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

<table>
<thead>
<tr>
<th>creational</th>
<th>structural</th>
<th>behavioural</th>
</tr>
</thead>
<tbody>
<tr>
<td>builder</td>
<td>adapter</td>
<td>chain of responsibility</td>
</tr>
<tr>
<td>factory</td>
<td>bridge</td>
<td>command</td>
</tr>
<tr>
<td>prototype</td>
<td>composite</td>
<td>interpreter</td>
</tr>
<tr>
<td>singleton</td>
<td>decorator</td>
<td>iterator</td>
</tr>
<tr>
<td></td>
<td>façade</td>
<td>mediator</td>
</tr>
<tr>
<td></td>
<td>flyweight</td>
<td>memento</td>
</tr>
<tr>
<td></td>
<td>proxy</td>
<td>observer</td>
</tr>
<tr>
<td></td>
<td></td>
<td>state</td>
</tr>
<tr>
<td></td>
<td></td>
<td>strategy</td>
</tr>
<tr>
<td></td>
<td></td>
<td>visitor</td>
</tr>
</tbody>
</table>
Iterator (UML)

Container
- Iterator CreateIterator()
- Append(i:Item)

Client

Iterator
- First()
- Next()
- isDone()
- currentItem()

List
- First()
- Next()
- isDone()
- currentItem()

Vector
- First()
- Next()
- isDone()
- currentItem()

ListIterator

VectorIterator
Iterator (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 Example (UML)

- Server
  - response handleRequest(request req)

- HTTPServer

- ServerDecorator
  - response handleRequest(request req)

- ServerLogger
  - response handleRequest(request req)
  - logRequest(request req)
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 req) {
    System.out.println("Server got request from " + req.from);
  }
}
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;
    }
}
```
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...
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

```java
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)

AbstractFactory
- CreateProductA()
- CreateProductB()

ConcreteFactory1
- CreateProductA()
- CreateProductB()

ConcreteFactory2
- CreateProductA()
- CreateProductB()

Client
  - AbstractProductA
  - AbstractProductB

  - ProductA1
  - ProductA2

  - ProductB1
  - ProductB2
Factory Example (Java)

```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());
    }
}
Common Design patterns

Proxy

Proxy(UML)

Client

Subject

RealSubject

Proxy

realSubject->Request

Client 

Subject

RealSubject

Proxy

realSubject->Request

Request()
Exercise: 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:

```java
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.
Some of the UML patterns in these slides were generated using the MetaUML Gallery of Patterns Copyright (C) 2005 Radu-George Radulescu, under the GNU GPL v 2.0.

This work is licensed under a Creative Commons Attribution-Noncommercial-Share Alike 3.0 Unported License.