Computer Science Introductory Course MSC - Software engineering

Lecture 3: Design patterns

Pablo Oliveira <pablo@sifflez.org>

ENST

Outline

1 Introduction

- What is a design pattern ?
- Categories of design patterns

2 Common Design patterns

- Iterator
- Decorator
- Singleton
- Visitor
- Factory
- Proxy

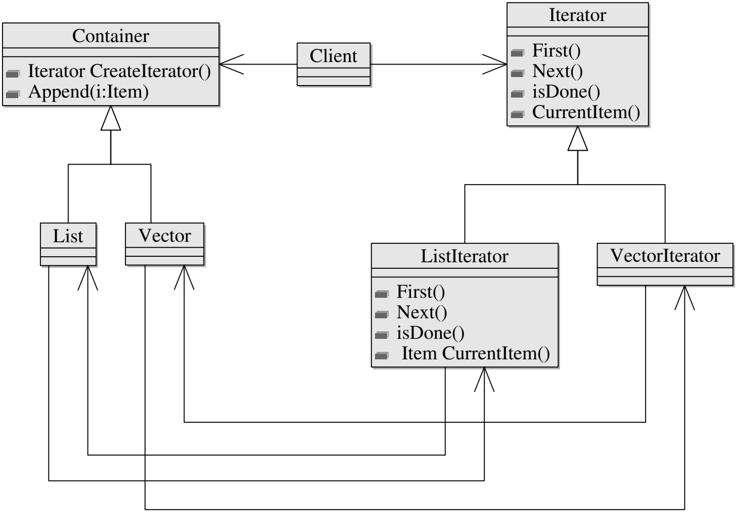
What is a design pattern ?

- Proposed by architect C. Alexander in 70ths.
- General reusable solution to a recurring problem.
- Must be adapted to each concrete case.
- Patterns allow to communicate complex principle using a common vocabulary.
- Describe software abstractions.
- Each programming language provides some patterns already included as idioms :
  - In java : encapsulation, subclassing, etc...
- Use design patterns wisely (sometimes they only clutter the problem), always adapt them to your particular problem and context.

Categories of design patterns

creational	*structural*	*behavioural*
builder	adapter	chain of responsability
factory	bridge	command
prototype	composite	interpreter
singleton	decorator	iterator
	façade	mediator
	flyweight	memento
	proxy	observer
		state
		strategy
		visitor

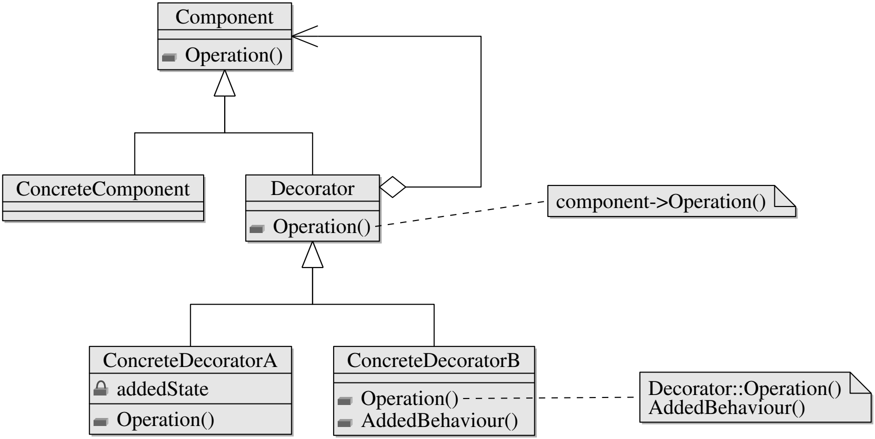
Iterator (UML)



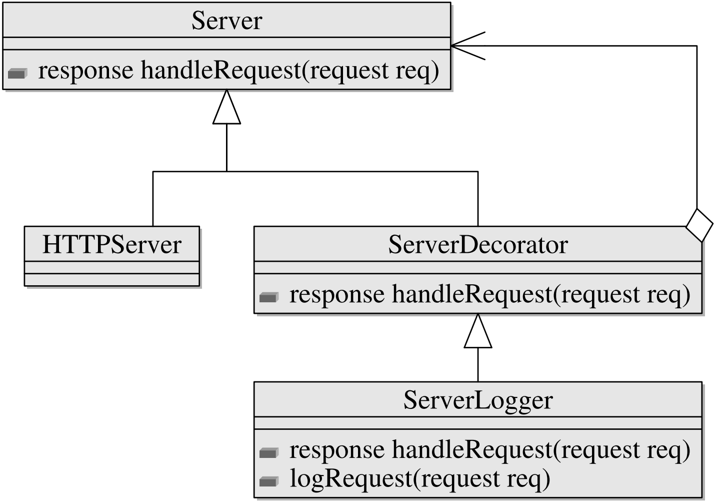
Iterator (Java)

```java
class Vector implements Container {
  private Item[] elements;
  private int last = -1;
  Vector(int size){elements = new Item[size];}
  Item get(int pos) {return elements[pos];}
  int getLast() {return last;}
  void Append(Item i){elements[++last] = i;}
  Iterator CreateIterator()
    {return new VectorIterator(this);}
}
class VectorIterator implements Iterator {
  private Vector v;
  private int cursor;
  VectorIterator(Vector v) {this.v = v;First();}
  void First() {cursor = 0;}
  void Next() {cursor++;}
  boolean isDone() {return cursor == v.getLast();}
  Item CurrentItem() {return v.get(cursor);}
}
```

Decorator (UML)



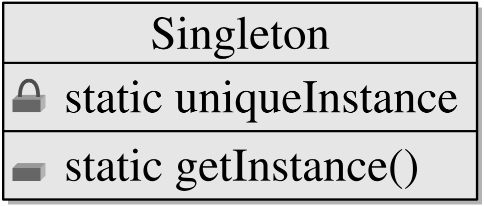
Decorator Example (UML)


Server
response handleRequest(request req)
HTTPServer
ServerDecorator
response handleRequest(request req)
ServerLogger
response handleRequest(request req)
logRequest(request req)

Decorator Example (Java)

```
interface Server {
  response handleRequest(request req);
}
abstract class ServerDecorator implements Server {
  protected Server decoratedServer;
  ServerDecorator(Server s) {decoratedServer = s;}
}
class ServerLogger extends ServerDecorator {
  ServerLogger(Server s){super(s);}
  response handleRequest(request req) {
    logRequest(req);
    return decoratedServer.handleRequest(req);
  }
  void logRequest(request red) {
    System.out.println
      ("Server␣got␣request␣from␣" + req.from);
  }
}
```

Singleton (UML)



Singleton

static uniqueInstance

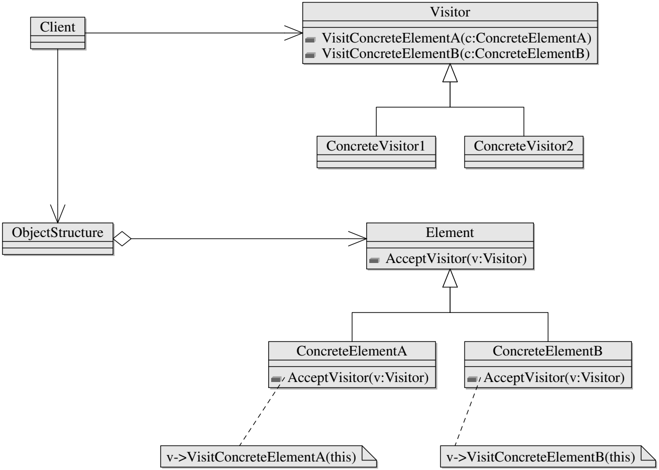
static getInstance()

The singleton pattern ensures that only a single instance of an object is ever created.

Singleton Example (Java)

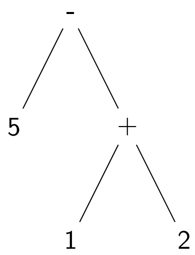
```java
public class Singleton {
   private static ClassicSingleton instance = null;
   protected Singleton() {} // no instantiation
   public static ClassicSingleton getInstance() {
      if(instance == null) {
         instance = new Singleton();
      }
      return instance;
   }
}
```

Visitor (UML)



The visitor pattern decouples the iteration over a structure and the operations made during the iteration

Visitor Example : Tree Visitor



We have a tree structure, and want to perform various algorithms on it. Each algorithm should be described in its own class...

Visitor Example (Java) : Object Structure

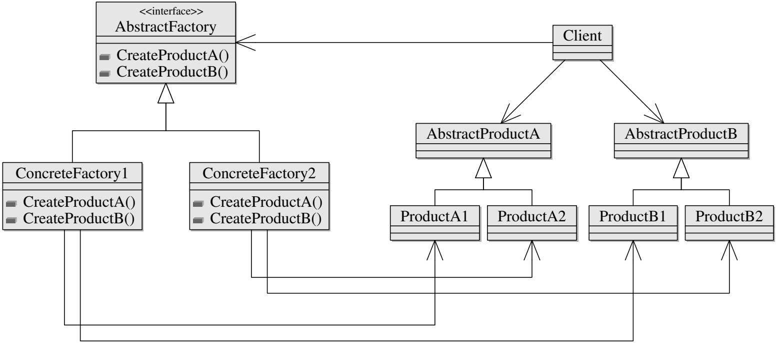
```java
abstract class TreeNode {
  TreeNode left, right;
}
class PlusN extends TreeNode {
  void acceptVisitor(TreeVisitor v)
    {v.visitPlus(this);}
}
class MinusN extends TreeNode {
  void acceptVisitor(TreeVisitor v)
    {v.visitMinus(this);}
}
class IntegerN extends TreeNode {
  Integer value;
  void acceptVisitor(TreeVisitor v)
    {v.visitInteger(this);}
}
```

Visitor Example (Java) : TreeVisitor

```
interface TreeVisitor {
  int visitInteger(IntegerN i);
  int visitPlus(PlusN p);
  int visitMinus(MinusN m);
}

class ReduceVisitor extends TreeVisitor {
  Integer value;
  void visitInteger(IntegerN i)
    {value = i.value;}
  void visitPlus(PlusN p)
    {
      p.left.acceptVisitor(this);
      Integer first = value;
      p.right.acceptVisitor(this);
      value += first;
    }
  ...
}
```

Factory (UML)

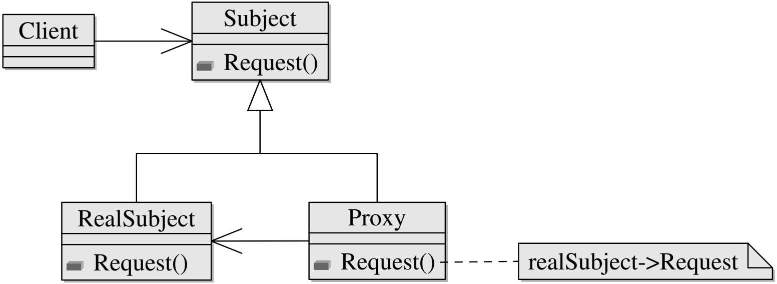

<<interface>>
AbstractFactory
CreateProductA()
CreateProductB()
Client
AbstractProductA
AbstractProductB
ConcreteFactory1
CreateProductA()
CreateProductB()
ConcreteFactory2
CreateProductA()
CreateProductB()
ProductA1
ProductA2
ProductB1
ProductB2

Factory Example (Java)

```
interface Button{};
interface TextBox{};

interface GUIFactory{
  public Button createButton();
  public TextBox createTextBox();
}
WindowsFactory implements GUIFactory{
  public Button createButton()
    {return new WindowsButton();}
  public TextBox createTextBox()
    {return new WindowsTextBox();}
}
class LinuxFactory implements GUIFactory {
  public Button createButton()
    {return new LinuxButton();}
...}
class Application {
  public Application(GUIFactory factory){
    Button button = factory.createButton();
      button.paint();
    }
  public static void main(String args[]){
    if (onWindows())
       new Application(new WindowsFactory());
    else
       new Application(new LinuxFactory());
  }}
```

Proxy(UML)


Client
Subject
Request()
RealSubject
Request()
Proxy
Request()
realSubject->Request

Exercice : Remote objects

- We are designing an application that manages a pool of objects of class Entry, some of them are local and some of them are on a remote server, we want to create a Proxy that enables us to access an Entry instance without worrying if the object is local or remote.
- You have already written these classes :

```
class Entry{
  EntryId uniqueId;
  String getData();
  void setData(String s);
}

class RemoteServer{
  public static String getData(EntryId id);
  public static void   setData(EntryId id, String s);
}
```

- Design a RemoteProxy class that makes remote/local access transparent.