Unit Testing and TDD


Slides material sources:
SWEBOK v3: IEEE Software Engineering Body of Knowledge
Working Effectively with Legacy Code, M. Feathers
Software Testing – A Craftsman’s Approach, P Jorgensen

What is software testing?
Software testing

*Testing* is the act of exercising software with *test cases*.

A test has two distinct goals:

- To **find failures** (*verification* aspect).
- To **demonstrate correct execution** (*validation* aspect).

**What is a test case?**
The essence of software testing is to determine a set of test cases for the item to be tested.

A test case is (or should be) a recognized work product.

A complete test case will contain a test case identifier, a brief statement of purpose, a description of preconditions, the actual test case inputs, the expected outputs, a description of expected post-conditions, and an execution history.

The execution history is primarily for test management use—it may contain the date when the test was run, the person who ran it, the version on which it was run, and the pass/fail result.

What is unit testing?
Unit testing

Common to most conceptions of unit tests is the idea that they are tests in isolation of individual components of software.

What are components?
- In unit testing, we are usually concerned with the most atomic behavioral units of a system.
- In procedural code, the units are often functions.
- In object oriented code, the units are classes.

Test harness is a generic term for the testing code that we write to exercise some piece of software and the code that is needed to run it.

Unit testing

Testing in isolation is an important part of the definition of a unit test, but why is it important?
- Error localization
  - As tests get further from what they test, it is harder to determine what a test failure means.
  - Often it takes considerable work to pinpoint the source of a test failure.
- Execution time
  - Larger tests tend to take longer to execute.
  - This tends to make test runs rather frustrating.
  - Tests that take too long to run end up not being run.
Unit testing

Here are qualities of good unit tests:
- They run fast.
  - If they don’t run fast, they aren’t unit tests.
- They help us localize problems.

A test is not a unit test if during execution the tested software:
- talks to a database.
- communicates across a network.
- You have to do special things to your environment (such as editing configuration files) to run it.

A unit test that takes 1/10th of a second to run is a slow unit test!!!

Unit testing

To put unit tests in place, we often have to change code!!

Why??

Dependency is one of the most critical problems in software development.

Often, classes depend directly on things that are hard to use in a test, they are hard to modify and hard to access.
Unit testing

When we want to get tests in place, there are two reasons to break dependencies:

- **Separation** - We break dependencies to separate from “other code” when we can’t get this other code into a test harness to run.

- **Sensing** - We break dependencies to some “other code” when that code is the only place we can sense the effects of our actions but we can’t easily access the state of the other code.
Unit testing

Faking collaborators

**Breaking dependencies on other code** is hardly ever that simple.

A useful technique is to substitute the “other code” upon which we depend with some “fake” code that is easy to instantiate and access.

A fake object is an object that impersonates some collaborator of a class when it is being tested.

- In object orientation, these other pieces of code are often called **fake objects**.
- **Stub** is another way to call these elements
Faking collaborators

In a point-of-sale system, we have a class called Sale. Whenever scan() is called, the Sale object needs to display the name of the item that was scanned, along with its price on a cash register display.

How can we test this to see if the right text shows up on the display?

If the calls to the cash register’s display API are buried deep in the Sale class, it's going to be hard.

Faking collaborators

We can move all of the display code form Sale over to ArtR56Display and have a system that does exactly the same thing that it did before.

Does that get us anything?
Faking collaborators

The Sale class can now hold on to either an ArtR56Display or something else, a FakeDisplay.

The nice thing about having a fake display is that we can write tests against it to find out what the Sale does.
Faking collaborators

Fake objects can be confusing in a way...

One of the oddest things about them is that they have two sides.

A more advanced type of fake is called a mock object.

Mock objects are fakes that perform assertions internally

i.e., we tell them what calls to expect, and then we tell them to check and see if they received those calls.
How do we create test cases?

Ad-hoc testing

Perhaps the most widely practiced technique is ad hoc testing:

Tests are derived relying on the software engineer's skill, intuition, and experience with similar programs.

Ad hoc testing can be useful for identifying test cases that not easily generated by more formalized techniques.
Input domain based techniques

Two widely known categories are:

- **Boundary value** testing techniques.
- **Equivalence class** testing techniques.

Code based techniques

Two widely known categories are:

- **Control flow** testing techniques.
- **Data flow** testing techniques.
Test-driven development

Test-driven development activities:

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

Test-driven development

Suppose we’re working on a financial application, and we need a class that is going to use some statistic mathematics to verify whether certain commodities should be traded.

The class must have a method that calculates something called the first statistical moment about a point.

We know the math, so we know that the answer should be -0.5 for the data series we code in the test.
Test-driven development

```java
public void testFirstMomen() {
    InstrumentCalculator calculator = new InstrumentCalculator();
    calculator.addElement(1.0);
    calculator.addElement(2.0);
    assertEquals(-0.5, calculator.firstMomentAbout(2.0), TOLERANCE);
}
```

Write a Failing Test Case

```java
public class InstrumentCalculator {
    public double firstMomentAbout(double point) {
        return Double.NaN;
    }
}
```

Get It to Compile

```java
public double firstMomentAbout(double point) {
    double numerator = 0.0;
    for (Iterator it = elements.iterator(); it.hasNext(); ) {
        double element = ((Double)it.next()).doubleValue();
        numerator += element - point;
    }
    return numerator / elements.size();
}
```

Make It Pass

**A additional feature:**
firstMomentAbout() must throw an exception when the elements list is empty

```java
public void testFirstMomen() {
    try {
        new InstrumentCalculator().firstMomentAbout(0.0);
        fail("expected InvalidBasisException");
    } catch (InvalidBasisException e) {
    }
}
```

Write a Failing Test Case

```java
public double firstMomentAbout(double point)
    throws InvalidBasisException {
    if (element.size() == 0)
        throw new InvalidBasisException("no elements");
    double numerator = 0.0;
    for (Iterator it = elements.iterator(); it.hasNext(); ) {
        double element = ((Double)it.next()).doubleValue();
        numerator += element - point;
    }
    return numerator / elements.size();
}
```

Get It to Compile

Make It Pass
Test-driven development

The class must have another method that calculates something called the second statistical moment about a point.

```java
public void testSecondMoment() throws Exception {
    InstrumentCalculator calculator = new InstrumentCalculator();
    calculator.addElement(1.0);
    calculator.addElement(2.0);
    assertEqual(0.5, calculator.secondMomentAbout(2.0), TOLERANCE);
}
```

Write a Failing Test Case

Get It to Compile

To make it compile, we can make a copy of the `firstMomentAbout` method and rename it so that it is now called `secondMomentAbout`.

```java
public double secondMomentAbout(double point) throws InvalidDataException {
    if (elements.size() == 0)
        throw new InvalidDataException("no elements");
    double numerator = 0.0;
    for (Iterator lt = elements.iterator(); lt.hasNext(); ) {
        double element = (Double)(lt.next());
        numerator += element - point;
    }
    return numerator / elements.size();
}
```
Test-driven development

```java
public double secondMomentAbout(double point)
    throws InvalidBasisException {
    if (elements.size() == 0)
        throw new InvalidBasisException("no elements");
    double numerator = 0.0;
    for (Iterator it = elements.iterator(); it.hasNext();)
        double element = (Double)it.next().doubleValue();
        numerator += Math.pow(element - point, 2.0);
    return numerator / elements.size();
}
```

Make It Pass

Test-driven development

```java
public double firstMomentAbout(double point)
    throws InvalidBasisException {
    return nthMomentAbout(point, 1.0);
}
public double secondMomentAbout(double point)
    throws InvalidBasisException {
    return nthMomentAbout(point, 2.0);
}

private double nthMomentAbout(double point, double n)
    throws InvalidBasisException {
    if (elements.size() == 0)
        throw new InvalidBasisException("no elements");
    double numerator = 0.0;
    for (Iterator it = elements.iterator(); it.hasNext();)
        double element = (Double)it.next().doubleValue();
        numerator += Math.pow(element - point, n);
    return numerator / elements.size();
}
```

Remove Duplication