Week 9 Announcements

- Video Assignment #2 → No late submission (hard deadline, Friday, 27 October) | 6 weeks (from the teaching break).
- Escape Room happened! How did it go? We hope it was fun! 😳
- Survey #2: it's open! | SELT is coming up soon!
- Lab 5 (Android I) marks: corrected and reloaded.
- GitLab Repo — Lecture Code: available!
- Group Project:
  - Checkpoint 2: next week! → Registration!
  - Voice your feature: closes today!
  - Surprise Feature! → Coming soon!
- Final exam: it will be in-person in the computing labs.
  - It's highly recommended to attend at least once to the labs to experience the ambience.

Australian National University

COMP2100/6442
Software Design Methodologies / Software Construction

Refactoring

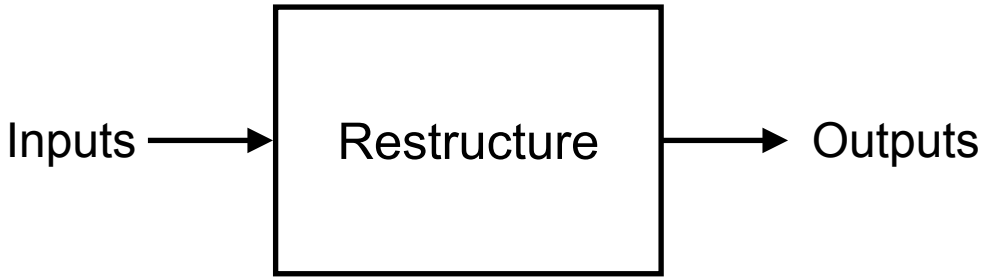
Nan Wang

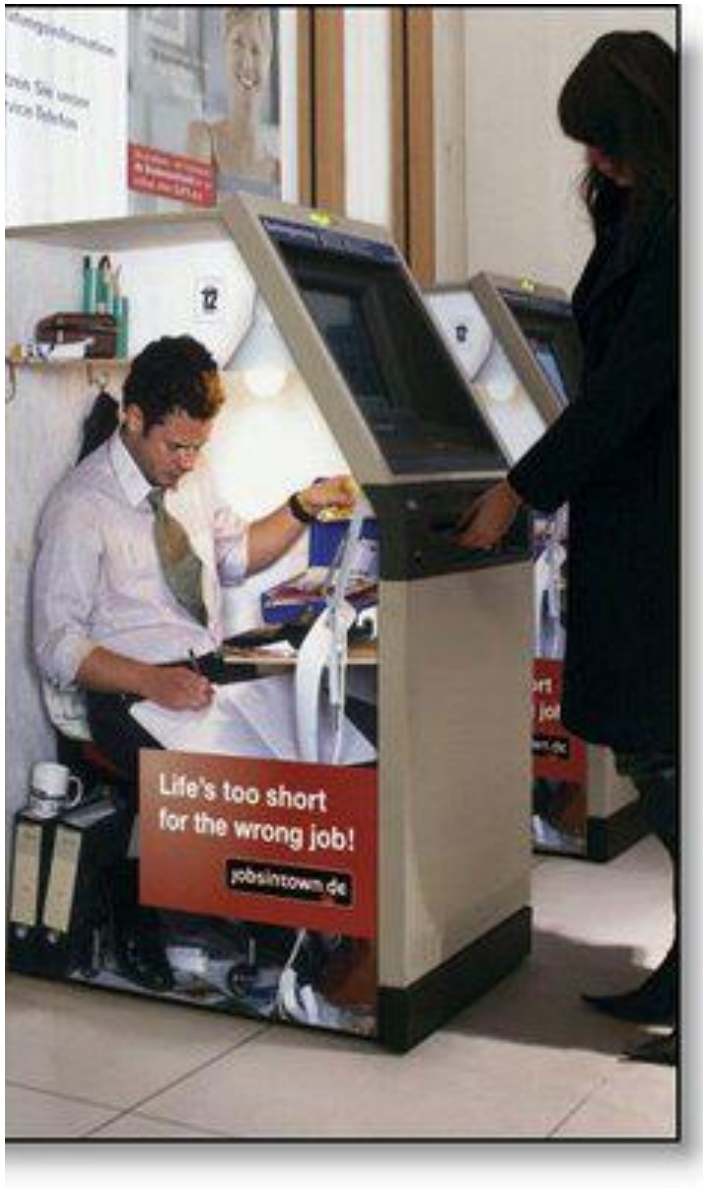
Outline

- Introduction to Refactoring
- SOLID Principles
- Common Code Smells
- Analysis of a group project
- Review of an exam question from a previous semester

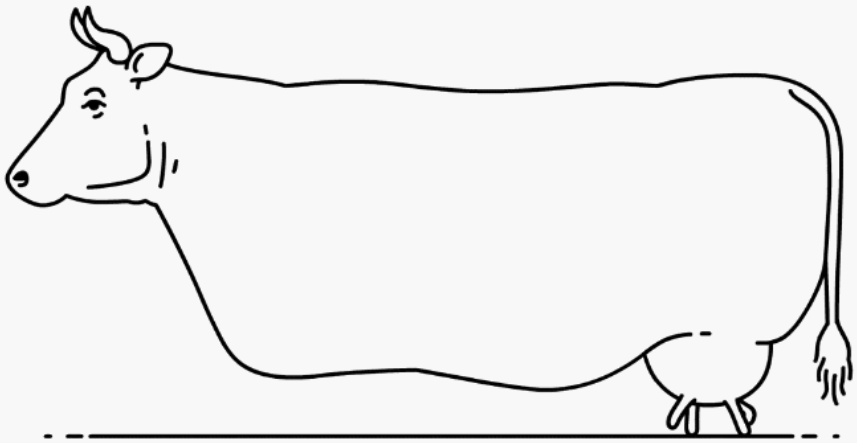
What is Refactoring?

- Refactoring is the process of restructuring a software system in a way that does not alter the external behaviour of the code but improves its internal structure.
- Refactoring is an effective approach to removing code smells in the software systems.





The systems need refactoring!



Code Smells

- Smells are certain structures in the code that indicate violations of fundamental design principles and negatively impact design quality

- Code smells are not bugs. Instead, they indicate weaknesses in design that may slow down development or increase the risk of bugs or failures in the future.

Code Smells

“If it stinks, change it.” — Kent Beck

Some common code smells:

- Mysterious names
- Hardcoding
- Duplicated code
- Long method & class
- Long parameter list
- Comments
- Primitive Obsession
- Dead Code

Goals of Refactoring

- Extensibility: The ease of a system that can be enhanced or extended for adding new functionalities.
- Performance: The ability of a system to meet timing requirements within given constraints such as speed, accuracy, memory usage etc.
- Maintainability: The ease with which a system can be modified to correct faults and improve performance or other attributes.
- Readability: The ease with which the code of a system can be understood.
- Reliability: The ability of the system to keep operating over time.
- Scalability: The ability of a system to maintain or improve performance with increased demand.

When to Refactor

- When you want to add new functionalities to the existing code.
- When you find code smells during a code review

Checklist of Refactoring

- The code should be improved after refactoring.
- New functionality should not be created during refactoring.
- All existing tests must pass after refactoring.
- Make small changes of code before testing during refactoring

SOLID Principle

- The principles were introduced by Robert C. Martin (Uncle Bob) in the early 2000s.
- The principles are a set of rules and practices for Object-Oriented software design that help developers create software that is easier to maintain, extend, and test.

SOLID Principle

- **S**ingle Responsibility
- **O**pen for Extension, Closed for Modification
- **L**iskov Substitution
- **I**nterface Segregation
- **D**ependency Inversion

Single Responsibility

- A class or a method should only have one responsibility. Besides, there should only be one reason to change.
- It aims to achieve modularisation.

Advantages:

- Fewer test cases
- Looser Coupling
- Greater reusability

Decoupling

- Coupling is the degree to which one class is dependent on another class.
- High coupling means components are tightly connected with each other.
- Decoupling aims to reduce the dependency among different components in a system.
- Components in a loosely coupled system can be replaced with alternative implementations that have the same functionality.

Modularisation

Single Responsibility

```
private JSON readFromInputStream(InputStream inputStream)
  throws IOException {
    StringBuilder resultStringBuilder = new StringBuilder();
    try (BufferedReader br
      = new BufferedReader(new InputStreamReader(inputStream))) {
        String line;
        while ((line = br.readLine()) != null) {
            resultStringBuilder.append(line).append("\n");
        }
    }
  return resultStringBuilder.toString().toJSON();
}
```

Open & Close

Classes should be open for extension but closed for modification. Try to avoid modifying existing code and causing potential new bugs.

Open & Close

It is better to make changes by adding to or building on top of existing code rather than modifying it.

Let's build a car! 

```
public class Car {

    private Engine engine;
    private Wheel wheels;
    private Door doors;

    public void run() { //code for running… }
}
```

Open & Close

Let's build a car that can fly! It will be cool! 

```
public class Car {

    private Engine engine;
    private Wheel wheels;
    private Door doors;
    private Wing wings;

    public void runAndFly() { //code for running and flying…}
}
```

Open & Close

Hey, let's build a car that can both fly and sail! 

```
public class Car {

    private Engine engine;
    private Wheel wheels;
    private Door doors;
    private Wing wings;
    private Propeller propeller;

    public void runAndFlyAndSail() { //code for running, flying and sailing}
}
```

Open & Close

Hey, why don't we go back to build a normal car! 

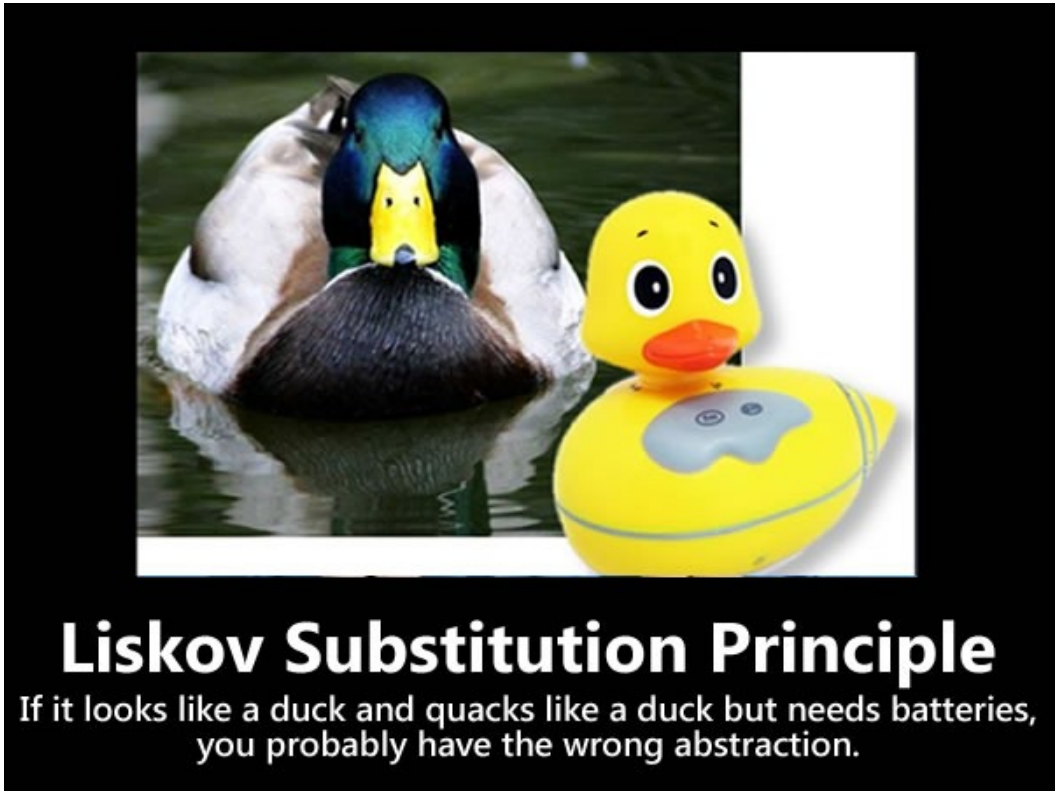
```
public class Car {

    private Engine engine;
    private Wheel wheels;
    private Door doors;

    public void run() { //code for running… }
}
```

Open & Close

```
public class FlyingCar extends Car {

   private Wing wing;
   public void runAndFly() { //code for running and flying… }
}

 public class FlyingSailingCar extends FlyingCar {

    private Propeller propeller;
    public void runAndFlyAndSail() { //code for running, flying and sailing… }
 }
```

Liskov Substitution

A class can be replaced with its subclass without disrupting the behaviour of our program.



Liskov Substitution

```
public interface Vehicle {

    void turnOnEngine();
    void refuel()
}
```

```
public class FuelVehicle implements Vehicle {

    void turnOnEngine() {
        System.out.println("Turn on engine!");
    }

    void refuel() {
       System.out.println("Refuel!");
    }
}
```

Liskov Substitution

```
public interface Vehicle {

    void turnOnEngine();
    void refuel()
}
```

```
public class ElectricVehicle implements Vehicle {

    void turnOnEngine() {
        throw new Exception("I don't have an engine");
    }

    void refuel() {
        throw new Exception("I don't consume fuel!");
    }
}
```

Interface Segregation

A class should only depend on the needed interfaces. Larger interfaces should be broken into smaller ones.

Interface Segregation

```
public abstract class Bird {
    abstract int layEggs()
    abstract void fly()
}
```

```
void method(Bird bird) {
   int eggs = bird.layEggs();
   bird.fly();
}
```

```
public class Pigeon extends Bird {
    public int layEggs() {
        return 3;
    }

    public void fly() {
        System.out.println("Flying in a pigeon style");
    }
}
```

What about goose?

Interface Segregation

```
public abstract class Bird {
    abstract int layEggs()
    abstract void fly()
}
```

```
void method(Bird bird) {
   int eggs = bird.layEggs();
   bird.fly();
}
```

```
public class Goose extends Bird {
    public int layEggs() {
        return 3;
    }

    public void fly() {
        throw new Exception("A goose cannot fly!");
    }
}
```

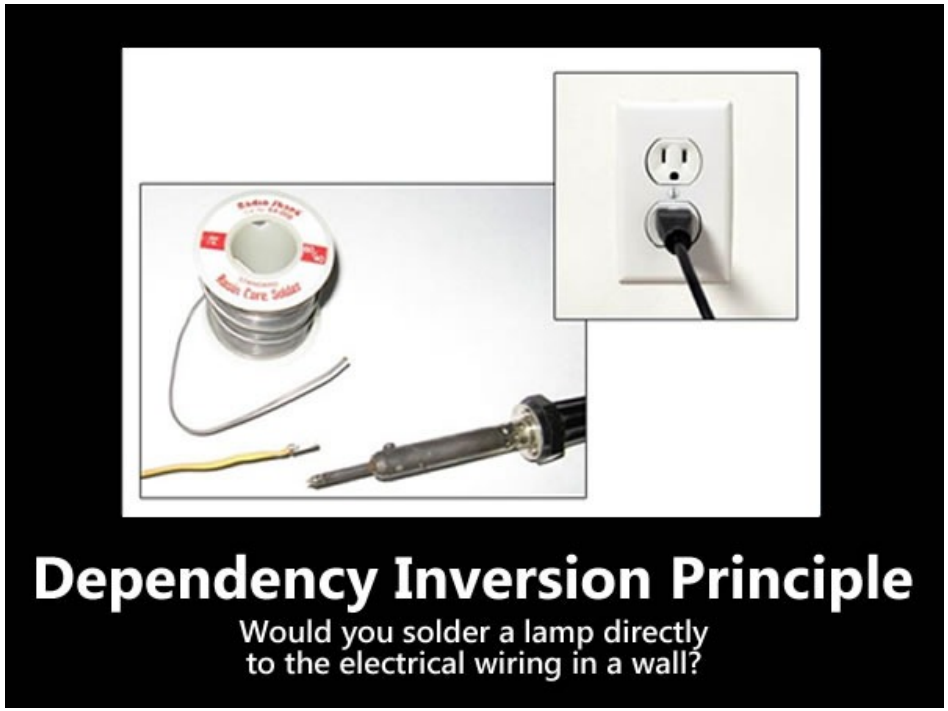
Interface Segregation

```
public interface Oviparous {
   int layEggs()
}

public interface Flyable {
   void fly()
}
```

```
public class Goose implements Oviparous{
   public int layEggs() {
      return 3;
   }
}

public class Pigeon implements Oviparous, Flyable
{
    public int layEggs() {
        return 3;
     }

   public int fly() {
      System.out.println("Flying in a bat style");
   }
}
```

Dependency Inversion

Dependency inversion refers to the decoupling of software modules. When high-level modules depend on low-level modules, they need to depend on abstractions.



Dependency Inversion

```
public class Program {
    private final JSON json;

    public void read(){ json.read(); }
    public void write() { json.write(); }
}
```

```
public class Program {
    private final XML xml;

    public void read(){ xml.read(); }
    public void write() { xml.write(); }
}
```

Dependency Inversion

```
public interface DAO {
   void read()
   void write()
}

public JSON implements DAO {
   void read() { }
   void write() { }
}

public XML implements DAO {
   void read() { }
   void write() { }
}
```

```
public class Program {

   private final DAO dao;

   public Program(DAO dao) {
      this.dao = dao;
   }

   public void read() { this.dao.read(); }
   public void write() { this.dao.write(); }
}
```

Don’t repeat yourself (DRY)

- Avoid copy-pasting your code in different places.
- You can extract a common logic into a new function.
- Greater reusability, maintainability, and testability

Keep it small and simple (KISS)

- Keep classes and methods small.
- Greater readability, and maintainability.

Mysterious Names

Functions, variables or classes are named in a way that does not communicate what they do or how to use them.

Some bad examples: a, b, c, aa, bb, cc, yorn, torf, etc.

```
int lucky1 = -1;
int lucky2 = 10;

int function luckyFunction(int lucky3) {
    return lucky3;
}
```

Mysterious Names

- Use polite words
- Create meaningful and unique names.
- Create descriptive names that communicate what they do.
- Use complete words rather than just acronyms.
- Follow the naming conventions of the programming languages

https://www.oracle.com/java/technologies/javase/codeconventions-namingconventions.html

Hardcoding

- Embedding data directly into the source code of a program or other executable object, as opposed to obtaining the data from external sources or generating it at runtime.
- Hard-coded data should only be limited to unchanging pieces of information, such as physical constants, version numbers and static text elements.
- Secure data such as passwords should never be hard-coded.

Duplicate Code

- Duplication usually occurs when multiple programmers are working on different parts of the same program at the same time.
- Merging duplicate code simplifies the structure of your code and makes it shorter.
- Facilitates the maintainability and testability.

```
public class A {

   void  readFileX()

}
```

```
public class B {

   void  readFileY()

}
```

Long Class & Method

- A long class or a long method contains too many lines of code.
- Many programmers prefer to add code to existing classes or methods rather than create new classes or methods.
- The longer the classes or methods, the harder to understand and maintain it
- Decompose it into smaller classes or methods

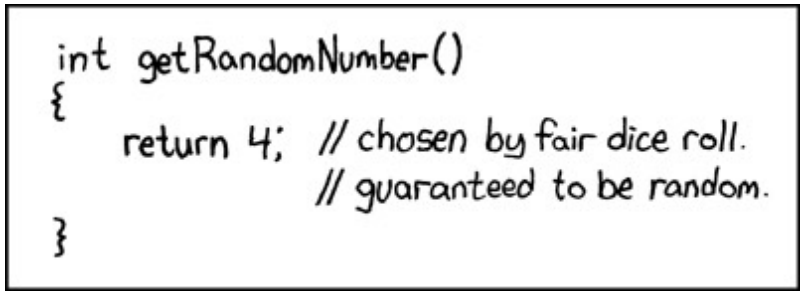
Long Parameter List

Some methods contains too many parameters.

void method(param1, param2, param3, param4, param5, param6, ….)

```
Object1 {
   param1, param2, param3
}
Object2 {
   param4, param5, param6
}
void method(Object1, Object2, ….)
```

Comments




```
1 // This is black magic
2 // from
3 // *Some stackoverlow link
4 // Don't play with magic, it can BITE.
```

```
1 /*
  ************************************************************
2 The reason this code does not work is on line 335
3 This line is there because this customer has not paid for the
  application in over 2 years.
4 ************************************************************
  */
```

Comments

- Comments should be intuitive and concise.
- Comments should help readers better understand the code.
- The best comment is a good name for method and class.
- Refrain from over-commenting, e.g. getter, setter.

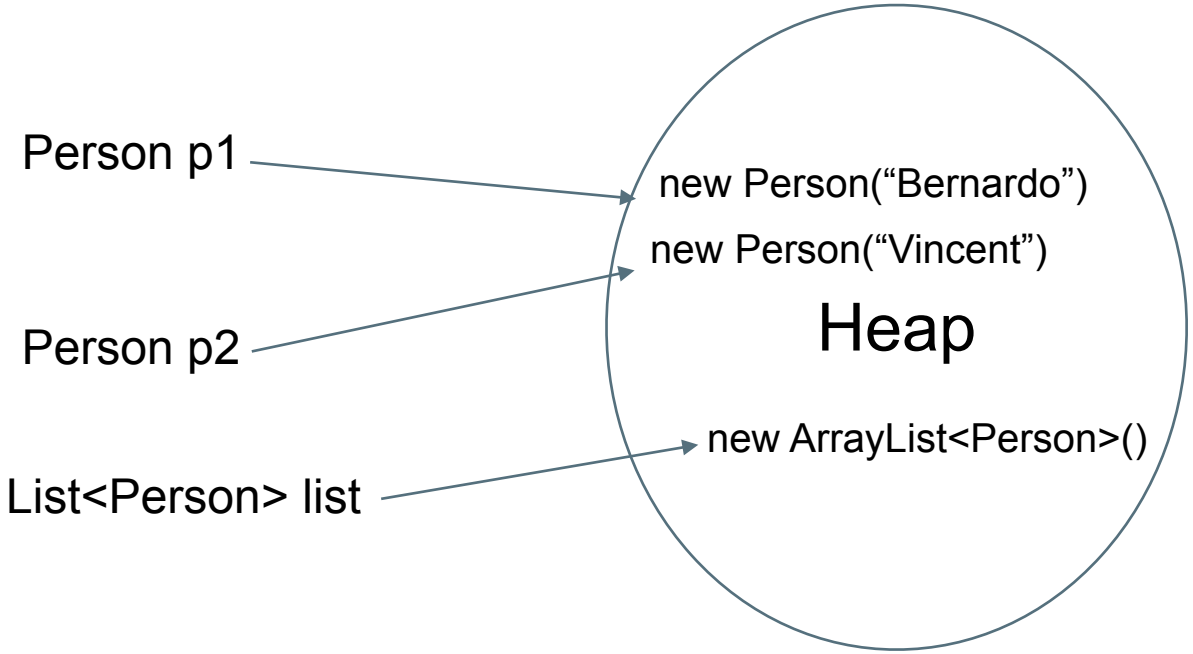
Primitive Obsession

- Use primitives such as int, String for coding information

```
public static final int MONDAY =1;
public static final int TUESDAY =2;
public static final int WEDNESDAY = 3;
public static final int MONDAY =“Monday”;
public static final int TUESDAY =“Tuesday”;
public static final int WEDNESDAY = “Wednesday”;

enum Day {MONDAY, TUESDAY, WEDNESDAY}
```

Mutable Object

- A mutable object can be changed after it is created.
- Make defensive copies when needed, e.g. object, list



Mutable Object

- A mutable object can be changed after it is created.
- Make defensive copies when needed, e.g. object, list

```
public void method (List<Person> persons) {
    List<Person> localPersons = new ArrayList<>(persons);
    // Process localPersons
}
```

Dead Code

- Delete unused code and unneeded files.
- Remove unneeded parameters
- Improve readability and reduce code size

References

- Refactoring: Improving the Design of Existing Code by Martin Fowler.
- Refactoring for software design smells by Ganesh Samarthyam, Girish Suryanarayana, and Tushar Sharma
- https://refactoring.guru/
- https://learn.microsoft.com/zh-cn/archive/blogs/gabriel_morgan/implementing-system-quality-attributes