COMP 3550

5.1 — COHESION, COUPLING, AND SEPARATION OF CONCERNS

Week 5: Design Principles & Refactoring

DESIGN QUALITY ISN’T OPTIONAL

> *Code is read more than it is written*

DESIGN QUALITY ISN’T OPTIONAL

- *Code is read more than it is written*.
- Clean design reduces bugs and speeds up dev overall
- Taking shortcuts never ends well long term

DESIGN QUALITY ISN’T OPTIONAL

- *Code is read more than it is written.*
- Clean design reduces bugs and speeds up dev overall
- Taking shortcuts never ends well long term
- A friend working at Google puts up their first code review to the team

DESIGN QUALITY ISN’T OPTIONAL

- *Code is read more than it is written.*
- Clean design reduces bugs and speeds up dev overall
- Taking shortcuts never ends well long term
- A friend working at Google puts up their first code review to the team
- This particular code review went through **23 rounds** of comments

DESIGN QUALITY ISN’T OPTIONAL

- *Code is read more than it is written.*
- Clean design reduces bugs and speeds up dev overall
- Taking shortcuts never ends well long term
- A friend working at Google puts up their first code review to the team
- This particular code review went through **23 rounds** of comments
- This means they recieved comments, went to fix their work as per the comments and **additional** changes were then requested **23 times** before their commit was merged in

DESIGN QUALITY ISN’T OPTIONAL

- *Code is read more than it is written.*
- Clean design reduces bugs and speeds up dev overall
- Taking shortcuts never ends well long term
- A friend working at Google puts up their first code review to the team
- This particular code review went through **23 rounds** of comments
- This means they recieved comments, went to fix their work as per the comments and **additional** changes were then requested **23 times** before their commit was merged in

If that doesn’t motivate design quality in code, I don’t know what will

HIGH COHESION

- A class should do one thing and do it well

```java
public class Student {
    private String name;
    private int id;

    // Constructor
    public Student(String name, int id) {
        this.name = name;
        this.id = id;
    }

    // Getters
    public String getName() {
        return name;
    }

    public int getId() {
        return id;
    }

    // Setters
    public void setName(String name) {
        this.name = name;
    }

    public void setId(int id) {
        this.id = id;
    }

    // toString method for easy printing
    @Override
    public String toString() {
        return "Student{name='" + name + "', id=" + id + "}";
    }
}
```

LOW COHESION

(look for **ANDs** in your description of responsibilities)

```java
import java.util.Map;

public class StudentReport {
    private String name;
    private int id;
    private Map<String, Double> grades;

    public StudentReport(String name, int id,
                Map<String, Double> grades) {
        this.name = name;
        this.id = id;
        this.grades = grades;
    }

    public double calculateGPA() {
        double total = 0;
        for (double grade : grades.values()) {
            total += grade;
        }
        return grades.isEmpty() ? 0 : total / grades.size();
    }
```

```java
    public String generateReportCard() {
        StringBuilder sb = new StringBuilder();
        sb.append("Report Card for ")
          .append(name).append(" (ID: ")
          .append(id)
          .append(")\n");
        for (Map.Entry<String, Double> entry : grades.entrySet()) {
            sb.append(entry.getKey())
              .append(": ").append(entry.getValue())
              .append("\n");
        }
        sb.append("GPA: ")
          .append(String.format("%.2f", calculateGPA()));
        return sb.toString();
    }
}
```

COUPLING

- How dependent are we on other components
  - effectively non-primitives

LOW COUPLING

```
public int rollDice(int sides) {
    Random random = new Random();
    return random.nextInt(sides) + 1;
}
```

- How do you test this? What if we want to change the Die to be a loaded die (tamper with probablities)?

LOW COUPLING

```
public int rollDice(Random random, int sides) {
        return random.nextInt(sides) + 1;
}
```

- Better....

LOW COUPLING

- BEST

```
public int rollDice(Dice random, int sides) {
        return random.nextInt(sides) + 1;
}
```

LOW COUPLING

- **BEST**

```java
public interface Dice {
    int nextInt(int bound);
}
```

```java
public int rollDice(Dice random, int sides) {
        return random.nextInt(sides) + 1;
}
```

LOW COUPLING

- **BEST**

```
public interface Dice {
    int nextInt(int bound);
}
```

```
public int rollDice(Dice random, int sides) {
      return random.nextInt(sides) + 1;
}
```

```
import java.util.Random;

public class StandardRandom implements Dice {
    private final Random random = new Random();

    @Override
    public int nextInt(int bound) {
        return random.nextInt(bound);
    }
}
```

- Now I can easily make a loaded die if I want to

LOW COUPLING

- **BEST**

```
public interface Dice {
    int nextInt(int bound);
}
```

```
public int rollDice(Dice random, int sides) {
      return random.nextInt(sides) + 1;
}
```

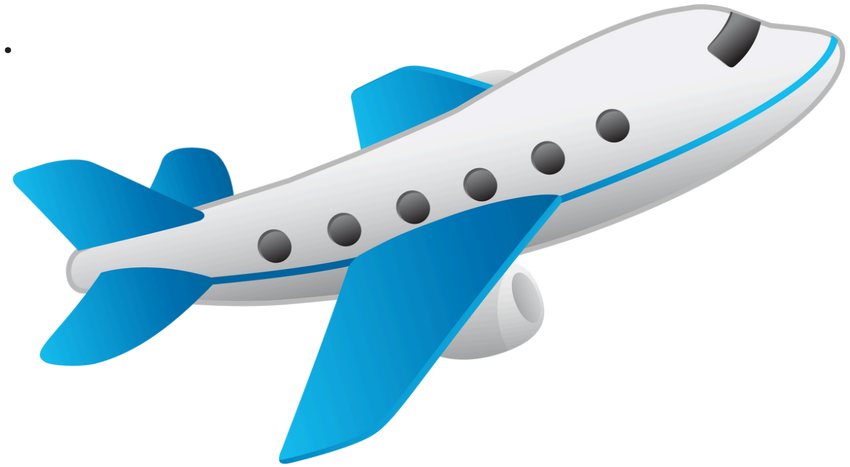
```
import java.util.Random;

public class StandardRandom implements Dice {
    private final Random random = new Random();

    @Override
    public int nextInt(int bound) {
        return random.nextInt(bound);
    }
}
```

- Now I can easily make a loaded die if I want to

- How many classes/interfaces are **too** much?
- Surely, you don't need this for **EVERY** possibility?

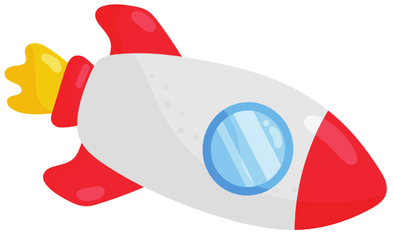
YAGNI

- You are correct, and don’t call me Shirley.
- **YAGNI**
  - **ya ain’t gonna need it.**



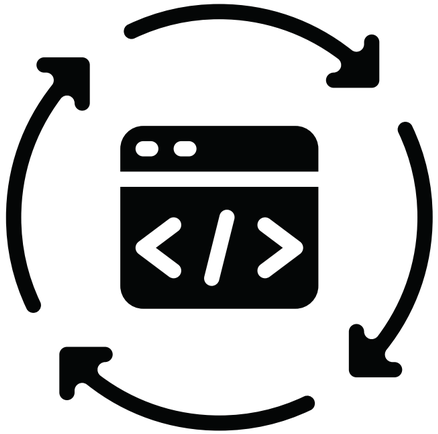
YAGNI

- You are correct, and don’t call me Shirley.
- **YAGNI**
  - **ya ain’t gonna need it.**
- We don’t build a rocketship to go to the corner store



YAGNI

- You are correct, and don’t call me Shirley.
- **YAGNI**
  - **ya ain’t gonna need it.**
- We don’t build a rocketship to go to the corner store
- **Built it as you need it (refactoring)**



SEPARATION OF CONCERNS (SOC)

- Split responsibilities:
  - logic vs. data vs. UI
- MVC and layered architecture are examples

SEPARATION OF CONCERNS (SOC)

- Split responsibilities:
  - logic vs. data vs. UI
- MVC and layered architecture are examples
- What does the **DSO/object** package do?
  - Represent what things are.
  - Think: data models or domain objects — they define structure, not behavior.
    - It describes the nouns, not the verbs.
    - It knows what it is, not what it does.
    - No behavior, no business, just blueprints.

SEPARATION OF CONCERNS (SOC)

- Split responsibilities:
  - logic vs. data vs. UI
- MVC and layered architecture are examples
- What does the **persistence** package do?
  - Save and fetch things — that’s it.
  - Think: repositories, DAOs, ORM mappings.
    - It talks to the database, not the business.
  - It knows how to store, not why to store.
  - It fetches the facts, not the reasons.

SEPARATION OF CONCERNS (SOC)

- Split responsibilities:
  - logic vs. data vs. UI
- MVC and layered architecture are examples
- What does the **logic/business** layer do?
  - Decide and coordinate.
  - Think: application rules, workflow, orchestration.
    - It’s the brain, not the hands or the eyes.
    - It makes decisions, not data, not display.
    - It tells others what to do but doesn’t do their job.

SEPARATION OF CONCERNS (SOC)

- Split responsibilities:
  - logic vs. data vs. UI
- MVC and layered architecture are examples
- What does the **presentation** layer do?
  - Talk to people.
  - It shows and takes input, no thinking, no storing.
  - It speaks human, not database or business.
  - It knows how to say it, not what to say.

MVC INSIDE THE PRESENTATION LAYER

- We often find we need further subpackages in our layers
- Persistence
  - Seperation by Stub/Real (maybe further)
  - Logic Layer
    - by feature or by functionality, *it depends*
- Presentation
  - Often MVC and the hardest for us to make decisions on what goes where

MVC INSIDE THE PRESENTATION LAYER

- **Model** (in **M**VC, this is a **UI model**, not domain model)
  - What does the UI model do? Hold the screen's memory.
  - “It’s what the page knows, not what the app knows.”
  - “It remembers what the user just saw or typed.”
  - “No business rules, just view state.”

MVC INSIDE THE PRESENTATION LAYER

- What does a **view** (M**V**C) do?
  - Show stuff, nicely.
  - “It paints pixels, not logic.”
  - “It looks smart, but doesn’t think.”
  - “It’s the outfit, not the brain.”

MVC INSIDE THE PRESENTATION LAYER

- What does a **controller** (MV**C**) do?
  - Translate clicks into calls.
  - It’s the bouncer, ‘You click? Go there.’
  - It hands off, it doesn’t handle logic.
  - It listens, decides where to send things — then gets out of the way.

TYING IT TOGETHER

- Please view the bonus video for an example of refactoring code which has:
  - poor SoC
  - low cohesion
  - high coupling
- into
  - Clear SoC
  - Highly Cohesibe
  - Low Coupled
- code

PAUSE & PROJECT REFLECT

- Once you have watched the refactor video I suggest you:

Look at one of your classes.
Is it doing too much?
Too tightly tied to something else?

Make some git issues and a plan to address these design quality concerns.