Lecture 18 — OOP Principles

CITS2005 Object Oriented Programming

Department of Computer Science and Software Engineering
University of Western Australia

Contents

- No textbook chapters
- “Clean Code” by Robert Martin is an option
- SOLID principles and writing “clean” code

SOLID

- SOLID is an acronym for some principles
- **S**ingle-responsibility principle
- **O**pen–closed principle
- **L**iskov substitution principle
- **I**nterface segregation principle
- **D**ependency inversion principle

SOLID

- Commonly attributed to Robert C. Martin
- They are general principles that (in theory) lead to "nice" code
- They are specifically for OOP
- Nice usually means things like: understandable, flexible, reusable, maintainable, makes you feel like a "real" engineer

Single Responsibility Principle (SRP)

- A class should have only one reason to change
- Single reason $\approx$ single responsibility
- Each class should focus on a single responsibility
- Translation: a class should do one thing
- Makes testing and maintaining code easier
- It is easier to think about how to use/test a class if it only does one thing

SRP Example: Violation

```java
public class FileManager {
    public String readFile(String filePath) { ... }
    public void writeFile(String filePath, String content) { ... }
    public String compressFile(String contents) { ... }
}
```

- This class handles multiple responsibilities: reading, writing, and compressing files
- Changes to any of these functionalities will affect the entire class

SRP Example: Adherence

```java
class FileReader {
    public String readFile(String filePath) { ... }
}

class FileWriter {
    public void writeFile(String filePath, String content) { ... }
}

class FileCompressor {
    public String compressFile(String filePath) { ... }
}
```

- Separate classes for each responsibility
- Easier to maintain and test individual functionalities

SRP Example: Violation

```java
public class MazeSolver {
    public void setPositionBlocked(int row, int col) { ... }
    public Path findPath(int sr, int sc, int dr, int dc) { ... }
}
```

- Similar to the Maze/MazeSolver from the lab
- The MazeSolver is both managing the details of the maze itself, and finding paths

SRP Example: Adherence

```java
class Grid {
    public void setPositionBlocked(int row, int col) { ... }
    public boolean getPositionBlocked(int row, int col) { ... }
}

public class MazeSolver {
    public void setMaze(Grid m) { ... }
    public Path findPath(int sr, int sc, int dr, int dc) { ... }
}
```

- Storing a grid and solving it as a maze are separated into different classes

Open-Closed Principle (OCP)

- Software entities should be open for extension but closed for modification
- “Open for extension” means we can add new functionality
- “Closed for modification” means we don’t need to change existing code to add new functionality
- Translation: add new features by making new classes/methods, not by modifying existing code
- Encourages the use of interfaces and abstract classes
- Reduces the risk of introducing errors in existing code when adding new features

OCP Example: Violation

```java
public enum ShapeType {
    CIRCLE, SQUARE
}

public class Shape {
    public ShapeType type;
    public double radius; // for Circle
    public double side; // for Square
}

public class AreaCalculator {
    public double area(Shape shape) {
        if (shape.type == ShapeType.CIRCLE) {
            return Math.PI * shape.radius * shape.radius;
        } else if (shape.type == ShapeType.SQUARE) {
            return shape.side * shape.side;
        }
        return 0;
    }
}
```

- If a new shape is added, we have to modify the AreaCalculator class

OCP Example: Adherence

```java
public interface Shape {
    double area();
}

class Circle implements Shape {
    private double radius;
    public double area() { return Math.PI * radius * radius; }
}

class Square implements Shape {
    private double side;
    public double area() { return side * side; }
}
```

- New shapes can be added and there is no need for `AreaCalculator`
- The `Shape` interface is “open”
- The specific implementations of area for `Circle` and `Square` are closed

Liskov Substitution Principle (LSP)

- Introduced by Barbara Liskov in 1987
- Subtypes should be substitutable for their base types without affecting the correctness of the program
- Translation: a subclass should be able to pretend to be its superclass without breaking anything
- Encourages proper inheritance and polymorphism
- Ensures that new derived classes do not introduce unexpected behavior

LSP Example: Violation

```java
class Bird {
    void fly() { ... }
}

class Penguin extends Bird {
    @Override
    void fly() {
        throw new UnsupportedOperationException("Penguins can't fly");
    }
}
```

- The `Penguin` class violates the LSP because it changes the behavior of the `fly` method
- Substituting a `Penguin` object for a `Bird` object could cause unexpected errors

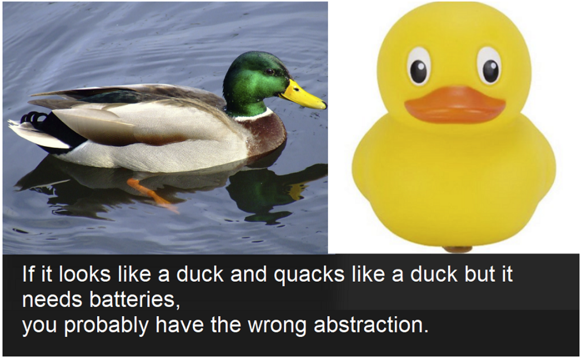
LSP Example: Adherence

```java
abstract class Bird {
    abstract void move();
}

class FlyingBird extends Bird {
    @Override
    void move() { fly(); }
    void fly() { ... }
}

class NonFlyingBird extends Bird {
    @Override
    void move() { walk(); }
    void walk() { ... }
}
```

Mid-lecture Break



Interface Segregation Principle (ISP)

- Clients should not be forced to depend on methods they do not use
- Translation: lots of smaller interfaces are usually better than one big class/interface
- If you have a large complicated class/interface, most methods/classes using it will only depend on part of it
- Make an interface for each of those parts
- Enhances code maintainability and readability

ISP Example: Violation

```java
public class GameEntity {
    public Model3D getModel() { ... }

    public Point3D getPosition() { ... }

    public int getHP() { ... }

    public int getAttack() { ... }
}

class Renderer {
    public void render(GameEntity entity) {
        // We don't care about getHP() and getAttack()
    }
}
```

ISP Example: Adherence

```java
interface Renderable {
    Model3D getModel();
    Point3D getPosition();
}

interface Fighter {
    int getHP();
    int getAttack();
}

public class GameEntity implements Renderable, Fighter {
    public Model3D getModel() { ... }

    public Point3D getPosition() { ... }

    public int getHP() { ... }

    public int getAttack() { ... }
}

class Renderer {
    public void render(Renderable entity) {
        // Much better
    }
}
```

Dependency Inversion Principle (DIP)

- High-level modules should not depend on low-level modules; both should depend on abstractions
- Abstractions should not depend on details; details should depend on abstractions
- Translation: don't store classes, store interfaces
- Makes code more flexible. If we change a class, we don't need to check all the classes that depend on it

DIP Example: Violation

```java
class EmailService {
    void sendEmail(String message, String recipient) { ... }
}

class Notification {
    EmailService emailService;

    void sendNotification(String message, String recipient) {
        emailService.sendEmail(message, recipient);
    }
}
```

- The `Notification` class directly depends on the `EmailService` class
- Violates the DIP as it makes it difficult to change or extend the notification mechanism

DIP Example: Adherence

```java
interface MessageService {
    void sendMessage(String message, String recipient);
}

class EmailService implements MessageService {
    @Override
    public void sendMessage(String message, String recipient) {
        // Not real code
    }
}

public class Notification {
    MessageService messageService;

    void sendNotification(String message, String recipient) {
        // Now we can change the message service whenever we want
        // Depending on an interface is better than depending on a class
        messageService.sendMessage(message, recipient);
    }
}
```

Mid-lecture Break