Objectives

- Design in the Small
- Code Smells
- Refactoring

Review

1. What is code coverage?
2. What is code coverage *criteria*?
   - Provide examples of code coverage criteria
3. How can you use/apply code coverage?
   - In what type of testing can code coverage be used?
4. What are the benefits and limitations of code coverage?

Review: Code Coverage

- Code coverage: the amount of code that your tests execute
- Code coverage criteria: metric used
  - Statement: number/% of statements executed
  - Branch: number/% of statements + branches (conditions, loops) executed
  - Path: number/% of paths executed

Review: Uses of Coverage Criteria

- "Stopping" rule → sufficient testing
  - Avoid unnecessary, redundant tests
- Measure test quality
  - Dependability estimate
  - Confidence in estimate
- Specify test cases
  - Describe additional test cases needed

Review: Coverage Limitations

- A test suite of test cases that all pass that has 100% [statement/branch/path] coverage of does **not** mean bug-free code
  - Errors of omission
    - Can’t cover what isn’t there
  - Different data values on same execution path may expose errors

Coverage + Other smarts to Create Good Tests → High-quality code

OBJECT-ORIENTED DESIGN PRINCIPLES

Designing Systems

All systems **change** during their life cycle

- Requirements change
- Misunderstandings in requirements
- New functionality

- Code must be ***soft***
  - Flexible
  - Easy to change
    - New or revised circumstances
    - New contexts
    - Fix bugs

Designing for Change Example

- July 2010, Oracle released Java 6 update 21
  - Generated java.dll replaced
    - COMPANY_NAME=Sun Microsystems, Inc. with
    - COMPANY_NAME=Oracle Corporation
- Change caused `OutOfMemoryError` during Eclipse launch
  - Eclipse versions 3.3-3.6 (widespread!)
  - Why? Eclipse used the company name in the DLL in startup (runtime parameters) on Windows
- Temporary Fix: Oracle changed name back
- Required changes to all Eclipse versions

Source: `http://www.infoq.com/news/2010/07/eclipse-java-6u21`

Designing Systems

All systems **change** during their life cycle

- Questions to consider:
  - How can we create designs that are stable in the face of change?
  - How do we know if our designs aren't maintainable?
  - What can we do if our code isn't maintainable?
- Answers will help us
  - Design our own code
  - Understand others' code

Designing Systems

All systems **change** during their life cycle

- Questions to consider:
  - **How can we create designs that are stable in the face of change?**
  - How do we know if our designs aren't maintainable?
  - What can we do if our code isn't maintainable?
- Answers will help us
  - Design our own code
  - Understand others' code

Best Practices Overview

- (DRY): Don’t repeat yourself
- Shy Code, Avoid Coupling
- Tell, Don’t Ask
- Avoid code smells

- SOLID
  - Single Responsibility Principle
  - Open-closed principle
  - Liskov Substitution Principle
  - Interface Segregation Principle
  - Dependency Inversion Principle

A lot of related fundamental principles. We have been using them/applying them, just haven’t named them.

Don’t Repeat Yourself (DRY): Knowledge Representation

> *Every piece of knowledge must have a single, unambiguous, and authoritative representation within a system*

- **Intuition**: when need to change representation, make in only one place
- Requires planning
  - What data needed, how represented (e.g., type)
  - Consider documentation as well

Don't Repeat Yourself (DRY): Knowledge Representation

> *Every piece of knowledge must have a single, unambiguous, and authoritative representation within a system*

- Example:
  - `Car` class defined constants for gears
  - `CarTest` should refer to those constants
    - Not redefine those gears, nor just hardcode numbers
    - The values are likely to change, so refer to the variables.

Don’t Repeat Yourself (DRY): Knowledge Representation

Every piece of knowledge must have a single, unambiguous, and authoritative representation within a system

- Example:
  - `Birthday` class had a month
    - Could be represented as a number and a String
  - Best: represent as a number (only), i.e., only one instance variable to represent the month
    - Get month String from the number (e.g., `MONTHS_OF_YEAR[month-1]`)
  - Why?

Don’t Repeat Yourself (DRY): Knowledge Representation

Every piece of knowledge must have a single, unambiguous, and authoritative representation within a system

- Example:
  - `Birthday` class had a month
    - Could be represented as a number and as a String
  - Best: represent as a number (only), i.e., only one instance variable to represent the month
    - Get month String from the number (e.g., `MONTHS_OF_YEAR[month-1]`)
  - Why? If need to update the month, just one variable needs to be updated, not two, which *can get out of sync*

Shy Code

- Goal: Won’t reveal *too much* of itself
- Otherwise: get *coupling*
  - Coupling: dependence on other code
  - Static, dynamic, domain, temporal

What techniques have we discussed for how to keep our code shy?

- Coupling isn’t always bad…
  - Can’t be completely avoided…
  - We want *shy* code – not completely isolated code

Achieving Shy Code

How can you make any field immutable?

- Private instance variables
  - Especially mutable fields
- Make classes public only when need to be public
  - i.e., accessible by other classes→ part of API
- Getter methods shouldn’t return private, mutable state/objects
  - Use `clone()` before returning

Coupling Overview

- Interdependence of classes
  - Dependence makes class susceptible to breaking if other class changes
- Class A is *coupled* with class B if class A
  - Has an object of type B
    - Instance variable, Parameter, return type
  - Calls on methods of object B
  - Is a child class of or implements B
- Goal: *Loose* coupling
  - Non-goal: no coupling

Static Coupling

- Code requires other code to compile
- Clearly, we need some static coupling!
  - Example: to display a line of text, we need the code for System.out
- Problem if you include more than you need

Static Coupling

- Code requires other code to compile
- Problem if you include more than you need
  - Example: poor use of inheritance
    - Brings excess baggage
    - Inheritance is reserved for “is-a” relationships
      - Base class should not include optional behavior
      - Not “uses-a” or “has-a”
- Solution: use *composition* or *delegation* instead

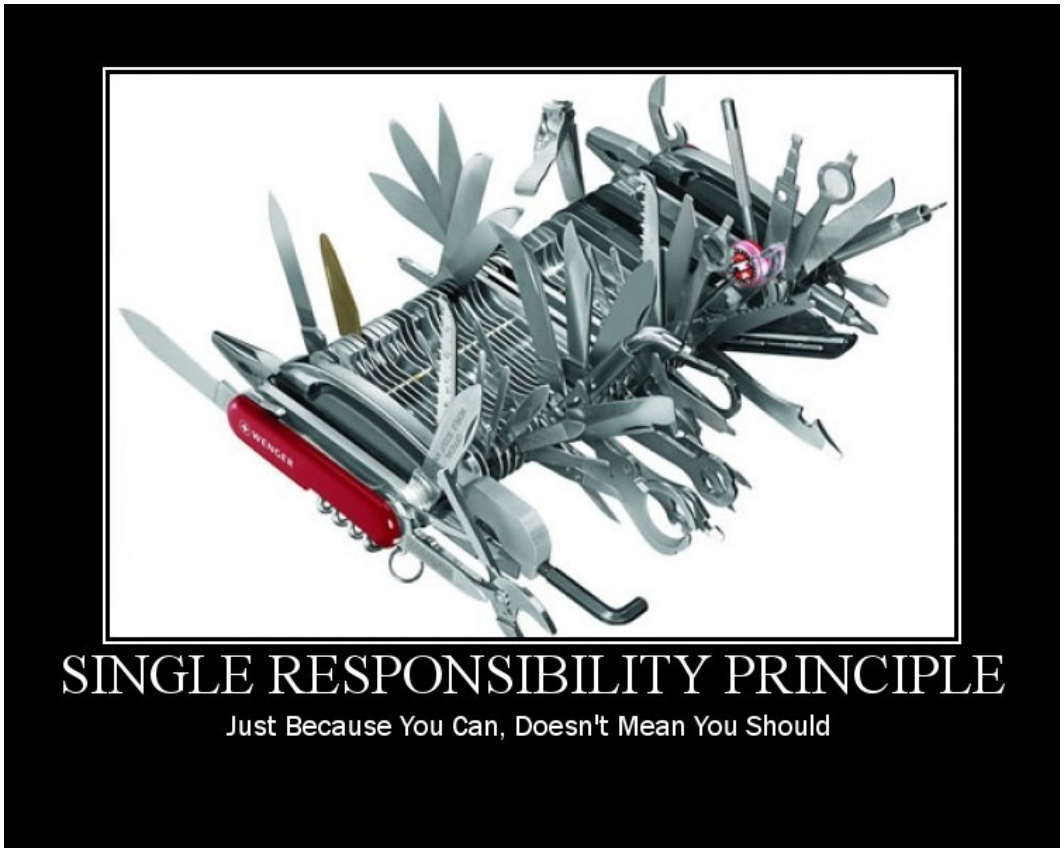
Static Coupling

- Code requires other code to compile
- Problem if you include more than you need
- Solution: use *composition* or *delegation* instead
  - Example: I created a class where I have keys associated with values. I shouldn't extend HashMap, but ***use*** a HashMap
  - Example: GamePiece class did not and *should not* include *chase* functionality
    - Only certain child classes need that functionality

Tell, Don’t Ask

- When designing methods, think of them as *sending a message*
  - Send a message
  - Get a response
- Method call: 1) sends a request to do something; 2) response is what is returned
  - Don’t ask about details
  - Black-box, encapsulation, information hiding
- Example: `hasSameBirthday(Birthday[] birthdays)`
  - Input: the array of birthdays to the method
  - Output: true/false if two people had the same birthday
    - Don’t need to know how it was determined; no printing of output

Single Responsibility Principle


WENGER
SINGLE RESPONSIBILITY PRINCIPLE
Just Because You Can, Doesn't Mean You Should

Single Responsibility Principle (SRP)

There should never be more than one reason for a class to change

- **Intuition**:
  - ➢ Each responsibility is an axis of change
    - More than one reason to change
  - ➢ Responsibilities become coupled
    - Changing one may affect the other
    - Code breaks in unexpected ways

This idea has come up before in class. Give an example of adhering to SRP.

Open-Closed Principle (OCP)

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

- Bertrand Meyer
  - Author of *Object-Oriented Software Construction*
    - Foundational text of OO programming
- Design modules that *never change* after completely implemented
- If requirements change, extend behavior by adding code
  - By not changing existing code → we 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?

OCP Solution: Use Abstraction

- Abstract base class or interface
  - ***Fixed*** abstraction → API
  - Cannot be changed (closed to modification)
- Derived classes: *possible behaviors*
  - Can always create new child classes of abstract base class
  - (Open to extension)

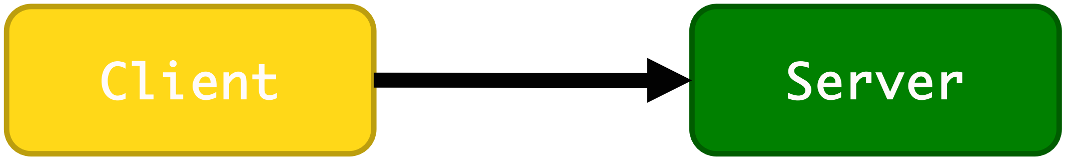
OCP Solution: Use Abstraction

- Abstract base classes or interfaces
  - Fixed abstraction → API
  - Cannot be changed (closed to modification)
- Derived classes: *possible behaviors*
  - Can always create new child classes of abstract base class
  - (Open to extension)
- Example: Create a new Baddie for Game
  1. Add a new Baddie class that derives from GamePiece
  2. Replace old goblin instantiation with new baddie in game
  3. DONE!

Not Open-Closed Principle

- `Client` uses `Server` class

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



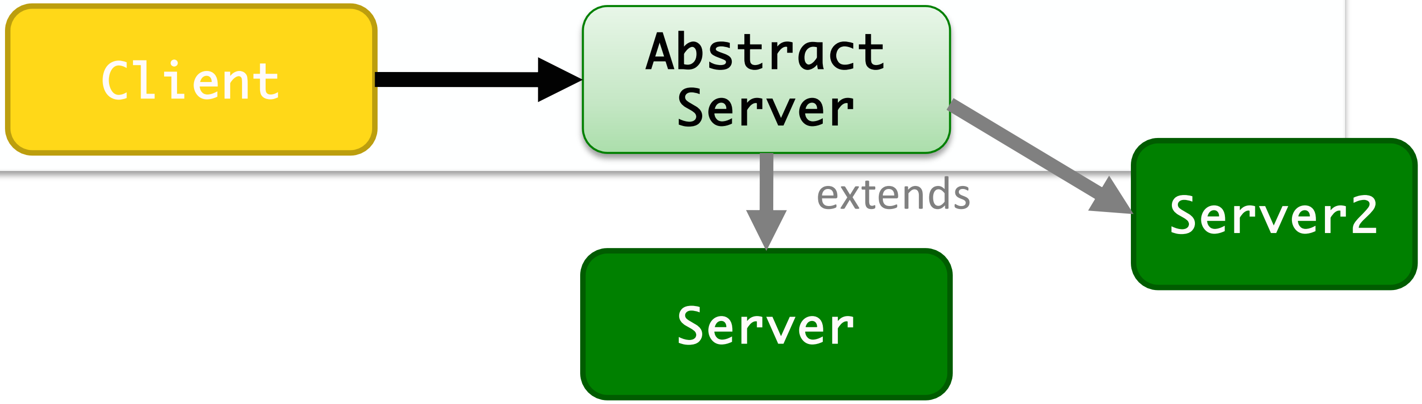
Open-Closed Principle

Or ServerInterface

- Client uses AbstractServer class

```
public class Client {
        public void method(AbstractServer x) {
                // method implementation uses only methods
                // from AbstractServer
                …

        }
}
```

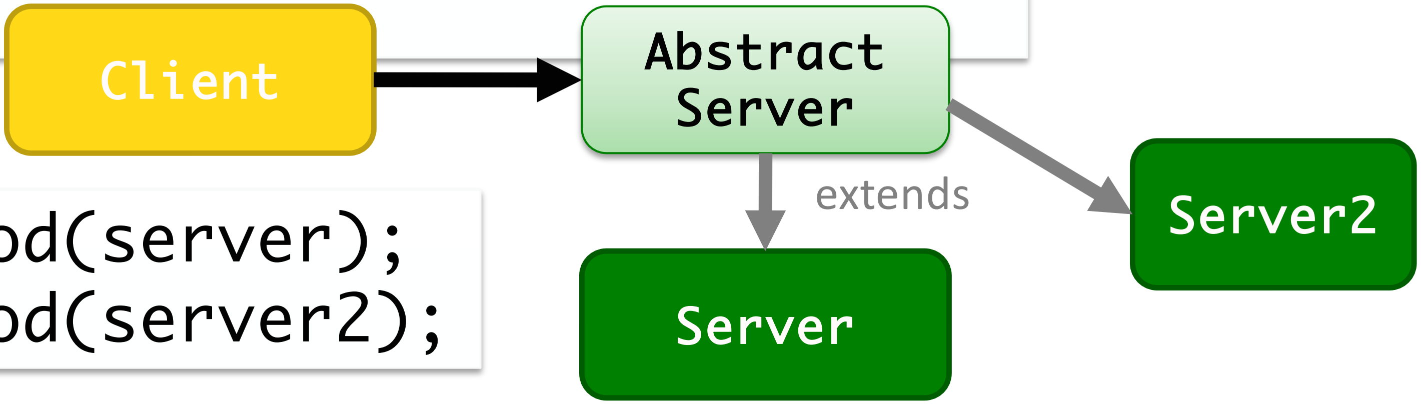


Open-Closed Principle

Or ServerInterface

- Client uses AbstractServer class

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



```
client.method(server);
client.method(server2);
```

Strategic Closure

- No significant program can be completely closed
- Must choose which changes to close
  - Requires knowledge of users, probability of changes

Goal: 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 a 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

Designing Systems

All systems **change** during their life cycle

- Questions to consider:
  - How can we create designs that are stable in the face of change?
  - **How do we know if our designs aren’t maintainable?**
  - **What can we do if our code isn’t maintainable?**
- Answers will help us
  - Design our own code
  - Understand others’ code

Code Smells

A hint in the code that something could be designed better

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

Code Smell Case Study: Duplicated Code

- What's the problem with duplicated code?
- Why do we like it?
  - What made us write the duplicated code?
- Refactor: How can we get rid of the duplicate code?
  - Consider different possibilities for where the duplicate code is
    - Same expression multiple times in a class
    - Duplicate code in 2 sibling child classes
    - Duplicate code in unrelated classes

Problem of Duplicated Code

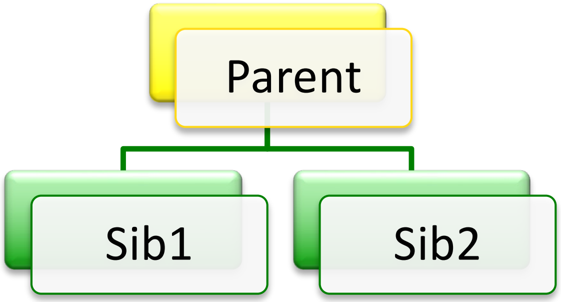
- If code changes, need to change in every location
- Duplicate effort to test code to make sure it works
  - More statements for test suite to test!
- When trying to search for code, may find a duplicate code→ not the one you're looking for
  - Increased effort in debugging

Duplicated Code Refactorings

- Consider: same expression multiple times in one class
- Solution: Extract method
  - Call method from those two places
- Benefits:
  - Reduces redundant code
  - Makes code easier to debug, test

Duplicated Code Refactorings

- Consider: duplicated code in 2 sibling child classes



- Solution: Extract method, put into parent class
  - Eclipse: extract method, pull up
- If similar but not duplicate, extract the duplicate code or parameterize

Duplicated Code Refactorings

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

Why so much time on duplicated code?
It’s a common yet costly problem.

Discussion: Duplicate Code

- Consider some code examples from the semester:
  1. `Object` and `Birthday` both have `equals(Object o)` methods
  2. `Goblin` and `Human` both have `takeTurn(Game game)` methods
- Do they have duplicate code? Were they poorly designed?

Discussion: Duplicate Code

- Consider some code examples from the semester:
  1. `Object` and `Birthday` both have `equals(Object o)` methods
  2. `Goblin` and `Human` both have `takeTurn(Game game)` methods
- Do they have du
  designed?

No! Having the same method signature does *not* necessarily mean that they have duplicate code.

Refactoring: Solution to Code Smells

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

After refactoring your code, what should you do next?

Process to Write Maintainable Code

Apply the design principles, but as your code evolves, you'll see that you didn't always adhere to the principles

1. Identify code smell
2. ***Refactor*** code to remove code smell
3. ***Test*** to confirm code still works!

Looking Ahead

- Testing project due Wednesday 11:59 p.m.
- Testing analysis due Thursday 11:59 p.m.
- Friday-Sunday: Exam 2
    - No class, I am available for office hours