Objectives

- Open-Closed Principle
- Code Smells
- Refactoring

Reflection on Project 1

- What were the difficult parts of Project 1?
- Did they get any easier?
- Did you develop a system or any techniques to make the process easier?
- In the future, how could you make the process easier?
- What do you think of JUnit in Eclipse?

Don't forget what you've learned.
Integrate testing into your development.

Review

- What is guaranteed in software development?
- What are some principles of design in Object-oriented Programming to address the challenge posed by that guarantee?
- What is the underlying theme of how to achieve those principles?

Review: Best Practices

- (DRY): Don't repeat yourself
- Single responsibility principle
- Shy
  - Avoid Coupling
- Tell, Don't Ask
- Open-closed principle
- Avoid code smells

Open-Closed Principle

- Bertrand Meyer
  - Author of *Object-Oriented Software Construction*
    - Foundational text of OO programming

Principle: Software entities (classes, modules, methods, etc.) should be **open** for **extension** but **closed** for **modification**

- Design modules that *never change* (after completely implemented)
- If requirements change, extend behavior by adding code
  - Don't change existing code → won't create bugs!

Attributes of Software that Adhere to OCP

- Open for Extension
  - Behavior of module can be extended
  - Make module behave in new and different ways
- Closed for Modification
  - No one can make changes to module

These attributes seem to be at odds with each other.
How can we resolve them?

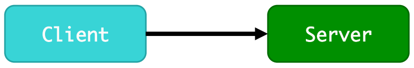
Using Abstraction

- Abstract base classes
  - Fixed abstraction → API
  - Cannot be changed
- Derived classes: *possible behaviors*
  - Can always create new child classes of abstract base class

Not Open-Closed Principle

- `Client` uses `Server` class

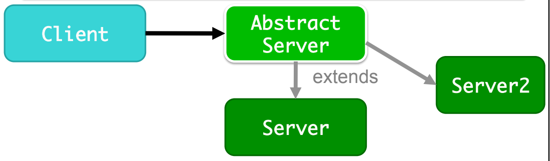
```
public class Client {
      public void method(Server x) {
      …
      }
}
```



Open-Closed Principle

- `Client` uses `AbstractServer` class

```
public class Client {
      public void method(AbstractServer x) {
      …
      }
}
```



Strategic Closure

- No significant program can be completely closed
- Must choose kinds of changes to close
  - Requires knowledge of users, probability of changes
  - Most probable changes should be closed

Heuristics and Conventions

- Member variables are private
  - A method that depends on a variable cannot be closed to changes to that variable
  - The class itself can't be closed to it
    - All other classes should be
- No global variables
  - Every module that depends on global variable cannot be closed to changes to that variable
  - What happens if someone uses variable in unexpected way?
  - Counter examples: `System.out, System.in`

➥Apply abstraction to parts you think are going to change

Code Smells

A hint in the code that something could be designed better

- Duplicated code
- Long method
- Large class
- Long parameter list
- Very similar subclasses
- Too many public variables
- Empty catch clauses
- Switch statements/long if statements
- Shotgun surgery
- Literals
- Global variables
- Side effects
- Using `instanceof`

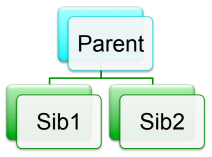
Duplicated Code

- What's the problem with duplicated code?
- Why do we like it?
  - ➢ What made us write the duplicated code?

What can we do when we have duplicated code?
(How can we get rid of the duplicate code?)

Duplicated Code

- Example: same expression in 2 methods of the same class
  - ➢ Solution: Extract method
  - ➢ Call method from those two places
- Example: duplicated code in 2 sibling subclasses



Duplicated Code

- Example: duplicated code in 2 sibling subclasses
  - ➢ Extract method, put into parent class
  - ➢ If similar but not duplicate, extract the duplicate code (or parameterize)
- Example: duplicated code in unrelated classes

Duplicated Code

- Example: duplicated code in unrelated classes
  - ➢ Ask: where does method belong?
  - ➢ One solution:
    - Extract class
    - Use new class in classes
  - ➢ Another solution:
    - Keep in one class
    - Other class calls that method

Refactoring: Solution to Code Smells

Refactoring: Updating a program to improve its design and maintainability *without changing its current functionality significantly*

- Example
  - ➢ Creating a single function that replaces 2 or more sections of similar code
    - Reduces redundant code
    - Makes code easier to debug, test

After refactoring your code, what should you do next?

Long Methods

- What's the problem with long methods?
- What made us write them?

Long Methods: Issues and Solutions

- Issues:
  - Hard to understand (see) what method does
  - Smaller methods have reader overhead
    - Look at code for called methods
    - But, should use descriptive names
- Solutions:
  - Find lines of code that go together (may be identified by a comment) and extract method

Large Class

- What's the problem?

Large Class

- Issue: Too many instance variables → trying to do too much (Single Responsibility)
- Solutions:
  - Bundle groups of variables together into another class
    - Look for common prefixes or suffixes
  - If includes optional instance variables (only sometimes used), create child classes
  - Look at how users use the class for ideas of how to break it up

Eclipse: Refactor → Extract Class or Extract Superclass

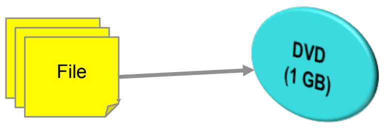
Long Parameter List

- More difficult to use (do I have everything?)
- If method signature changes, have a lot of places to change
- Solutions: Use objects
  - Instead of separate parameters for an object's data
  - Group parameters together

Eclipse: Refactor → Introduce Parameter Object OR Refactor → Change Method Signature

Bin-Fitting Problem

- Classic CS problem: fit as many of something (A) into as few (B) as possible
- Example
  - A: Files, which have a size
  - B: CDs or DVDs (Disks)



Heuristics

- Worst fit
  - Store file in disk with most free space
- In-order worst fit
  - Put files on disk, in order seen
- In-decreasing-order worst fit
  - Sort files by size
  - Put on disks

Finding the Disk With Most Free Space

- Keep the disks in sorted order by their free space
  - ➢ Java class: `PriorityQueue`
    - Uses `compareTo` method or `Comparator`

Getting A Solution

- Import → General → Existing project into Workspace
  - ➢ Archive file: /home/courses/cs209/handouts/bins.tar
- Try running Bin.java
  - ➢ Run options
  - ➢ Argument: data/example.txt

Refactoring Discussion

Looking at the `main` method on the handout…

- How clearly written is the code?
- What, if any, comments might be helpful within the code?
- Does it satisfy its role as a tutorial?
- What, if any, suggestions does this code make about how the remaining parts of the assignment will be written?
- How would you test this code for bugs?

Assignment 10: Code Critique & Refactoring

- Given: a problem specification and a solution to the problem
  - ➢ You refactoring your own code is emotional
  - ➢ More objective with someone else's solution
- Goals
  - ➢ Read and understand someone else's code
    - Haven't done much of this in Java
  - ➢ Critique code (do you smell something?)
    - Identify, articulate problems
  - ➢ Refactor code to solve problems identified
  - ➢ Write tests to verify the code