MTAT.03.094
Software Engineering

Lecture 08: Architecture and Design II

Dietmar Pfahl
email: dietmar.pfahl@ut.ee

Fall 2020
Schedule of Lectures

Week 01: Introduction to SE
Week 02: Requirements Engineering I
Week 03: Requirements Engineering II
Week 04: Analysis
Week 05: Development Infrastructure
Week 06: Continuous Development and Integration
Week 07: Project Estimation / Architecture and Design I
Week 08: Architecture and Design II
Week 09: Verification and Validation I
Week 10: Verification and Validation II
Week 11: Refactoring (and TDD)
Week 12: Agile/Lean Methods
Week 13: Industry Guest Lecture
Week 14: Course wrap-up, review and exam preparation
Week 15: Reserve time slot (no lecture scheduled as of today)
Structure of Lecture 08

• Why Architecture?
• Terminology: Architect, Architecting, Architecture
• Viewpoints and View Models
• Notation
• Architecture & Design Patterns
Design Patterns, Styles, and DSSAs

Design Patterns

- Design Patterns – Elements of Reusable Object-Oriented Software by Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides, 1994 (1st ed.)
Design Patterns

• A design pattern is a way of reusing abstract knowledge about a problem and its solution.

• A pattern is a description of the problem and the essence of its solution.

• It should be sufficiently abstract to be reused in different settings.

• Pattern descriptions usually make use of object-oriented characteristics such as inheritance and polymorphism.

ELEMENTS:

Name
A meaningful pattern identifier.

Problem description.

Solution description (might have an example)
Not a concrete design but a template for a design solution that can be instantiated in different ways.

Benefits and Consequences
The results and trade-offs of applying the pattern.
The Observer Pattern

Name: Observer

Problem description

• Situations where multiple displays of state are needed.

Solution description

• Separates the display of object state from the object itself.
• See UML description.

Consequences

• Optimisations to enhance display performance are difficult.
The Observer Pattern – Inbuilt in JDK

Inbuilt Observer Pattern in Java:
Java has an inbuilt Observer pattern implementation using:
• Class java.util.Observable (represents Subject [=publisher])
• Interface java.util.Observer (represents an Observer [=subscriber])

Concrete Observers in Java need to implement the Observer interface, whereas concrete Subject needs to extend Observable to provide its own notification logic.

Source:
http://www.javabrahman.com/design-patterns/observer-design-pattern-in-java/
The Observer Pattern – Example Implementation (Class Diagram)

Source: http://www.javabrahman.com/design-patterns/observer-design-pattern-in-java/
The Observer Pattern – Example Implementation (Code)

```java
import java.util.Observable;
import java.util.Observer;

public class Publisher extends Observable{
    public void changeStateTo(String newStateName){
        this.setChanged();
        this.notifyObservers(newStateName);
    }
}

public class Subscriber1 implements Observer{
    String currentPublisherState;
    @Override
    public void update(Observable o, Object arg) {
        System.out.println("New state received by Subscriber 1:");
        this.currentPublisherState=(String)arg;
    }
}

public class Subscriber2 implements Observer{
    String currentPublisherState;
    @Override
    public void update(Observable o, Object arg) {
        System.out.println("New state received by Subscriber 2:");
        this.currentPublisherState=(String)arg;
    }
}

public class Client {
    public static void main(String args[]){
        Publisher publisher=new Publisher();
        publisher.addObserver(new Subscriber1());
        publisher.addObserver(new Subscriber2());
        publisher.changeStateTo("assigned A");
        publisher.changeStateTo("assigned B");
        publisher.changeStateTo("assigned C");
    }
}
```

Source:
http://www.javabrahman.com/design-patterns/observer-design-pattern-in-java/
The Observer Pattern – Example Implementation (Code)

```java
import java.util.Observable;
import java.util.Observer;

public class Publisher extends Observable{
    public void changeStateTo(String newStateName){
        this.setChanged();
        this.notifyObservers(newStateName);
    }
}

public class Subscriber1 implements Observer{
    String currentPublisherState;

    @Override
    public void update(Observable o, Object arg) {
        System.out.println("New state received by Subscriber 1:");
        System.out.println(arg.toString());
        this.currentPublisherState=(String)arg;
    }
}

public class Subscriber2 implements Observer{
    String currentPublisherState;

    @Override
    public void update(Observable o, Object arg) {
        System.out.println("New state received by Subscriber 2:");
        System.out.println(arg.toString());
        this.currentPublisherState=(String)arg;
    }
}

public class Client {
    public static void main(String args[]){
        Publisher publisher=new Publisher();
        publisher.addObserver(new Subscriber1());
        publisher.addObserver(new Subscriber2());
        publisher.changeStateTo("assigned A");
        publisher.changeStateTo("assigned B");
        publisher.changeStateTo("assigned C");
    }
}
```

Output:

New state received by Subscriber 2:assigned A
New state received by Subscriber 1:assigned A
New state received by Subscriber 2:assigned B
New state received by Subscriber 1:assigned B
New state received by Subscriber 2:assigned C
New state received by Subscriber 1:assigned C
The Observer Pattern – Example Implementation (Code)

Lets quickly go through what’s there in Java’s example’s class diagram & corresponding code –

- Publisher is the Subject. It extends java.util.Observable.
- Subscriber1 & Subscriber2 are the Observers. They implement java.util.Observer.
- Client first initiates Publisher. It then adds one instance each of Subscriber1 & Subscriber2 to Publisher’s list of Observers.
- Client then invokes method changeStateTo() with new state value as “assigned”. Internally, Publisher then initiates notifyObservers() with this new state value. Before notifying, Publisher calls setStateChanged() which is a requirement of the java’s observer pattern implementation.
- update() methods of Subscriber1 and Subscriber 2 are called internally by the notifyObservers() method and the new state value received by both in the parameter arg is printed.
Note on Observer Pattern in JDK

Observer interface is deprecated since Java 9

- One of its cons is that Observable isn't an interface but a class, that's why subclasses can't be used as observables.
- Also, a developer could override some of the Observable's synchronized methods and disrupt their thread-safety.

Use instead `PropertyChangeListener` interface

- see https://www.baeldung.com/java-observer-pattern
Three Types of Patterns

**Creational patterns:**
- Deal with initializing and configuring classes and objects

**Structural patterns:**
- Deal with decoupling interface and implementation of classes and objects
- Composition of classes or objects

**Behavioral patterns:**
- Deal with dynamic interactions among societies of classes and objects
- How they distribute responsibility

<table>
<thead>
<tr>
<th>Scope of pattern</th>
<th>Creational</th>
<th>Structural</th>
<th>Behavioral</th>
</tr>
</thead>
<tbody>
<tr>
<td>Class-level patterns</td>
<td>Factory method</td>
<td>Adapter</td>
<td>Interpreter</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Template Method</td>
</tr>
<tr>
<td>Object-level patterns</td>
<td>Abstract Factory</td>
<td>Adapter</td>
<td>Chain of Responsibility</td>
</tr>
<tr>
<td></td>
<td>Builder</td>
<td>Bridge</td>
<td>Command</td>
</tr>
<tr>
<td></td>
<td>Prototype</td>
<td>Composite</td>
<td>Iterator</td>
</tr>
<tr>
<td></td>
<td>Singleton</td>
<td>Decorator</td>
<td>Mediator</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Façade</td>
<td>Memento</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Proxy</td>
<td>Flyweight</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Observer</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>State</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Strategy</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Visitor</td>
</tr>
</tbody>
</table>
Three Types of Patterns

<table>
<thead>
<tr>
<th>Scope of pattern</th>
<th>Type of pattern</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td><strong>Creational</strong></td>
</tr>
<tr>
<td></td>
<td><strong>Structural</strong></td>
</tr>
<tr>
<td></td>
<td><strong>Behavioral</strong></td>
</tr>
<tr>
<td>Class-level patterns</td>
<td>Factory method</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>Object-level patterns</td>
<td>Abstract Factory</td>
</tr>
<tr>
<td></td>
<td>Builder</td>
</tr>
<tr>
<td></td>
<td>Prototype</td>
</tr>
<tr>
<td></td>
<td>Singleton</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Singleton**
Singleton

- Singleton pattern is one of the simplest design patterns in Java

- This pattern involves a single class which is responsible to create an object while making sure that only a single object gets created.

- This class provides a way to access its only object which can be accessed directly without need to instantiate the object of the class
Singleton

Useful for:

- Access to hardware resources such as printers (print spooler implemented as singleton multiple concurrent access and deadlocks)
- Producing logs (logger utility is implemented as singleton; again, avoids potential confusion due to multiple concurrent access to logger)
- Configuration file (if implemented as singleton allows only one access, loading the config data into the object once → performance advantage because it avoids repeated reading from config file)
- Cache (similar to config file; if cache object is created as a singleton object, it can have a global point of reference and for all future calls to the cache object the client application will use the in-memory object)
Singleton

Implementation:

Step 1

Create a Singleton Class.

`SingletonObject.java`

```java
public class SingletonObject {

    // create an object of SingletonObject
    private static SingletonObject instance = new SingletonObject();

    // make the constructor private so that this class cannot be
    // instantiated
    private SingletonObject(){
    }

    // Get the only object available
    public static SingletonObject getInstance(){
        return instance;
    }

    public void showMessage(){
        System.out.println("Hello World!");
    }
}
```
Singleton

Implementation:

Step 2

Get the only object from the singleton class.

SingletonPatternDemo.java

```java
public class SingletonPatternDemo {
    public static void main(String[] args) {
        //illegal construct
        //Compile Time Error: The constructor SingleObject() is not visible
        //SingleObject object = new SingleObject();

        //Get the only object available
        SingleObject object = SingleObject.getInstance();

        //show the message
        object.showMessage();
    }
}
```
Benefits of Design Patterns

• Design patterns enable large-scale reuse of software architectures and also help document systems
• Patterns explicitly capture expert knowledge and design tradeoffs and make it more widely available
• Patterns help improve developer communication
• Pattern names form a common vocabulary

More on Design Patterns: Mini-Tutorials by Derek Banas
https://www.youtube.com/playlist?list=PLF206E906175C7E07
Benefits of Design Patterns

In my Design Patterns Video Tutorial I will cover all of the most common design patterns. I will also explain when to use them and other topics on OOP design principles.
Further Reading

• Jan Bosch, Design & Use of Software Architectures, 2000.
• George Fairbanks: Just Enough Software Architecture, 2012.
• Erich Gamma et al., Design Patterns: Elements of Reusable Object-Oriented Software, 1995.
• Mary Shaw and David Garlan, Software Architecture; Perspectives of an Emerging Discipline, 1995.
• Richard Taylor et al.: Software Architecture, University of California at Irvine, Lecture 2011.
• Hans van Vliet: Software Architecture, Free University of Amsterdam, Lecture 2008.
Next Lecture

• Date/Time:
  • Friday, 30-Oct, 10:15-12:00

• Topic:
  • Verification and Validation (Testing) I

• For you to do:
  • Continue working on Homework 4
  • Go to Assessment Labs next week!