

Contents

I	Resources	2
II	Paradigmes	4
1	Abstraction	5
2	Inheritance	6
2.1	Accessing Members	6
2.2	Pitfalls	6
2.2.1	Constructor Calls an Overridable Method	6
2.3	Liskov Substitution Principle	7
3	Encapsulation	8
4	Polymorphism , Method Overriding	9
4.1	Virtual method Invocation	9
4.2	Method Overloading	9
4.3	Method Overriding	9
III	Design Principles	11
5	Separate What Vary from What Stays the Same	12
6	Program to an Interface, not an Implementation	13
7	Favour Composition over Inheritance	14
8	Strive for loosely coupled designs between objects that interact	15
IV	Design	16
9	Requirementments	17
9.1	Requirement	17

9.2 Use Cases	17
V Creational Patterns	18
10 Singleton	20
10.1 Problem Description	20
10.2 Implementation	20
VI Structural	22
11 Decorator	24
VII Behavioural Patterns	25
12 Observer	27
12.1 Problem Description	27
12.2 Implementation	27
13 Strategy	29
13.1 Problem Description	29
13.2 Anti-Patterns	29
13.2.1 Voiding by Overriding with Empty Methods	29
13.2.2 Tag Subclasses with Interfaces	29
13.3 Impementation	29

Part I

Resources

- UML Distilled textbook by Martin Fowler
- Object-Oriented Software Engineering Practical Software Development using UML and Java (second ed.), Lethbridge, Laganieri
- Head First Object Oriented Analysis and Design, McLaughlin, Pollice, West
- Head First Design Patterns, Freeman, Freeman

Part II

Paradigmes

Chapter 1

Abstraction

There are different meanings of abstraction. One of them is the ability to capture real world entities as classes. Two types of abstractions in Java:

- **interfaces**, used to define expected behaviour. Implementation **is hidden from a client**.
- **abstract classes**, used to define incomplete functionality.

Chapter 2

Inheritance

The ability of subclass to derive members (fields and methods) from ascendands. In java only single parent class is allowed. It is an **'is-a'** relationship. **Derived class inherits all members present in the base class. However not all of them are accessible.** This is ruled by access modifiers used in the base class.

2.1 Accessing Members

It is valid to instantiate an object with a subtype. The instantiated reference variable allows an access to those members (and their variations) which are present in their type. It is still possible to access subtype members using cast:

```
Parent childParent = new Child();  
\\ access to a member as it is defined in the Parent class.  
childParent.field...
```

```
\\ access to a member as it is defined in the Child class  
((Child)childParent).field
```

Pay attention to the syntax of above cast!

*** Those method which are overridden are accessible as usual.**

In order to call methods that are not overridden the reference variable used to access the methods (here - *pc*) must be cast to (*Child*)

2.2 Pitfalls

2.2.1 Constructor Calls an Overridable Method

1. call to constructor in **child class**
2. it calls **parent class constructor** first

3. if there is a call to overridable method it calls **textchild version of the method**
4. **ERROR!** The call **will fail** if the method references some uninitialised variable. **The variable can be initialised only when the control returns to child constructor - in steps which will follow!**

2.3 Liskov Substitution Principle

Whenever an instance of some class is expected in a program, one can supply an instance of subclass of the class

Chapter 3

Encapsulation

Inner details of classes can be hidden by making them private and acceptable through public API only - getters (accessors) and setters (mutators).

Chapter 4

Polymorphism , Method Overriding

'*Many forms*'. implemented by

- subclass specialisation (*is-a*' relationship
- Liskov substitution principle
- virtual method invocation)

Polymorphism is usually achieved by method overriding. It utilises method dynamic binding.

4.1 Virtual method Invocation

Method calls are dynamically dispatched based on runtime type of the receiver object.

4.2 Method Overloading

Overloading means that two or more methods have the same name but different signature. They **must have different parameters** (number of them and/or types), they **may have different return type and access modifiers** (private, protected, etc) - see Rules.

4.3 Method Overriding

It means that new implementation is provided to an inherited method. This is annotated by `@Override`. The overridden method in a superclass can be still called when invoked using **super.** keyword.

Rules

- **final, static, private** methods can't be overridden.
- access modifier of overriding method **must not be more restrictive**.
- **no new checked exception** can be thrown
- if return type is a reference type, then it can be original type or any descendant of this type (**covariant return type**).

Private methods can't be overridden because they are excluded from inheritance (not visible from within subclasses).

Static methods reside in static context, they belong to class, not to objects. Therefore they are not inherited, too. Hence, they can't be overridden.

However, they can be **shadowed** - subclass can have static method with the same name, as its parent class. A method call is bind to first method with a proper signature - starting from the class in current context and going up the inheritance tree.

Part III

Design Principles

Chapter 5

Separate (Encapsulate) What Vary from What Stays the Same

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

Chapter 6

Program to an Interface, not an Implementation

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

Chapter 7

Favour Composition over Inheritance

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

Chapter 8

Strive for loosely coupled designs between objects that interact

Loosely coupled designs allow us to build flexible OO systems that can handle change because they minimize the interdependency between objects.

Part IV

Design

Chapter 9

Requirements

9.1 Requirement

This is a single need of a system. Details what system should **do** or **be**.

9.2 Use Cases

Describes what system does to accomplish a particular **one particular customer goal**. Each use case details exactly what a system should **do**. It captures potential requirements of a new system. It can discover what is **missing** in requirements. **Use cases and requirements should exhaustively match each other.**

Each use case has a single goal, but may provide one or more scenarios. Constituents:

- **clear value** -
- **start and stop condition**
- **external initiator** - an actor outside the system
- **main path**
- **alternate path** - optional. May **branch** or **extend** existing path - main or alternate as well.

Part V

Creational Patterns

Those patterns specify how an object can be created.

Chapter 10

Singleton

10.1 Problem Description

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.

10.2 Implementation

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 singleton 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 be possible to subclass and implement Cloneable interface. The final modifier prevents singleton from cloning.

Part VI

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.

Chapter 11

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 exactly the same methods. Decorator classes just adjust and modify existing interface. It is often used when large number of subclasses is required to solve all possibilities, like for instance servicing IO operations. The drawback is that we must create multiple objects in order to get single functionality we need.

Part VII

Behavioural Patterns

Those patterns capture particular types of actions within a program.

Chapter 12

Observer

12.1 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**.

12.2 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 *Observable* 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 Observable{  
  
    ...  
  
    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){
        ...
    }
}
```

Chapter 13

Strategy

13.1 Problem Description

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

13.2 Anti-Patterns

13.2.1 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.

13.2.2 Tag Subclasses with Interfaces

Another maintenance nightmare:

1. terrible code duplication, extremely bug-prone.

13.3 Implementation

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 run-time.