Objectives

- Testing wrap up
- Design in the Small

1

Review

1. What is code coverage? Code coverage criteria?
2. How can you use code coverage?
3. What are the benefits and limitations of code coverage?

2

Review: Code Coverage

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

3

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

4

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

5

Categories of Testing
(Non-Exhaustive)

- Black-box testing
- White-box testing
- Non-functional testing
- Acceptance testing

Ideas or definitions of any of these?

6

Categories of Testing
(Non-Exhaustive)

- Black-box testing
  - Test *functionality* (e.g., the calculator)
  - No knowledge of the code
  - Examples of testing: boundary values
- White-box testing
  - Have access to code
  - **Goal**: execute *all* code
- Non-functional testing
  - Performance testing
  - Usability testing (HCI)
  - Security testing
  - Internationalization, localization
- Acceptance testing
  - Customer tests to decide if accepts product

More Testing Tools, Frameworks

- Mockito
  - Mock objects before have other code
  - Allows you to test in isolation, e.g., mock the payment system so you focus on your code
- Cucumber
  - Behavior-driven development
  - Language parser: Gherkin
- Many more

OBJECT-ORIENTED DESIGN PRINCIPLES

9

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

10

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 name in the DLL in startup (runtime parameters) on Windows
- Temporary Fix: Oracle changed name back
- Requires changes to all Eclipse versions

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

11

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

12

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

Overview Best Practices

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

A lot of similar, related fundamental principles

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)

15

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

16

Single Responsibility Principle

17

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

We've talked about this idea in this class. Give an example of adhering to SRP.

18

Example


```
interface Network {
      public void connect();
      public void disconnect();
      public void send(String s);
      public String receive();
}
```

- Reasonable interface
- But has more than one responsibility
- Check:
  - Change for different reasons? Called from different parts of program?

Example


```
interface Network {
      public void connect();
      public void disconnect();
      public void send(String s);
      public String receive();
}
```

- Reasonable interface
- But has more than one responsibility
- In Java
  - Socket class does connect/disconnect
  - Use separate Streams to send and receive data on the Socket

Shy Code

- Won't reveal too much of itself
- Otherwise: get *coupling*
  - Coupling: dependence on other code
  - Static, dynamic, domain, temporal
- Coupling isn't always bad…
  - Can't be completely avoided…

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

Achieving Shy Code

- Private instance variables
  - Especially mutable fields

How can you make any field immutable?

- 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

- 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 or implements class 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
  - Example: I am creating a class where I have keys associated with values. I shouldn't extend HashMap, but use a HashMap
  - Example: GamePiece class 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: sends a request to do something
  - Don't ask about details
  - Black-box, encapsulation, information hiding
- Example: isPalindrome(String s)
  - Input: the "raw" string to the method
  - Output: if it's a palindrome or not
    - Don't need to know how the spaces and casing were ignored

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
  - 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?

Using Abstraction

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

Using Abstraction

- Abstract base classes or interfaces
  - Fixed abstraction → API
  - Cannot be changed
- Derived classes: *possible behaviors*
  - Can always create new child classes of abstract base class
- Assignment 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) {
     …
     }
}
```

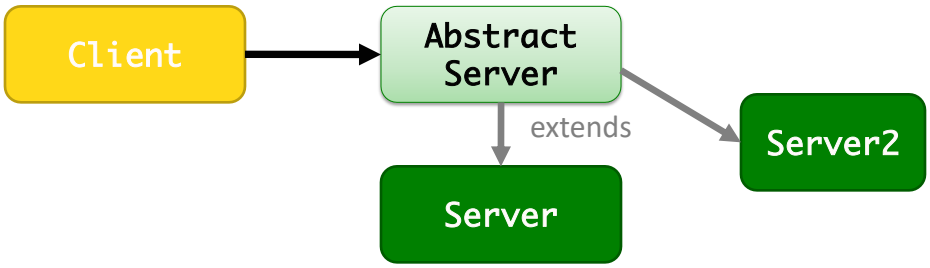
33

Open-Closed Principle

Or `ServerInterface`

- `Client` uses `AbstractServer` class

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



34

Strategic Closure

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

Goal: Most probable changes should be closed

35

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

36

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

37

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`

38

Process to Write Maintainable Code

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

1. Identify code smell

2. ***Refactor*** code to remove code smell

Code Smell: Duplicated Code

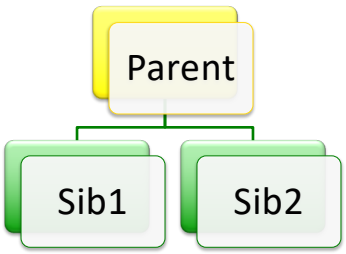
- What's the problem with duplicated code?
- Why do we like it?
  - What made us write the duplicated code?
- What can we do when we have duplicated code?
  - (How can we get rid of the duplicate code?)
  - Consider different possibilities for where the duplicate code is

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 in at least one method of a class
  - Solution: Extract method
  - Call method from those two places
- Consider: duplicated code in 2 sibling child classes



Duplicated Code Refactorings

- Consider: duplicated code in 2 sibling child classes
  - Extract method, put into parent class
    - Eclipse: extract method, pull up
  - If similar but not duplicate, extract the duplicate code or parameterize
- Consider: duplicated code in unrelated classes

Duplicated Code Refactorings

- Consider: 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

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

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 method 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?

Revised Process to Write Maintainable Code

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

1. Identify code smell

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

Code Smell: Long Methods

- What's the problem with long methods?
- What made us write them?
- How can we fix them?
- What is an issue with lots of short methods?

47

Looking Ahead

- More code smells
- Wed – testing project analysis due

48