This study guide contains the slides from past classes, and the draft slides for future classes (obviously, these latter are subject to change). Over the course of the semester I will add study questions and additional exercises as time allows.

Contents

1 Introduction ........................................... 3
2 Bugs and tests ........................................ 6
3 Test-driven design .................................... 10
4 Diving into writing tests ............................... 14
   4.1 Using JUnit ........................................... 14
   4.2 Black-box (functional) and white-box (structural) testing ........................................... 16
   4.3 Some particulars of object-orientation ........................................... 17
5 Adapting existing code for testing ..................... 21
   5.1 Introduction ........................................... 21
   5.2 Background: Design patterns ........................................... 24
   5.3 Seams ........................................... 25
   5.4 Quickly adding testable changes ........................................... 29
   5.5 Getting classes into a test harness ........................................... 36
   5.6 Getting methods into a test harness ........................................... 44
   5.7 Characterization tests and classes ........................................... 56
   5.8 Dealing with libraries ........................................... 63
   5.9 Dealing with mysterious code ........................................... 64
   5.10 Dealing with non-object oriented code ........................................... 65
   5.11 Dealing with giant classes and methods ........................................... 67
6 The test plan ........................................... 75
7 Test strategies .......................................... 78
   7.1 Static testing ........................................... 78
   7.2 Other approaches to functional testing ........................................... 79
   7.3 Structural testing ........................................... 79
8 Testing interactive systems ............................. 81
   8.1 Web site and services testing ........................................... 81
   8.2 Testing GUIs ........................................... 84
9 Testing real-time systems ................................ 86
## Tentative semester schedule

See the course homepage for updates and full assignment details. This schedule is tentative, and may be revised as the semester unfolds. In particular, major course events may shift by a class period or two within the first two weeks of class as excused absence schedules arrive. Quizzes, and assignments due later in the semester, may not appear in this schedule. See the course homepage or the online version of this document for updates.

### Friday, February 1 — Lecture 1
- Lectures meet in Centennial 2310
- Introduction, bugs and tests ($\S1, 2$)
- Deadlines: Open-door office hours poll due — Read Everett & McLeod Ch. 1-4 — Read Feathers Ch. 1-2 — Notification of travel dates due

### Monday, February 4 — Lecture 2
- JUnit ($\S4.1$)

### Friday, February 8 — Lecture 4
- Can there ever be enough tests? ($\S4.2$)
- Deadlines: Read Feathers Sec. 8.1 (p. 88-94)

### Monday, February 11 — Lecture 5
- Test-driven development ($\S3, 4.3$)
- Deadlines: Tool demos Round 1 preferences due

### Wednesday, February 13 — Lecture 6
- Object-oriented oddities ($\S4.3$)
- Deadlines: Assignment 1 due

### Friday, February 15 — Lecture 7
- Object-oriented oddities (wrapup), testing in an imperfect world, design patterns ($\S5.1, 5.2$)

### Monday, February 18 — Lecture 8
- Fakes and mocks, preprocessor and link seams ($\S5.1, 5.3$)
- Deadlines: Read Feathers Ch. 4 — Tool demos Round 1 machine needs/room preferences due

### Wednesday, February 20 — Lecture 9
- Object seams, Sprout Method, Sprout Class ($\S5.3, 5.4.1$)
- Deadlines: Pick a conference time in the D2L Assignments 1/2 forum by 8:00am today — Read Feathers Ch. 5-6 (through p. 67) — Assignment 2 presentations begin

### Friday, February 22 — Lecture 10
- Wrap Method, Wrap Class ($\S5.4.2$)
- Deadlines: Read rest of Feathers Ch. 6 — Assignment 2 due

### Monday, February 25 — Lecture 11
- Harnessing classes despite their parameters ($\S5.5.1$)
- Deadlines: Read Feathers Ch. 9 through p. 113

### Wednesday, February 27 — Lecture 12
- Hidden dependencies, further constructor instantiations, global dependencies ($\S5.5.2, 5.5.3, 5.5.4$)
- Deadlines: Read Everett & McLeod Sec. 7.5-7.6 — Read rest of Feathers Ch. 9

### Friday, March 1 — Lecture 13
- Harnessing methods ($\S5.6.1, 5.6.2, 5.6.3$)
- Deadlines: Read Feathers Ch. 10 — Assignment 3 due

### Monday, March 4 — Lecture 14
- Breaking out methods, characterization tests, characterization classes ($\S5.6.4$)
- Deadlines: Read Feathers Ch. 13

### Wednesday, March 6 — Lecture 15
- Dealing with libraries, inscrutable code, non-object-oriented languages ($\S5.8, 5.9, 5.10$)
- Deadlines: Read Feathers Ch. 14-19 (they are all short)

### Friday, March 8 — Lecture 16
- The huge class, the huge method ($\S5.11$)
- Deadlines: Read Feathers Ch. 20-22

### Monday, March 11 — Tool demos
- Code coverage tools (JaCoCo [Smidl], PITest [Johnson], DBUnit [Soich])
- Deadlines: Assignment 4 due

### Wednesday, March 13 — Tool demos, review
- Continuous integration (GitLab CI [Harrington], Jenkins [LaFleur]), Mockito for mocking [Patchen]

### Friday, March 15 — Exam 1

### Monday, March 18 — Spring break starts

### Monday, March 25 — Group speed dating

### Wednesday, March 27 — Lecture 17
- The test plan ($\S6$)
- Deadlines: Read Everett & McLeod Ch. 5

### Friday, March 29 — Lecture 18
- Documentation standards ($\S6$)
- Deadlines: Assignment 5 due

### Monday, April 1 — Lecture 19
- Static testing ($\S7.1$)
- Deadlines: Read Everett & McLeod Ch. 6

### Wednesday, April 3 — Tool demos

### Friday, April 5 — Project selection presentations

### Monday, April 8 — Lecture 20
- Other approaches to functional testing
- Deadlines: Read Everett & McLeod Ch. 7

### Wednesday, April 10 — Lecture 21
- Structural testing ($\S7.3$)
- Deadlines: Read Everett & McLeod Ch. 8

### Friday, April 12 — Lecture 22
- Web site and service testing ($\S8.1$)

### Monday, April 15 — Lecture 23
- GUI testing, tool demos ($\S8.2$)

### Wednesday, April 17 — Lecture 24
- Real-time systems ($\S9$)

### Friday, April 19 — Lecture 25
- Performance testing

### Monday, April 22 — Project check-in

### Wednesday, April 24 — Lecture 26
- Automated testing

### Friday, April 26 — Tool demos

### Monday, April 29 — TBD

### Wednesday, May 1 — TBD

### Friday, May 3 — Project work day (tentative)

### Monday, May 6 — Project work day (tentative)

### Wednesday, May 8 — Final lecture

### Friday, May 10 — Exam 2

### Tuesday, May 24 — Group project demos, 7:45am
1 Introduction

Some famous software failures

• NASA’s Mars Climate Orbiter
  – September 1999, crashed due to a units integration fault
  – “The MCO MIB has determined that the root cause for the loss of the MCO spacecraft was the failure to use metric units in the coding of a ground software file, 'Small Forces,' used in trajectory models. Specifically, thruster performance data in English units instead of metric units was used in the software application code titled SM_FORCES (small forces).”

  – Poor testing of safety-critical software can cost lives
  – 6 accidents, 3 patients killed due to radiation overdoses
  – “The second Yakima accident was again attributed to a type of race condition in the software - this one allowed the device to be activated in an error setting (a 'failure' of a software interlock) […]
    A lesson to be learned from the Therac-25 story is that focusing on particular software bugs is not the way to make a safe system. Virtually all complex software can be made to behave in an unexpected fashion under certain conditions. […]
    The software should be subjected to extensive testing and formal analysis at the module and software level; system testing alone is not adequate.”

• Ariane 5 Explosion

• Intel’s Pentium FDIV Fault

• Blackout of the Northeast in 2003

• …and on and on!

The cost of software failures


• “[T]he national annual costs of an inadequate infrastructure for software testing is estimated to range from $22.2 to $55.5 billion”

• Better approaches to software testing could cut this amount in half!


Ten principles of testing

1. Business risk can be reduced by finding defects.
2. Positive and negative testing contribute to risk reduction.
3. Static and execution testing contribute to risk reduction.
4. Automated test tools can contribute to risk reduction.
5. Make the highest risks the first testing priority.
6. Make the most frequent business activities (the Pareto Principle) the second testing priority.
7. Statistical analyses of defect arrival patterns and other defect characteristics are a very effective way to forecast testing completion.

8. Test the system the way customers will use it.

9. Assume the defects are the result of process and not personality.

10. Testing for defects is an investment as well as a cost.

Some terminology

- Verification and validation
  - Verification: are we building the thing right?
  - Validation: are we building the right thing?

- White-box and black-box
  - White-box testing (structural testing) allows us access to the source under test
  - Black-box testing (functional testing) hides the source under test

Testing activities

1. Identify an objective to be tested
2. Select inputs
3. Compute the expected outcome
4. Set up the execution environment
5. Execute the program
6. Analyze the results

Some specific benefits of software testing

- Evaluate software under development
  - (Does it meet the specification? How it is progressing)
- Make judgments about the quality or acceptability of a software product
- Discover problems (uncover faults and errors)
- Make sure we did not introduce any new faults
And even earlier testing

<table>
<thead>
<tr>
<th>Source</th>
<th>Requirements</th>
<th>Design</th>
<th>Code</th>
<th>Test</th>
</tr>
</thead>
<tbody>
<tr>
<td>[Boehm, 1981]</td>
<td>1</td>
<td>5</td>
<td>10</td>
<td>50</td>
</tr>
<tr>
<td>[Hoffman, 2001]</td>
<td>1</td>
<td>3</td>
<td>5</td>
<td>37</td>
</tr>
<tr>
<td>[Cigala, 2003]</td>
<td>1</td>
<td>3</td>
<td>7</td>
<td>51</td>
</tr>
<tr>
<td>[Rothman, 2000]</td>
<td>5</td>
<td>33</td>
<td>75</td>
<td></td>
</tr>
<tr>
<td>[Rothman, 2000] Case B</td>
<td>10</td>
<td>40</td>
<td></td>
<td></td>
</tr>
<tr>
<td>[Rothman, 2000] Case C</td>
<td>10</td>
<td>40</td>
<td></td>
<td></td>
</tr>
<tr>
<td>[Rothman, 2002]</td>
<td>1</td>
<td>20</td>
<td>45</td>
<td>250</td>
</tr>
<tr>
<td>[Pavlina, 2003]</td>
<td>1</td>
<td>10</td>
<td>100</td>
<td>1000</td>
</tr>
<tr>
<td>[McGibbon, 2003]</td>
<td>5</td>
<td>50</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

| Mean | 1 | 7.3 | 25.6 | 177 |
| Median | 1 | 5 | 10 | 50.5 |

Table 1: Normalized Cost-to-Fix Estimates

Exercises and study questions

1. What are the ten principles of testing which Everett and McLeod identify? Give examples of how each one applies both in software-related and non-software-related contexts.

2. What is the difference between verification and validation?

3. What is the general workflow for testing activities? Emphasize the necessity of each step of the workflow by discussing the consequences of omitting it.

4. What are the benefits of software testing? How does each correspond to a risk incurred by not testing?

5. Describe how the cost of repairing defects in software systems has been found to change across the software lifecycle.

What we'll look at in this class

First, we'll start with one particular tool, JUnit

- For testing in Java, which you already know
- Before Spring Break, you'll explore similar and related tools

Then: how do you start testing in the real world?

- Most of the time, you'll be joining projects, not starting them
- How do you get the benefits of testing in this situation?

After Spring Break, we'll widen our inquiry

- To the role of testing in the broader software engineering process
- To the details of testing in various other scenarios

What you'll do in this class

- In the first half, several exercises with JUnit
- Demonstrations of various testing tools
About a week before Spring Break
Throughout the second half of the semester

• A large project
  – Adopt an open-source project
  – Bring testing to it

• There will also be two mid-terms
  – Each covering half of the semester
  – At the end of each half

• All throughout, reading
  – Two textbooks, available at the rental office
  – Read the lecture’s chapter before each lecture

2 Bugs and tests

An early taxonomy
From Beizer (Software Testing Techniques, 1983):

1. Requirements
   • Incorrect requirements, incomplete requirements

2. Features and Functionality
   • Mishandled exceptions, feature/function correctness

3. Structural Bugs
   • Control flow and sequencing

4. Data
   • Data access

5. Implementation and Coding
   • Style violations, typographical error

6. Integration
   • External interfaces and timing

7. System and Software Architecture
   • Poor performance, bad software architecture

8. Test Definition and Execution
   • Poor documentation, incomplete tests

9. Unclassified defects
The IEEE spec

- Nomenclature (p.5): Problem vs. failure vs. defect vs. fault
  - Hierarchy (p.3-4)
  - Kinds of reports
- Examples (p.11)
- Scope (p.1) — this is not everything!

The checklist mentality

- Inspired by the rigorous checklist protocols of flight crews
  - An expert (pilot) verifies that a series of conditions are present
- SPRAE is a "checklist" for software testing
  - Specification
  - Premeditation
  - Repeatability
  - Accountability
  - Economy
- From *A Perspective on Teaching Software Testing*, Edward A. Jones, 2000

Specification

A "formal" document (requirements specification) on what the program should do

- It is the basis of a serious testing effort
  - "No specification, no test"
  - Without a specification, how do we know what the right result of a test is?
- The specification is the answer to the question "Why are you testing that?"

What are some different "types" of requirements specifications?

Premeditation

Pre-planning in detail — test plans, environments, data, scripts, schedules

- All documented!
- There is art in deciding what to document
  - One the one side: the stereotype of software management as an excessive generator of useless documents — and consuming too many resources from actual developing and coding
  - On the other side: projects fails to a risk against which we did not adequately test
- Another very important intention to be recorded: What is success?
  - How do we know when the tests are passed, and we can stop?
Repeatability
The testing process and its results must be repeatable and independent of the tester

- Anyone on the test team must be able to run the test suite
  - And maybe people not on the test team too
  - Includes the test build and test script

What are test scripts/builds for simple "batch" programs? How about programs that interface with the user?

Accountability
The test process must produce an audit trail.

- The test results should be saved in the test execution log, while errors should be in the discrepancy log
- Answers the questions “What was tested?”, “What were the results?”, and “What were the errors?”

What are some benefits of this documentation?

Economy
Testing should not require excessive human or computer resources.

- "How much testing is enough?" is often "How much time/money do we have left?"

What things can help our test economy?

Exercises and study questions
(2.1) List and describe the none classifications of Beizer’s taxonomy of software defects.
(2.2) What does the acronym SPRAE stand for?
(2.3) What is the role of the specifications document in testing? How do we regard that document under SPRAE?
(2.4) What does "premeditation" refer to under SPRAE? How is it recorded?
(2.5) Describe what in SPRAE tells us about when we can stop testing.
(2.6) What does "repeatability" or "reproducibility" of a test suite entail?
(2.7) What artifacts arise from the "accountability" aspect of SPRAE? What specific questions must our process of accountability address? What benefits do the accountability artifacts bring?
(2.8) What does "accountability" refer to in the SPRAE regime?

Four testing strategies
1. Static testing
   - The book talks about static testing in the context of documents
   - (This is the only sort of testing that applies to non-code artifacts)
   - We’ll also look at the static analysis of the code itself

2. Black-box testing
   - Tests which run code, but with reference only to its APIs

3. White-box testing
   - Letting the structure of our code inform our test design

4. Performance testing
   - Real time, CPU time, memory, other machine resources
**Functional testing**
Another name for black-box testing, sometimes used for white-box testing

- Test the specified properties of the software against the implementation
- Often driven by *use cases*
  - Good source for functional test cases
  - Use cases are a less good for performance tests
  - Sometimes good for white-box tests

**Functional tests are often interactive**

- User navigation
  - Tester follows a script of (common? expected common?) user actions
  - Generally manual
  - Incorporates the user/operator manual
  - Complicated with stateless aspects of the system
- Transaction screen testing
  - Focus on the aspects of the user experience which cause a measurable change in some external storage
- Transaction flow testing
  - Sequences of changes, and the ability to transition among them
  - Usually combined with transaction screen testing
- Report screen and report flow testing
  - Again related to transaction screen testing
  - Focus on the correct reporting of information by the program
  - Consistency of information sent to multiple places
- Database-oriented testing

**Regression testing**

- Functional tests are often the basis of regression testing
- Structured to re-run frequently
  - Check that new code does not break old code
- Deprecation and code removal

---

**Exercises and study questions**

(2.9) What are the four strategies into which we categorize tests?
(2.10) What is the relationship between use cases and testing? What testing strategies do they enable?
(2.11) List the different kinds of interactive functional tests, and describe each.
(2.12) Describe the process of interactive user navigation testing.
(2.13) Describe transaction screen testing, and the related categories of functional tests associated with them.
(2.14) Define regression testing, and describe how it can be implemented and enforced.
3 Test-driven design

Test-driven development
TDD is a simple idea for structuring your activity when programming

When coding, repeat the following steps:

1. Write a failing test case
2. Get it to compile
3. Get it to pass
4. Remove duplication

TDD Example

- Working on a financial application
- Need a class to make decisions as to whether certain commodities should be traded
  - A very mathematical question, so we’ll need to support various operations
- Specifically, we need to calculate the first statistical moment about a particular point
  - Don’t have a method for it yet
  - But our stats experts gave us a simple example for this test:

```java
@Test public void testFirstMoment() {
    final InstrumentCalculator calc = new InstrumentCalculator();
    calc.addElement(1.0);
    calc.addElement(2.0);
    assertEquals("First moment about 2.0 for {1.0,2.0} within tolerance",
                 -0.5, calc.firstMomentAbout(2.0), TOLERANCE);
}
```

Making testFirstMoment compile

- This won’t even compile right now
  - We might already have a class InstrumentCalculator, but we haven’t written firstMomentAbout yet!
  - So we add a stub for the method
- We don’t want the new test to pass just by coincidence, so we make it return an absurd value

```java
public class InstrumentCalculator {
    // ... keeping what’s already here

    public double firstMomentAbout(double point) {
        return Double.NaN;
    }
}
```
Making testFirstMoment pass

- The algorithm for calculating the first moment is standard, so we look it up and implement it

```java
public double firstMomentAbout(double point) {
    double numerator = 0.0;
    for(final double element : getElements()) {
        numerator += element - point;
    }
    return numerator / elements.size();
}
```

- This change is actually larger than a typical response to a test under TDD
  - That’s OK when we’re introducing a standard algorithm
  - We’re not developing the algorithm itself — we already know the right algorithm
  - We’re just implementing a given standard algorithm, so the code we introduce to make the test pass can be bigger

Remove duplication

- At this point, there isn’t duplication
- So nothing to do for this step!

Write another failing test case

- The code we just added makes one test pass, but it’s not hard to conceive of cases which will fail
- There’s a division in the algorithm: are we safe against division by zero?
  - And what should happen when we call firstMomentAbout() with an empty data set?
- Write another test for this case!

```java
@Test(expected = InvalidBasisException.class)
public void testEmptyFirstMoment() {
    new InstrumentCalculator().firstMomentAbout(0.0);
    fail("Expected InvalidBasisException");
}
```

Making testEmptyFirstMoment compile

What do we need to do to make testEmptyFirstMoment compile?

- If InvalidBasisException is not already part of InstrumentCalculator’s package, we must create it
- Otherwise it compiles
Making `testEmptyFirstMoment` pass

- We’ll need to throw an `InvalidBasisException` when there are zero elements
- So a revised `firstMomentAbout`:

```java
public double firstMomentAbout(double point) {
    if (getElements().isEmpty()) {
        throw new InvalidBasisException();
    }

double numerator = 0.0;
for(final double element : getElements()) {
    numerator += element - point;
}
return numerator / elements.size();
}
```

- Now the test passes!

Remove duplication

- Still no duplication from our changes. Onward!

Write a failing test case

- Our next task is to write a routine for the second statistical moment about a point.
- So we write a test for this case. Our stats experts again gave us a simple example:

```java
@Test public void testSecondMoment() {
    final InstrumentCalculator calc = new InstrumentCalculator();
calc.addElement(1.0);
calc.addElement(2.0);
assertEquals("Second moment about 2.0 for {1.0,2.0} within tolerance",
    0.5, calc.secondMomentAbout(2.0), TOLERANCE);
}
```

Making `testSecondMoment` compile

- The problem is again that we do not define the method we are now testing
- And again we make it compile by adding a vacuous definition of the method. We’ll just copy `firstMomentAbout` and change the name:

```java
public double secondMomentAbout(double point) {
    if (getElements().isEmpty()) {
        throw new InvalidBasisException();
    }

double numerator = 0.0;
for(final double element : getElements()) {
    numerator += element - point;
}
return numerator / elements.size();
}
```
Making testSecondMoment pass

- Unsurprisingly, the code for the first moment does not satisfy the second moment’s test!
- But the algorithm for the second moment is very similar, and we only need to make one change: from

  ```java
  numerator += element - point;
  ```

  to

  ```java
  numerator += Math.pow(element - point, 2.0);
  ```

- And now it passes!

Remove duplication

- This time around there’s definitely duplication — we have two methods that are almost completely identical!
- In fact, the algorithm for any of the statistical moments has only the same variation that we see here
- So the best way to remove this duplication is with a more general private method `nthMomentAbout` which the others call

  ```java
  public double nthMomentAbout(double point, double n) {
      if (getElements().isEmpty()) {
          throw new InvalidBasisException();
      }
      double numerator = 0.0;
      for(final double element : getElements()) {
          numerator += Math.pow(element - point, n);
      }
      return numerator / elements.size();
  }
  ```

  ```java
  public double firstMomentAbout(double point) {
      return nthMomentAbout(point, 1.0);
  }
  ```

  ```java
  public double secondMomentAbout(double point) {
      return nthMomentAbout(point, 2.0);
  }
  ```

- We already have tests in place, so we can be confident in this change

It’s OK to duplicate!

- This example seems draconian
  - Adding methods that we know are wrong!
  - Copying a method outright!
- But the point of TDD is that we are freed from worrying about more than one thing at a time
  - We might be setting up a test
  - Or we might be writing code for a new feature, but *never at the same time as setting up a test*
– Or we might be refactoring away some horrible duplication, but never at the same time as setting up a test or writing new code
– Do one thing at a time, and do it right

• With legacy code — code we didn’t write and may not completely understand — being able to copy code wholesale is valuable
  – See the two methods side-by-side
  – The original is unchanged while we develop the new one
• And the tests we build up make the later refactoring to remove duplication much less risky

Exercises and study questions

(3.1) What are the steps of test-driven design?
(3.2) Why (and when) are new tests under TDD expected to fail, and possibly not even to compile?
(3.3) Why is code duplication accepted under TDD? How does TDD support the removal of this duplication?
(3.4) Under TDD, when do we remove tests?
(3.5) What is the benefit of code duplication under TDD when addressing legacy code?

4 Diving into writing tests

4.1 Using JUnit

JUnit

• For specifying and running functional tests in Java
• A separate test for every method
  – Use Java annotations to mark the test
• The JUnit executable finds test methods, runs them, reports the results

JUnit example

From the JUnit wiki:

• A simple program

```java
public class Calculator {
    public int evaluate(final String expression) {
        int sum = 0;
        for (String summand: expression.split("\+"))
            sum += Integer.valueOf(summand);
        return sum;
    }
}
```

JUnit example

• Test Calculator with

```java
import static org.junit.Assert.assertEquals;
import org.junit.Test;
public class CalculatorTest {
    @Test public void evaluatesExpression() {
        final Calculator calculator = new Calculator();
```
final int sum = calculator.evaluate("1+2+3");
assertEquals(6, sum);
}
}

- The @Test annotation: how JUnit finds tests
- Method takes no parameters
- The class holding the tests has a zero-argument constructor
- Assertions

JUnit assertions
- There are special methods for asserting things which should be true
- Javadoc API org.junit.Assert
  - Parameter order: expected value, then actual value
  - Pointer equality vs. equals
  - Always give a message!
- Matchers: org.hamcrest.CoreMatchers
  - Allows conditions to be stated separated from the structure of the unit test
    - assertThat("good", allOf(equalTo("good"),
        startsWith("good")))
    - assertThat("good", not(allOf(equalTo("bad"),
        equalTo("good"))))
    - assertThat("good", anyOf(equalTo("bad"),
        equalTo("good")))
    - assertThat(new Object(), not(sameInstance(new Object())));

Exceptions
- Can also specify cases which we expect to fail

@Test(expected = IndexOutOfBoundsException.class)
public void empty() {
    new ArrayList<Object>().get(0);
}

- Other tools for finer-grained detection of particular statements in a method

Groups of tests
- Suites
  - Simple hierarchy
  - Define a suite by including certain classes in it
  - Pick the suite to run, instead of all tests
- Categories
  - Like a suite, but pick only a subset of each class’s tests
  - For example: tag fast tests and slow test
Other JUnit features include

- Specify order of tests
- Specify parameters for multiple runs of the same test
- Impose timeouts
- Plugins for continuous testing in IDEs

4.2 Black-box (functional) and white-box (structural) testing

Black box vs. white box testing
Under black-box testing, we do not refer to the production code when writing tests

- Perhaps we do not have access to it
- Perhaps it has not yet been written

When we write tests with the source code in hand, we call it white-box testing

Techniques for designing functional tests

- Equivalence classes
  - Inputs that should all be treated the same way
  - All categories and relationships the same for data in the equivalence class
  - Test only one representative of the class

- Boundary values
  - Identify points in a range of data where behavior changes
  - Then, identify an epsilon — the minimum distance from the border where the behavior would be expected to settle into an equivalence class
  - An equivalence class for a range could then be the epsilon at each end, and a point in the middle

- Expected results coverage
  - Often there will be discrete classes or levels of output
  - Design equivalence classes to produce each possible output value
  - Or combine equivalence classes in the possible ways to generate each output

- Intuition for things that tend to be tricky
  - Error handling
  - Data conversions
  - Dates and their manipulation
**White box testing**

When we also look at the source code, we have new strategies for designing tests in terms of the *coverage* of the source code

- **Statement coverage**
  - What percentage of the statements in a program are executed during testing?
  - High coverage in a complex system is very difficult to achieve

- **Branch coverage**
  - Make sure both branches of each true/false test has been tested

- **Compound condition coverage**
  - Extends branch coverage to individual components of a boolean formula
  - Test all of the combinations of booleans which will trigger one or the other branch

- **Loop coverage**
  - Find each possible loop through the code
  - Explicit via *for*, *while*, etc.
  - Other jumping mechanisms
  - Make sure each loop body executed 0 times, 1 times, various boundary times

- **Path coverage**
  - All possible paths through a program
  - (For some finite sense of *all*)

- **Other techniques for white-box testing**
  - Dates (again!)
  - Empties
  - Buffer overflow

---

**Exercises and study questions**

(4.1) Describe the difference between white-box and black-box testing.
(4.2) What circumstances necessitate black-box testing?
(4.3) List and describe three general approaches to black-box testing.
(4.4) What are equivalence classes, and what is their application to testing?
(4.5) What are boundary values of equivalence classes? How do they guide test design?
(4.6) How can equivalence classes and expected results coverage interact when designing tests?
(4.7) What are some of the flaws arising from data conversions which testing can address? From dates and their representations?
(4.8) List and define five forms of test coverage. What levels of each notion of coverage are possible in a nontrivially-sized software project? What is feasible?
(4.9) What usages in source code should attract particular attention under white-box testing?

---

**4.3 Some particulars of object-orientation**

**Object-oriented software**

- The testing ideas we’ve seeing
  - Evolved over time
To address projects in all languages

- But some common language paradigms and use cases have unique challenges and distinct error patterns
- Object-oriented languages are one such case
  - Relationships in object-oriented programs are complex
    - Classes, inheritance, polymorphism, dynamic binding can obscure what code actually runs
  - The subtype relationship: if \( A <: B \) then an instance of \( A \) should be freely substitutable where an instance of \( B \) is required
    - But compiler cannot normally check whether \( A \)’s methods adhere to the same specification as \( B \)’s methods
    - Only whether the types are compatible

Classes as the basis of testing

We’ve seen how JUnit take classes as the basic unit of testing
This gives us four levels of testing

- **Intra-method** — covering one method
  - Traditional unit testing
- **Inter-method** — multiple methods of one class tested in concert
  - Traditional module testing
- **Intra-class** — construct tests for a single class
  - Usually sequences of method calls within the class
- **Inter-class** — more than one class tested at a time
  - Focus on their interactions
  - A form of integration testing

Faults specific to object-oriented programming

Assume that \( V \) extends \( W \), and that both classes have a method \( m \)

```java
void f(boolean b) {
    final W o;
    if (b) {
        o = new V();
    } else {
        o = new W();
    }
    o.m();
}
```

- Which code is run?
A more complicated example

- (All methods public)

- What happens for `new A().d()`?
  \[
  \begin{align*}
  A:: & \quad d() \rightarrow g() \rightarrow h() \rightarrow i() \rightarrow j() \rightarrow l() \\
  B:: & \quad h() \rightarrow i() \rightarrow k() \\
  C:: & \quad i() \rightarrow j() \rightarrow l()
  \end{align*}
  \]

- What happens for `new B().d()`?

- What happens for `new C().d()`?

- This is called a *yo-yo diagram*

- Even more complicated: which stateful fields are consulted along each path?

Categories of flaws typical to OOP

- Describing typical sorts of flaws helps us identify good test cases

- Eight kinds of faults and anomalies arising from inheritance and method polymorphism
  - ITU — inconsistent type use (context swapping)
  - SDA — state definition anomaly
  - SDIH — state definition inconsistency arising from field hiding
  - IISD — indirect inconsistent state definition
  - ACBP — anomolous construction behavior from polymorphic method calls
  - ACBI — anomolous construction behavior from unexpected initialization order
  - IC — incomplete construction
  - SVA — state visibility anomaly

- These errors are not unique to object-oriented software!
  - Their manifestation will be different in other language styles
  - The vocabulary for discussing them will likely vary
  - They may be more insidious in the object-oriented style
ITU: inconsistent type use (context swapping)

- Arises from casting a subclass instance back-and-forth from a superclass
  - When the specification of the classes differ, their operations may be incompatible
  - Consider a Stack class implemented by extending the general mutable ArrayList class
  - The general add and remove methods break the abstraction of the stack’s push and pop methods
  - Programs that rely on a balanced numbers of calls to push and pop may fail unexpectedly
- Not about overriding methods
  - Superclass methods not hidden, even for a subclass with an idea to use them internally only

SDA: state definition anomaly

- When the state interactions of overridden methods vary too greatly from the original methods
- “Safe” sequences of method calls on the superclass may cause errors in the subclass
- Violates the notion of what a subtype should be

SDIH: state definition inconsistency arising from field hiding

- Overriding local fields may disrupt superclass methods
- Information in the hidden fields becomes inaccessible to superclasses

IISD: indirect inconsistent state definition

- Think about a superclass $D$ with stateful field $x$ and method $m$
- Then its subclass $E$ introduces a stateful method $e$
- None of $D$’s methods will call $e$, of course
- But if $m$ is also overridden, it might call $e$
- $e$ might make changes to the state which invalidate a series of method calls involving $m$

ACBP: anomalous construction behavior from polymorphic method calls

Remember that in many object-oriented languages, superclass constructors may be called by default

- ACBP involves a constructor which calls a method of the object under construction
- A subclass may override this method
- Under dynamic dispatch, the subclass’s overriding method would be called from the superclass constructor
- Even when the overridden method is stateless, its result builds the state of the instance
- So the overridden method may introduce a faulty initial state of new instances

ACBI: anomalous construction behavior from unexpected initialization order

ACBI again involves a constructor which calls a method $m$ of the object under construction

- Then a subclass may override $m$ with a method that refers to stateful fields set up by the superclass constructor
- The subclass method could use uninitialized values to build the state of new instances
IC: incomplete construction

• In C++ (though not Java), object fields values are *undefined* before a constructor sets them — they may have literally any value
  – Not an issue in Java: its early focus on applet security lead to default initial values for all types
• But even in Java, neglecting field initializations can lead to unexpected situations

SVA: state visibility anomaly

• Certain stateful fields in a superclass are *private*, managed by a non-private method \( m \)
• Methods in classes overriding \( m \) must call \code{super.m} or those fields cannot be correctly managed
  – Subclasses of those methods lose the ability to access \code{super.m}

5 Adapting existing code for testing

5.1 Introduction

Why do we change software?

• Add features
• Fix defects
• Refactoring — improve software structure
• Optimize — improve software performance
• Also: four ways software can change
  – In structure
  – New functionality
  – Alter functionality
  – Resource usage
• And we can make a chart . . .

Connecting reasons and ways

Typically:

<table>
<thead>
<tr>
<th></th>
<th>Add feature</th>
<th>Fix defect</th>
<th>Refactor</th>
<th>Optimize</th>
</tr>
</thead>
<tbody>
<tr>
<td>Structure</td>
<td>Changes</td>
<td>Changes</td>
<td>Changes</td>
<td>—</td>
</tr>
<tr>
<td>New functionality</td>
<td>Changes</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>Alter functionality</td>
<td>—</td>
<td>Changes</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>Resource usage</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>Changes</td>
</tr>
</tbody>
</table>
TDD with legacy code

• When we are working with legacy code, there is an important caveat
• We must be testing methods on these classes all of the time
• So for legacy code, TDD is really a five-step loop
  1. Get the class you want to change under test
  2. Write a failing test case
  3. Get it to compile
  4. Get it to pass, trying not to change existing code
  5. Remove duplication
• Getting the class under test does not necessarily mean writing tests for all of the legacy methods
• But in general it does mean instantiating the class in a test harness
  – This can be hard
  – There’s a related problem as well — just because we can instantiate the class in the test harness does not mean that run any particular method in the the test harness
  – We’ll look at some common problems and solutions next

Identifying an obstacle

• Code dependencies keep us from testing and easily changing code
  – Can’t create just a few classes for tests, must create many
  – Can’t isolate effects of changes
  – Can’t measure effects and outputs
• Two distinct reasons to break up dependencies:
  – Because we need to sense
    * But we can’t access values because they are too tightly tied to consuming modules
  – Because we need to separate
    * But it’s hard to structure the code so that a reasonable test compiles and runs

Example: a network bridge controller

```java
public class NetworkBridge {
    public NetworkBridge(EndPoint[] endpoints) {
        // ...
    }
    public void makeRouting(String sourceID, String destID) {
        // ...
    }
    // ...

    // Functionality includes:
    – Accepts an array of network locations (EndPoints)
    – Manages their configuration via local hardware
    – Translates endpoint-to-endpoint routes from method calls to the hardware
```
* Uses methods on `EndPoints` which communicate via socket to remote devices

- How do we feasibly test this class?
  - Will we need some spare hardware?
  - Will we need to wire up a testbed network?
  - Will we need to break in to the network wires to test the signals which are actually transmitted?
  - Or at each remote device?

Fake it!

- Replace objects of actual program classes with *fake* objects

  - Example, a point-of-sale scanner

    ```
    PointOfSale
    scan(barcode : String)
    ```

    When you scan something, it displays the item on a register

- The legacy system hard-codes output to a particular hardware display (the Model R56)
  - Want to test the display output, without actually using the display hardware

Make space for the fakes

- First redesign to separate the display class

  ```
  PointOfSale
  scan(barcode : String)
  ```

  ```
  ModelR56Display
  showLine(line : String)
  ```

- Then use interfaces to make the display interchangeable

  ```
  PointOfSale
  scan(barcode : String)
  ```

  ```
  Display
  ≪interface≫
  showLine(line : String)
  ```

  ```
  ModelR56Display
  ```

  ```
  FakeDisplay
  ```

Fake objects support real tests!

- "But that’s not real testing!"

- Yes, it is
  - Tests on `PointOfSale` can examine *one* class
  - Other tests can verify the "real" `ModelR56Display`
  - Test the integration in a separate test
Mock objects

- *Mock* objects carry the idea of fake objects one step further
- Mock objects know the expectations of a particular test
- Set up tests by giving mock objects a checklist
- Departure from the checklist fails the test

```java
public @Test void displayAnItem() {
    final MockDisplay display = new MockDisplay();
    display.setExpectation("showLine", "Milk $3.99");
    final PointOfSale sale = new Sale(display);
    sale.scan("11203402");
    display.verify();
}
```

5.2 Background: Design patterns

Design Patterns

- Common techniques in programming
- Larger than a few statements
  - Ways to set up and use certain phenomena
  - A vocabulary for discussing programs structures
- Some patterns now considered "antipatterns"
- A number of patterns will come up as we discuss testing and modifying code
  - (And yes, I do realize that you do not learn an inventory of design patterns in any prerequisite course — we will go over patterns as we need them.)

**Factory Method**

The Factory abstract object creation

- Strictly speaking the only language mechanism for creating an object is constructor invocation
- But constructors are inflexible — cannot be overridden, must be specifically named
- With a Factory, we abstract out object creation behavior
  - A Factory is an instance satisfying an interface like
    ```java
    public interface Factory<A> {
        public A build();
    }
    ```
  - Creating objects via a factory allows us to vary the actual type of created objects
  - Variations give the `build` method arguments
Singleton
Sometimes there should only be one instance of a class
In the Singleton pattern:

- The constructor is private
- The sole instance of the class is bound to a static field

```java
public class Unique {
    public static final INSTANCE = new Unique();
    private Unique() {
    }
    // Methods follow here
}
```

5.3 Seams
Object seams

- Dynamic dispatch
  - What does $f$ print?
    ```java
class A {
    method f() { return 1; }
}
class B extends A {
    method f() { return 2; }
}

class M {
    method n() {
        A a = new B();
        a.f()
    }
}
```
  - Working with Java gives us a certain expectation, but in fact this is a property of the language
    * Under static dispatch, the declared type $A$ determines the dispatched method
    * Under dynamic dispatch, the instantiates type $B$ determines the dispatched method

Object seams

- Interfaces

  - What method is called by `cell.recalculate()`?
  - If we can change which `recalculate` is called in that line of code without changing the code around it, then that call is a seam
This method call is a seam

```java
public Spreadsheet buildMySheet(Cell cell) {
    // ...
    cell.recalculate();
    // ...
}
```

- Where is the enabling point?
  - In the argument list
  - In a test, we can choose which subclass of `Cell` to instantiate and pass

But not all method calls are seams!

```java
public Spreadsheet buildMySheet() {
    // ...
    final Cell cell = new FormulaCell(this, "A1", ",=A2+A3")
    // ...
    cell.recalculate();
    // ...
}
```

- There is no enabling point

Are static methods seams?

```java
public Spreadsheet buildMySheet(Cell cell) {
    // ...
    recalculate(cell);
    // ...
}
private static void recalculate(Cell cell) {
    // ...
}
```

- If it’s `private` then we can make it into a seam safely:

```java
public Spreadsheet buildMySheet(Cell cell) {
    // ...
    recalculate(cell);
    // ...
}
protected void recalculate(Cell cell) {
    // ...
}
```

- Extend class, override `recalculate` to change behavior

Seam roundup

- Link seams
  - Applies to global functions, entire class definitions, etc.
  - A little heavyweight, since we’re setting up an entire alternate build environment
• Preprocessor seams
  – Applies to languages with `#define` and `#include` (or similar)
  – Less heavyweight than link seams

• Object seams
  – Largely applies only to object-oriented languages
  – More flexible and appealing than link and preprocessor

Object seams
• Dynamic dispatch
  – What does `f` return?
    ```java
class A { method f() { return 1; } }
class B extends A { method f() { return 2; } }

class M {
  method n() {
    A a = new B();
    a.f()
  }
}
```
  – Working with Java gives us a certain expectation, but in fact this is a property of the language
    * Under *static dispatch*, the declared type `A` determines the dispatched method
    * Under *dynamic dispatch*, the instantiates type `B` determines the dispatched method

Object seams
• Interfaces

```
Cell
\downarrow
reccalculate()

ValueCell
\downarrow
reccalculate()

FormulaCell
\downarrow
reccalculate()
```

– What method is called by `cell.recalculate()`?
– If we can change which `reccalculate` is called in a line of code which calls `reccalculate`, *without* changing the code around it, then that call is a seam

This method call is a seam
```
public Spreadsheet buildMySheet(Cell cell) {
  // ...
  cell.recalculate();
  // ...
}
```

• Where is the enabling point?
  – In the argument list
  – In a test, we can choose which subclass of `Cell` to instantiate and pass
But not all method calls are seams!

```java
public Spreadsheet buildMySheet() {
    // ...
    final Cell cell = new FormulaCell(this, "A1", "=A2+A3")
    // ...
    cell.recalculate();
    // ...
}
```

• There is no enabling point

Are static methods seams?

```java
public Spreadsheet buildMySheet(Cell cell) {
    // ...
    recalculate(cell);
    // ...
}
private static void recalculate(Cell cell) {
    // ...
}
```

• If it’s private then we can make it into a seam safely:

```java
public Spreadsheet buildMySheet(Cell cell) {
    // ...
    recalculate(cell);
    // ...
}
protected void recalculate(Cell cell) {
    // ...
}
```

– Extend class, override `recalculate` to change behavior

Seam roundup

• Link seams
  – Applies to global functions, entire class definitions, etc.
  – A little heavyweight, since we’re setting up an entire alternate build environment

• Preprocessor seams
  – Applies to languages with `#define` and `#include` (or similar)
  – Less heavyweight than link seams

• Object seams
  – Largely applies only to object-oriented languages
  – More flexible and appealing than link and preprocessor
5.4 Quickly adding testable changes

5.4.1 Sprout Method and Sprout Class

Remember why we brought up seams

- Seams are one way to break dependencies
- Breaking dependencies in turn supports making (safe) code changes and adding tests
- Does adding tests actually happen? Do people add tests with code changes?
- What are the actual costs?
- Scenario: write some test for code you are about to change
  - Two hours to write the tests
  - Fifteen minutes to make the change

So was it worth it?

- If it took 15 minutes to make a change, that certainly means we did it right the first time
  - That won’t always be true!
  - We might have misunderstood how the software works
  - We might introduce a bug with the change
  - And anything like those would probably rely on the tests to fix
- Don’t know when we’ll be back in that code again
  - Best case: editing the same code for the next round of work
    In this case, immediately begin to recoup cost
  - Worst case: that code isn’t changed again for years
    - Maybe we won’t change it, but will we need to ever read the code?
    - Will it be easier to understand with smaller classes and with tests?
    - Typically, changes cluster

In general, testing raises productivity

- But when adding testing to an existing project, there’s a mountain to climb
  - Once you cross the mountain, it’s great!
  - But the crossing is arduous
- The process of making code testable does tend to improve code
  - Subsequent changes become easier
  - Team morale can improve by lowering the baseline level of frustration

The economics is more difficult

- Very difficult to estimate how long refactoring will take
- When the time is right, tempting to hack in the code “and test later”
  - Coming back tends to be hard
  - Not motivating work
  - Before climbing the mountain, going back is the sort of work we tend to avoid
Dilemma: pay now or pay more later

• Why "more"?

• Some ways to mitigate when we don’t have lots of time, but need to change code not under a good test regime
  – When changing a class, can we instantiate it in a test harness?
    * Might be easier than it first seems
    * Dependencies might not be as bad as we think?
  – If not: can we implement the needed change with fresh code?

Add a new method

• When we can implement with fresh code, do so in a new method

• We can most easily add tests for new methods

• Testing the method is not the same thing as testing call points!
  – But it’s at least some testing

Example

class TransactionGateway {
    public void postEntries(List<Entry> entries) {
        for(final Entry entry : entries) {
            entry.postDate();
        }
        transactionBundle().add(entries);
    }
}

• Change: we need to make sure that none of the new entries are already in transactionBundle()

Could just do this

class TransactionGateway {
    public void postEntries(List<Entry> entries) {
        List<Entry> entriesToAdd = new LinkedList<Entry>();
        for(final Entry entry : entries) {
            if (!transactionBundle().contains(entry)) {
                entriesToAdd.add(entry);
                entry.postDate();
            }
        }
        transactionBundle().add(entriesToAdd);
    }
}

• But it’s invasive

• No separation between new and old code

• Mixing two operations

• With the new temporary variable, opportunities for this method to become even more muddy
A better way

- Add a new method

```java
List<Entry> uniqueEntries(List<Entry> entries) {
    final List<Entry> result = new ArrayList<Entry>();
    for (final Entry entry : entries) {
        if (!transactionBundle().contains(entry)) {
            result.add(entry);
        }
    }
    return result;
}
```

- Easy to add tests for this behavior

- Then use new method in the old method

```java
public void postEntries(List<Entry> entries) {
    final List<Entry> entriesToAdd = uniqueEntries(entries);
    for (final Entry entry : entriesToAdd) {
        entry.postDate();
    }
    transactionBundle().add(entriesToAdd);
}
```

This is the Sprout Method technique

1. Identify where we need to make a code change
2. Can we formulate the change as a sequence of statements at one place in a method?
   - Write down a call to a new method, and comment it out
   - Yes, before writing the method itself — this gives us the sense of what the call will look like in context
3. Determine the local variables we need from the source method, and make them arguments to the call
4. Will the sprouted method need to return a result?
   - If so, change the commented-out call to save its return value
5. Develop and test the sprout method
6. Remove the comment in the source method to enable the call

Using Sprout Method

- Can be difficult to isolate arguments
- Consider passing `null`
- Consider making the sprout a public static method
  - Avoid the need to instantiate the class to test the sprout method
- Giving up on testing the original method?
- Clear separation of old/new code
  - Clear interface
  - Identify all local variables affected
The *Sprout Class* technique

- A limit of *Sprout Method*: to test the method, we must usually instantiate its class
- May not be reasonable to break dependencies to allow us to instantiate the class (for testing) if we are short on time
- *Sprout Class* says:
  - Create a new class with (a single method with) the new functionality
  - The original method instantiates the new class, and calls its method
- The new class is usually easier to put under test
  - Although again, not the call site

**Sprout Class steps**

1. Identify where we need to make a code change
2. Can we formulate the the change as a sequence of statements at one place in a method?
   - Name the new class after this sequence
   - Write down a call to create an instance of the new class and then call its method, and comment those lines out
   - Again, *before* writing the method itself — this gives us the sense of what the call will look like in context
3. Determine the local variables we need from the source method, and make them arguments to the class constructor
4. Will the method call to the sprouted class need to return a result?
   - If so, change the commented-out call to save its return value
5. Develop the sprout class test-first
6. Remove the comment in the source method to enable the object creation and method call

**Sprout Class can detract from design**

- At least, in the short term
- Tiny little classes!
- Things that would have ideally stayed in one class end up in sprouts just to make safe change possible
- Can mitigate by reusing patterns in existing code
- Sprouted classes can be the basis for subsequent (better) changes
5.4.2  

*Move Method* and *Move Class*

Another example — paying an employee

```java
class Employee {
    // ...
    public void pay(final Date date) {
        final Money amount = new Money();
        for(final Timecard card : timecards) {
            if (payPeriod.contains(date)) {
                amount.add(card.getHours() * getPayRate());
            }
        }
        getPayDispatcher.pay(this, date, amount);
    }
}
```

- New requirement: every time we pay an employee, we must add the employee’s name to a file for some reporting system
  - Where does this code go?
  - The `pay` method does seem easiest
  - But these are two different operations. Will we always want to do them together?

Instead of adding to `pay`, a new method

```java
class Employee {
    // ...
    private void dispatchPayment(final Date date) {
        final Money amount = new Money();
        for(final Timecard card : timecards) {
            if (payPeriod.contains(date)) {
                amount.add(card.getHours() * getPayRate());
            }
        }
        getPayDispatcher.pay(this, date, amount);
    }

    public void pay(final Date date) {
        logPayment();
        dispatchPayment(date);
    }

    private void logPayment() {
        // ...
    }
}
```

This is the *Wrap Method* technique

1. Identify a method we need to change
2. Can we formulate the change as a sequence of statements in the method?
   - Rename the method and make it private
• Create a new wrapper method with the same name and signature as the old one

3. Place a call to the renamed old method in the wrapper method

4. Develop and test a new method for the new feature, and call it from the wrapper method

Wrap Method
Another way to add new, tested functionality where we can’t easily write tests for existing code

• Does not increase size of existing methods

• Primary disadvantage: can lead to poor names

Wrap Method variation
When we don’t care about having the same name for the wrapper method with both functionalities:

1. Identify a method we need to change

2. Can we formulate the change as a single sequence of statements?
   • Develop and test a new method for the new feature

3. Create a wrapper method that calls both the new method and the old method

What is Wrap Class?
How do we make Wrap Method into a class-level operator?

• Recall the same example

```java
public class Employee {
    // ...
    public void pay(final Date date) {
        final Money amount = new Money();
        for(final Timecard card : timecards) {
            if (payPeriod.contains(date)) {
                amount.add(card.getHours() * getPayRate());
            }
        }
        getPayDispatcher.pay(this, date, amount);
    }
}
```

Use the Decorator pattern

• Turn Employee into an interface
   -- And something like StandardEmployee implementing with the original implementation

• Write a new class LoggingEmployee which also implements the interface, but delegates to an Employee instance’s methods

```java
public class LoggingEmployee implements Employee {
    private final Employee employee;
    public LoggingEmployee(final Employee employee) {
        this.employee = employee;
    }
    // ...
}
```
// Other methods just delegate to employee
public void pay(final Date date) {
    logPayment();
    employee.pay(date);
}
private void logPayment() {
    // ...
}

- LoggingEmployee is a decorator for Employee
- Most of its methods will just call the corresponding method on its employee field
- We can nest different decorations of Employee:
  Employee e = new LoggingEmployee(
      new PretaxInvestmentEmployee(
          new StandardEmployee()));

Or a more minimal wrapping

public class LoggingPayDispatcher {
    private final Employee employee;
    public LoggingPayDispatcher(final Employee employee) {
        this.employee = employee;
    }
    public void pay(final Date date) {
        logPayment();
        employee.pay(date);
    }
    private void logPayment() {
        // ...
    }
    // Note --- not necessarily any other methods
}

Essence of Wrap Class

- Add behavior leaving the original class’s behavior intact
- If there are many calls to the value we will wrap, then the full Decorator-style wrapping may be best
- If the new behavior is limited to a few places with few other calls on the instance, the more minimal wrapping may be preferable

General case of Wrap Class

1. Identify a method where we need to make a change
2. Can we formulate the the change as a sequence of statements in the method?
   - Create a class that accepts an instance of the class we are going to wrap in its constructor, and stores it in a private field
3. Create a method in the wrapper class that does the new work
4. Write another method in the wrapper class that calls both the new work method, and the old method on the wrapped class
5. Instantiate the wrapper class where we need to enable the new behavior
**A Wrap or a Sprout?**

- *Sprout Method* and *Wrap Method* are not that different
  - *Wrap Method* suggests that the new feature is independent and equally important as the old feature
  - *Sprout Method* allows the new work to take place at a particular point relative to the old work; *Wrap Method* is appropriate when they are separable, and possibly can happen in any order

- Two cases that point to *Sprout Class*:
  1. The new class’s responsibility is separate from the original, which still has its own relevance
     - For example in tax-calculation software, code for deductions relevant only at certain times of the year
  2. We cannot instantiate the original class in a test harness

- *Wrap Class* has a higher threshold. Two cases that point to *Wrap Class*:
  1. The behavior to be added is completely independent from what’s already in the class
     - Don’t want to "pollute" the old class with unrelated or low-level details
  2. The original class is so big already that we hesitate to make it worse
     - Wrapping separates the big old class, and makes a roadmap for future changes
     - This can seem like a silly change, but there is psychological value is separating the old pre-testing code from additions made under a discipline of testing
     - A clear signal of progress over the mountain!

**5.5 Getting classes into a test harness**

**5.5.1 Despite irritating parameters**

**A dialog between pair programmers**

- We start off eager and buoyant!
  - "This will be easy!! All we have to do is make the Floogle class flumoux a little bit!!"

- Then they have a look at the Floogle class
  - "We need to add a method here, and change this method here, and of course get the class into the test harness. This won’t be too bad!!"

- Then they look at the constructors for Floogle
  - "Well, the simplest constructor takes three parameters. But maybe it won’t be hard to construct it."

- Parameters can be very *irritating*

**An example**

In the code for a billing system, an untested class *CreditValidator*

```java
public class CreditValidator {
    public CreditValidator(CreditMaster master,
                            String validatorID,
                            RGHConnection connection) {
        // ...
    }

    public Certificate validateCustomer(Customer customer)
        throws InvalidCredit {
```
Here’s a test that will definitely fail to even compile

@Test public void testNewCreditValidator3() {
    final CreditValidator cv = new CreditValidator();
}

• Remember the TDD process: this is a perfect place to start
• Construction tests can look odd
  – “It’s just creating an object! How can a test not call any methods?”
  – But it’s actually fine.
    • The TDD process normally does suggest that the solution to making the test pass should be in the development code
    • But when we’re starting with legacy code, lack of a test harness for the class is a fine “failure” to address
    • And remember our additional step for legacy code

Making the test compile

public CreditValidator(CreditMaster master,
        String validatorID,
        RGHConnection connection) {
    // ...
}

• The CreditMaster argument retrieves the local information used to make a decision
  – Needs just local files
    final CreditMaster
        master = new CreditMaster("crm2.mas’, true);
• The validator ID is easy to produce — just a string, that probably has something to do with our login
• The RGHConnection is the tricky part
  – Connection to a network server which responds to credit report requests
  – It’s a bad idea to put a specific network connection into a test
  – The machine may be down, or we may be developing offline, or we may have to pay for each connection
  – This is an Irritating Parameter!

Addressing the Irritating Parameter
Often the easiest way is again to apply the Extract Interface pattern
• Convert the original class into an interface
• Use the original class as the interface’s canonical implementation, and instantiate that class rather than the

original: From to

```
• Then we can add a fake implementation for testing

Fake connections for real tests

• We can use the fake connection to make our test for the CreditValidator compile and pass:

```java
@Test public void testNewCreditValidator3() {
    final CreditMaster
        master = new CreditMaster("crm2.mas", true);
    final RGHConnection
        connection = new FakeRGHConnection();
    final CreditValidator
        cv = new CreditValidator(master, "acct504",
                                 connection);
}
```

• What if we notice that this constructor does not actually call any methods on the CreditMaster instance?
  – Could we just pass null?
  – "That’s a terrible idea! Passing null is just inviting trouble! It’s horrible practice!"
  – But the standards we use for test code are different
  – The problem with using null in production code is that it can cause errors in far-flung parts of the program
    * So in production code, of course: explicitly use null only as a very last resort
  – But this is a test harness — if the null causes an error, we will know immediately!
Aside: the Null Object pattern

- Consider a function mapping employee IDs to the instance of an Employee class for that employee
- What should the function do if passed an ID which does not correspond to an employee?
- Throwing an exception means that every time we call the function, we must be prepared for an error
- In some cases, a special class for vacuous instances may be helpful

![Diagram of Employee and NullEmployee classes]

- Instead of returning null, return the singleton instance of the NullEmployee
- Avoids null dereferences, since this is an actual instance whose methods simply do nothing
- Shields client code for error checking
- Caveat: can lead to things like counting errors since we no longer need to clean up source data

An alternative to Extract Interface: Subclass and Override Method

- Useful when the inconvenient dependency is not hardcoded into a constructor
- For example: the constructor of RGHConnection actually calls its own connect method to set up the connection

  - For a test we could create a subclass that overrides connect to do nothing

![Diagram of RGHConnection and RGHConnectionWithoutConnect classes]

5.5.2 Despite hidden dependencies

**A Hidden Dependency**

Consider this class from a mailing list system:
public class MailingListDispatcher {
    private final MailService service;
    private int status;

    public MailingListDispatcher() {
        this.service = new MailService();
        this.status = MAIL_OK;
        final int clientType=12;
        service.connect();
        if (service.getStatus() == MS_AVAILABLE) {
            service.register(this, clientType, MARK_MSGS_OFF);
            service.setParam(clientType, ML_NOBOUNCE | ML_REPEATOFF);
        } else {
            status = MAIL_OFFLINE;
        }
        // ...much more...
    }

    void sendMessage(final String message) { ... }
    void addRecipient(int id, EmailAddress addr) { ... }
    // ...various other methods...

    • Uses new to create a MailService instance
    • Does detailed configuration of MailService here, instead of in that class
    • Uses a magic number — what is 12 supposed to mean!?
    • MailingListDispatcher has a dependency on MailService, hidden in the constructor

    Using Parameterize Constructor against Hidden Dependency
    • We can add a parameter to the constructor to expose the dependency

    public class MailingListDispatcher {
        private final MailService service;
        private int status;

        public MailingListDispatcher(MailService service) {
            this.service = service;
            this.status = MAIL_OK;
            final int clientType=12;
            service.connect();
            if (service.getStatus() == MS_AVAILABLE) {
                service.register(this, clientType, MARK_MSGS_OFF);
                service.setParam(clientType, ML_NOBOUNCE | ML_REPEATOFF);
            } else {
                status = MAIL_OFFLINE;
            }
            // ...much more...
        }

        • We can also Preserve Signatures to make this change very safe

        public MailingListDispatcher() {
            this(new MailService());
        }
    }
Extract and Override Factory Method against Hidden Dependency
An alternative to Parameterize Constructor

public MailingListDispatcher() {
    this.service = makeMailService();
    // ...rest of method...
}

protected MailService makeMailService() {
    final MailService service = new MailService();
}

- Dependency is still in constructor, but the factory method is a seam we can exploit
  - Testing version of class overrides factory method to return a "fake" instance
- Can put additional configuration into factory method as well
- Disadvantage: hard to use in C++
  - Constructors cannot call virtual methods (that is, methods resolved by dynamic dispatch) resolved in sub-classes

Extract and Override Getter against Hidden Dependency

- Sometimes we can avoid the problems with Extract and Override Factory Method
- Replace uses of the field with calls to a getter method
- Getter method builds actual instance at its first reference

public MailingListDispatcher() {
    this.serviceSetup = false;
    this.status = MAIL_OFFLINE;
    // ...rest of old constructor...
}

private MailService service;
private boolean serviceSetup;
private int status;
protected MailService getMailService() {
    if (!serviceSetup) {
        this.service = new MailService();
        final int clientType=12;
        service.connect();
        if (service.getStatus() == MS_AVAILABLE) {
            this.status = MAIL_OK;
            service.register(this, clientType, MARK_MSGS_OFF);
            service.setParam(clientType, ML_NOBOUNCE | ML_REPEATOFF);
        } else {
            status = MAIL_OFFLINE;
        }
        this.serviceSetup = true;
    }
    return this.service;
}
5.5.3 Despite additional instantiations in the constructor

Another use of Extract and Override Factory Method: against Construction Blob

• Parameterize Constructor is an easy way to remove dependencies hidden in a constructor

• But sometimes there’s just too many dependencies/values to remove
  – Too many parameters to the constructor as a result
  – Or worse, some of the objects are used to construct other objects

```java
public class WatercolorPane {
  WatercolorPane(Form border, WashBrush brush,
                  Pattern backdrop) {
    // ...  
    Panel anteriorPanel = new Panel(border);
    anteriorPanel.setBorderColor(brush.getForegroundColor());
    Panel backgroundPanel = new Panel(border, backdrop);
    FocusWidget cursor = new FocusWidget(brush,
                                          backgroundPanel);
  }
}
```

• What if we want to sense via the cursor?

Supersede Instance Variable against Construction Blob

• Define a setter to overwrite the field

```java
public void supersedeCursor(FocusWidget cursor) {
  this.cursor = cursor;
}
```

• Easy in languages with automatic garbage collection (Java, C#, Scala, etc) — but much trickier in C++
  – Do we garbage collect the old cursor immediately?

• As a general rule. avoid Supersede Instance Variable unless we can’t avoid it

5.5.4 Despite global dependencies

Global variables, global dependencies

• Global variables are among the hardest dependencies to cope with

• There are a number of patterns that can resolve simple cases
  – Parameterize Constructor
  – Parameterize Method
  – Extract and Override Call

• A more subtle form of global variable shows up in the Singleton pattern

```java
SomeClass someObject
  = ObjectSupply.getInstance().getObject(key);
```

– getInstance is a static method returning the one instance of an ObjectSupply
– Can guarantee that it is a singleton by (for example) making the constructor private
– Note that many people now consider Singleton to be an anti-pattern, an pattern of bad practice
Why is Singleton bad?

- Consider these statements:

```java
final Account example = new Account();
example.deposit(1);
int balance = example.getBalance();
```

What can these statements effect?

- Maybe some static variables in `Account` (although that’s questionable practice)
- Maybe some other `Account` instances or other objects stored in static fields (although that’s terrible, terrible practice in general)
- We aren’t passing other objects to the constructor or method, so it’s a very safe assumption that no other objects are changed by these statements
- The local variable is set, but that’s obvious — no surprise there

- Using complicated global variables turns this reasoning on its head
  - Method calls on global objects intentionally open us to non-local effects
  - What other objects might have been called (registered) with the `Singleton`, and so subject to (obscure) changes by a later call to the `Singleton` from elsewhere?

Faking a Singleton

A `Singleton` is tricky and unsatisfying to fake

- Remember: a static method like `getInstance()` will be backed by a (usually private) static field storing the singleton

- Add a new static method to the singleton class

```java
public class SomeClass {
    // ...leave everything else intact...
    public void setTestingInstance(SomeClass newInstance) {
        this.instance = newInstance;
    }
}
```

- But we still have to deal with the private constructor

  - Often the constructor is private for a very good reason
    * Two database connections?
    * Two consumers of more than half the available disk space?
    * Two controllers for fuel rods in a nuclear reactor?
  
  - (Of course if these reasons do not apply, we can be comfortable about a public constructor)
  
  - There is no silver bullet to forbid making additional instances in the production code but allowing it in the test code
Compromise: protected constructor

- A protected constructor allows subclassing, but not instantiation outside of that class.
- We’d need some team agreement to only subclass in testing code
  - Possibly, this can be enforced with development/repository tools.
- To make alternate singletons for testing
  - Subclass and Override Method to create sensing objects which do not e.g. contact the database server.
  - Extract Interface if there are more extensive dependencies.
- This overall pattern is called Introduce Static Setter.

Singleton can be an indicator of a design flaw

- Remember the example of a database connection in the Singleton.
- Set us this way so any of the program classes can interact with the database.
- Why are so many classes interacting with the particular database?
- If the program actually does anything besides access a database, then a better design would be for most classes to do that other thing, and a few classes take care of actual database interaction.

A global may not actually be used globally

- If a Singleton is really used in only a few places, Parameterize Constructor and Parameterize Method can be effective in localizing it.
- But there is a space penalty, since every object could then store it in a field.

5.6 Getting methods into a test harness

Harnessing methods

- So we’ve made a class accessible to tests.
- But that’s just half the battle
  - Now we actually have to write tests.
  - Will we actually be able to call the necessary methods?
- Sometimes we can call methods without instantiating a class at all
  - If the method is static, or if we can easily make it static.
- But sometimes there will be other difficulties.

Why might it be hard to test a method?

- It might not be accessible to the test
  - For example, it could be private.
- It might be hard to construct its parameters.
- It might have undesirable side effects
  - Like modifying a database or launching a missle.
- We might need to sense test results through an object the method uses.
5.6.1 Despite access permissions

The Hidden Method

- In other words: *private* and *protected* methods
- Can we test through a *public* method?
  - Often, *private* methods implement steps of a *public* method
  - Saves the trouble of finding alternate access
  - Good to have extra tests of public methods anyway
  - Avoids overgeneralizing tests of a *private* method, with calls that could not be made in practice
- But sometimes, that’s not feasible
  - Might be too difficult to set up calls through the public method
  - Might want concrete feedback and illustrative tests specific to the *private* method

Make the Hidden Method public

- Certainly a direct solution!
- Does this bother us?
  - It’s a sign that our class is doing too much
  - Rather than maintain the poor design, better to fix it
- Why might the method be private?
  1. It’s just a utility. Client classes wouldn’t care about it
     - Not very severe
     - An extra public method would be forgivable
     - We should give thought to whether it would be better to move the method to another class
  2. It would be difficult for clients to use this method without adversely affecting results from other methods on the class
     - More serious
     - But there’s a remedy: move the method to another class
- Does this sound strident?
  - Nonetheless: Good design is testable; untestable design is bad

An example — some nasty (but realistic) legacy code

```java
public class CCAImage {
    private void setSnapRegion(int x, int y, int dx, int dy) {
        // ...
    }
    public void snap() {
        // ...
    }
    // ...and a whole lot more...
}
```

- Takes pictures in a security system
– (Wait, why is the functionality for snapping a picture in an image class?)

• The API to the physical camera hardware has changed, so we need to update setSnapRegion

• snap can make several calls to setSnapRegion
  – Depends on whether the subject is in motion

• Private variables of the CCAImage class govern target tracking
  – Calls to setSnapRegion from outside of snap would disrupt the tracking
  – So we really do not want to make this method public

• Aside: how we get into this mess?
  – The underlying issue is that the CCAImage class just does too much
  – The real solution is to break up the class into smaller classes
    * We’ll look at approaches to this later
    * But it’s time-consuming — what can we do quickly?

Again, a compromise

• Make the method protected instead of private

• Write a subclass of CCAImage for testing
  – Override setSnapRegion with a public method
  – The new method just calls the parent method

• Is this OK?
  – We just gave convincing reasons why the method should not be public. How is this different?
  – The real question: is making the method protected a fair trade for getting it under test?
  – It probably is
  – Subclasses can call the method, but that’s probably OK
  – Will the risk motivate us to refactor and break up the overstuffed class?

Another possible solution: reflection

• Some languages (including Java) have a reflection library, which allows us to query the structure of classes and methods at runtime

• We can also build calls to methods at runtime this way, and avoid certain compile-time checks
  – In Java’s reflection library, we can get around private modes
  – The disadvantage is that reflection-based method invocation is difficult and verbose
5.6.2 Despite subclass limits

**The "Helpful" Language Feature**

Consider this method:

- Takes a list of uploaded files from a web client
- Iterates through each one
- Returns a list of streams associated with files matching certain criteria

```java
public void List<Stream> getKsrStreams(HttpFileCollection files) {
    final ArrayList<Stream> result = new ArrayList<Stream>();
    for (final HttpPostedFile file : files) {
        if (file.name.endsWith(".ksr")
            || (file.name.endsWith(".txt")
                && file.contentLength > MIN_LEN)) {
            // ...
            result.add(file.getStream());
        }
    }
    return result;
}
```

- We’d like to make changes to, and perhaps refactor, this method
  - For testing, we’d like to create an HttpFileCollection, and populate it with sample data
- **HttpPostedFile has no public constructor**
  - Worse, the class is final, and is defined in a 3rd-party library
  - So we don’t have source code

**What can we do?**

- Are there superclasses which are not final?
  - We could create our own subclasses to vary behavior, if the superclasses have enough common functions
- Another pattern that can be useful: **Skin and Wrap the API**
  - We can’t turn the original class into an interface
  - But we can write:
    - An interface with methods mirroring the original class
    - A wrapper for an object of the original class which delegates to the original class’s methods, and implements the new interface
    - Another test class which implements the new interface
  - So now we pass collections of objects implementing the new interfaces
    - The cost is that in the production code, we must iterate through the original collection, and wrap the objects for our new collection
5.6.3 Despite side-effects

This class does it all

```java
public class AccountDetailFrame extends Frame
    implements ActionListener, WindowListener {
private TextField display = new TextField(10);
    // ...

    public AccountDetailFrame(...) { ... }

    public void actionPerformed(ActionEvent event) {
        String source = (String)event.getActionCommand();
        if (source.equals("project activity")) {
            DetailFrame detailDisplay = new DetailFrame();
            detailDisplay.setDescription(getDetailText() + " " + getProjectionText());
            detailDisplay.show();
            String accountDescription = detailDisplay.getAccountSymbol();
            accountDescription += "; ";
            // ...
            display.setText(accountDescription);
            // ...
        }
    }

    // ...other methods...
}
```

• It creates GUI components
• It receives notifications from the components via this handler
• It calculates what it needs to display
• It displays it
• All of this in a strange way:
  – It builds up detailed text
  – And then creates and displays it in another window
  – Then it pulls information from the new window, processes it, and displays it in one of its own components
• But we shouldn’t call up a GUI from the test harness
  – So these are Undetectable Side-Effects

How do we test an Undetectable Side-Effect?

• We divide it
• First, separate to remove the dependence on the ActionEvent class

48
public void actionPerformed(ActionEvent event) {
    String source = (String)event.getActionCommand();
    performCommand(source);
}

public void performCommand(String source) {
    if (source.equals("project activity")) {
        DetailFrame detailDisplay = new DetailFrame();
        detailDisplay.setDescription(
            getDetailText() + " " + getProjectionText());
        detailDisplay.show();
        String accountDescription = detailDisplay.getAccountSymbol();
        accountDescription += "; ";
        // ...
        display.setText(accountDescription);
        // ...
    }
}

How do we test an Undetectable Side-Effect?

Next, we separate storage for the new frame

private DetailFrame detailDisplay;

public void performCommand(String source) {
    if (source.equals("project activity")) {
        detailDisplay = new DetailFrame();
        detailDisplay.setDescription(
            getDetailText() + " " + getProjectionText());
        detailDisplay.show();
        String accountDescription = detailDisplay.getAccountSymbol();
        accountDescription += "; ";
        // ...
        display.setText(accountDescription);
        // ...
    }
}

How do we test an Undetectable Side-Effect?

Next, extract the code that uses the DisplayFrame into methods

private DetailFrame detailDisplay;

public void performCommand(String source) {
    if (source.equals("project activity")) {
        detailDisplay.setDescription(getDetailText() + " " + getProjectionText());
        String accountDescription = detailDisplay.getAccountSymbol();
        accountDescription += "; ";
        // ...
        display.setText(accountDescription);
        // ...
    }
void setDescription(String description) {
    detailDisplay = new DetailFrame();
    detailDisplay.setDescription(description);
    detailDisplay.show();
}

String getAccountSymbol() {
    return detailDisplay.getAccountSymbol();
}

How do we test an Undetectable Side-Effect?
And repeat with other GUI components

public void performCommand(String source) {
    if (source.equals("project activity")) {
        setDescription(getDetailText() + " " + getProjectionText());
        String accountDescription = getAccountSymbol();
        accountDescription += "; ";
        // ...
        setDisplayText(accountDescription);
        // ...
    }
}

void setDisplayText(String description) {
    display.setText(description);
}

Now can we test the monster?

public void performCommand(String source) {
    if (source.equals("project activity")) {
        setDescription(getDetailText() + " " + getProjectionText());
        String accountDescription = getAccountSymbol();
        accountDescription += "; ";
        // ...
        setDisplayText(accountDescription);
        // ...
    }
}

• We can apply Subclass and Override Method
  – Override methods like setDescription and setDisplayText to sense (without GUI use)
  – Override getDetailText and getProjectionText to return test values
  – And performCommand is now testable!

Command/query separation
• Design principle first described by Bertrand Meyer
• A method should be a command or a query, but not both
– Command modify the state of an object, but return no result
– Queries return a value, but do not change the state
• Makes the purpose of methods clearer
  – Do not need to look at the source to figure out possible side-effects

5.6.4 Big changes for hard cases

Worst-case scenarios: Break Out Method Object

• We discussed obstacles to getting a method into a test harness: Hidden Method, The "Helpful" Language Feature, Undetectable Side Effect
• But these are specific cases
• A general approach for methods is the Break Out Method Object pattern
  – It’s an amped-up generalized version of Sprout Class

Steps for Break Out Method Object

1. Create a new class to house the method code
2. Create a constructor for the new class with exactly the arguments of the method
   • If the original method uses fields or other methods of the original class, insert a new first argument of the original class to the new class constructor
3. For each argument of the new class constructor, declare an field with the same name and type as the constructor argument
   • Then in the new class constructor, assign the constructor argument value to the corresponding field
4. Create an empty execution method on the new class
   • Often just called run
5. Copy the body of the old method into the new execution method
6. Use the compiler to find where the old method body is using methods or fields from the old class
   • Address each to make the new class compile
   • Often just by qualifying the references with the old class argument
   • Possibly making methods public, or adding getters for private fields
7. Change the old method to create an instance of the new class and call its execution method
8. Use Extract Interface to break dependencies on the old class, any other classes
Example — (some of) a complicated GUI class

```java
public class GDIBrush {
    private void drawPoint(int x, int y, Color color) {
        // ...
    }

    public void brush(Vector<Point> renderingRoots, ColorMatrix colors, Vector<Point> selection) {
        for(final Point p : renderingRoots) {
            int n;
            // ...much prep omitted...
            drawPoint(p.x, p.y, colors.get(n));
        }
    }

    // ...many other methods, fields omitted...
}
```

- We cannot spare the effort to get GDIBrush itself under test
- But we need to test the brush method

First we create the new class

```java
public class Renderer {

    public Renderer() {
    }

}
```

Then copy the old method arguments to the constructor
public class Renderer {
    public Renderer(Vector<Point> renderingRoots,
                    ColorMatrix colors,
                    Vector<Point> selection) {
    }
}

And the body of `draw` uses other resources from `GDIBrush`

public class Renderer {
    public Renderer(GDIBrush gdiBrush,
                    Vector<Point> renderingRoots,
                    ColorMatrix colors,
                    Vector<Point> selection) {
    }
}
Next we create instance fields and assign them from the constructor

```java
public class Renderer {
    private GDIBrush gdiBrush;
    private Vector<Point> renderingRoots;
    private ColorMatrix colors;
    private Vector<Point> selection;

    public Renderer(GDIBrush gdiBrush,
                    Vector<Point> renderingRoots,
                    ColorMatrix colors,
                    Vector<Point> selection) {
        this.gdiBrush = gdiBrush;
        this.renderingRoots = renderingRoots;
        this.colors = colors;
        this.selection = selection;
    }
}
```

---

Add a stub for the execution method

```java
public class Renderer {
    private GDIBrush gdiBrush;
    private Vector<Point> renderingRoots;
    private ColorMatrix colors;
    private Vector<Point> selection;

    public Renderer(GDIBrush gdiBrush,
                    Vector<Point> renderingRoots,
                    ColorMatrix colors,
                    Vector<Point> selection) {
        this.gdiBrush = gdiBrush;
        this.renderingRoots = renderingRoots;
        this.colors = colors;
        this.selection = selection;
    }

    public void run() {
    }
}
```
Copy the old method body into the execution method

```java
public class Renderer {
    private GDIBrush gdiBrush;
    private Vector<Point> renderingRoots;
    private ColorMatrix colors;
    private Vector<Point> selection;

    public Renderer(GDIBrush gdiBrush,
                    Vector<Point> renderingRoots,
                    ColorMatrix colors,
                    Vector<Point> selection) {
        this.gdiBrush = gdiBrush;
        this.renderingRoots = renderingRoots;
        this.colors = colors;
        this.selection = selection;
    }

    public void run() {
        for(final Point p : renderingRoots) {
            int n;
            // ...much prep omitted...
            drawPoint(p.x, p.y, colors.get(n));
        }
    }
}

... and qualify method references with gdiBrush

```
In the original class — `drawPoint` must now be public

```java
public class GDIBrush {
    public void drawPoint(int x, int y, Color color) {
        // ...
    }

    public void brush(Vector<Point> renderingRoots,
                       ColorMatrix colors,
                       Vector<Point> selection) {
        for(final Point p : renderingRoots) {
            int n;
            // ...much prep omitted...
            drawPoint(p.x, p.y, colors.get(n));
        }
    }

    // ...many other methods, fields omitted...
}
```

In the original class — call the break-out method

```java
public class GDIBrush {
    public void drawPoint(int x, int y, Color color) {
        // ...
    }

    public void brush(Vector<Point> renderingRoots,
                       ColorMatrix colors,
                       Vector<Point> selection) {
        new Renderer(this, renderingRoots, colors, selection).run();
    }

    // ...many other methods, fields omitted...
}
```

5.7 Characterization tests and classes

Tests for managing change

- Why do we have tests?
  - Find bugs
    - In new code under TDD, most tests pass
      * With automated testing, these tests keep us from adding new bugs
      * (With manual testing only, we tend to get behind)
  - What do we do with legacy code?
    - In general, finding bugs is not a problem
    - In general, we don’t have time to fix all the bugs
    - Transition from tests that specify to tests that preserve
      * We may not have a spec for the legacy software
      * If we do, it’s probably not developed through to test cases
* But we have the current state of the software as a reference point
* So we want to know when we change what a program does
* Assess whether a change is for good or for ill
  – For legacy software, we write *characterization* tests
    * For a particular area of the code which we need to change
    * Record what it does now, so we can diagnose changes we introduce

**Writing characterization tests**

1. Get classes/methods into a test harness
2. Write an assertion that you know will fail
3. Let the failure tell you what the behavior actually is
4. Change the test to expect the actual behavior
5. And repeat

**The answer probably isn’t *fred***

```java
void testGenerator() {
    final PageGenerator pg = new PageGenerator();
    assertEquals("fred", pg.generate());
}
```

• But we can still run the test and let *junit* tell us what actually happens

```
junit.framework.ComparisonFailure: expected:<fred>
but was::<>
```

• So we update the test

```java
void testGenerator() {
    final PageGenerator pg = new PageGenerator();
    assertEquals("", pg.generate());
}
```

• And so on: further tests for what we see from *generate* after various operations on/additions to *pg*

**When do we stop?**

• The empty document example is a good example of where we *start*
• But how do we know when we have *enough* tests?
• One metric: consider these tests to be about understanding code
• When the tests cover the cases we’d think about when trying to figure out the code, we may have gone far enough
• These are *not* black-box tests!
• *The Method Use Rule*: before you use a method in a legacy system, write tests for it
  – Tests as a form of document: they reflect when we know to be true about the a method
Characterizing classes

- We have a class, and we want to figure out what to test
- How do we do it?
  - First, try to figure out what it does at a high level
    - Write tests for the simplest things we can imagine it doing
    - Then see where curiosity leads us

Guidelines for characterizing classes

1. Look for tangled pieces of logic
   - For difficult sections of code, consider introducing a \textit{sensing variable}
   - Sensing variables can help us make sure we execute particular pieces of code

2. Make a list of things that could go wrong
   - As you study a class or a method
   - See if you can formulate tests that trigger the faults you find

3. Think about the inputs you supply under test
   - What happens at extreme values?

4. Look for invariants
   - Are there conditions which should be true at all times during the lifetime of the class?
   - Attempt to write tests to verify them
   - Refactoring may be needed to reveal the invariants, or to enable testing them
   - Sensing variables may be useful here as well

Sensing variables

- New variables or flags
- Not read by other code fragments
- So not accessed by other parts of the production code
- So often intended to be temporary
  - A flag set when a particular branch is executed
  - A record of list nodes which are changed
- Can guide testing
  - Prove that our test data is sufficient to cover code of interest
  - Remove the variable once tests and test data are in place

Tests make documentation

- Start with easy cases
- Move on to idiosyncrasies
- Add to class documentation as well
What happens when we find bugs?

• If we are adding tests to existing test-free code, we will probably find bugs

• Should you fix them?
  – Depends on the situation
  – If the code has never been deployed, you should fix the bug
  – If the code has been deployed, a client may depend on the behavior
    * Even though you see it as a bug!
    * Think about what ripple effects fixing the bug may cause
  – On balance, tend to fix bugs
    * Especially if the behavior is clearly wrong
    * If you only suspect the code to be in error, mark it as suspicious and file a bug or ticket to discuss/track it
    * Find out quickly if an issue is really a bug, and how best to deal with it

Targeted testing

• Computing the value of fuel in a leased tank

```java
public class FuelShare {
    private long cost=0;
    private double corpBase=12.0;
    private ClientAlphaLease lease;

    public void addReading(int gallons, Date when) {
        if (lease.ismonthly()) {
            if (gallons < Lease.CORP_MIN) {
                cost += corpBase;
            } else {
                cost += 1.2 * priceForVolume(gallons);
            }
        }

        // ...

        lease.postReading(readingDate, gallons);
    }

    // ...other methods...
}
```

• We want to move the top-level if-statement to a new method, and then move that method to the `ClientAlphaLease` class
  – Assume we already have basic characterization tests for the class

Where the refactoring takes us
We can imagine how the code will look after the refactoring
public class FuelShare {
    private long cost=0;
    private double corpBase=12.0;
    private ClientAlphaLease lease;

    public void addReading(int gallons, Date when) {
        cost += lease.computeValue(gallons, priceForVolume(gallons));
        // ...
        lease.postReading(readingDate, gallons);
    }

    // ...other methods...
}

public class ClientAlphaLease extends Lease {
    public long computeValue(int gallons, long totalPrice) {
        long thisCost=0;
        if (lease.ismonthly()) {
            if (gallons < Lease.CORP_MIN) {
                thisCost += corpBase;
            } else {
                thisCost += 1.2 * totalPrice;
            }
        }
        return thisCost;
    }

    // ...other methods...
}

• What kind of tests do we need to make sure the refactoring is correct?

Tests for assuring the refactoring

• We know we won’t change some logic

if (gallons < Lease.CORP_MIN) {
    cost += corpBase;

    This will be exactly the same flow of logic, so we can be slack about testing it.

• But this part of the original logic is more transformed:

} else {
    cost += 1.2 * priceForVolume(gallons);
}

It becomes

} else {
    thisCost += 1.2 * totalPrice;
}

– It may not be *that* big of a change
– But it’s enough that we should make sure that our tests cover the else-branch

60
Is the else-branch covered?
Recall the original code:

```java
public void addReading(int gallons, Date when) {
    if (lease.ismonthly()) {
        if (gallons < Lease.CORP_MIN) {
            cost += corpBase;
        } else {
            cost += 1.2 * priceForVolume(gallons);
        }
    }
    // ...
    lease.postReading(readingDate, gallons);
}
```

- To cover the else-branch we seem to need:
  - A monthly lease
  - To `addReading` with more gallons than `CORP_MIN`

- Is there some other way that the test could pass?
  - Here, the reasoning may be simple enough that we can be sure there isn’t
    * But remember those ellipses!
  - But in more complicated cases, `Insert Sensing Variable` is our ally

```java
public boolean ranElseBranch=false;
public long computeValue(int gallons, long totalPrice) {
    long thisCost=0;
    if (lease.ismonthly()) {
        if (gallons < Lease.CORP_MIN) {
            thisCost += corpBase;
        } else {
            thisCost += 1.2 * totalPrice;
            ranElseBranch=true;
        }
    }
    return thisCost;
}
```

A failure out of a success out of a failure

- So now we are confident that the `control flow logic` of our code means that the test covers the else-branch
- But what if there is some special behavior relating to the inputs?
- What if the code used `double` to represent dollars instead of `int` and `long` to represent cents?

```java
public class FuelShare {
private double cost=0.0;
private double corpBase=12.0;
private ClientAlphaLease lease;

public void addReading(int gallons, Date when) {
    if (lease.ismonthly()) {
        if (gallons < Lease.CORP_MIN) {
...
cost += corpBase;
} else {
    cost += 1.2 * priceForVolume(gallons);
}

// ...

lease.postReading(readingDate, gallons);

// ...other methods...

– Never even mind that floating-point rounding error are probably leaking cents everywhere!

**Beware of changes in representations**

- Every conversion from `double` to `int` is an opportunity for an error

- Are our characterization tests too much about a "sunny day"?
  - Not testing special conditions
  - Only verifying that certain behaviors are present
  - Without working through edge cases, can we infer that all behaviors are really preserved?

- We must make sure that we exercise all branches *and* all conversions!
  - By manually calculating the the expected values for a piece of code
  - By using a debugger to step through assignments to expose the lower-level conversions our inputs trigger
  - By using *Insert Sensing Variables* to verify that both branches and conversions are exercised
  - The most valuable characterization tests exercise a specific path, and exercise each conversion along the path

**Another option: characterize a smaller chunk of code**

- Note the ellipses: `convertValue` is really a bigger method than we’re writing out

```java
public void addReading(int gallons, Date when) {
    if (lease.ismonthly()) {
        if (gallons < Lease.CORP_MIN) {
            cost += corpBase;
        } else {
            cost += 1.2 * priceForVolume(gallons);
        }
    }
    // ...
    lease.postReading(readingDate, gallons);
}
```

- It might be easier to extract methods smaller than all of `computeValue`, and write tests to characterize these smaller pieces

- Refactoring tools may not always extract methods in the way we’d like
  - But manually moving the code is always an option

62
A last checklist for writing characterization tests for a change

1. First write tests for the area where you need to make a change
   • The first goal is to understand (and document) the goals and behavior of the class and relevant methods
2. Then consider the specific things you want to change
   • Write tests focusing more closely on these specifics
3. When extracting or moving functionality, focus on preserving combinations of branches and operations
   • Verify that corresponding execution paths are preserved
   • Verify that the new code is connected properly to the old code
   • Investigate and exercise conversions and other implicit operations

5.8 Dealing with libraries

Good libraries, bad libraries

• Or rather: good use of libraries, bad use of libraries
• In general, libraries are great assets
  – Don’t reinvent the wheel
  – Stand on the shoulders of giants
  – Obviously modular
• But good things can go bad
  – What if the vendor jacks up the price of licenses?
  – What if the great open-source project becomes abandonware?
  – What if support for a platform we’re targeting never comes together?
  – What if our architecture evolves and the library is no longer a good fit?
• As soon as we start using it, a library becomes a dependency!
  – We know the trouble dependencies can be
  – But we also know how to structure our code so we can break from them when we need

Approaching a library

• Every hard-coded use of a library class or method is a lost opportunity for a seam
  – Consolidate references to a library in a small portion of your code
  – Use internal APIs to build bridges between your program’s needs and the library’s functionality
  – Use wrapper classes for library objects which must commingle with your application’s objects
  – In some cases generics can help, if application classes are parameterized over library classes
• Beware their Singleton!
  – Does the library assume that there will be only one instance of certain classes?
  – Will this cause difficulties with faking and mocking?
  – Would you want (but have no way) to Introduce Static Setter?
  – Wrapping their singletons may be the best (or only) option
• Protect your sensing!
  – Sometimes libraries restrict overriding
    • In Java, this means **final** classes and objects
    • In C++, this means not declaring methods to be **virtual**
  – It is very hard to work around these restrictions
  – Many of the techniques we’ve studied rely on being able to declare subclasses and override methods

5.9 Dealing with mysterious code

Scratch refactoring

• When trying to figure out legacy code, remember that copying source code is cheap

• **Scratch refactoring** refers to refactoring that you do solely for the purpose of understanding code
  – Don’t plan on keeping it — be ready to just throw your changes away and get a fresh copy from the repository
  – Don’t worry about testing — if doesn’t matter if you break something, because you’re going to throw it away

• **Don’t get attached**
  – Since we aren’t being careful (meaning: writing tests in a disciplined way), we might introduce changes and errors
  – This is a technique for getting started with code, so we might get wrong first impressions
    • The structure we end up after a scratch refactoring may not actually be the best restructuring
    • The changes we need to introduce may require a different refactoring than the scratch result

Delete unused code

• If we can determine that certain code is never used, *delete it*

• Worried that it will turn out to be useful later?
  – That’s why we have version control systems
  – We can retrieve it later if needed
  – But less code is easier to understand

The Sprawl

• Long-lived applications sprawl
  – It’s not that it was planned that way — they very likely did start out with thoughtful planning and a crisp architecture
  – But time and deadlines, changes, mission creep, and personnel changes take a toll
  – If you know only a portion of the code well, you’ll probably gravitate towards those when adding changes — whether or not they are the ideal starting point

• Lack of awareness of the architecture leads to a system degrading
  – The system can be so complex that it takes a long time to get the big picture
  – The system can be so complex that there is no big picture
  – The deadline pressure can make you lose sight of the big picture
• So what happens?
  – Maybe it becomes impossible to make changes
  – Maybe we consider a re-write
  – Maybe there’s just a continual drag, taking longer than it should to make changes

5.10 Dealing with non-object oriented code

Testing in non-object-oriented languages

• Specifically, in procedural languages like C
• It’s a problem of seams
  – Programs in procedural languages tend to have fewer seams
  – Partly it’s a property of the languages; there is less mechanism for abstraction, so there will be fewer points where we can make an alternate concrete realization
  – So it is harder to break dependencies
• We’ve already discussed using link and preprocessor seams
  – Can we exploit object seams in any way?

We are all object-oriented now

• Many “classic” procedural languages have evolved to include class and object concepts
  – Most C compilers compile C++ as well
  – Fortran and COBOL have object-oriented extensions
  – Basic (at least on Microsoft systems) is fully object-oriented
• So the bulk of the legacy code may be procedural, but we can use object-oriented techniques to introduce tests
  – Caveat: it is not always trivial to make (for example) a C program compile under C++!

Encapsulate Global References

Consider a packet scanner written in C

```c
int scan_packets(struct rnode_packet *packet, 
    int flag) { 
    struct rnode_packet *current = packet; 
    int scan_result, err=0; 

    while (current) { 
        scan_result = loc_scan(current->body, flag); 
        if (scan_result & INVALID_PORT) { 
            ksr_notify(scan_result, current); 
        } 
        /* ... */ 
        current = current->next; 
    } 

    return err; 
}
```

• We want to put this function under test, but it calls another function ksr_notify with global side-effects
**First step: make a class**

Find the declaration of the global function, and wrap it in a new class:

```cpp
class ResultNotifier {
public:
  virtual void ksr_notify(int scan_result,
                           struct rnode_packet *packet);
}
```

- Add a default implementation of the class

```cpp
eextern "C" void ksr_notify(int scan_result,
                            struct rnode_packet *packet);
```

```cpp
void ResultNotifier::ksr_notify(int scan_result,
                                struct rnode_packet *packet) {
  ::ksr_notify(scan_result, packet);
}
```

Notice that we **Preserve Signatures**!

- Then declare a global instance of ResultNotifier.

```cpp
ResultNotifier globalResultNotifier;
```

**Now invoke the global function through the ResultNotifier**

So for example in the original function,

```cpp
extern ResultNotifier globalResultNotifier;

int scan_packets(struct rnode_packet *packet,
                 int flag) {
  struct rnode_packet *current = packet;
  int scan_result, err=0;

  while (current) {
    scan_result = loc_scan(current->body, flag);
    if (scan_result & INVALID_PORT) {
      globalResultNotifier.ksr_notify(scan_result, current);
    }
    /* ... */
    current = current->next;
  }

  return err;
}
```

- The compiler will tell us about all of the places where we need to qualify `ksr_notify`

**What does this get us?**

- Nothing, yet

- Next we can apply *Encapsulate Global References* again on `scan_packets` itself, and wrap it in a class Scanner

66
• Now we can break the implicit dependency of Scanner on ResultNotifier
  • Instead of the hardcoded reference to globalResultNotifier, give Scanner’s constructor a parameter and private field of type ResultNotifier
  • Now we can write test instances of Scanner which use mock ResultNotifier instances

It’s all object-oriented

• We are taking the view that every procedural program is actually an object-oriented program
• It’s just that to start there’s only one class, for which there is one instance.
• So we can understand Encapsulate Global References as just the same as Sprout Class
  • Once we break out a few classes, we can apply all of the techniques that we’ve been studying

5.11 Dealing with giant classes and methods

The Bloat

• Or, what happens when a class gets too big
  • Too many API methods
  • Too many fields to understand the state of objects
  • Too many tasks that the class takes care of
  • Too much is hidden — there’s probably too many private methods as well
  • Too hard to test
• It’s the downside of encapsulation
  • Good to hide certain details
  • Good to restrict the API especially for stateful change
  • But when we encapsulate too much, the code inside can rot and fester
  • Too hard to understand the consequences of changes
  • Hard to enumerate (and thus test) all feature interactions
• We’ve already talked about changing big classes
  • Sprout Class pushes new functionality into a new class
  * Sprout Method does at least name the new responsibility so the further bloating is clear
  • But sometimes we must (or should, or just can) address the root cause

The Single-Responsibility Principle

• Every class should have a single responsibility, a single purpose in the system
• Of course the idea of responsibility is nebulous
  • Does that mean that each class should have only one method?
  • It’s more about the main purpose
### Example

<table>
<thead>
<tr>
<th>RuleParser</th>
</tr>
</thead>
<tbody>
<tr>
<td>String current</td>
</tr>
<tr>
<td>HashMap&lt;String, Integer&gt; variables</td>
</tr>
<tr>
<td>int currentPosition</td>
</tr>
</tbody>
</table>

- public int evaluate(String) |
- private int branchingExpression(Node left, Node right) |
- private int causalExpression(Node left, Node right) |
- private int variableExpression(Node node) |
- private int valueExpression(Node node) |
- private String nextTerm() |
- private boolean hasMoreTerms() |
- public void addVariable(String name, int value) |

- Evaluates strings containing rule expressions in some language
- What is its main responsibility?
  - Clear from the name: it *parses*
  - No wait, it’s clear from the first public method: it *evaluates*
  - Which is it?
  - And it seems to manage variables too
  - Maybe it has several responsibilities?

### Seven guidelines for teasing out class responsibilities

1. **Group the methods**
2. **Look at private methods**
   - We’ve discussed testing private methods before
     - If you need to test it, it shouldn’t be private
     - If making it public bothers you, it may be part of a separate responsibility, and should be on a different class
3. **Break up methods**
   - Long methods may operate at multiple layers of abstraction
     - Use of a particular library API
     - Hardcoded database references
     - GUI operations
   - We can extract these operations into separate methods
     - In the short run, more methods
     - May reveal additional/better method groupings
4. **Look for internal relationships**
   - It is rarely the case that each of a class’s methods use all of the class’s fields
   - Make a quick sketch
     - A graph of methods and fields, with links between a method and each field it uses
     - Look for clusters, and consider moving these methods and fields into their own class
5. Look for the primary responsibility
   • Can you pick one of the class’s tasks above others you might identify?
   • The other tasks become candidates for extraction
   • Interface segregation
     – Is the class responsible to different client classes for different sets of methods?
     – We can describe the class methods in several different interfaces
     – Client classes can be aware of a single interface, rather than the entire implementing class

6. In a pinch, try scratch refactoring

7. Focus on the current work

**In the RuleParser**

• An easy grouping

```
<table>
<thead>
<tr>
<th>Method Grouping</th>
</tr>
</thead>
<tbody>
<tr>
<td>evaluate</td>
</tr>
<tr>
<td>branchingExpression</td>
</tr>
<tr>
<td>causalExpression</td>
</tr>
<tr>
<td>variableExpression</td>
</tr>
<tr>
<td>valueExpression</td>
</tr>
</tbody>
</table>
```

• So in a full redesign this design would make sense

```
RuleEvaluator
   public int evaluate(String)
   public void addVariable(String, int)

SymbolTable
   public void addVariable(String, int)

Expression
   public int evaluateWith(SymbolTable)
```

```
RuleParser
   public Expression parse(String)

TermTokenizer
   public String nextTerm()
   public boolean hasMoreTerms()
```

• Or, maybe we decide to separate only some of the method groups

• Method grouping is a good team exercise as well, capturing the whole team’s knowledge about a class

**Interface segregation example**
• Can be a precursor to separating the class

Monster methods
Two extremes of large methods

• The bulleted method
  – Little indentation, indicating few nested control structures

• The snarled method
  – Very deep indentation, indicating many nested structures

Most are a little bit of each. In either case, tool support is important

• We need to move code around, extracting methods and maybe classes

• Much less error-prone

• With a trusted refactoring tool, we don’t need to back the code extraction with tests
  – Or at least, not as many

• Separate use of automated refactoring from manual refactoring

• Goals of extracting code from a monster method
  1. Separate logic from awkward dependencies
  2. Introduce seams to support tests which in turn support further refactoring

The start of a long method

```java
public class CommoditySelectionPanel {
    public void update() {
        if (commodities.size() > 0
            && commodities.getSource().equals("local")) {
            listbox.clear();
            for(Commodity commodity : commodities) {
```
if (commodity.isTwilight() && !commodity.match(broker)) {
    listbox.add(commodity.getView());
}
}

// much, much more
}
// ...

• It’s a panel, suggestive of display, but also doing filtering

With tool support

• Name high-level functionality

• And at the same time prepare dependencies to be broken

public class CommoditySelectionPanel {
    public void update() {
        if (commoditiesReadyForUpdate()) {
            listbox.clear();
            for(Commodity commodity : commodities) {
                if (commodity.isTwilight() && !commodity.match(broker)) {
                    listbox.add(commodity.getView());
                }
            }
            // much, much more
        }
    }

    boolean commoditiesReadyForUpdate() {
        return commodities.size() > 0 && commodities.getSource().equals("local");
    }

    // ...
}

With tool support

public class CommoditySelectionPanel {
    public void update() {
        if (commoditiesReadyForUpdate()) {
            clearDisplay();
            for(Commodity commodity : commodities) {
                if (commodity.isTwilight() && !commodity.match(broker)) {
                    listbox.add(commodity.getView());
                }
            }
        }
    }
}
// much, much more
}

boolean commoditiesReadyForUpdate() {
    return commodities.size() > 0 && commodities.getSource().equals("local");
}

void clearDisplay() {
    listbox.clear();
}

// ...
}

With tool support

public class CommoditySelectionPanel {
    public void update() {
        if (commoditiesReadyForUpdate()) {
            clearDisplay();
            for(Commodity commodity : commodities) {
                if (isSingleBroker(commodity)) {
                    listbox.add(commodity.getView());
                }
            }
        }
        // much, much more
    }

    boolean commoditiesReadyForUpdate() {
        return commodities.size() > 0 && commodities.getSource().equals("local");
    }

    void clearDisplay() {
        listbox.clear();
    }

    boolean isSingleBroker(Commodity commodity) {
        return commodity.isTwilight() && !commodity.match(broker);
    }

    // ...
}

With tool support

public class CommoditySelectionPanel {
    public void update() {
        if (commoditiesReadyForUpdate()) {
            clearDisplay();
            for(Commodity commodity : commodities) {
                if (isSingleBroker(commodity)) {
                    displayCommodity();
                }
            }
        }
        // much, much more
    }

    boolean commoditiesReadyForUpdate() {
        return commodities.size() > 0 && commodities.getSource().equals("local");
    }

    void clearDisplay() {
        listbox.clear();
    }

    boolean isSingleBroker(Commodity commodity) {
        return commodity.isTwilight() && !commodity.match(broker);
    }

    // ...
}
boolean commoditiesReadyForUpdate() {
    return commodities.size() > 0 && commodities.getSource().equals("local");
}

void clearDisplay() {
    listbox.clear();
}

boolean isSingleBroker(Commodity commodity) {
    return commodity.isTwilight() && !commodity.match(broker);
}

void displayCommodity() {
    listbox.add(commodity.getView());
}

// ...
boolean isSingleBroker(Commodity commodity) {
    return commodity.isTwilight() && !commodity.match(broker);
}

void displayCommodity() {
    listbox.add(commodity.getView());
}

// ...

So what now?

• The code isn’t really that different now
  – Same structure, just calling methods

• But we’ve introduced seams against dependencies
  – Subclass and Override Method to sense through displayCommodity and clearDisplay

• There’s still filtering in a display class
  – But we’d need to separate the methods anyway to extract them into a better class design

• The automated tool makes coarse work safe

Without tool support

Errors to watch for when refactoring without tool support

1. Breaking the connection between variables
   • Forget to pass a variable to an extracted method
   • Declare it local instead of adding it as a parameter

2. Extract a method to a name that accidentally hides or overrides another method

3. Pass the wrong parameters, or return a wrong value
   • Or get the type wrong

Some precautions we can take

• Introduce sensing variables

• Start small
  – Small in the number of lines of code
  – A small coupling count
    * The number of variables we must pass in or out of the method

Extracting smaller methods first can help reveal structure in the monster

  – In bulleted methods, this often means extracting users of local variables
  – Might be many lines, but with 0-counts

• Can we prioritize the main logic of the method?
  – Are there secondary features we extract with lighter testing?
- Calculated risk

- *Break Out Method Object*
  - Local variables become instance variables
  - Must easier to test

**Some techniques**

1. Skeletonize methods — especially snarled methods
   - Extract the substructures of a control structure
   - Reveal the logic behind the various decisions and outcomes
   - As we did with CommoditySelectionPanel — extract the branches of a conditional to clarify the higher-level purpose of the conditional

2. Find sequences — especially bulleted methods
   - Extract control structures
   - Reveal the sequence of operations which the lengthy structures obscure
   - As we did with CommoditySelectionPanel — reveal the sequence of steps when ready for an update

*Wait* — those are contradictory
   - Appropriate to different goals
   - We’ve seen both already

3. Extract to the current class first
   - Less error-prone
   - Worry about the monster method first, single responsibility after

4. Extract small piece first

5. Be prepared to redo extractions
   - Rely on the repository to restore milestone states
   - It’s OK if you don’t immediately see the best structure!

6 **The test plan**

**Elements of a typical test plan**

1. Specifying exactly what artifacts we are testing
   - The programs, systems, hardware

2. Goals and rationals
   - Rationals refer to the specification and analysis of risks
   - Goals should be measurable

3. Scope and limitations
   - Distinguish what we will and will not test
   - Identify things which are expensive to test but post a low risk on failure
4. Subject matter experts (SMEs)
   • For consultation on the business relevance of risks, establishment of criteria, interpretation of results
   • Could be from an outside source for client needs, inside source for sales focus, etc.

5. Development experts
   • For languages, databases, hardware specs, 3rd-party software
   • Used in project but maybe not familiar to test team

6. Test data

7. Test environments
   • Must exactly reflect the target platform
   • Reproducibility, both for accountability and for re-checking after corrections

8. Test strategy
   • When we do each sort of test

9. Details for each development phase

10. Overall testing schedule
    • Integration of the test strategy into the overall development schedule

**Details for each development phase**

What’s a development phase?
   • In general, phases include: preliminary investigation, preliminary construction, final construction, delivery and installation, patches and new versions

What details do we need for each phase?
   • Conditions for starting and finishing
   • Test cases
   • Test case writing schedule
     – Developing basic test case description into statements in terms of the specification
   • Test case execution schedule
     – Awareness of dependencies among tests, reuse of virtual environments, etc. for an optimal schedule
   • Analysis and reporting schedule
     – Later adjustment may be needed if (when) testing uncovers surprises!
Test cases
The case writers detail:

• A title and short description — for human readers
• The phase where this case is relevance
• The goals and how we measure them, for this case
• Proposed data, tools, and procedure
  – For setting up, cleaning up, and rerunning

As testing proceeds we record:

• Date and location of each run of this test
• Recordable results from each step of test execution
  – Including the defects each run finds
  – And a summary: success or failure?

Writing the plan and cases

• Usually, we need to plan testing alongside the development of the requirements, spec, etc.
• The typical proceeding of effort usually follows the bathtub diagram:

IEEE Standard for Software Test Documentation

7 Test strategies

Recall: Four testing strategies

1. Static testing
   - Documents
   - Static code analysis
2. Black-box testing
   - Tests which run code, but with reference only to its APIs
3. White-box testing
   - Letting the structure of our code inform our test design
4. Performance testing
   - Real time, CPU time, memory, other machine resources

7.1 Static testing

IEEE Standard for Software Test Documentation


Static testing
- Of documents
  - The only form of testing for documentation
- Of code

What documents can we test?
- Project manager
  - Requirements, project planning documents
- Software designer and developer
  - Use cases, designs and specifications, many others
  - Code
- Testers’ documentation
- Administrator’s and users’ guides
  - Installation, operation, maintenance
  - Manuals, training material, online help text
Testing documentation

- Cosmetic vs. content inspection
  - Fix spelling, grammar and other such minor issue before the real static test
  - These errors are distracting!
  - Static testers could end up proofreading instead of focusing on content

- Modes of static testing
  - Desk checking
  - Inspections
  - Walk-throughs

Checking code as a document

- National Weather Service checklist
- Yeary’s Java checklist
- IEEE Standard for Software Reviews and Audits

Static analysis of code

- Static as opposed to dynamic
- Automatically checking code as a document
- Like compiler warnings
- Can be very sophisticated — checking for possible null pointer references
- Can be very simple — looking for variables in the same scope whose names are too similarly spelled
- Can check for required code documentation as well
- We’ll look at one particular tool FindBugs next week
  - A future homework will involve FindBugs, so if you use your own machine, think about installing it

7.2 Other approaches to functional testing

QuickCheck

- Another tool for functional testing without tests
  - [https://www.youtube.com/watch?v=AfaNEebCDos](https://www.youtube.com/watch?v=AfaNEebCDos)

7.3 Structural testing

Structural testing

- Refers to tests on the software platform which are not the system under development
- Operating system, network drivers, security systems, data management
- Why test these things?
  - Verify designers’ understanding and expectations
  - Verify platform implemented according to its specification
  - Reduce opportunities for failures after delivery
- White-box techniques are typically strictly excluded
  - Open-source systems may be an exception
**Interface testing**
Designing tests according to the published API of the system services

1. Tests that intercept the call to an external service
   - Check that the passed information matches our understanding of the API
2. Re-activate the connection to the external service
   - Examine the results returned and callbacks from the service
3. Test the program’s use of these results by "stubbing" the return values
   - Isolates our system’s response from any possible errors in transmission
4. Send requests from the test system rather than the application under test
   - But make the application handle responses, and verify that they are correct

**Security testing**
Beyond the scope of this class
- (But there will be an elective offered in the fall)
- Use of protocols
- Implementation of protocols

**Installation testing**
- We’d described user manuals as something we can test statically
- But it’s also important to test in practice
- Set up hardware and system exactly as the intended production platform
  - And possibly there are several variations
- Follow published installation instructions
  - Which may now mean simply running a wizard
  - Follow all paths of options and configuration choices

**Smoke test**
- Given a correct *installation*, can the system be *configured*
- Often too many configurations to test all
  - Identify the most important configurations with respect to customer needs
- And following each configuration, run use cases sufficient to identify correct configuration

**Aside: another meaning of smoke test**
- In recent agile strategies of programming
- The system should always work
  - Even if it just starts and then shuts down (correctly) at first
- The smoke tests verify this basic ability to turn the system over
Aside: one more meaning of smoke test

• In a large system, the full test suite will invariably take a long time to run — hours, perhaps days
• The build verification test (BVT) aka smoke test is a small subset of the full system
  – Should run in a small number of minutes
  – Should test the most important cases and a broad variety of software functions
• Intended to diagnose whether the software runs at all
  – Expect major bugs to fail the smoke test

Administration testing

• Like use case testing, but focused on tasks that require internal administrator privilege
  – Adding users
  – Managing security settings
  – Building master files and lists
• Often possible earlier in the projects
  – These features often completed earlier

Backup and restore testing

• How does the application interact with backup systems? What can be backed up?
• Restoration process
  – Assembling synchronized backups if separate
  – Blame identification
• To test, must trigger abnormal exit from program
  – Then restart, and compare restored state to desired state

8 Testing interactive systems

8.1 Web site and services testing

Testing web applications

The web is a distinctive programming and testing environment
• Deployed, distributed and remote
• Accessible globally, so timezones and languages can be much more significant
• But not globally accessible without consideration for disabilities
• Exceptionally competitive market
  – High user expectations
  – Often very low cost to moving to a competing site
• Not just client-server, but typically many servers
• HTTP protocol is stateless
  – But state preserved in cookies, session objects, server databases
  – Implemented by multiple technologies
  – Not always clear how to test the interaction of the different technologies
Surveying the testing space

- Testing static hypertext web pages
- Testing dynamic web applications
- Testing web services
- A test case for any of these is a series of interactions between clients and servers

Testing static hypertext web pages

- Unvarying and the same to all users
- Usually, but not always, stored as an HTML file
- Testing looks for flaws in the web site structure
  - Dead links
  - Inconsistent navigation
  - Dead-ends
- Lots of tools for this
  - I’ll schedule demos next week, for after spring break

Testing dynamic web applications

- Dynamic web pages created by a program on demand
- Contents and structure may be determined by previous interactions with
  - The particular user
  - Web server state
  - User location
  - User browser
  - Time of day
- A web application is a full application deployed across the web
- The tester usually does not have access to at least some of the server-side software and data
  - Client-side vs. server-side testing

Client-side testing

- Need a way to generate data for various web forms/fields at each stage of testing

Test strategies:

1. Explore action sequences starting from some URL
2. Use user session data
   - Typically stored in server logs
3. Bypass testing
   - Most web pages with forms validate their inputs before actually sending them to the server
   - Bypass this check by script to send noncomplying data to test reaction
Server-side testing and web applications

- Much harder to reason about control flow
  - "Back" and "refresh" buttons
  - Hand-editing URLs
  - Users can modify hidden form fields, edit or delete cookies
  - Server behavior can be modified by code loaded through normal web operations (J2SE, .NET) — components not available until after system is deployed
- Traditional control-flow and coverage analyses do not suffice
  - The possible paths cannot be known statically
- These techniques are still in research

Testing web services

- For example, XML/SOAP
- In general, a distributed, modular application whose components communicate structured-format data
- Many of the same difficulties as web applications
  - Services from multiple servers
  - Dynamic discovery of new services
- Testing efforts so far focus on the messages
  - Making sure that XML schemas are respected

Server-side testing and web applications

- Much harder to reason about control flow
  - "Back" and "refresh" buttons
  - Hand-editing URLs
  - Users can modify hidden form fields, edit or delete cookies
  - Server behavior can be modified by code loaded through normal web operations (J2SE, .NET) — components not available until after system is deployed
- Traditional control-flow and coverage analyses do not suffice
  - The possible paths cannot be known statically
- These techniques are still in research

Testing web services

- For example, XML/SOAP
- In general, a distributed, modular application whose components communicate structured-format data
- Many of the same difficulties as web applications
  - Services from multiple servers
  - Dynamic discovery of new services
- Testing efforts so far focus on the messages
  - Making sure that XML schemas are respected
8.2 Testing GUIs

Testing web applications
The web is a distinctive programming and testing environment

- Deployed, distributed and remote
- Accessible globally, so timezones and languages can be much more significant
- But not globally accessible without consideration for disabilities
- Exceptionally competitive market
  - High user expectations
  - Often very low cost to moving to a competing site
- Not just client-server, but typically many servers
- HTTP protocol is *stateless*
  - But state preserved in cookies, session objects, server databases
  - Implemented by multiple technologies
  - Not always clear how to test the interaction of the different technologies

Surveying the testing space

- Testing static hypertext web pages
- Testing dynamic web applications
- Testing web services
- A test case for any of these is a series of interactions between clients and servers

Testing static hypertext web pages

- Unvarying and the same to all users
- Usually, but not always, stored as an HTML file
- Testing looks for flaws in the web site structure
  - Dead links
  - Inconsistent navigation
  - Dead-ends
- Lots of tools for this
  - I’ll schedule demos next week, for after spring break
Testing dynamic web applications

- Dynamic web pages created by a program on demand
- Contents and structure may be determined by previous interactions with
  - The particular user
  - Web server state
  - User location
  - User browser
  - Time of day
- A web application is a full application deployed across the web
- The tester usually does not have access to at least some of the server-side software and data
  - Client-side vs. server-side testing

Client-side testing

- Need a way to generate data for various web forms/fields at each stage of testing

Test strategies:

1. Explore action sequences starting from some URL
2. Use user session data
   - Typically stored in server logs
3. Bypass testing
   - Most web pages with forms validate their inputs before actually sending them to the server
   - Bypass this check by script to send noncomplying data to test reaction

Server-side testing and web applications

- Much harder to reason about control flow
  - "Back" and "refresh" buttons
  - Hand-editing URLs
  - Users can modify hidden form fields, edit or delete cookies
  - Server behavior can be modified by code loaded through normal web operations (J2SE, .NET) — components not available until after system is deployed
- Traditional control-flow and coverage analyses do not suffice
  - The possible paths cannot be known statically
- These techniques are still in research
Testing web services

- For example, XML/SOAP
- In general, a distributed, modular application whose components communicate structured-format data
- Many of the same difficulties as web applications
  - Services from multiple servers
  - Dynamic discovery of new services
- Testing efforts so far focus on the messages
  - Making sure that XML schemas are respected

Server-side testing and web applications

- Much harder to reason about control flow
  - "Back" and "refresh" buttons
  - Hand-editing URLs
  - Users can modify hidden form fields, edit or delete cookies
  - Server behavior can be modified by code loaded through normal web operations (J2SE, .NET) — components not available until after system is deployed
- Traditional control-flow and coverage analyses do not suffice
  - The possible paths cannot be known statically
- These techniques are still in research

Testing web services

- For example, XML/SOAP
- In general, a distributed, modular application whose components communicate structured-format data
- Many of the same difficulties as web applications
  - Services from multiple servers
  - Dynamic discovery of new services
- Testing efforts so far focus on the messages
  - Making sure that XML schemas are respected

9 Testing real-time systems

Real-time systems

- Must respond to externally-generated stimuli in a specific finite period
- Sometimes called reactive
  - Recognize changes via sensors
  - Respond by influencing the environment via actuators
- Often embedded into a larger engineering system
- Examples
  - Controller for a robot arm on a conveyor belt, getting messages from other robots and sensors on the belt
  - A flight monitoring system that expects responses from aircraft systems within 30 seconds of e.g. a landing authorization
Real-time vocabulary

- **Deadline** — just what it sounds like
- **Timeliness** — the ability of software to meet time constraints
- **Real-time application** — set of tasks implementing a functionality
  - *Periodic* tasks — activated at a fixed frequency
    * So all the times when tasks are activated are known
  - *Aperiodic* tasks — can be activated at any point in time
    * To achieve timeliness, must be specified with constraints on their activation pattern — *sporadic*
    * For example: a minimum interval time between sporadic tasks
- **Response time** — time from activation to completion of execution
  - *Execution order* of tasks can effect response time

Real-time systems and testing

- **Observability** — ability to monitor of log what a real-time system does
  - Usually increased via *probes* in the system internals
  - *Probe-effect:* removing the probes can change the system timing and invalidate tests
    * Common solution: leave them in, but direct output to something that takes the same amount resources
  - Storage constraints may make logging difficult in the deployed system

Remember the R from SPRAE?

- Repeatability
- Often called reproducibility in real-time testing
- Event-triggering and dynamic scheduling make reproducibility very hard to achieve in real-time systems
- Response time will depend on factor beyond just software
- Repeatable testing requires a high effort of control

Goal of testing

- Find the longest execution time (aka worst-case)
- Very difficult to estimate
- So testers try to make the software run for as long as possible
- **Timeliness fault** — a mistake in implementation that causes incorrect temporal behavior
  - For example, a bug in a conditional may cause a loop to execute more times than planned
  - Two tasks interfering
  - Environment (sensors, activators) behave differently than expected
- Two kinds of errors
  - *Timeliness fault* — failure to meet the software’s *internal* requirements
    * Usually hard to detect, requires access to logs and knowledge of system
  - *Timeliness failure* — a violation of a time constraint that can be observed externally
Real-time testing
Some different approaches, mostly depending on formal models of the software system

- Timed Petri nets, constraint graphs, clock region graphs, temporal logic, timed automata
- Less formal techniques: genetic algorithms
- Statically generate execution orders