Testing

Christopher Simpkins
chris.simpkins@gatech.edu

Unit Tests and Functional Tests

Unit tests are tests of individual system components

- usually automated (code that tests other code)
- written by developers
- can be white box or black box
- often tests internal design elements that don't map cleanly to user-facing features

Functional, or acceptance tests

- black box
- usually not automated - test script performed by QA or customer
- ideally written by customer, expressing the customer's requirements for a feature to be "done"

Most of our discussoin of testing will focus on unit testing.

Errors, Faults, Test Cases

Defects are not free, somebody gets paid for making them.
– Deming

- Failure: incorrect behavior of a component (a symptom)
- Fault, a.k.a. Bug or Defect: incorrect program or data object (results from error)
- Error: human mistake, by commision or omission

A *test case* is a single input to the system with an expected output.

- If actual output differs we have a failure
- A good test case causes a failure which helps us find the bug that cased it

White Box Tests

- We use the code to develop our tests
- Execute all lines of code by taking each branch (100% coverage)

Given:

```
public static int fac(int n) {
    if (n < 0) throw new IllegalArgumentException();
    else if (n == 0) return 1;
    else return n * fac(n - 1);
}
```

This test contains cases that cover each branch:

```
@Test
public void testFac() {
  try {
      fac(-1);
      fail("Negative argument didn't cause IllegalArgumentException.")
  } catch(IllegalArgumentException e) {}
  assertEquals("Failed for n == 0.", 1, fac(0));
  assertEquals("Failed for n > 0.", 120, fac(5));
}
```

Black Box Tests

Instead of looking at code, we test the specification (e.g., Javadoc)

- Boundary conditions (on and off boundary)
- Equivalence partitions (one case per)
- Plausible faults (specific values)

Ex: Write a unit test for `public static int abs(int n)` in `java.lang.Math`, which returns the absolute value of an integer `n`.

```
@Test
public void absPositivesNegatives() {
    assertEquals("Incorrect for positive numbers", 1, Math.abs(1));
    assertEquals("Incorrect for negative numbers", 1, Math.abs(-1));
}
```

We include an assert for each equivalence partition. Notice that this is similar to branch coverage in white box testing.

For detailed information on unit testing with JUnit, see http://junit.org/ and look at the test code in tomcat-todo (in `src/test/java` and in `build.xml` to see how to invoke unit tests with Ant).

Test-Driven Development

- **First Law** You may not write production code until you have written a failing unit test.
- **Second Law** You may not write more of a unit test than is sufficient to fail, and not compiling is failing.
- **Third Law** You may not write more production code than is sufficient to pass the currently failing test.

Consequence: tests and code are written together in an interleaved fashion.

Attitude: tests must be maintained to the same high standards as production code.

Clean Tests

```
public void testGetPageHieratchyAsXml() throws Exception {
  crawler.addPage(root, PathParser.parse("PageOne"));
  crawler.addPage(root, PathParser.parse("PageOne.ChildOne"));
  crawler.addPage(root, PathParser.parse("PageTwo"));
  request.setResource("root");
  request.addInput("type", "pages");
  Responder responder = new SerializedPageResponder();
  SimpleResponse response =
    (SimpleResponse) responder.makeResponse(
      new FitNesseContext(root), request);
  String xml = response.getContent();
  assertEquals("text/xml", response.getContentType());
  assertSubString("<name>PageOne</name>", xml);
  assertSubString("<name>PageTwo</name>", xml);
  assertSubString("<name>ChildOne</name>", xml);
}
...
```

Imagine several test methods written with this level of detail and duplication.

Domain-Specific Testing Language

The test methods of the previous slide follow a build-operate-check pattern. This pattern can be represented by a domain-specific testing API:

```
public void testGetPageHierarchyAsXml() throws Exception {
  makePages("PageOne", "PageOne.ChildOne", "PageTwo");
  submitRequest("root", "type:pages");
  assertResponseIsXML();
  assertResponseContains(
    "<name>PageOne</name>", "<name>PageTwo</name>",
    "<name>ChildOne</name>" );
}
```

- Each of the test methods use helper methods developed specifically for tests: makePages (build), submitRequest (operate), and asserts (check).
- Testing API helps us write clean tests: clear, simple, and densely expressed.

One Assert per Test

Some people one assert-per-test rule, as in this case:

```
public void testGetPageHierarchyAsXml() throws Exception {
  givenPages("PageOne", "PageOne.ChildOne", "PageTwo");
  whenRequestIsIssued("root", "type:pages");
  thenResponseShouldBeXML();
}

public void testGetPageHierarchyHasRightTags() throws Exception {
  givenPages("PageOne", "PageOne.ChildOne", "PageTwo");
  whenRequestIsIssued("root", "type:pages");
  thenResponseShouldContain(
    "<na  me>PageOne</name>", "<name>PageTwo</name>",
    "<name>ChildOne</name>"
  );
}
```

Applying the one-assert-per-test rule leads to less clean code. We prefer a one-concept-per-test rule.

One Concept per Test

```
public void testGetPageHierarchyAsXml() throws Exception {
  makePages("PageOne", "PageOne.ChildOne", "PageTwo");
  submitRequest("root", "type:pages");
  assertResponseIsXML(); assertResponseContains(
    "<name>PageOne</name>", "<name>PageTwo</name>",
    "<name>ChildOne</name>" );
}
```

The one-concept-per-test rule, as hown here, leads to cleaner code (less duplication, clearer intent).

F.I.R.S.T.

Five additional rules for clean tests: Test should be fast, indepedent, repeatable, self-validating, and timely.

- **Fast** Tests should be fast. They should run quickly. When tests run slow, you won't want to run them frequently. If you don't run them frequently, you won't find problems early enough to fix them easily. You won't feel as free to clean up the code. Eventually the code will begin to rot.
- **Independent** Tests should not depend on each other. One test should not set up the conditions for the next test. You should be able to run each test independently and run the tests in any order you like. When tests depend on each other, then the first one to fail causes a cascade of downstream failures, making diagnosis difficult and hiding downstream defects.

F.I.R.S.T

- **Repeatable** Tests should be repeatable in any environment. You should be able to run the tests in the production environment, in the QA environment, and on your laptop while riding home on the train without a network. If your tests aren't repeatable in any environment, then you'll always have an excuse for why they fail. You'll also find yourself unable to run the tests when the environment isn't available.
- **Self-Validating** The tests should have a boolean output. Either they pass or fail. You should not have to read through a log file to tell whether the tests pass. You should not have to manually compare two different text files to see whether the tests pass. If the tests aren't self-validating, then failure can become subjective and running the tests can require a long manual evaluation.

F.I.R.S.T.

And finally:

- **Timely** The tests need to be written in a timely fashion. Unit tests should be written just before the production code that makes them pass. If you write tests after the production code, then you may find the production code to be hard to test. You may decide that some production code is too hard to test. You may not design the production code to be testable.

JUnit embodies these rules. Use Junit.