Lecture 5: Design Principles

Review so far…

- W1: Intro ✓
- W2: Modelling with UML ✓
- W3-4: Design patterns ✓
- **W5-6: Design principles & system architecture**
- W7-8: Testing
- W9: Continuous integration
- W10: Review

Today's Plan

- 10:05-10:55:
  - Quiz 5 (assessed, correctness)
  - Main principles: Coupling and cohesion
  - PI 5 (assessed, participation)…
- Break
- 11:05-11:55:
  - …PI 5 (assessed, participation)
  - Other principles: Abstraction, flexibility, testability …
- Long break
- 13:00-14:00 OR 14:00-15:00 in BO1028: Lab 5 Design Principles

Quiz 5

- Join YACRS
  - Session: 1216
- It's design principle quiz time!

Q5.1 Coupling and Cohesion

A software engineer should always aim to:

- A: Increase both coupling and cohesion
- B: Decrease both coupling and cohesion
- C: Increase coupling and decrease cohesion
- D: Decrease coupling and increase cohesion

Q5.1 Coupling and Cohesion (solution)

A software engineering should always aim to:

- A: Increase both coupling and cohesion
- B: Decrease both coupling and cohesion
- C: Increase coupling and decrease cohesion
- **D: Decrease coupling and increase cohesion [correct]**

Q5.2 Abstraction

A software engineer should aim to:

- A: Keep the level of abstraction as high as possible
- B: Keep the level of abstraction as low as possible
- C: Keep a balanced level of abstraction
- D: Not worry about abstraction

Q5.2 Abstraction

A software engineer should aim to:

- **A: Keep the level of abstraction as high as possible – maintains flexibility and gives more options during development [correct]**
- B: Keep the level of abstraction as low as possible
- C: Keep a balanced level of abstraction
- D: Not worry about abstraction

Q5.3 Packages

Java packages are:

- A: Not useful when describing systems

- B: Difficult to represent in UML

- C: A key way to represent the divide and conquer principle
- D: Only able to have a low level of cohesion

Q5.3 Packages

Java packages are:

- A: Not useful when describing systems
- B: Difficult to represent in UML
- **C: A key way to represent the divide and conquer principle [correct]**
- D: only able to have a low level of cohesion

Design

Properties of "good" design:

- Increasing profit by reducing cost and increasing revenue
- Ensuring that we actually conform with the requirements
- Accelerating development
- Increasing qualities such as
  - Usability
  - Efficiency
  - Reliability
  - Maintainability
  - Reusability
  - …

Why are design principles important?

- Design principles provides a framework for talking about design decision and making principled (rational?) decisions (on a slightly higher level than e.g. design patterns)
- You are probably already using many of these, giving these principles the proper names/descriptions will improve your usage of them
- They can provide rationale for design decisions to make sure you come up with a “good” design
- They will improve your working practice and make your programming, managing, teaching life easier

DESIGN Principles

- **Divide and Conquer**
- **Cohesion**
- **Coupling**
- **Keep the level of abstraction as high as possible (already covered)**
- Increase reusability
- Reuse existing designs and code
- Anticipate obsolescence
- Design for portability
- Design for testability
- Design defensively

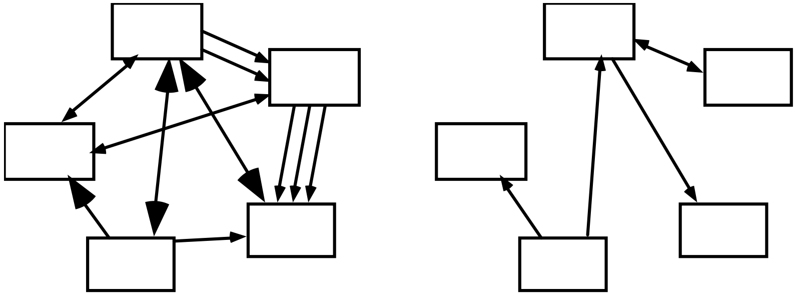
Divide and conquer

Divide and conquer (combine)

- Trying to deal with something big all at once is normally much harder than dealing with a series of smaller things
  - Separate people can work on each part.
  - An individual software engineer can specialize.
  - Each individual component is smaller, and therefore easier to understand.
  - Parts can be replaced or changed without having to replace or extensively change other parts.

Coupling and Cohesion

- Cohesion
  - How related things in a given class are
- Coupling
  - How dependent classes are on each other (between classes)
- Design to **increase cohesion** and **decrease coupling**



Cohesion

High Cohesion is Good

- Functional
- Layer
- Communicational
- Sequential
- Procedural
- Temporal
- Utility

Functional Cohesion

- Keep related functions together and keep everything else out.

- No side-effects (same result if you put in the same input at any point, no internal states)

- Benefits to the system:
  - Easier to understand
  - More reusable
  - Easier to replace

Layer Cohesion

- Lower level layers provide services to higher level layers via a well defined interface
  - Separate concerns into highly cohesive packages

Communicational Cohesion

- Grouping together classes and methods for accessing the same data
  - For example: Methods for reading/writing to logs, a specific database

Sequential, Procedural, and Temporal Cohesion

- All relate to functionalities that work together, but all refer to slightly different relationships.

- **Sequential** – Order of sequence is important because one method acts as input to the next

  - For example, a dialogue prompt that uses input from prompt for following methods

- **Procedural** – Methods are called one after another but don’t depend on each other
  - For example, functions with prepare lines of a log file.
- **Temporal** – Methods must be called together at a specific time
  - For example, a startup routine.

Utility Cohesion

- Grouping modules that are re-used but don't have a natural home
  - For example: Math or ImageIO Class

Coupling review

Coupling: High Coupling is Bad

- Content
- Common
- Control
- Data
- Routine
- Type Use
- Import
- External

Content Coupling

- Content from one module to another is not masked, but accessed directly
- Values from one object are modified directly by another or cached
  - Make your instance values private
  - Use getters/setters to control access

Content Coupling - example

Imagine a student class with a student ID attribute.

```
class Student(String )
{
       public String studentId;
       …
}

Student myStudent = new Student(“12345678”);

// Option A:
myStudent.studentId = “000000000000000000”;
>> [nothing]

// Option B:
myStudent.setStudentId(“000000000000000000”);
>> “Invalid ID, too long”
```

Common Coupling

- Global attributes shared between components
  - Components using the global attribute become coupled
    - Difficult to access control
    - Low Readability
    - Changes to the global variable affect many modules

Control Coupling

- The behaviour of a module is controlled by flags
  - Issue: One module must know the flags and logic of another module

- Solution: Polymorphism,…

Control Coupling

```
…
public drawRoutine(Command command) {
        if (command.equals(“Draw Circle”) {
                drawCircle();
        }
        else {
                drawRectangle();
        }
} …
```

Refactored using polymorphishm (introducing stamp coupling)…

```
public drawRoutine(Shape shape) {
        shape.draw()
}
```

Control Coupling

```
…
public runCode(String name) {
        if (IGNORE_CODE) {return;}

        …
} …
```

```
…
public myCode(String name) {
        if (DEBUG_CODE) {
                System.out.print(name);
        }
        …
} …
```

Stamp Coupling

- Stamp coupling occurs when a data type is used as a parameter to an method.
  - If a module takes an object argument and only uses a very small amount of the data contained

For example:

```
produceReport(Student student)
```

```
produceReport(String studentID)
```

Stamp Coupling

```
public interface Identifier
{
  public abstract String getId();
  public abstract String …
}

public class Student implements Identifier {…}
public class Employee implements Identifier {…}

public class Report
{
  public void produceReport(Identifier o)
  {...}
  ...
}
```

Data coupling

- Data coupling occurs when many variables are given as simple arguments – all function that uses this must parse all of them.

- The more arguments a method has, the higher the coupling (and complexity)

- Stamp (object) vs Data (simple, but many) trade-off

```
…
enrollStudent(String studentName, int studentId,
      Enrollment enrollmentRole)

enrollStudent(Student student)
…
```

Routine Coupling

- Routine coupling occurs when several functions need to be called together to work
- Possible solution: reduce routine call coupling by writing a single routine that encapsulates the sequence.

Routine Coupling - Example

```
…
myTouchPoint.drawBorder(x,y);
myTouchPoint.drawBackground(x,y);
myTouchPoint.drawShadow(x,y);
…
```

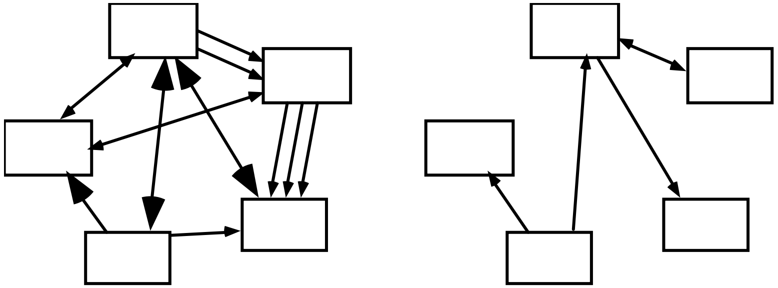
OR

```
myTouchPoint.draw(27,56);
…
public void draw(Int x, Int y) {
       this.drawBorder(x,y);
       this.drawBackground(x,y);
       this.drawShadow(x,y);
}
```

Type Use Coupling

- If a class uses a data type defined in another class, this introduces coupling

- Can you implement an associations without a type coupling ? Perhaps use interfaces in some way and a pattern…



Import and External Coupling

- import coupling occurs when one module imports the functionality provided by another
- external coupling occurs when an application depends on the configuration of the environment.
  - Sometimes the Façade pattern can help

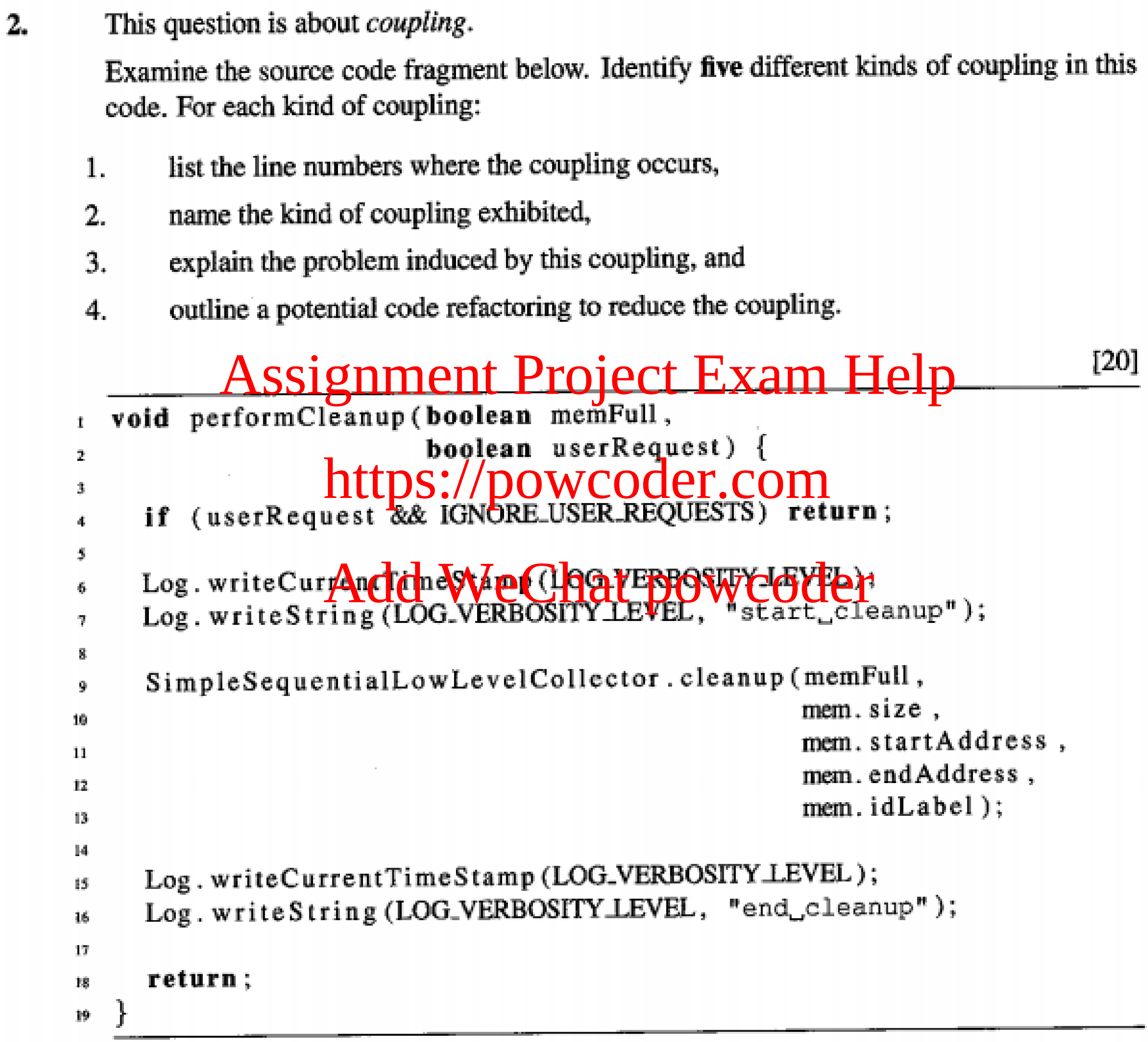
Break

PI 5

- YACRS session: 1217

- On which line does content coupling occur?

PI 5.1 Coupling and cohesion exam question



2. This question is about *coupling*.

Examine the source code fragment below. Identify **five** different kinds of coupling in this code. For each kind of coupling:

1. list the line numbers where the coupling occurs,
2. name the kind of coupling exhibited,
3. explain the problem induced by this coupling, and
4. outline a potential code refactoring to reduce the coupling.

[20]

```
1  void performCleanup(boolean memFull,
2                      boolean userRequest) {
3
4    if (userRequest && IGNORE_USER_REQUESTS) return;
5
6    Log.writeCurrentTimeStamp(LOG_VERBOSITY_LEVEL);
7    Log.writeString(LOG_VERBOSITY_LEVEL, "start␣cleanup");
8
9    SimpleSequentialLowLevelCollector.cleanup(memFull,
10                                             mem.size,
11                                             mem.startAddress,
12                                             mem.endAddress,
13                                             mem.idLabel);
14
15   Log.writeCurrentTimeStamp(LOG_VERBOSITY_LEVEL);
16   Log.writeString(LOG_VERBOSITY_LEVEL, "end␣cleanup");
17
18   return;
19 }
```

PI 5.1 Exam question

- **Routine Coupling**, Lines must be called together to create intended effect, Lines 6 -7, 15-16
  - Solution: write a separate function to handle this, e.g. easy to change file format etc

- **Common Coupling**, LOG VERBOSITY LEVEL, IGNORE_USER_REQUEST could be a global variable, Lines 6-7, 15-16
  - Solution:

- **Control Coupling**, Some flag (could also be variables) appear to be passed in to control this functions behaviours , Line 4. User would need to know the logic of the flags
  - Normally, mem object could e.g. contain it's own clean-up
  - Line 4: Move the logic outside

- **Data Coupling**: Many attributes or provided when arguments could be simplified, Lines 9 to 13
  - Solution: Parse in object, at least only parse necessary arguments

- **Content Coupling**: Attributes are accessed directly rather than through a defined function or interface, Line 9-13
  - Solution: parse in object and ensure to get and set on private.

- **External**: Memory structure seems to be assumed

Summary

- Design principles (especially cohesion and coupling)
- General principles; not rules…
  - Modelling techniques from W2 can help visualize and describe a **domain** and inform the **design**
  - The design patterns you have seen in W3-4 can help minimize coupling – but not necessarily required to ensure low coupling
- Lab 5 at 13:00 or 14:00 in BO1028

Next Week

- Design Principles (cont.) and **System Architecture (including documentation)**
- Read:
  - Chapter 9.5 ->