Objectives

- Coverage tools
- Object-oriented Design Principles
  - Design in the Small
  - DRY
  - Single responsibility principle
  - Shy
  - Open-closed principle

COVERAGE TOOLS

Coverage Tools

- Coverage is used in practice
- Don't need to figure out coverage manually
- Available tools to calculate coverage
  - Examples for Java programs: Cobertura, Clover, JCoverage, **Emma**
  - Measure statement, branch/conditional, method coverage

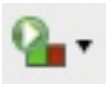
Eclipse Plugin: EclEmma for Coverage

- Eclipse can be extended through ***plugins***
  - Provide additional functionality
- EclEmma Plugin
  - Records executing program's (or JUnit test case's) coverage
  - Displays coverage graphically

Installing Emma in Eclipse

1. From your Eclipse menu select *Help* → *Eclipse Marketplace*.
2. Search for "EclEmma".
3. Hit *Install* for the entry "EclEmma Java Code Coverage".
4. Follow the steps in the installation wizard.

Demonstration



- Execute test with coverage

Note: Coverage and Testing Project

- You won't be able to leverage coverage for the testing project
  - Even if you write code, it's not _my_ code.
- Challenge of test-driven development (TDD)
  - Common practice in industry

OBJECT-ORIENTED DESIGN PRINCIPLES

Designing Systems

All systems **change** during their life cycle

- Requirements change
- 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

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)

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 SRP.

Shy Code

- Won't reveal too much of itself
- Otherwise: get *coupling*
  - Static, dynamic, domain, temporal
- Coupling isn't always bad…

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

Static Coupling

- Description: 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

Dynamic Coupling

- Description: Code uses other code at runtime
  - `getOrder().getCustomer().getAddress().getState()`
- Why a problem: Relies on several objects/classes and their state
  - If others change, my code has to change
- Solution: Talk *directly* to code

Domain Coupling

- Description: Business rules, policies are embedded in code
- Why a problem: if change frequently, code has to change frequently
- Solution: put into another place (metadata)
  - Database, property file
  - Process the rules

Temporal Coupling

- Description: Dependencies on time
  - Order that things occur
  - Occur at a certain time
  - Occur by a certain time
  - Occur at the same time
- Solution: Write *concurrent* code

Tell, Don't Ask

- 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

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?*

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