design-patterns-notes

m.sz

October 22, 2018

Contents

Part I

Design Principles

Chapter 1

Separate What Vary from What Stays the Same

Extracting changing parts and encapsulating them separatly allows future changes without affecting existing parts of code.

Chapter 2

Program to an Interface, not an Implementation

Here by 'an interface' an API is meant, not the java.

Chapter 3

Favour Composition over Inheritance

Allows dynamic changes of behaviour in runtime. See: Strategy Pattern.

Part II

Design Patterns

Chapter 4

Creational

Those patterns specify how an object can be created.

4.1 Singleton

4.1.1 Problem Descritpion

All cases, when **exactly one instance** is required:

- network session
- solar system, Milky Way

There is a modification to the pattern possible to create a pool of fixed amount of objects.

4.1.2 Implementatio

The key is to:

- make the **class final**
- make the **constructor private**. **The constructor is mandatory!** Without it default 'package-protected' constructor will be created.
- **static** *object* variable and its **static** *getter*
- **static getter** returning new object only if there is no object present

```
final class Singleton{
  // private & static instance variable.
  private static Singleton obj;

  // private constructor
```

```
  private Singleton(..){
    ...
  }

  // the only publicly exposed member - instance getter
  public static Singleton getObj(){
    return obj == null? new Singleton() : obj;
  }
}
```

Object getter is the only publicly exposed API. It returns private static instance variable holding the singleon object. If the object is null, it calls private constructor. There is no other way to call the constructor (and to instantiate another object).

final class modifier is added in order to prevent pattern break by subclassing. Without the keyword it would bo possible to subclass and implement Cloneable interface. The final modifier prevents singleton from clonning.

Chapter 5

Structural

The way the objects are connected. It allows to implement particular project constraints. The overall idea is to implement such solutions that future changes to the system won't require changes to existing code.

5.1 Decorator

This is a use of layered objects to dynamically and transparently add responsibilities to individual objects. They preserve original API - we use them by invoking exacty the same methods. Decorator classes just adjust and modify existing interface. It is often used when large number of subclasses is required to solve al possibilities, like for instance servicing IO operations. The drawback is that we must to creat multiple objects in order to get single functionality we need.

Chapter 6

Behavioural

Captures particular types of actions within a program.

6.1 Observer

6.1.1 Problem Description

The requirement is that objects update their state in respond to changes which occur in some other object. As it is common problem the solution is supported by standard Java libraries - **Observer** and **Observable**.

6.1.2 Implementation

- **Observable** class
  - **keeps track** of all objects which need to be notified about the changes which are of their interest
  - **notifies** observing objects when the changes happen
  - We need a call to **setChanged()** somewhere in the *Observable* class in order to update the flag indicating that a change has happened. **notify()** tests the flag to decide whether *Observers* have to be notified or not. That is the reason why we **must not just instantiate an object of *Observeable* class, we must extend this class**
- **Observer** class. Objects of this class **registers** by Observable in order to be **updated** when some specific change occurs.

```
class ObserverableClass extends Observabe{

  ...

  public void notify(Observer o){
```

```
    setChanged();
    super.notify(b);     \\ or implement own way to notify
  }
}

class ObserverClas implements Observer{
  Observable notifier;

  ...

  public void register(){
    notifier.addObserver(this);
  }

  public void update(ObservableClass oc){
    ...
  }
}
```

6.2 Strategy

6.2.1 Problem Description

Derived classes cherrypick behaviours(algorithms). Adding new behaviour in the parent class results that unwanted behaviour may appear in some child classes.

6.2.2 Anti-Patterns

Voiding by Overriding with Empty Methods

1. code duplication (many empty methods)
2. need to go over all subclasses each time new feature is added.

Tag Subclasses with Interfaces

Another maintenance nightmare:

1. terrible code duplication, extremaly bug-proned.

6.2.3 Impementation

1. apply 'separate what vary from what stays the same' principle - extract away optional features.
2. apply 'favour composition over inheritance' principle - root each tree of extracted features with an interface.

3. apply 'program to interface, not to implementation' principle - assign extracted behaviour as an interface type to allow behaviour change in runtime.