**Façade**
(simply relays calls to a set of methods
an example is the Session Enterprise Java Beans)

**Bridge**
(avoid the following situation)

One must create two distinct sources, but their sole difference is
"extends Window"
versus "extends NewWindow"
Two hierarchies interleaved
Bridge (the red-blue lines)

Either red or blue is instantiated, at run time

OS Implementation

- YWindow
  - OS Y
- XWindow
  - OS X

OS independent hierarchy

- Window
  - abstract class
  - SimpleWindow
  - IconWindow
  - FramedIconWindow

BridgePatternExample.java

(as in the CVS, project Patterns)

- Shape
  - drawingAPI
  - resizeByPercentage()

- DrawingAPI1
  - drawCircle()

- DrawingAPI2
  - drawCircle()

- CircleShape
  - drawingAPI
  - draw()

<<exclusifs>>

another variant
Factories

- **Abstract Factory**
  
  *The client gets a factory corresponding to a specified domain and then generates objects by calling it. All generated objects belong to the specified domain.*

- **Factory Method**
  
  *The client gets objects that are created according to some individual characteristic. The objects of the same application may belong to different domains.*

**Abstract factories** are implemented with the help of **factory methods**!

---

**Abstract Factory**

- **Client**
- **Factory1**
  - `create()`
- **Factory2**
  - `create()`
- **ConcreteWindow1**
- **ConcreteWindow1**

Of course the factories also inherit a common interface, not shown here.
Abstract factory (with interfaces)

AbstractFactory
- staticCreateFactory()
- CreateProductA()
- CreateProductB()

ConcreteFactory1
- CreateProductA()
- CreateProductB()

ConcreteFactory2
- CreateProductA()
- CreateProductB()

AbstractProductA

AbstractProductB

ConcreteProductA1

ConcreteProductA2

ConcreteProductB1

ConcreteProductB2

Client

Factory Method

Client

AbstractCreator
- factoryMethod(param)

ConcreteCreator
- factoryMethod(param)

Product

create object of type x or y

The same factory creates different objects
A Simple Factory Method Pattern

class Complex {
    public static Complex fromCartesian(double real, double imag) {
        return new Complex(real, imag);
    }

    public static Complex fromPolar(double rho, double theta) {
        return new Complex(rho * java.lang.Math.cos(theta),
            rho * java.lang.Math.sin(theta));
    }

    private Complex(double a, double b) {
        // ...
    }
}

One method for each kind of object because it is not possible to differentiate them by their parameters.

Exercice 3

Créer une factory qui génère les composants suivants:

1. Factory de fenêtres GUI
2. Factory d'objets qui utilisent la console d'Eclipse

- La factory à créer est spécifiée dans une propriété
- Les méthodes read du GUI et de la console sont bloquantes
Properties

// From the user properties:
Properties myProperties = new Properties();
FileInputStream
    in = new FileInputStream("myProp.properties");
myProperties.load(in);
in.close();
String dType = myProperties.getProperty("displayType");

// From the system properties:
dtype = System.getProperty("terminalType");

Synchronisations of threads

If the call to notify() is done before the call to wait() the wait() will wait the next call to notify(). One should thus memorize the signal in an attribute
Java Reflection

**Instanciation**

```java
String dType = "pack.Classname";
Class dFactoryClass = Class.forName(dType);
dFactoryClass.newInstance();
```

**Appel d'une classe statique**

```java
Class dFactoryClass = Class.forName(dType);
Method meth = dFactoryClass.getMethod("getDisplayFactory", new Class[0]);
DisplayFactory dFactory = (DisplayFactory)meth.invoke(null, new Object[0]);
```

Drawback of Reflection

- The availability of the class called in by reflection is only checked at runtime.
- In order to clean an application, an IT team in a bank had eliminated all programs that had not been used for 6 months. At the December, 31 closure, the programs had been destroyed ….
Flyweight
(poids mouche, poids plume)

- the program must handle lists containing many pieces
- many of them are similar
- using an object for each one would use up much memory space
- a typical example is a text, in which each letter has specific parameters (font, face, size…)

```
HashMap<SpecialChar>

GraphicCharFactory
pool
get (char, font): SpecialChar

ArrayList<SpecialChar>
text (only pointers)
```

'get' returns the pointer to the SpecialChar, and creates it if it does not exist already.
Flyweight

The GoF book describes the flyweight within a hierarchy, but the hierarchy is independent from the flyweight pattern.

The Flyweight is used only at the line level.

```
GraphicCharFactory
| pool
| get(character, fontFace)

FlyweightExample
| text
| main()
```

```
<<HashMap>>

GraphicChar
| String fontFace
| printAtPosition(c, x)
```

```
<<ArrayList>>

FlyweightExample.java
(as in CVS, project Patterns)"
Each time the subject is called, it calls a method in all the observers.

Thus at the end of each method called in the subject, there is a call to the list of observers.

The observers are registered, sometimes by themselves.

The observers must of course have a common interface.

They are also called listeners!

```java
void notify() {
    for (Observer o: pool) {
        o.update();
    }
}
```
Observer

Observer
– update()
– it often register itself in the subject

Subject
– notify()
– addObserver()
– removeObserver()

Template (uses inheritance)

Algorithm expressed as a sequence of operations.
These operations must be replaced easily in the source

operation 1
operation 2
...


Template Method  (trivial)

Different concrete class implement different algorithms with the same sequence of operations.