Objectives
• Coverage tools
• Object-oriented Design Principles
   Design in the Small

Project 1 Questions?

Project 1 Notes
• Test-driven development
  ➢ Incomplete comments, pre-/post conditions
  ➢ Make reasonable assumptions
    • Document assumptions in your test code
  ➢ Write the specification that code has to pass
• Organizing tests
  ➢ Can have multiple test classes
    • Organize by fixture, functionality, all pass, all errors
• Independent test cases
  ➢ Each tests different functionality
  ➢ Should only have one failure
    • Easier to locate the bug
• Handling error cases
  ➢ Sometimes an exception is the expected result
    Add an "expected" attribute:
    ```java
    @Test(expected=IndexOutOfBoundsException.class)
    public void testIndexOutOfBoundsException() {
      ArrayList emptyList = new ArrayList();
      Object o = emptyList.get(0);
    }
    ```

Project 1 Notes
• Do not change the Car class’s API or it’s package
  ➢ Otherwise, won’t work with my Car class
• May want to write code for Car class to help you figure out tests

Review
• How do we know when we’ve tested enough?
• How can we use coverage criteria?
• Why is coverage not enough?
  ➢ What can we do to improve testing?
Analogy: Map coverage

Goal: Expose all the “scarecrows”

On CSI

http://i.imgur.com/prFIq.jpg

Coverage Tools
• Coverage is used in practice
• You don’t need to figure out coverage manually
• Available tools to calculate coverage
  ➢ Examples for Java programs: Clover, JCoverage, Emma
  ➢ Measure statement, branch/conditional, method coverage

Eclipse Plugin: EclEmma for Coverage
• Eclipse can be extended through plugins
  ➢ Provide additional functionality
• EclEmma Plugin
  ➢ Records executing program’s (or JUnit test case’s) coverage
  ➢ Displays coverage graphically

Demonstration
• Execute MediaItemTest with Coverage

Installing Emma in Eclipse At Home
• Under Help → Install New Software
  ➢ Add… a new remote site
    ➢ Name: EclEmma
    ➢ URL: http://update.eclemma.org/

• Select to install Emma
  ➢ Go through process
• Restart Eclipse
The Bug

Set is calling equals(Object o) when it adds an element

What method did we define?

```java
public boolean equals(Bigram b) {
    return b.first == first && b.second == second;
}
```

We overloaded the equals method.

Using @Override annotation

```java
@Override
public boolean equals(Bigram b) {
    return b.first == first && b.second == second;
}
```

Compiler tells us there is a problem.

How do we fix?

Designing Systems

- All systems change during their life cycle
  - Changes in requirements
  - Misunderstandings in requirements
- Code must be soft
  - Flexible
  - Easy to change
  - New or revised circumstances
  - New contexts
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

- (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

Dry: Knowledge Representation

- **Intuition:** when need to change code, make in only one place
  
  Every piece of knowledge must have a single, unambiguous, and authoritative representation within a system

- Requires planning
  - What data needed, how represented (e.g., type)

Single Responsibility Principle

- **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

Example

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

- Reasonable interface
- But has two responsibilities
  - Can you group the functionality into two responsibilities?
- Check:
  - Change for different reasons? Called from different parts of program?

Shy Code

- Won’t reveal too much of itself
- Otherwise: get coupling
  - Static, dynamic, domain, temporal

- Coupling isn’t always bad…
Achieving Shy Code

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

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

How can you make any field immutable?

• Think of methods as “sending a message”
   Method call: sends a request to do something
   Don’t ask about details
   Black-box, encapsulation, information hiding
   Return: answer

Tell, Don’t Ask

• Code requires other code to compile
   Not really a bad thing
   BUT don’t drag in 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”
   Want composition or delegation instead

Static Coupling

Dynamic Coupling

• Code uses other code at runtime
   getOrder().getCustomer().getAddress().getState()
   Relies on several objects/classes and their state
   Talk directly to code

Domain Coupling

• Business rules, policies are embedded in code
   Problem if change frequently
   Code will have to change frequently

• Put into another place (metadata)
   Database, property file
   Process the rules
Temporal Coupling

- Dependencies on time
  - Order that things occur
  - Occur at a certain time
  - Occur by a certain time
  - Occur at the same time

⇒ Write concurrent code

Project 1 Due on Friday