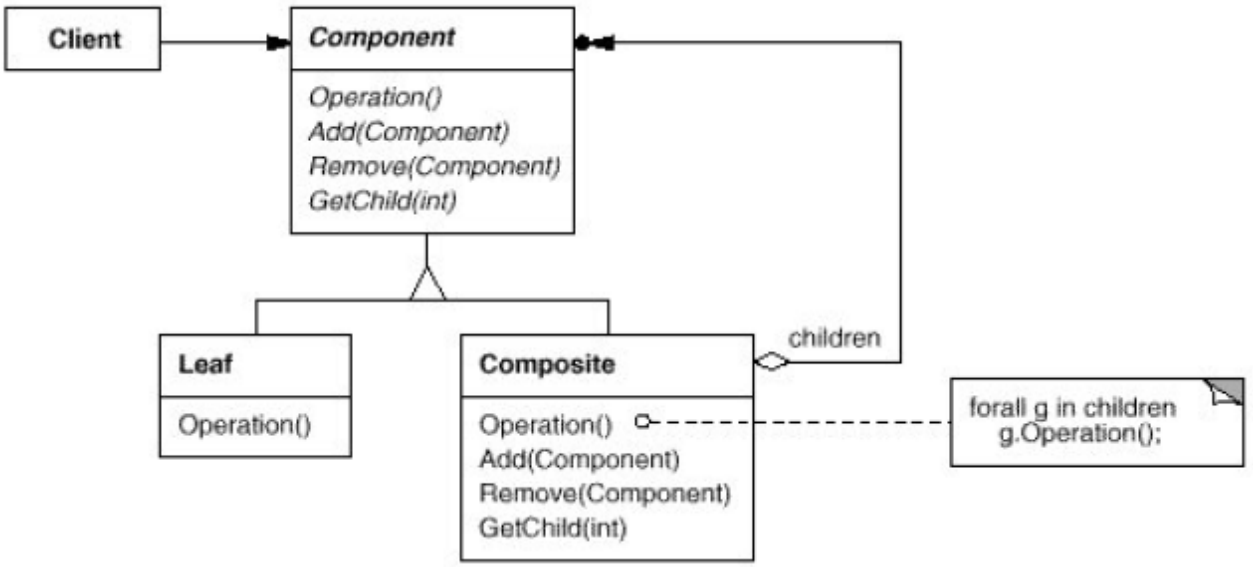
- **Adapter**
  - Convert the interface of a class into another interface clients expect.
  - Adapter lets classes work together that couldn't otherwise because of incompatible interfaces.
  - يجعل من غرض موافق لأخر
- **Bridge**
  - Decouple an abstraction from its implementation so that the two can vary independently.
  - يفصل بين تجريد غرض و تنفيذاته
- **Composite**
  - Compose objects into tree structures to represent part-whole hierarchies.
  - Composite lets clients treat individual objects and compositions of objects uniformly.
  - يعتمد على أغراض بدائية و مكونات منها

Structural Patterns (2)

- **Decorator**
  - Attach additional responsibilities to an object dynamically.
  - Decorators provide a flexible alternative to subclassing for extending functionality.
  - يضيف خدمات للغرض ديناميكياً و بشكل عودي
- **Facade**
  - Provide a unified interface to a set of interfaces in a subsystem.
  - Facade defines a higher-level interface that makes the subsystem easier to use.
  - يخفي البنية المعقدة لنظام
- **Flyweight**
  - Use sharing to support large numbers of fine-grained objects efficiently.
  - أغراض صغيرة الحجم مكرسة لتكون متشاركة
- **Proxy**
  - Provide a surrogate or placeholder for another object to control access to it.
  - غرض يحجب آخر

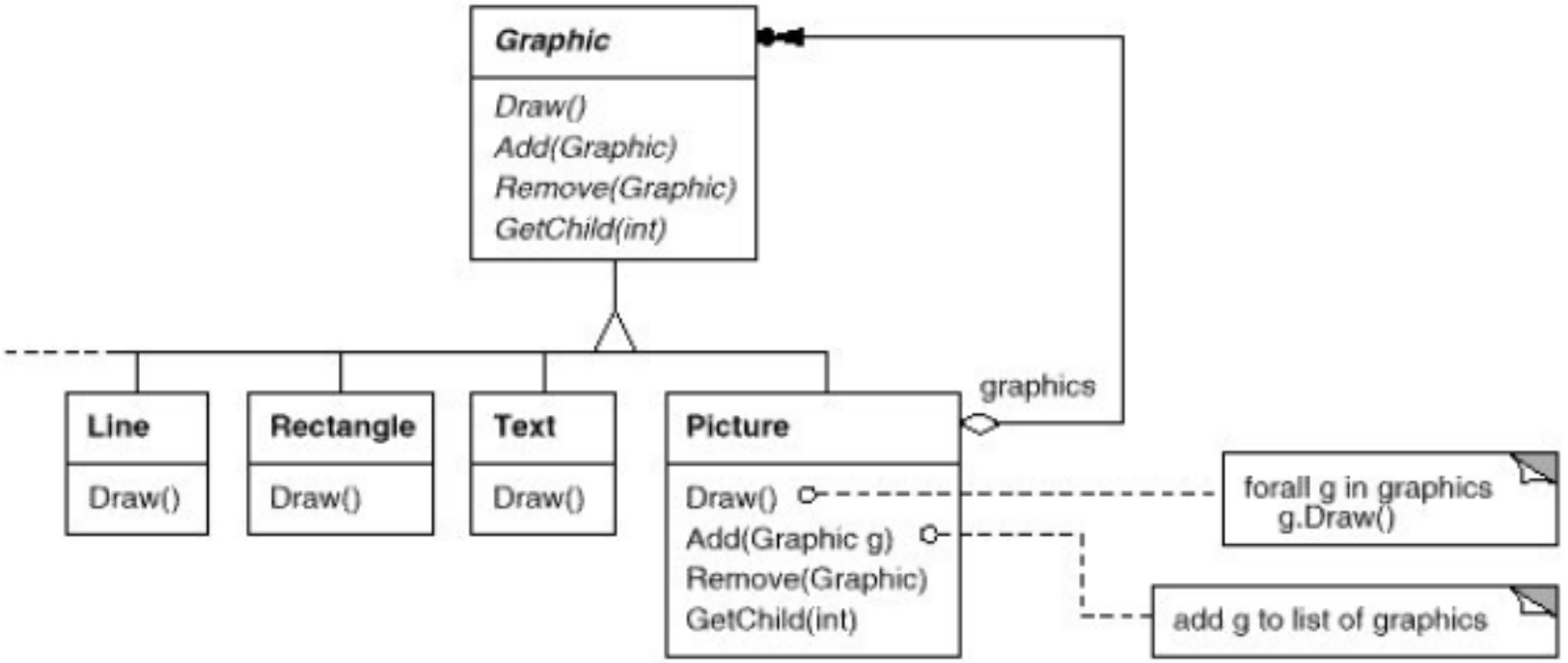
Composite

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

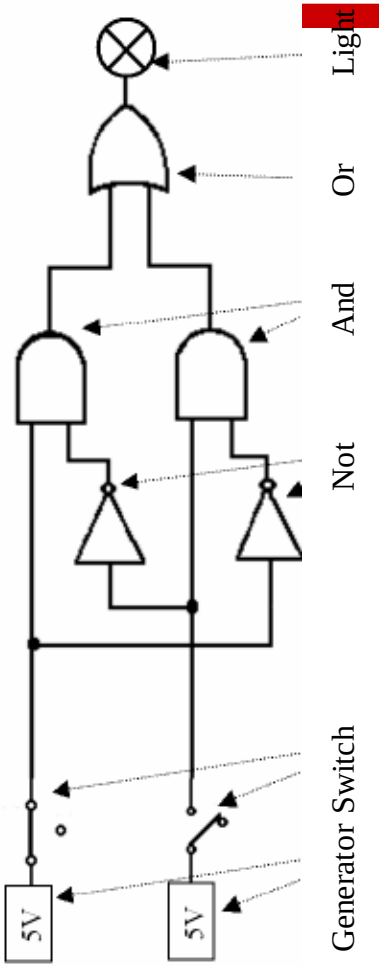


Composite –Example 1

- Awt/Swing Components and their method of *repaint()*.



Composite –Example 2



نريد نمذجة دارات منطقية اعتباراً من Components كما في الشكل التالي :

الـ Component يملك على الأكثر دخلين و خرج وحيد.

من أجل بناء دارة فإن خرج Component يجب أن يتصل إلى مدخل Component .

خرج الـ Component يمكن أن يكون فعالاً أو لا. هذا يتوقف على نوع الـ Component و قيم دخله.

نقول إن دخلاً فعالاً إذا كان موصولاً إلى خرج فعال و غير فعال إذا كان موصولاً إلى خرج غير فعال.

الـ Generator لا يملك دخلاً و خرجه دوماً فعال.

الـ Light لا يملك خرجاً و هي مضيئة إذا كانت فعالة.

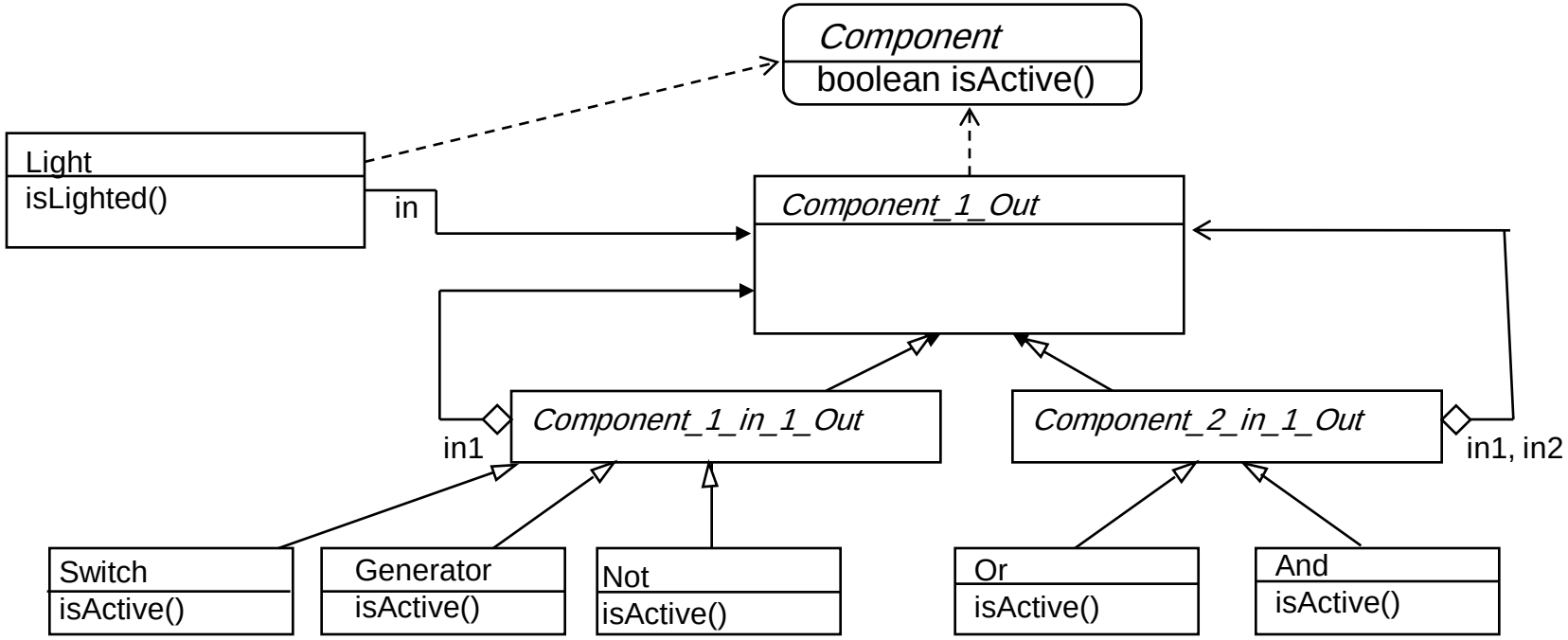
الـ Switch يمكن أن يكون مفتوحاً {غير فعال} أو موصولاً {فعالاً}.

الـ Not و الـ Switch هما بخرج وحيد و دخل وحيد.

يمكننا أن نرى أيضاً Components بدخلين و خرج وحيد كالـ And و الـ Or.

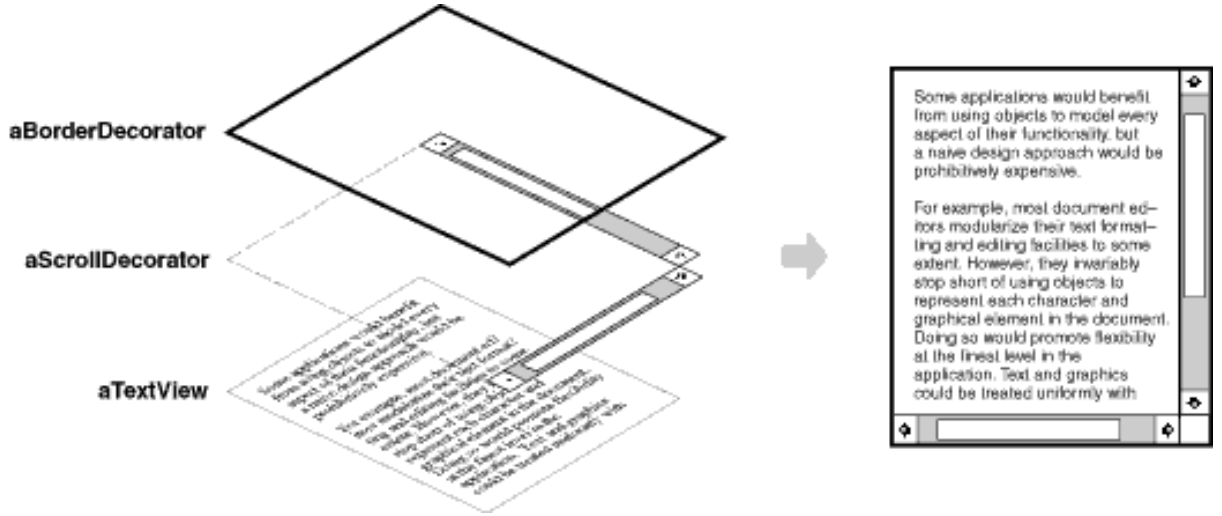
دخل الـ Component يتعلق عموماً بالـ Component المتصل به. إذا لم يتصل الدخل بـ Component فهو null. إذا حولنا حساب خرج Component الذي يملك دخلاً null فإنه يجب رفع Exception NullComponentInputException.

Composite –Example 2



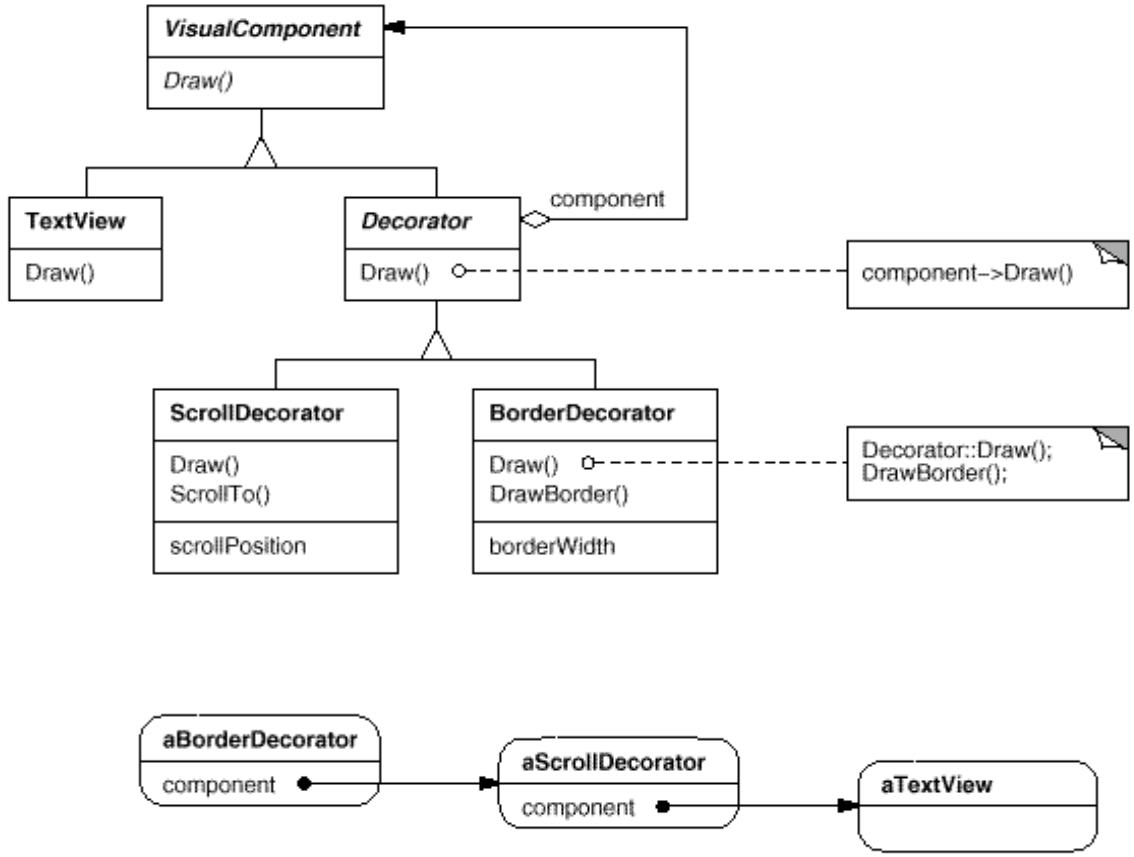
Decorator (1)

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



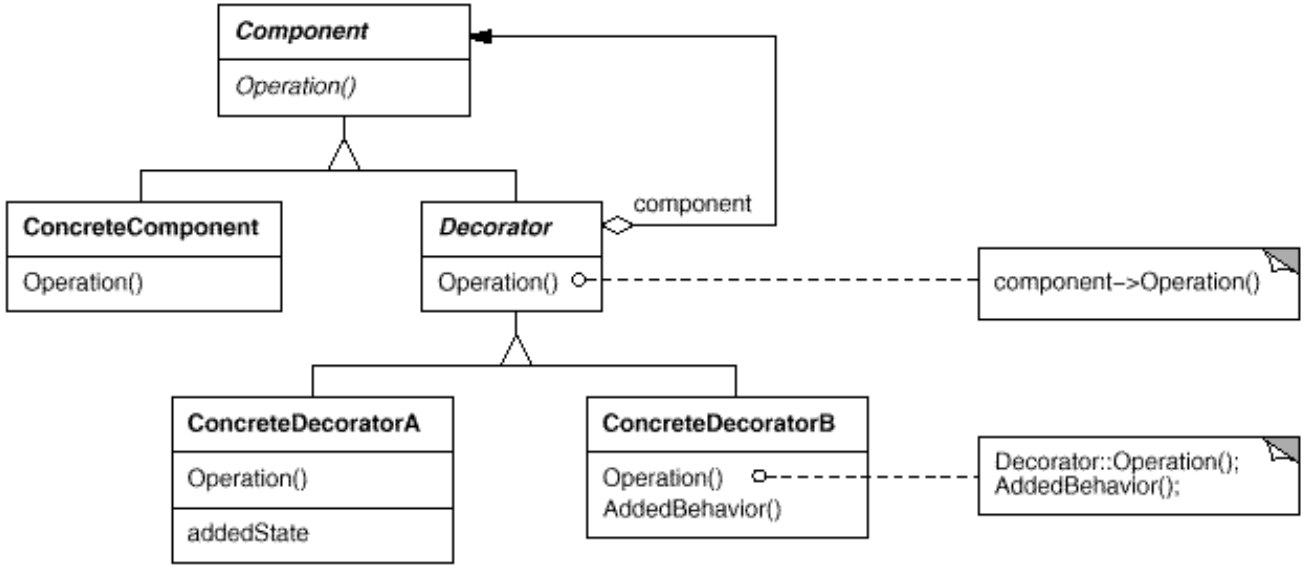
- Ex : I/O Streams in Java

Decorator (2)


VisualComponent
Draw()
TextView
Draw()
Decorator
Draw()
component
component->Draw()
ScrollDecorator
Draw()
ScrollTo()
scrollPosition
BorderDecorator
Draw()
DrawBorder()
borderWidth
Decorator::Draw();
DrawBorder();
aBorderDecorator
component
aScrollDecorator
component
aTextView

Decorator (3)

- Structure



- When to use :
  - to add responsibilities to individual objects dynamically and transparently.
  - for responsibilities that can be withdrawn.
  - when extension by subclassing is impractical. Sometimes a large number of independent extensions are possible and would produce an explosion of subclasses to support every combination.

Example of debugger(1/12) : Decorator + Composite + Singleton

- حل سيء باستخدام println("info about the bug")
- استخدام الواجهة Debugger
  - DebuggerToScreen
  - WithoutDebugger
  - DebuggerToFile
- تحسين إضافي باستخدام DeuggerToStream
  - DebuggerToScreen
  - DebuggerToFile
- DebuggerInWindow
- Usage

Example of debugger(2/12) : Decorator + Composite + Singleton

- الحل السيئ

```
public class BadSolution
{
  public static void main(String[] args)
  {
    System.out.println(« Begin of program.");

    System.out.println("Begin of For.");

    for(int i=0; i<10; i++)
    {
      System.out.println("Begin of step i=" + i);
      // ...
      System.out.println("End of step i=" + i);
    }

    System.out.println("End of For.");

    //
    // Several ohers thousands of System.out.println().
    //

    System.out.println("End of program.");
  }
}
```

- لماذا هذ الحل سيئ
  - إزالة التنقيح
    - <<< إزالة البرنامج
  - تغيير التنقيح نحو ملف أو إضافة معلومات إضافية أو ... أو ...
    - <<< تغيير البرنامج

Example of debugger (3/12) : Decorator + Composite + Singleton

```
public interface Debugger
{
        public void display(String message);
}
```

- استخدام الواجهة Debugger
  - الهدف = فصل الاستخدام عن كيفية تنفيذه
  - يمكن إضافة وظائف جديدة عند الحاجة

```
public class DebuggerToScreen implements Debugger {
        public void display(String message)
        {
                System.out.println(message);
        }
}
```

```
public class WithoutDebugger implements
        Debugger {
        public void display(String message)
        { //nothing ....          }
}
```

```
public class DebuggerToFile implements Debugger {
        public DebuggerToFile(String nnameFile) throws
        FileNotFoundException {
                FileOutputStream f =   new FileOutputStream (nameFile);
                sf = new PrintStream (f);
        }
        public void display(String message)
        {sf.println(message);}
}
```

Example of debugger (4/12) : Decorator + Composite + Singleton

```
import java.io.*;
public class DebuggerToStream implements Debugger {
        PrintStream outStrm;
        public DebuggerToStream(PrintStream strm) {
                outStrm = strm;
        }
        public DebuggerToStream(OutputStream strm) {
                this( new PrintStream(strm) );
        }

        public void display(String message) {
                outStrm.println(message);
        }

}
```

- التحسين باستخدام DebuggerToStream
  - يجمع الكود المشترك من الـ DebuggerToScreen و الـ DebuggerToFile
  - منهج display() وحيد

```
public class DebuggerToScreen
        extends DebuggerToStream {
        public DebuggerToScreen() {
                super(System.out);
        }
}
```

```
public class DebuggerToFile
        extends DebuggerToStream {
        public DebuggerToFile(String nameFile)
        throws java.io.FileNotFoundException {
                super( new FileOutputStream (nomFichier) );
        }
}
```

Example of Debugger (5/12) : Decorator + Composite + Singleton

```
public class DebuggerInWindow extends javax.swing.JFrame implements Debugger {
	protected javax.swing.JTextArea zoneDisplay;
	public DebuggerInWindow() {
		super("Debugger Window");
		super.setSize(400, 300);
		zoneDisplay = new javax.swing.JTextArea(40, 20);
		zoneDisplay.setEditable(false);
		Container contentPane = super.getContentPane();
		contentPane.setLayout(new java.awt.BorderLayout());
		contentPane.add(BorderLayout.CENTER,
				new JScrollPane(zoneDisplay));
		super.show();
	 }

	public void Display(String message)
	{
		zoneDisplay.append(message + "\n");
	}
}
```

Example (6/12) : Decorator + Composite + Singleton

```
public class SolutionWithDebugger {
        public static void main(String[] args) throws Exception {
                main("debugger to screen", new DebuggerToScreen());
                main("debugger to file", new debuggerToFile("dbg.txt"));
                main("debugger in window", new DebuggerInWindow());
                main("without debugger", new WithoutDebugger());
        }
        public static void main(String title, Debugger dbg) {
                S.o.p(« Execution with " + title + ".");
                main(dbg);
        }
        public static void main(dbg) {
                dbg.display("Begin of program.");
                dbg.display("Begin of For.");
                for(int i=0; i<10; i++) {
                                dbg.display("Begin of step i=" + i);
                                // ...
                                dbg.display("End of step i=" + i);
                }
                dbg.display("End of For");
                //....
                dbg.display("End of program.");
        }
}
```

- الميزات :
  - يمكننا تغيير نوع التنقيح بسهولة
- السيئات :
  - يجب نقل المنقح ضمن كل البرنامج
- الحل :
  - استخدام النموذج Singleton

Example of debugger (7/12) : Decorator + Composite + **Singleton**

```
public class SolutionWithDebugger {
  public static void main(String[] args) throws Exception {
    S.o.p("Execution without debugger.");
    main();
    main("To Screen", new DebuggeToScreen());
    main("To File", new DebuggerToFile("dbg.txt"));
    main("In Window", new DebuggerInWindow());
    main("Withut Debugger", new WthoutDebugger());
  }

public static void main(String title, Debugger dbg){
    S.o.p("Executin with Debugger " + title + ".");
    SingletonDebugger.setInstance( dbg );
    main();
    }

 public static void main() {
    SingletonDebugger.display("Begin of program");
    SingletonDebugger.display("Begin of for");
    for(int i=0; i<10; i++) {
        SingletonDebugger.display("Begin of step i=" + i);
        // ...
       SingletonDebugger.display("End of step i=" + i);
    }
    //...
  }
}
```

- يمكننا تغيير نوع التنقيح بشكل ديناميكي
  - و بدون تغيير للبرنامج الذي يستخدم المنقح

```
public abstract class SingletonDeboggeur {
        protected static Debugger instance = null;
        public static Debugger getInstance()
        {
                if(instance == null)
                                instance = new
                                WithoutDebugger();
                return instance;
        }
        public static void setInstance(Debugger dbg)
        {
                instance = dbg;
        }
        public static void display(String message) {
                getInstance().display(message);
        }
}
```

Example of debugger (7/12) : Decorator + Composite + Singleton

- لنفرض أننا نريد إضافة التاريخ إل رسائل التنقيح
  - و لكن سنحتاج إلى صف جديد للإظهار في نافذة و آخر للإظهار على الشاشة و آخر لإظهار ضمن ملف و ...

```
public class DebuggerToScreenWithDate
      extends DebuggerToScreen {
      public void display(String message) {
        super.dsplay( "[" + new java.util.Date() + "]:
       " + message );
  }
}
```

```
public class DebuggerInWindowWithDate
      extends DebuggerInWindow{
      public void display(String message) {
        super.dsplay( "[" + new java.util.Date() + "]: " + message );
  }
}
```

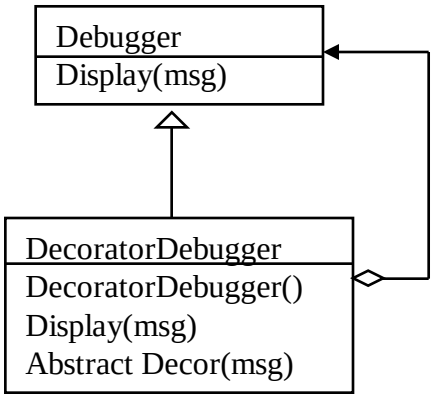
Example of debugger (8/12) : **Decorator** + Composite + Singleton

```
public abstract class DecoratorDebugger implements
        Debugger {
    protected Debugger dbgr;
    public DeboggageDecore(Debugger d) {
        this.dbgr = d;
    }

     public void display(String message) {
        dbgr.display ( decor(message) );
    }

    protected abstract String decor(String message);
}
```

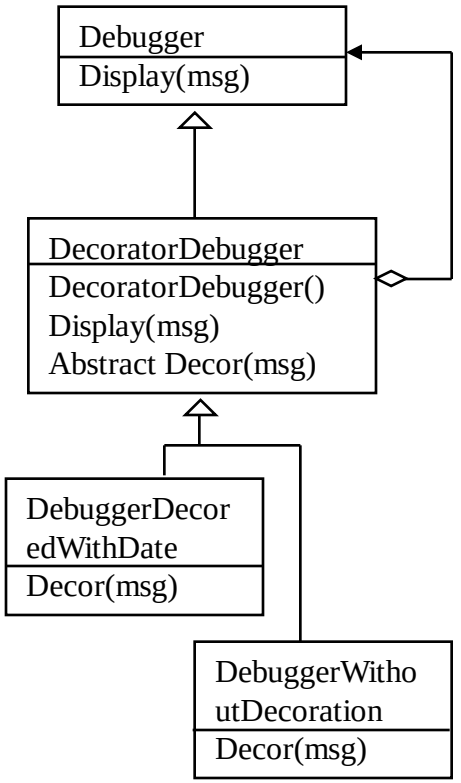
- لنفرض أننا نريد إضافة التاريخ إل رسائل التنقيح
- سنستخدم الموذج التصميمي Decorator و الذي يسمح بتغيير سلوك غرض فردي دون إنشاء صفوف مشتقة من صف هذا الغرض
  - سنقوم بإنشاء صف لزخرفة الأغراض



Example of debugger (9/12) : **Decorator** + Composite + Singleton

```
public class DebuggerDecoredWithDate
      extends DecoratorDebugger {
      public DebuggerDecoredWithDate (Debugger d){
          super(d);
      }
      protected String decor(String message) {
         return "[" + new java.util.Date() + "] : " +
         message;
      }
}
```

```
public class DebuggerWithoutDecoration
      extends DecoratorDebugger {
      public DebuggerDecoredWithDate (Debugger d){
          super(d);
      }
      protected String decor(String message) {
         return message;
      }
}
```



Example of debugger (10/12) : **Decorator** + Composite + Singleton

```
public class SolutionWithDecoration {
  public static void main(String[] args) throws Exception
  {
    Debugger toScreen = new DebuggerToScreen();
    Debugger toFile = new DebuggerToFile("dbg.txt");
    Debugger inWindow = new DebuggerInWindow();

    SolutionWithDebogger.main("to screen with decoration",
        new DebuggerDecoredWithDate(toScreen) );

    SolutionWithDebogger.main("to file with decoration",
        new DebuggerDecoredWithDate(toFile) );

    SolutionWithDebogger.main(« in window with decoration",
        new DebuggerDecoredWithDate(inWindow) );
  }
}
```

- يمكننا استخدام الـ Decorator بأسلوب آخر

```
SingletonDebugger.setInstance( new
DebuggerDecoredWithDate(new
      DebuggerInWindow() ) );

SolutionWithDebugger.main();
```

- هل تتذكرون شيئاً مشابهاً بالـجافا؟

Example of debugger (11/12) : Decorator + **Composite** + Singleton

- تحسين للحل السابق هو من خلال عزل سلوك الزخرفة في واجهة منفصلة.

```
public interface Decoration {
    public String decor(String message);
}
```

```
public class WithoutDecoration
       implements Decoration {
    public String decor(String message){
        return message;
  }
}
```

```
public class DecorationWithDate
       implements Decoration {
    public String decor(String message){
        return "[" + new java.util.Date() +
         "] : " + message;
  }
}
```

```
public class CompositeDebugger extends
        DecoratorDebugger {
    protected Decoration decoration;
    public CompositeDebugger(Debugger dbg,
                        Decoration dec){
        super(dbg);
        this.decoration = dec;
   }
   protected String decor(String message) {
       return decoration.decor(message);
   }
   public void setDecoration(Decoration dec) {
        this.decoration = dec;
   }
}
```

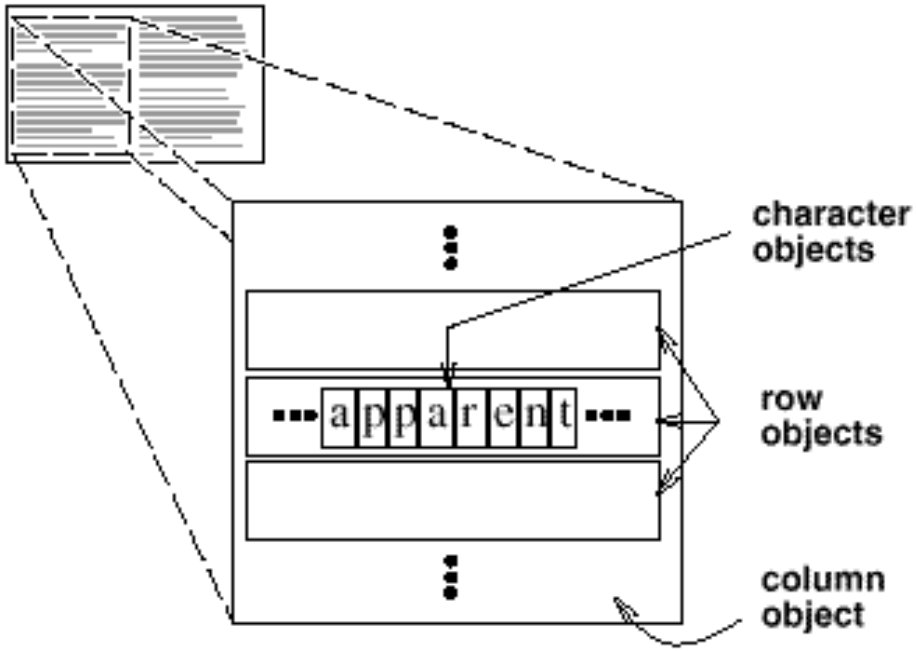
Example of debugger (12/12) : Decorator + **Composite** + Singleton

```
public class SolutionByComposition
{
  public static void main(String[] args) {
    Debugger toScreen = new DebuggerToScreen();
    Debugger toFile = new DebuggerToFile("dbg.txt");
    Debugger inWindow = new DebuggerInWindow();
    Decoration NoDecoration = new WithoutDecoration();
    Decoration withDate = new DecorationWithDate();
    main("to screen without decoration", toScreen, NoDecoration);
    main(« in Window without decoration", inWindow, NoDecoration);
    main("to screen with decoration", toScreen, withDate);
    main("to file with decoration", toFile, withDate);
    main("in window with decoration", inWindow, withDate);
  }
  public static void main(String title,  Debugger dbg,
                          Decoration dec) {
      SolutionWithDebugger( title, new CompisteDebugger(dbg, dec) );
  }
}
```

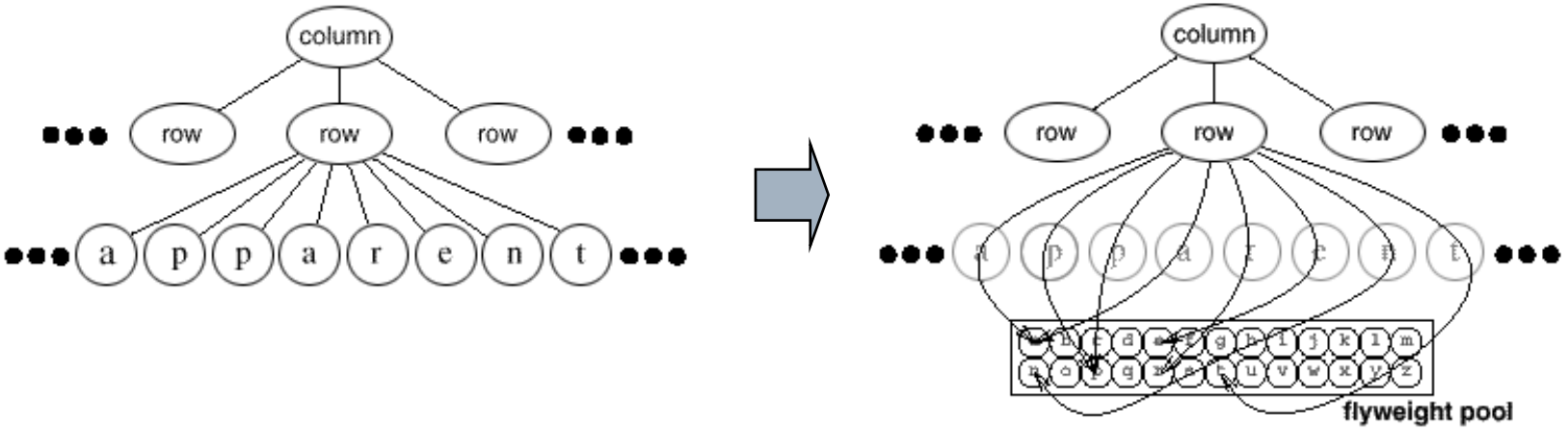
- الميزات :
  - يمكننا إضافة تنفيذات جديدة للـ Debugger
  - يمكننا إضافة تنفيذات جديدة للـ Decoration
  - يمكننا تجميع الاثنين
  - لا حاجة لتعديل الصفوف الموجودة مسبقاً
- يجب أن نسعى بحلولنا الغرضية التوجه أن تكون
  - Extensible
  - Reusable
  - maintenable

Flyweight (1)

- Use sharing to support large numbers of fine-grained objects efficiently.

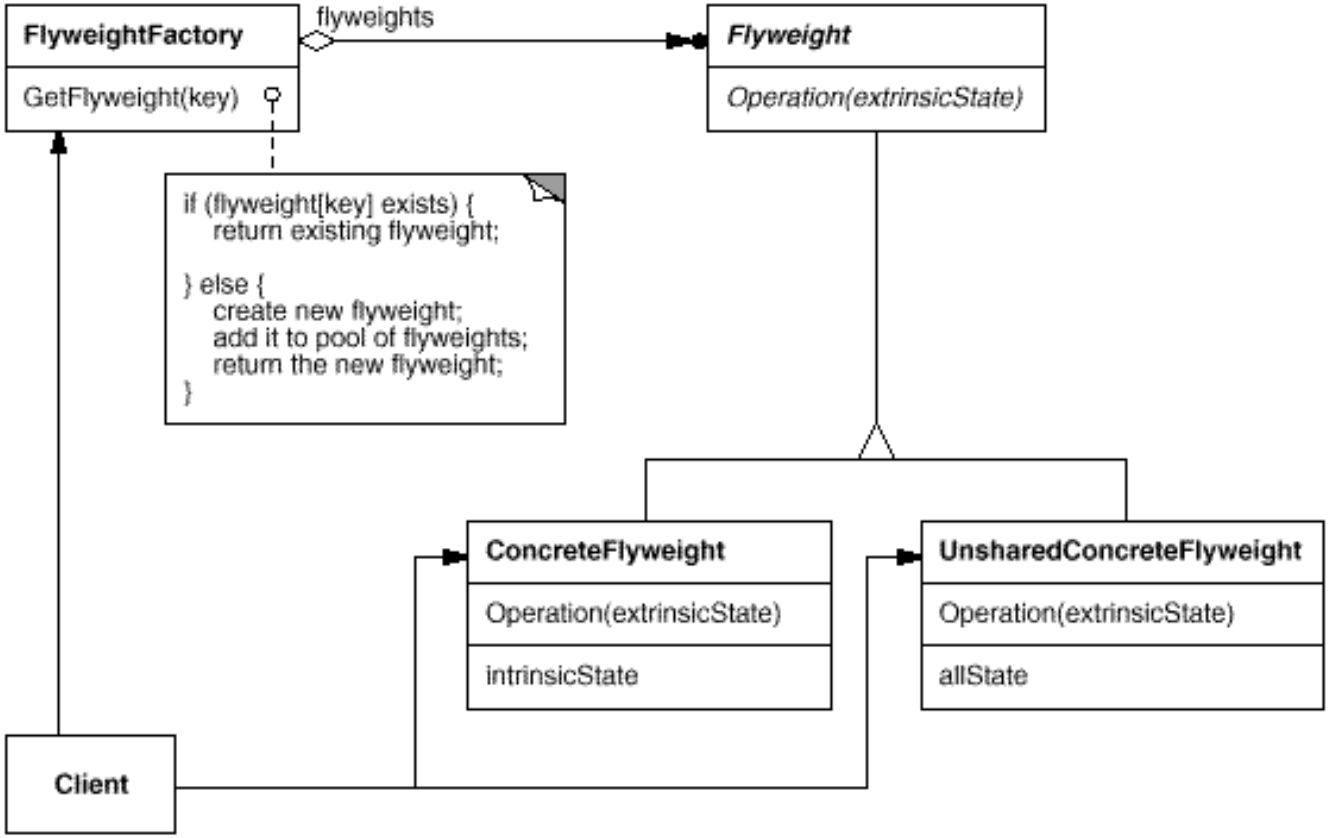


Flyweight (2)

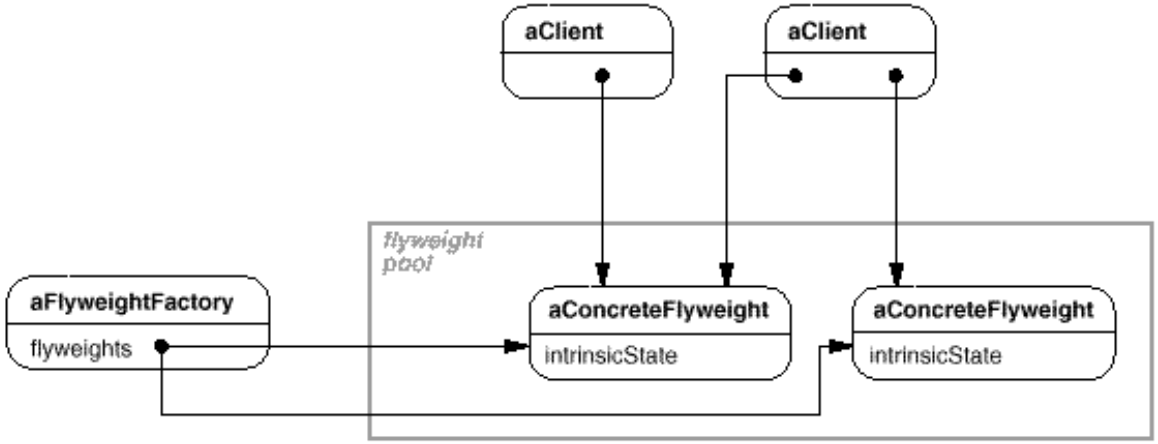

column
row
row
row
a
p
p
a
r
e
n
t
column
row
row
row
flyweight pool

Flyweight (3)

- Structure



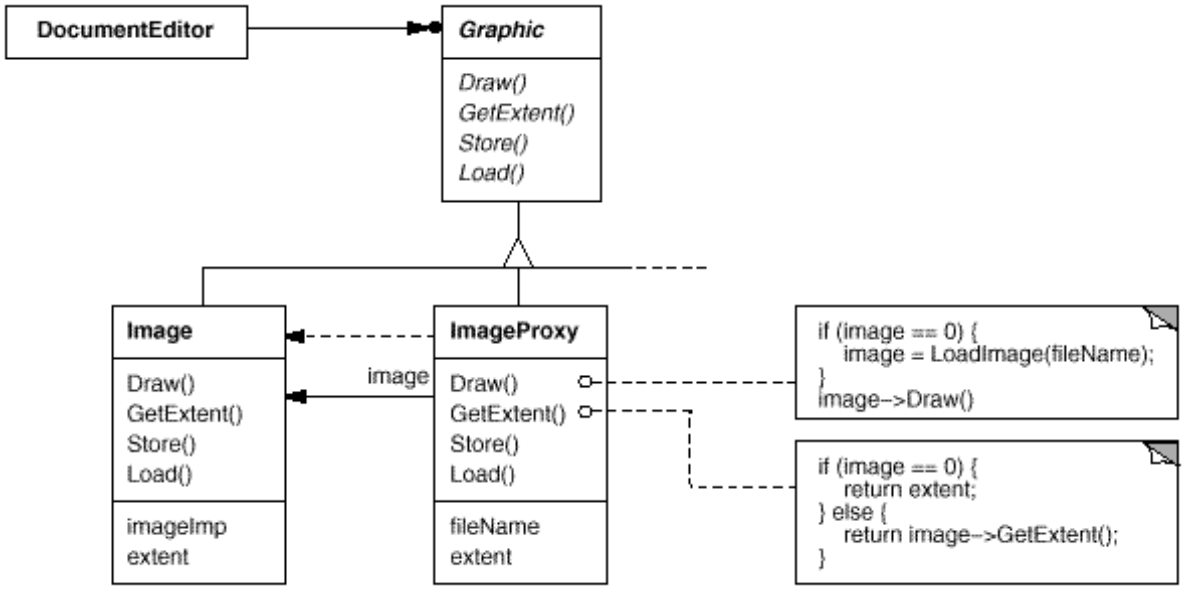
Flyweight (4)

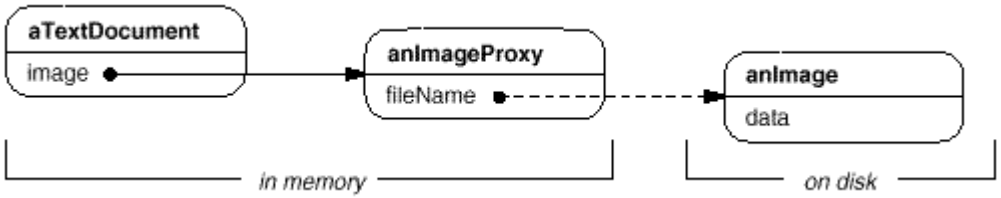


- Applicability
  - An application uses a large number of objects.
  - Storage costs are high because of the sheer quantity of objects.
  - Most object state can be made extrinsic.
  - Many groups of objects may be replaced by relatively few shared objects once extrinsic state is removed.
  - The application doesn't depend on object identity.

Proxy (1)

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



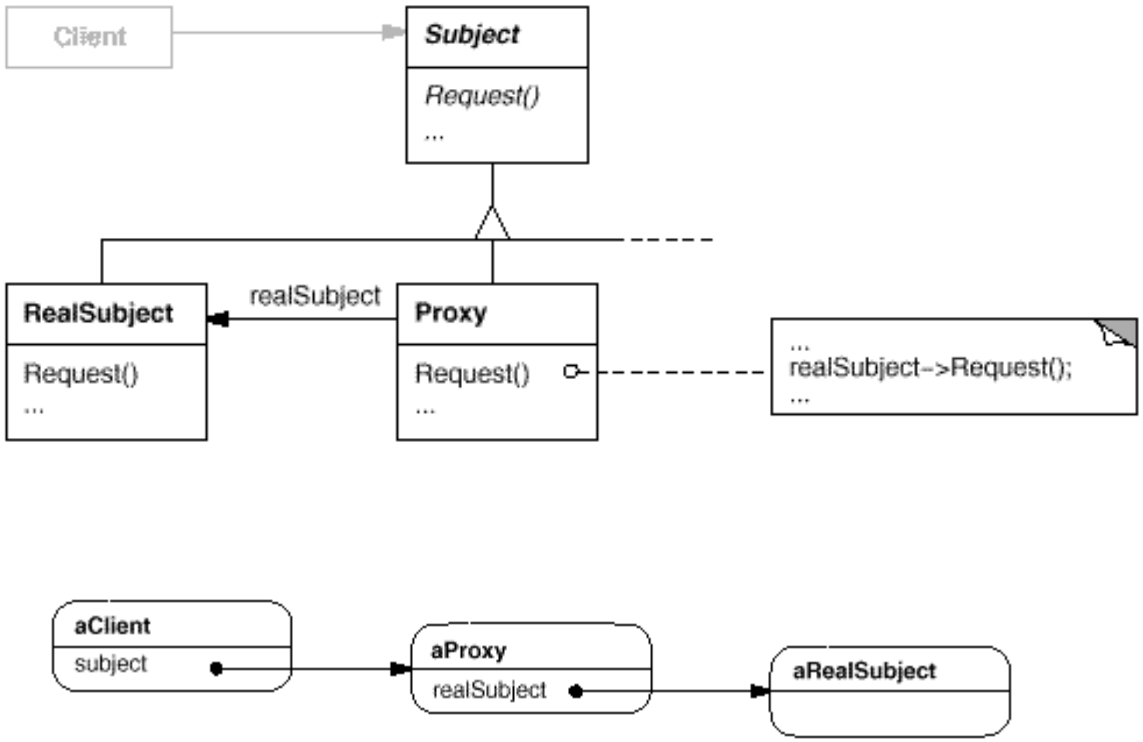


Proxy (2)

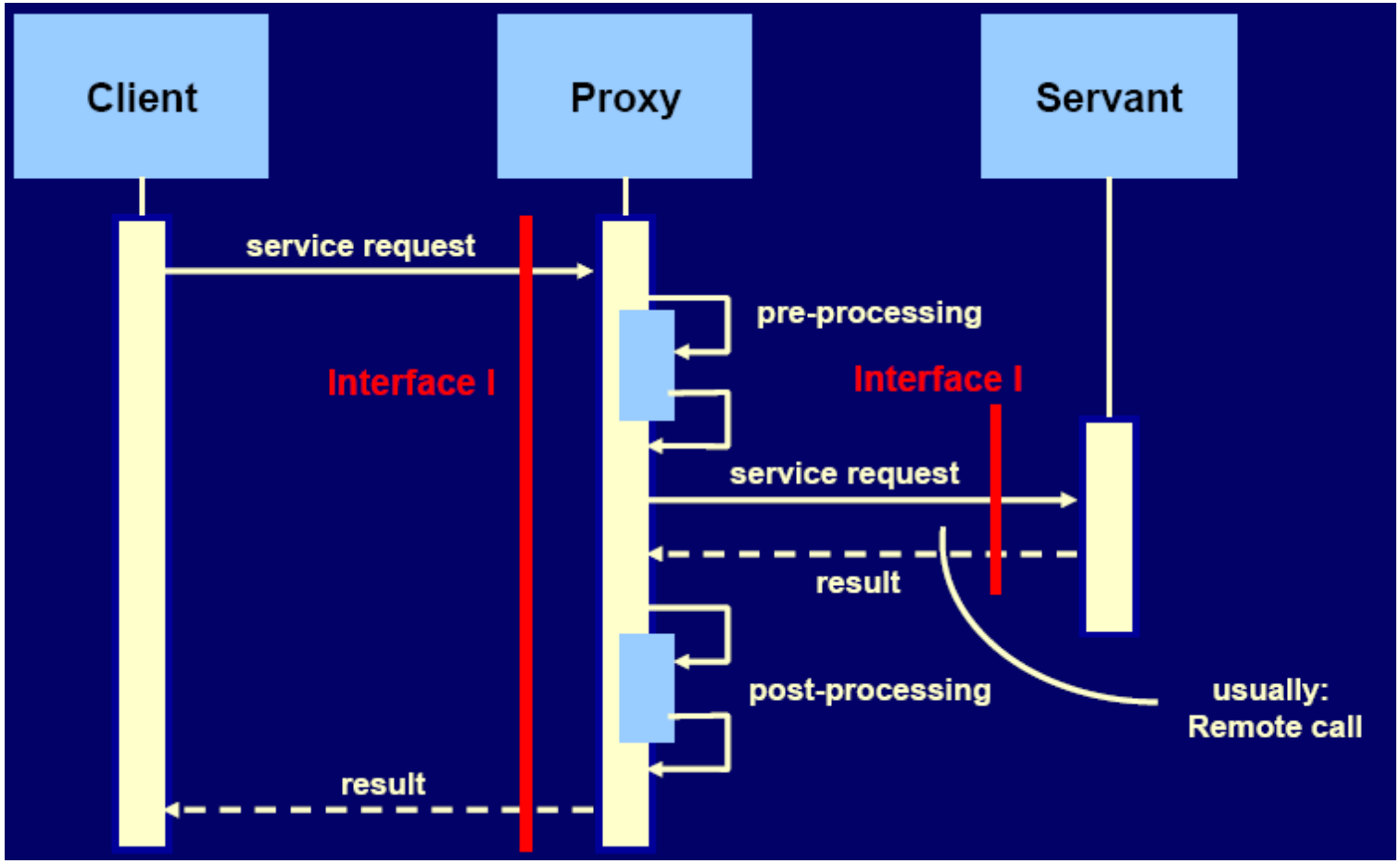
- Applicability :
  - A **remote proxy** provides a local representative for an object in a different address space.
  - A **virtual proxy** creates expensive objects on demand.
  - A **protection proxy** controls access to the original object. Protection proxies are useful when objects should have different access rights.
  - A **smart reference** is a replacement for a pointer that performs additional actions when an object is accessed:
    - counting the number of references to the real object>>> free this object when there are no more refereces
    - loading a persistent object into memory when it's first referenced.
    - checking that the real object is locked before it's accessed to ensure that no other object can change it.

Proxy (3)

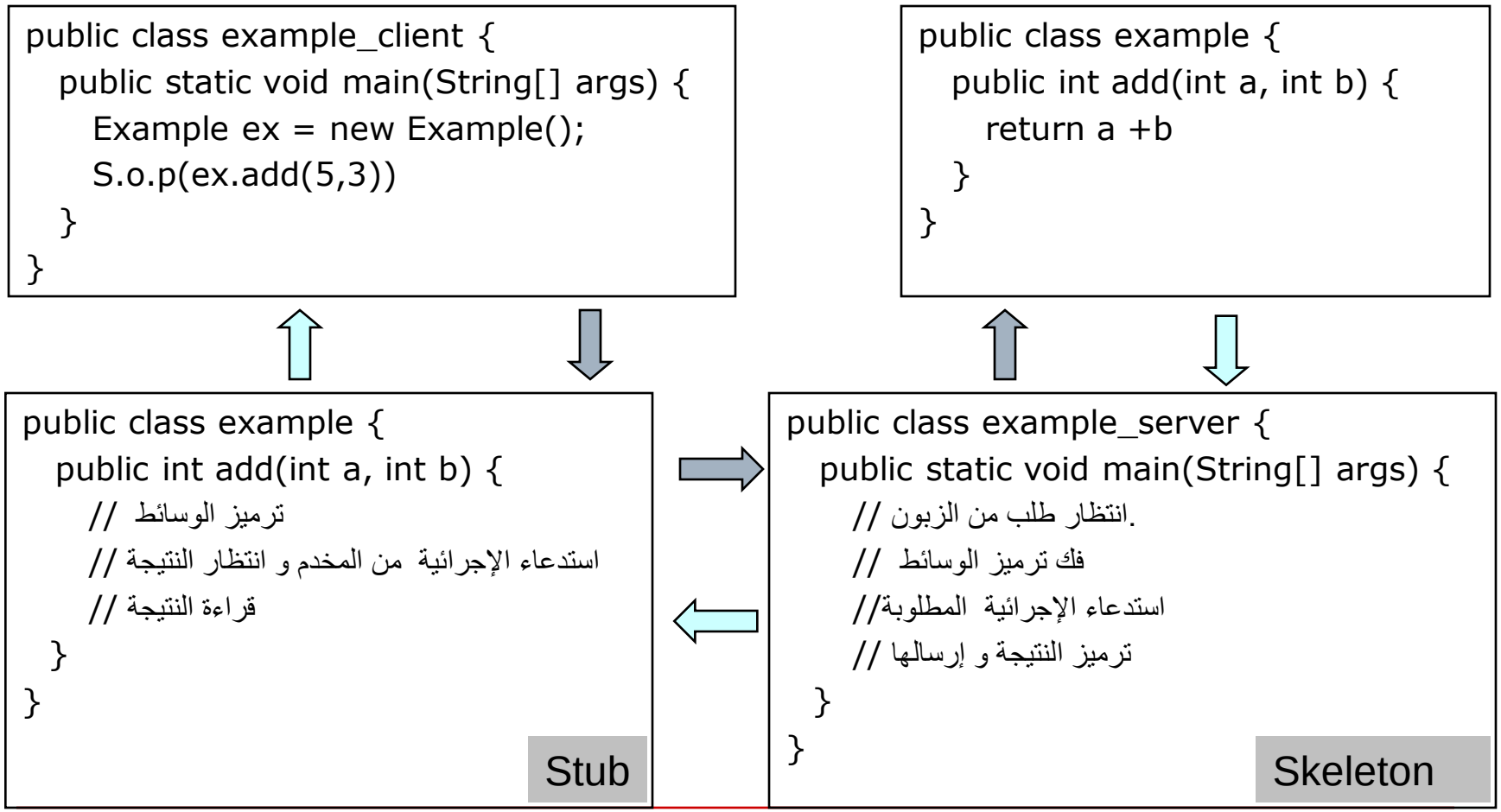
- Structure



Proxy (4) : remote proxy

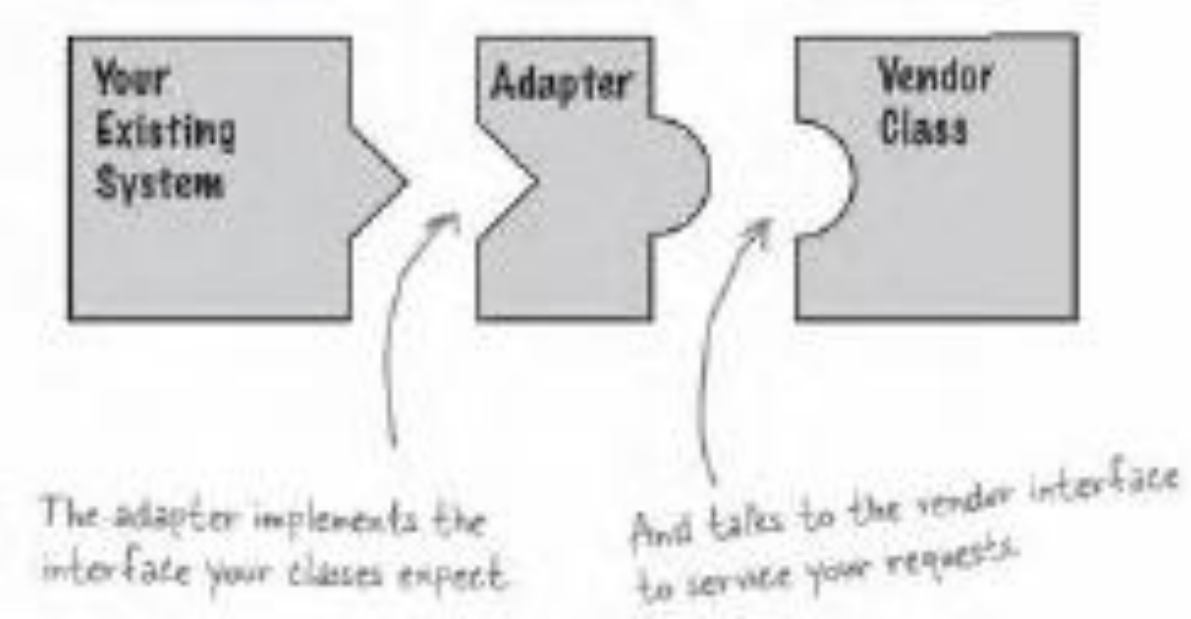


Proxy (5) : Stub & Skeleton



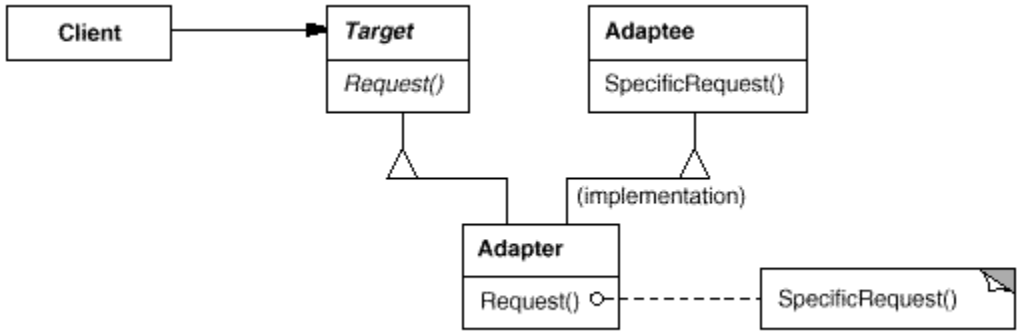
Adapter (1)

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

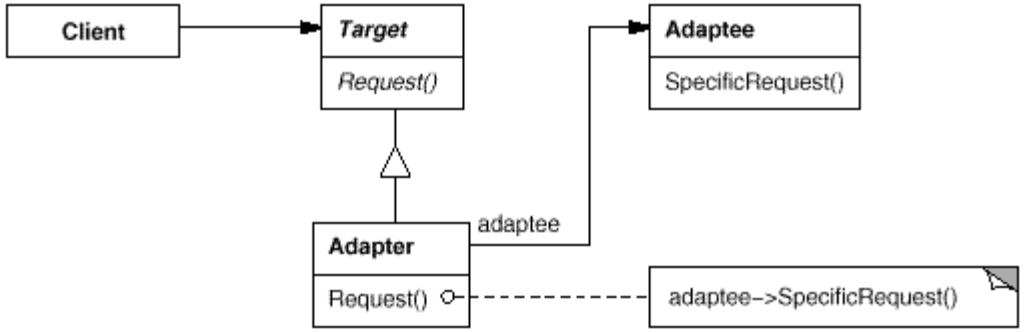


Adapter (2)

- Class Adapter pattern

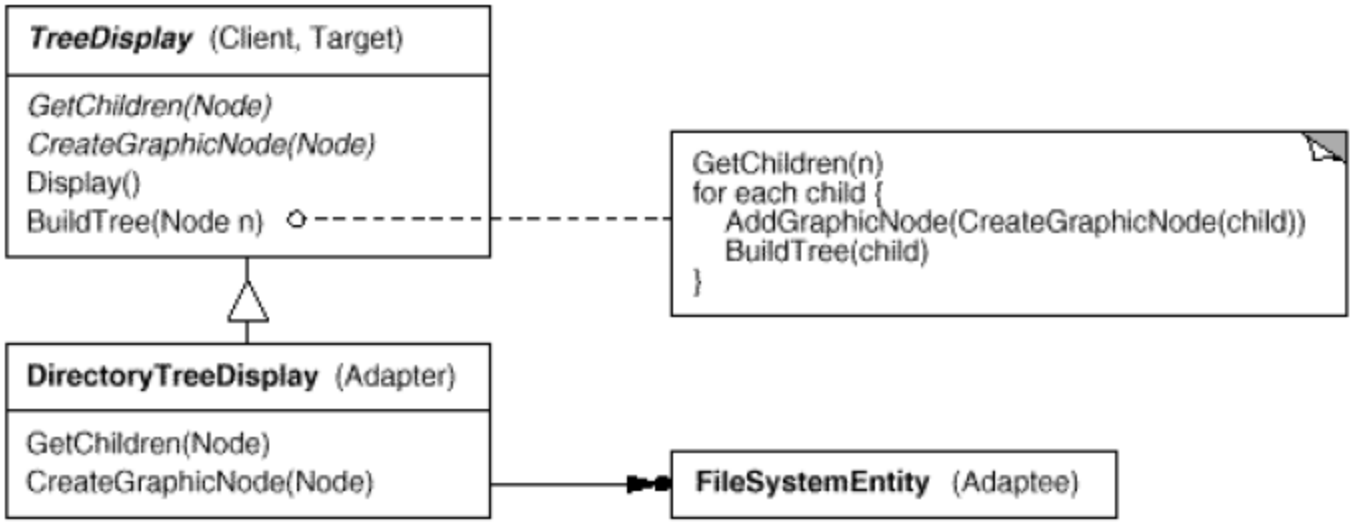


- Object Adapter pattern



Adapter (3)

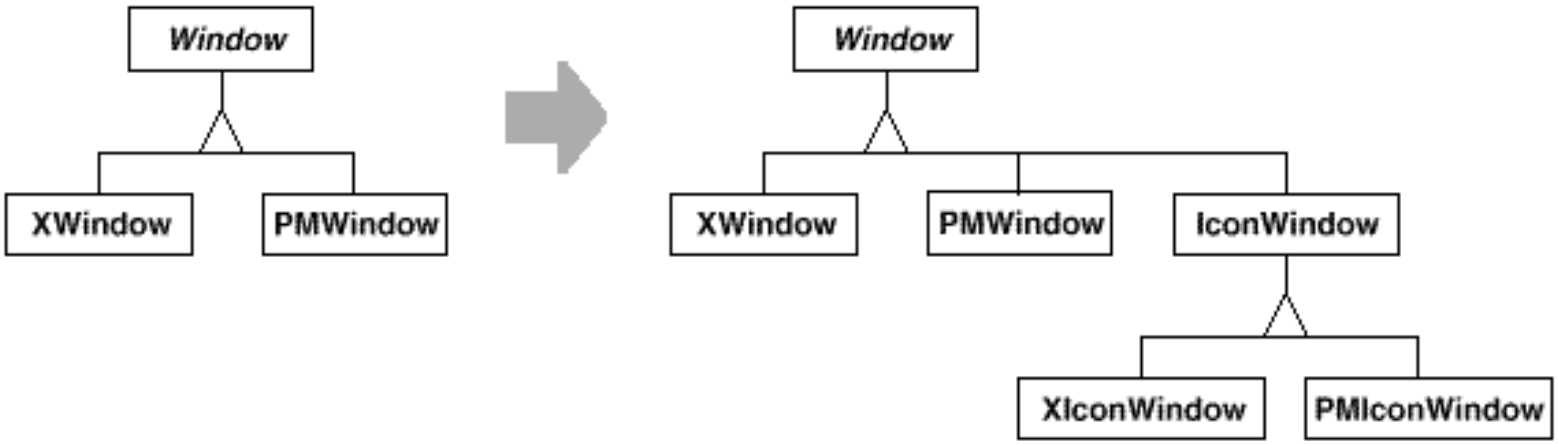
- Example :



- To use when :
  - a class exists, but its interface does not match the one you need.
  - you need to use several existing subclasses, but it's impractical to adapt their interface by subclassing every one. An object adapter can adapt the interface of its parent class.

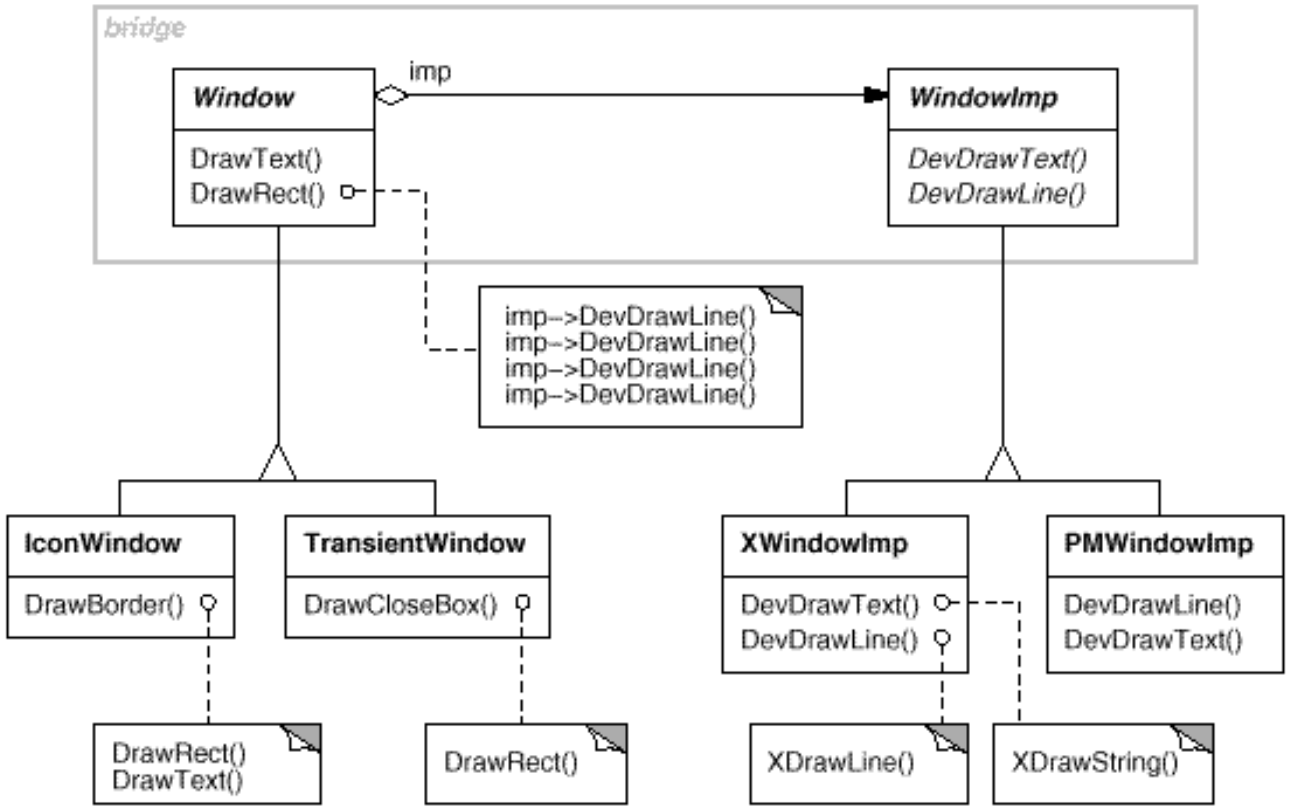
Bridge (1)

- Decouple an abstraction from its implementation so that the two can vary independently
- problem :



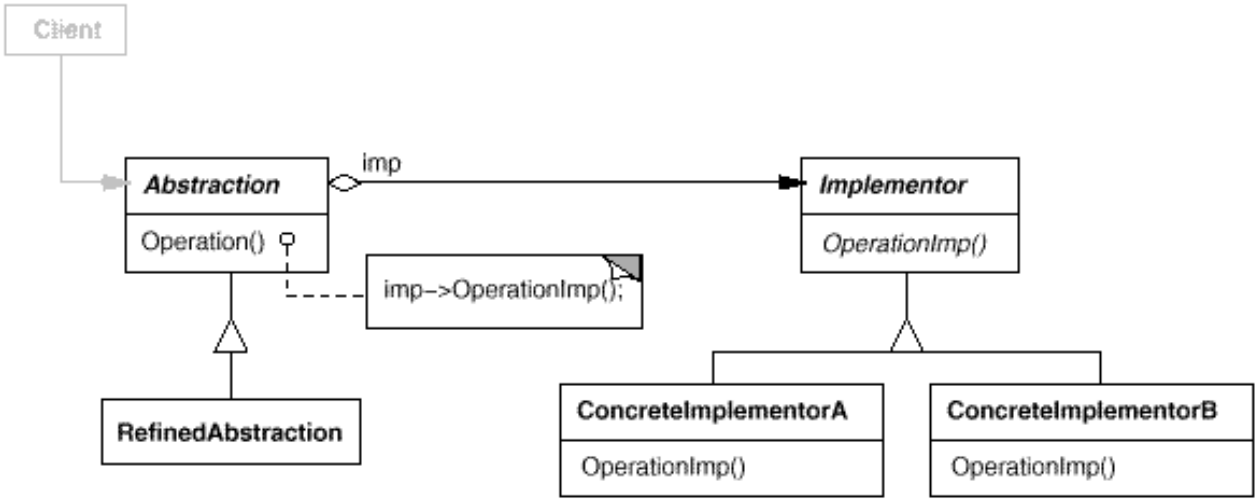
Bridge (2)

- Solution with the Bidge pattern :



Bridge (3)

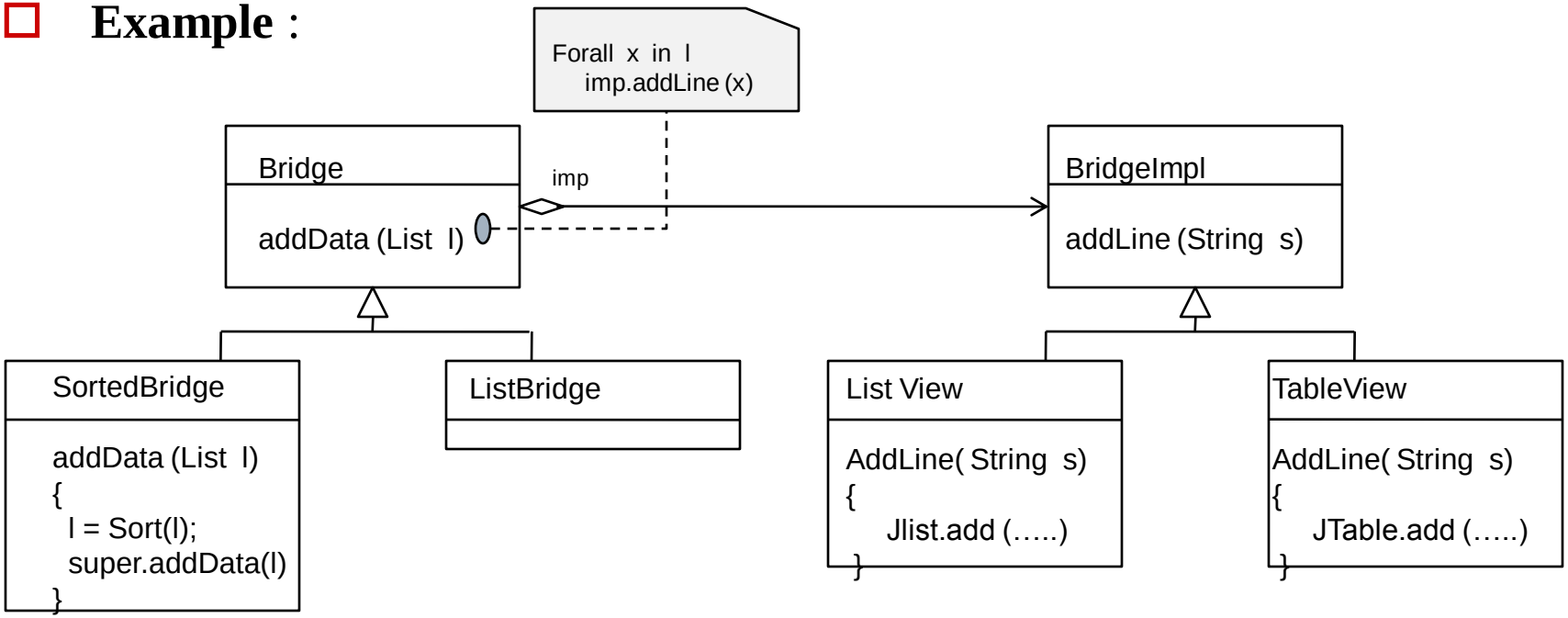
- Structure :



- To use when :
  - to avoid a permanent binding between an abstraction and its implementation. (example, the implementation must be selected or switched at run-time).
  - both the abstractions and their implementations are independently extensible by subclassing.
  - changes in the implementation of an abstraction should have no impact on clients; that is, their code should not have to be recompiled.

Bridge (4)

- **Example** :



Structural Design Patterns ملخص عن أهداف الـ

- **Composite** = Compose objects into tree structures to represent part-whole hierarchies. Composite lets clients treat individual objects and compositions of objects uniformly.
- **Adapter** = Convert the interface of a class into another interface clients expect. Adapter lets classes work together that couldn't otherwise because of incompatible interfaces.
- **Bridge** = Decouple an abstraction from its implementation so that the two can vary independently.
- **Decorator** = Attach additional responsibilities to an object dynamically. Decorators provide a flexible alternative to subclassing for extending functionality.
- **Façade** = Provide a unified interface to a set of interfaces in a subsystem. Facade defines a higher-level interface that makes the subsystem easier to use.
- **Flyweight** = Use sharing to support large numbers of fine-grained objects efficiently.
- **Proxy** = Provide a surrogate or placeholder for another object to control access to it.