Contents

Part I

Resources

- UML Distilled textbook by Martin Fowler
- Object-Oriented Software Engineering Practical Software Development using UML and Java (second ed.), Lethbridge, Laganiere
- Head First Object Oriented Analysis and Design, McLaughlin, Pollice, West
- Head First Design Patterns, Freeman, Freeman
- `https://www.visual-paradigm.com/tutorials/`

Part II

Paradigmes

Chapter 1

Abstraction

1There are different meanning of abstractin. 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 folow!**

2.3 Liskov Substitution Principle

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

Chapter 3

Encapsulation

Inner details of classes can be hidden by making them private and acceptible 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.

4.3.1 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 overriden 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 overriden.
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

OCP, Open-Closed Principle

Open classes for extensions, but close for modifications. We allow changes but without modifying existing code. This can be achieved using inheritance or by composition and delegating some feature to an external class.

Chapter 6

DRY, Don't Repear Yourself

ONE requirement in ONE place implemented ONE single time. No duplicated code. If any changes are required in the future then we will implement them just in ONE place only. No risk of loosing consistency.

Chapter 7

Cohesion. SRP. Single Responsibility Principle.

. Each Class Should Have Only One Reason to Change. **Do one thing well and don't pretend to be something else**. When a class has more than on reason to change then this class probably **tries to do too much**. Cohesion measures the degree of connectiviness between elements of some component - class, module. The higher cohesion the more well-defined responsibilities of each class in a software. Cohesive class means:

- class focuses on one specific task
- if something seems to be unrelated then it might belong to some other class.

Simple test:
'The X Ys itself' f.i. 'The Observer registers itself (by..).

Chapter 8

LSP, Liskov Substitution Principle

Subtypes must be substituable for their base class. We must be able to use subclasses in all those places where we can use their base classes. Otherwise, it indicates the inheritance implemented in a wrong way.

Chapter 9

Separate (Encapsulate) What Vary from What Stays the Same

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

Chapter 10

Program to an Interface, not an Implementation

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

Chapter 11

Favour Composition over Inheritance

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

Chapter 12

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.

Chapter 13

Encapsulate Away Common Behaviour

Part IV

Design

Chapter 14

Requiremments

14.1 Terminology

14.1.1 Commonalities

What does the system have in common with something already known? What do we know about the system for sure?

14.1.2 Variabilities

Something about the system that we are sure that the system is NOT. Something the system is not like.

14.1.3 A Feature

A high-level description of what the system is supposed to do. Can be used to figure **requirements** neede to implement this feature.

14.1.4 Requirement

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

14.2 Use Case Diagram

Constituents:

- **an actor** - a stickman
- **a system** - big box
- **use cases** - ovals inside the box.

14.3 Use Cases

Describes what system does to acomplish 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 us **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.

Chapter 15

Class Diagram

15.1 Design - Textual Analysis

15.1.1 Each Noun in a Use Case or in the Requirements have a Potential to be a Class

15.1.2 [Each Verb in a Use Case can be a Method]

15.2 Design - Building Blocks

15.2.1 Classes

- **abstract** classes are *italicised*
- **reference attributes** are ommited.

15.2.2 Relationships

Description

They are possible relations between classes:

- dependency - **dotted arrow**
- association/delegation - **solid arrow**; this is the case when we use some functionality of an external class without modifying it; association specialisations:
  - composition - **solid diamond** at target side; the same lifetime, children dependent. Example - House and Rooms. Deleting House destroys its Rooms as well
  - aggregation - **hollow diamond** at target side. Independent children with independent lifycycle. Example: Team, Player. Delete the Team, Players can go to another Team.

- generalisation - **hollow arrow** at target side
- specialisation

Association - target element is a part of source element.

Building Block

- an arrow starting at source and ending at target side
- target identifier by target side of the connection line
- a multiplicity indicator at target side (1, 2, *)

15.2.3 Operations

We don't model constructors!

Building Blocks

- method identifier
- list of parameter types
- return:
  - **void** - nothing
  - [ ] - indicates a multiple type.
    [*] - indicates an unlimited number of objects.

c

15.2.4 Interfaces

`<<interface>>`

Part V

Creational Patterns

Those patterns specify how an object can be created.

Chapter 16

Singleton

16.1 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.

16.2 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 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.

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 17

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.

Part VII

Behavioural Patterns

Those patterns capture particular types of actions within a program.

Chapter 18

Observer

18.1 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**.

18.2 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){
    ...
  }
}
```

Chapter 19

Strategy

19.1 Problem Description

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

19.2 Anti-Patterns

19.2.1 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.

19.2.2 Tag Subclasses with Interfaces

Another maintenance nightmare:

1. terrible code duplication, extremaly bug-proned.

19.3 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.