Credit Where Credit is Due

- Significant sections of this lecture are derived from “Pragmatic Unit Testing.”
- Andrew Hunt and David Thomas
- An excellent, practical book. You should buy it.
- Available in Java and .NET flavors.
Today’s Lecture:

- Unit Tests
  - Testing classes and methods against a contract
- Unit testing is good for YOU!
- Testing Harnesses*
  - Making tests automatic, repeatable and independent
- Mock Objects*
  - Testing one piece of code at a time

*Demo included!

Unit Tests

- Do you spend a large amount of time using the debugger?
- Do you ever find yourself saying things like,
  - “That’s impossible!”
  - “I don’t understand how this could happen.”
- Unit tests be a big help.
Unit Tests: Definitions

- Unit tests are **whitebox** tests written by developers, and designed to **verify** small **units** of program functionality.

- **Key Metaphor: I.C. Testing**
  - Integrated Circuits are tested individually for functionality before the whole circuit is tested.
Unit Tests: Definitions

- Unit tests are **whitebox** tests written by developers, and designed to **verify** small units of program functionality.

- **Whitebox** – Unit tests are written with full knowledge of implementation details.

- **Developers** – Unit tests are written by you, the developer, concurrently with implementation.
Unit Tests: Definitions

- Unit tests are whitebox tests written by developers, and designed to verify small units of program functionality.

- Small Units – Unit tests should isolate one piece of software at a time.
  - Individual methods and classes

- Verify – Make sure you built ‘the software right.’ Testing against the contract.
  - Contrast this with validation.
Testing Against a Contract

- A method’s contract is a statement of the responsibilities of that method, and the responsibilities of the code that calls it.

- Think, a legal contract
  - If you pay me exactly $30,000
  - I will build a new room on your house

- Helps to pinpoint responsibility.

More on Contracts

- Methods and objects all have contracts!
  - Sometimes they are explicit
  - Sometimes implicit

- Let’s see some examples...
Implicit Contracts

- Sometimes the contract exists implicitly in the code and the mind of the programmer.

```java
public boolean isThisALeapYear(Calendar today)
{
    return (today.get(Calendar.YEAR) % 4 == 0);
}
```

Informal Contracts

- Sometimes a method’s contract is informally described in comments.
Informal Contracts

/** Applies a move to a board. This assumes that the move is one that was returned by getAllMoves. Upon applying the move, it will also update the value of the board and switch the board's turn. */

public void applyMove(Move mv) {
    byte row = 0, col = 0, bck = 0, ...;
    byte opTurn = (mTurn == BLK) ? WHT : BLK;
    OthelloMove appM = null;
    boolean good = false;
}

Pre/Post Conditions, Invariants

• You may remember these from early computer science classes.
  • And you may never use them!

• **Precondition**
  • Things that must be true of parameters and fields for call to be ‘legal.’

• **Postcondition**
  • Things this method guarantees will be true of fields and the return value after being called.

• **Invariants**
  • Something that will always be true.
  • Usually describe objects and fields.
Pre/Post Conditions, Invariants

```java
public class BankingExample {

    public static final int MAX_BALANCE = 1000;
    //Invariant: The balance will always be greater than
    //zero, but less than MAX_BALANCE.
    private int balance;

    //Precondition: amount is greater than zero
    //Postcondition: the new balance is set to the
    //old balance plus amount.
    public void credit(int amount) { ... }

    //Precondition: amount is greater than zero
    //Postcondition: balance set to the old balance
    //minus amount
    public void debit(int amount) { ... }
}
```

Machine-Readable

```java
public class BankingExample {

    public static final int MAX_BALANCE = 1000;
   //@ invariant balance >= 0 && balance <=MAX_BALANCE;
    private int balance;

    //@ requires amount > 0;
    //@ ensures balance = \old(balance) + amount;
    public void credit(int amount) { ... }

    //@ requires amount > 0;
    //@ ensures balance = \old(balance) - amount;
    public void debit(int amount) { ... }
}
```
Today’s Lecture:

• Unit Tests
  • Testing against a contract
• Unit testing is good for YOU!
• Testing Harnesses*
  • Making tests automatic, repeatable and independent
• Mock Objects*
  • Testing one piece of code at a time

*Demo included!

Unit testing is good for YOU!

• Unit testing
  • Seems like a good idea, in theory.
  • Often, people just don’t do it.
  • Let’s look at some common excuses why developers often don’t.
Writing Unit Tests Takes Too Long!

- Unit testing implies a pay-as-you-go model, rather than pay-at-the-end.
- But there’s more
  - Unit testing implies linear work, rather than exponential.
  - Think of clearing a field
    - Regular mowing, versus
    - Bushwhacking

Linear vs. Exponential Work

- Unit testing implies:
  - Steady productivity throughout the development cycle.
- Without unit testing:
  - Productivity starts off higher, but dives at the end when the testing starts.
  - Relearn code you wrote weeks or months ago.
Questions Worth Asking

- How much time do you spend debugging code you or others have written?
- How much time do you spend reworking code that you thought was working but turned out to have major bugs?
- How much time do you spend isolating a bug to its source?

- Often, this time add up fast.
- Unit testing can help reduce it.

It’s Not My Job to Test!

- If you’re worried about taking your testers’ jobs, don’t!
  - They have plenty to worry about with integration, acceptance tests, etc.

- As programmers, our job is to create working code.
  - Until you write a unit test, you have no idea.
They Aren’t in the Process!

- Often developers say things like,
  - “Our company runs different types of tests.”
  - “Our test machine isn’t set up for unit tests.”
  - “We have a different process.”

Unit Tests are Personal

- Unit tests test the code you write.
- They are meant to be run on a developer’s workstation.
  - If they are not part of source control, no problem!
  - If no one else on your team uses them, no problem!
Unit Tests are Personal

- Think of unit testing the same way you think of your text editor.
  - “I use Notepad, he uses Emacs.”
  - The main difference being, the relative quality of your code.

- Of course, there are benefits to a culture of unit testing.
  - Automated regression tests & source control
  - Easier Integration
  - But it isn’t necessary to reap the benefits

The Take-Away Message

- Unit tests are a tool, just like an IDE, that help you, the individual developer, write better code.
Today’s Lecture:

- Unit Tests
  - Testing against a contract
  - Unit testing is good for YOU!
- Testing Harnesses*
  - Making tests automatic, repeatable and independent
- Mock Objects*
  - Testing one piece of code at a time

*Demo included!

Testing Harnesses

- Testing harnesses are tools that help manage and run your unit tests.
- Help us to achieve three properties of good unit tests, which are:
  - Automatic
  - Repeatable
  - Independent
Meaning...

- **Automatic**
  - With one touch, our tests should be run and checked for completion. We want a fuzzy feeling with as little work as possible.
- **Repeatable**
  - Any developer can run the tests and they will work right away.
- **Independent**
  - Your tests can be run in any order and they will still work.

JUnit: A Java Unit Testing Harness

- Provides one-touch functionality for running all of your tests.
- Easy to verify success or failure.
- Source of failure is immediately obvious.
JUnit is Also a Testing Framework

- We write tests using code included in the JUnit framework.
  - @Test annotation tells the harness that you have written a test.
  - org.junit.Assert is full of helpful assertion tools.

JUnit Demo Time

- Testing the ShoppingCart
Other Helpful JUnit Features

• @BeforeClass
  • Run once before all test methods in class.

• @AfterClass
  • Run once after all test methods in class.

• Together, these methods are used for setting up computationally expensive test elements.
  • E.g., database, file on disk, network...

Other Helpful JUnit Features

• @Before
  • Run before each test method.

• @After
  • Run after each test method.

• Make tests independent by setting and resetting your testing environment.
  • E.g., creating a fresh object
for each class:
    setUpBeforeClass();

for each test:
    setUp();
    run test;
    tearDown();

tearDownAfterClass();

---

Helpful JUnit Assert Statements

- **assertEquals**(float expected, float actual, float delta)
  - Used for so that floating point equality is unnecessary.

- **assertSame**(Object expected, Object actual)
  - Tests for two objects are the same in memory.

- **assertNull**(java.lang.Object object)
  - Asserts that a reference is null.

- **assertNotNull**(String message, Object object)
  - Many ‘not’ asserts exists.
  - Most asserts have an optional message that can be printed.
Today’s Lecture:

- Unit Tests
  - Testing against a contract
- Unit testing is good for YOU!
- Testing Harnesses*
  - Making tests automatic, repeatable and independent
- Mock Objects*
  - Testing one piece of code at a time

*Demo included!

Unit Testing and Isolation

- Unit testing is all about isolating bugs.
  - When a unit test fails, we should know almost exactly in the source code where the bug lies.
- Mock objects to the rescue!
  - Allow us this isolation.
Unit Testing and Speed

• Running our tests should be fast...
  • If they aren’t people won’t run them.
• But what about bringing up and down environment code?
  • E.g., network sockets, databases, date-related code

• Mock objects to the rescue!
  • We make our own, simplified versions.

Unit Testing and Unusual Situations

• We want to test our code in weird situations.
  • E.g., daylight-saving time, network outages, file permission errors
• We can’t force a network outage.
  • At least, not in a repeatable way...

• Mock objects to the rescue!
  • We define the behavior.
Additional Benefit: Protocol Checks

• We want to make sure our code uses other code correctly.
  • E.g., network sockets are open before they are read.

• Mock objects to the rescue!
  • Protocol conformance can be verified.

How EasyMock Works
How EasyMock Works

getSize() returns 8

getParent() returns null
How EasyMock Works

getSize() returns 8

getParent() returns null
EasyMock Demo Time

- Exceptional Conditions (NTP)
- Interacting Code (AST)
- Protocol Conformance (Iterator)

- (easymock.org, for more!)

Other Neat Features

- EasyMock has a ton of features.
  - Stub behavior
    - When you don’t really care if or when a method is called.
  - Nice mocks
    - Return defaults instead of throwing exceptions.
  - Check calling order between several mocks
  - Mock Reset
  - Argument Matchers
  - Different behavior for same call
  - Intricate return behavior
Take-Away Points

- Unit tests are tests by and for programmers.
  - Think of them as a tool, like an IDE.
- Testing harnesses and mock objects make the hard parts easier.
  - Automatic, repeatable, independent
- Unit test generation is a viable option.
  - Helps to achieve high code coverage.
  - Be careful about code intent versus implementation.

The end

Slides and source code available online.