Design Patterns

Christopher Alexander is an architect noted for his theories about design. He produced and validated a pattern language designed to empower any human being to design and build at any scale.

The concept of design patterns has been reused in IT by E. Gamma, R. Helm, R. Johnson and J. Vlissides: also known as the Gang of Four or GoF. They have sold 500,000 copies of their book.

Abstract factory, Adapter, Composite, Decorator, Factory methods, Observer, Strategy, Template method…

E. Gamma is currently heading the Eclipse project

Other authors have written books about patterns
Design Patterns (basic concepts)

class SomeClass extends SuperClass{
    private String str;
    public String getString() {
        return str;
    }
}

Signature
All signatures = type (or subtype)
The whole = class (subclass or derived class)
The subclass inherits (extends) the superclass

Design Patterns

• An object is an instance of a class
• An abstract class has abstract operations
• A concrete class is one that is not abstract
• A concrete class that inherits an abstract class must override the abstract operations
• A first-class operator or function is one that can be defined during the execution of a program, stored in a variable, passed as arguments to other functions, and returned as the values of other functions.
Inheritance / Composition / Forward

class SuperClass {
    private String name;
    public String getName() { return name; }
}
class InheritanceClass extends SuperClass{
}

    ci = new InheritanceClass();
    ci.getName();

Inheritance / Composition / Forward

class SuperClass {
    private String name;
    public String getName() { return name; }
}
class CompositionClass {
    SuperClass element;
}

    cc = new CompositionClass()
    cc.element.getName();
Inheritance / Composition / Forward

class SuperClass {
    private String name;
    public String getName() { return name; }
}
class ForwardClass {
    SuperClass element;
    public String getName() {
        return element.getName();
    }
}

Cd = new ForwardClass();
cd.getName();

Delegation = composition + forward + environment

class SuperClass {
    private String name;
    public String getName(DelegationClass env) {
        return env.xxxxx+name;
    }
}
class DelegationClass {
    SuperClass element;
    String xxxxx;
    public String getName() { return element.getName(this); }
}

// see GoF's state design pattern for an example
Two GoF principles

• Program to an interface, not an implementation

• Favor object composition over class inheritance

Declare interfaces, not classes,

```java
public interface Interface {
    public String method();
}

public class SubClass implements Interface{
    public String method() {
        return null;
    }
}
```
…debugging and maintenance become difficult!

Example:

```java
public class Application {
    Interface intf;

    public static void main(String[] args) {
        Application a = new Application();
        a.intf = new SubClass();
        a.execute(args);
    }

    void execute(String[] args) {
        intf.method();
    }
}
```

Where do I find the source of method()?

Note that you don’t need to implement a new interface to insert new attributes!! If an attribute is added to an object, it does not disturbs the other uses of the object.

Refactoring may also be called to rescue!
The GoF patterns

Design Patterns –
Elements of Reusable Object-Oriented Software,
E. Gamma…

Inheritance (in GoF’s book)
Singleton

Only one singleton is instantiated

One does not know the order in which the Class\textsubscript{i} are instantiated

All Class\textsubscript{i} are not always present, they have no references to each other and there may be other such classes later

Who does create the Singleton?

---

public class ClassicSingleton {
    private static ClassicSingleton instance = null;

    protected ClassicSingleton() {     // Exists only to defeat instantiation  }

    public static ClassicSingleton getInstance() {
        if (instance == null) {
            instance = new ClassicSingleton();
        }
        return instance;
    }
}

// from Javaworld
Singletons

- Global variable!
- Don’t overuse them!
- Only for data that are meaningful in the whole program
- Not more than a few singletons in a program

(Complex) example of use

*Threadlocal* is a singleton that may be attached to the current thread, the one that executes the code of the current method.

It is managed by the JVM.
ThreadLocal singleton

The class introduced into ThreadLocal can be accessed in the following way:

```
x = HolderObject.get()  x = HolderObject.get()  x = HolderObject.get()
Thread1

y = HolderObject.get()  y = HolderObject.get()
Thread2
```

Thanks to the JVM, there is one singleton per thread,

```
x != y,        x always the same, y always the same
```

(Complex) example of use

```java
class HolderObject {
    public static ThreadLocal<MyObject> currentObject = new ThreadLocal<MyObject>() {
        protected synchronized MyObject initialValue() {
            return new MyObject();
        }
    };

    public static MyObject get() {
        return currentObject.get();
    }
}
```

// At the initialization, the object ThreadLocal calls method initialValue(),
// defined by the developer. This method returns an object.
// This object is then stored in ThreadLocal located in currentObject.
// Moreover, the JVM saves and restores this object at each context switch
(Complex) example of use

class HolderObject {
    public static ThreadLocal< HolderObject >
    currentObject = new ThreadLocal< HolderObject >() {
        protected synchronized HolderObject initialValue() {
            return new HolderObject();
        }
    }
    public static HolderObject get() {
        return currentObject.get();
    }
    Type userAttribute = "initval";
}

// Usually the object that is stored is the HolderObject itself

ThreadLocal

Can be used to replace parameters in a method call, but in very specific situations, such as

- Process support (task control block, semaphore, identifier)
- Transaction manager (EJB3)
Exercice 2

Créer un object dont la méthode `run` est exécutée sur un thread.
Cette méthode contient une boucle infinie et affiche toutes les secondes un identificateur unique mémorisé dans le singleton `ThreadLocal` du thread.
Créer deux de ces objets et lancez-les sur deux threads.
Pour générer des identificateurs uniques, utiliser un deuxième singleton (global celui-là).
Attention, sa méthode doit utiliser `synchronized`!

```java
class Singleton {
    static Singleton single = null;
    int i = 10;
    public static Singleton instantiate() {
        if (single == null) {
            single = new Singleton();
        }
        return single;
    }
    public int get() {
        return i++;
    }
}
```

Quand on a obtenu un ID, il ne faut plus utiliser le singleton depuis le même thread, car les autres objets feront changer le contenu de la variable utilisée pour générer les Ids.

```java
new Thread(new Thr()).start();
new Thread(new Thr()).start();
```
**Creation d’un objet ThreadLocal**

class HolderObject {
    public static ThreadLocal<HolderObject> currentObject = new ThreadLocal<HolderObject>() {
        protected synchronized HolderObject initialValue() {
            return new HolderObject();
        }
    };
    public static String getString() {
        return currentObject.get().str;
    }
    public static void setString(String s) {
        currentObject.get().str = s;
    }
    String str = null;
}


**Class or object adapter**

**Goal:**

The **class** adapter implements a new interface that provides a new functionality or an existing functionality under a new name.

In the class adapter, the code that implements or support the new functionality is called from an auxiliary object that extends the object with the available functionality.

The **object** adapter **delegates** the call to the available object.
Design Pattern: Adapter
(Inheritance / Composition / Delegation)+Interface

Existing class
Required interface
Adapter
Just forwards the calls

Design Pattern: Adapter

Existing class
Required interface
Adapter
Just forwards the calls

Existing object
Existing class
Two different possibilities

Also possible
Java implementation of a class adapter

```java
interface Stack {
    void push (Object o);
}

/* Double Linked List */
class DList {
    public void insertTail (Object o) { ... }
}

/* Adapt DList class to Stack interface */
class DListImpStack extends DList implements Stack {
    public void push (Object o) {
        insertTail (o);
    }
    // forwards the calls
}
```

Java implementation of a object adapter

```java
interface Stack {
    void push (Object o);
}

/* Double Linked List */
class DList {
    public void insertTail (Object o) { ... }
}

/* Adapt DList class to Stack interface */
class DListImpStack extends DList implements Stack {
    DList dl = new DList ();
    public void push (Object o) {
        dl.insertTail (o);
    }
}
```
Composite: tree structure
(one must be able to walk through and add/remove groups or leaves in the same way)

Model of a diagram
for a graphical application

```
Composite
  Component
    Operation()
    Add(Component)
    Remove(Component)
    getChild(int)

Leaf
  Operation()

Composite
  super group
    group
      circle
      square
    square
    line
```

```
Composite
  Component
    Operation()
    Add(Component)
    Remove(Component)
    getChild(int)

Leaf
  Operation()

Composite
  super group
    group
      circle
      square
    square
    line
```
Implementation of the Composite

abstract class GraphicComponent {
    abstract public void print();
}

class CompositeGraphic extends GraphicComponent {
    private ArrayList<Graphic> mChildGraphics = new ArrayList<Graphic>();
    public void print() {
        for (Graphic graphic : mChildGraphics) {
            graphic.print();
        }
    }
}

class Ellipse extends GraphicComponent {
    public void print() {
        System.out.println("Ellipse");
    }
}

- All nodes are GraphicComponent with a print() method
- root.print() prints all leaves

Composite: object diagram

Each object extends abstract class GraphicComponent to make all objects compatible
Composite: other implementation details

• Parent references may be handled

• But if parents are multiple reverse path ambiguous → Flyweight

Composite: other implementation details

Where to define the children list and the getChild method?

• Component class (abstract class)
  – all elements similar (transparency)
  – but one may try to retrieve a child from a leaf

• Composite class
  – one can check that elements are retrieved only from composite (safety)
  – But elements are dissimilar
Decorator

BufferedReader br =
    new BufferedReader(
        new InputStreamReader(
            new ByteArrayInputStream(
                “A text to read”.getBytes()
            )
        )
    );

Decorator (class diagram)
Decorator

- More flexibility than inheritance
- Several identities (addresses)!
- Keep interface simple
- Decorator’s abstract class optional
- For more complex cases → Strategy

x = new HScrollDecorator (new VScrollDecorator (new SimpleWindow()));
Decorator

HScrollDecorator
Constructor(x)
{    super(x)    }
draw()

draw()

VScrollDecorator
Constructor(x)
{    super(x)    }
draw()

draw()

SimpleWindow
draw()

generateDescription()

Public String generateDescription()
{
    return
    decoratedWindow.getDescription()
    + ", including vertical scrollbars";
}

Public String generateDescription()
{
    return "simple window";
}

Public String generateDescription()
{
    return
decoratedWindow.getDescription()
    + ", including horizontal scrollbars";
}
Decorator  (without the intermediary abstract classes)

```
Decorator
    HScrollDecorator
        Constructor(x) {
            super(x)
            decorWindow = x
        }

Decorator
    VScrollDecorator
        Constructor(x) {
            super(x)
            decorWindow = x
        }

SimpleWindow
```

```java
Window (interface) draw()

HScrollDecorator Window decorWindow
Constructor(x) {
    super(x)
    decorWindow = x
}

VScrollDecorator Window decorWindow
Constructor(x) {
    super(x)
    decorWindow = x
}

SimpleWindow
```