“Every Thing You Always Wanted to Know About Unit-Testing*

*But Were Afraid to Ask”

“Unit-testing, what is it?”

```
import org.junit.*;
import org.junit.runner.JUnitCore;

public class Calc_Test {
  @Test public void testAdd() {
    Assert.assertEquals(9, Calc.add(4, 5));
  }

  @Test public void testSub() {
    Assert.assertEquals(-8, Calc.sub(3, 11));
  }

  @Test(expected = ArithmeticException.class)
  public void testDivideZero() {
    Calc.divide(3, 0);
  }

  public static void main(String[] args) {
    JUnitCore.main(Calc_Test.class.getName());
  }
}
```

“What makes it successful?”

Class Scheme

- Overview
  - 383 LOC
  - 12 public instance method
  - 2 instance fields
  - Quite low level

Bug in Scheme.join():

```
public Scheme join(final Scheme other) {
   final HashSet<Attribute> set = new HashSet<Attribute>();

   for(int i = 0; i < attributes.length; ++i)
      set.add(attributes[i]);

   for(int i = 0; i < attributes.length; ++i)
      set.add(other.attributes[i]);

   return new Scheme(true, set.toArray(new Attribute[set.size()]));
}
```

Class Relation

- Overview
  - 499 LOC
  - 6 public instance method
  - 4 instance fields (incl. inherited)
  - Higher level class

- Bug in Relation.join():

```
for (Tuple<V> tb : other) {
   PartialKey key = new PartialKey(len, tb, bindices);
    Set<Tuple<V>> set = atuples.get(key);
    if (set == null || set.isEmpty())
      continue;
    for (Tuple<V> ta : set) {
      TupleBuilder<V> builder = new TupleBuilder<V>(newScheme.arity);
      populateTuple(builder, a2new, ta);
      populateTuple(builder, a2new, tb);
      result.add(builder.toTuple());
   }
}
```

Direct Test Did not Catch the Bugs

- Scheme_Tests passes through Scheme.join() twice
- Relation_Tests passes through Relation.join()'s inner loop 12 times

A Higher Level Tests Caught the Bugs

- JTL_Tests passes through…
  - Scheme.join() 103,458 times
  - Relation.join() 14,300 times

- Conclusion:

Most bug prevention power is due to higher level functional tests

“Real programmers don't need tests
to write perfect code”

Reality

Code decays

Reality

Small change, Huge effect

Reality

Halting Problem

Reality

The smaller, the better

Need to change code to clean it up

Changes are risky

Small changes are safer

“We have a huge collection of unit tests.
They run every night”

Tight feedback loop =>

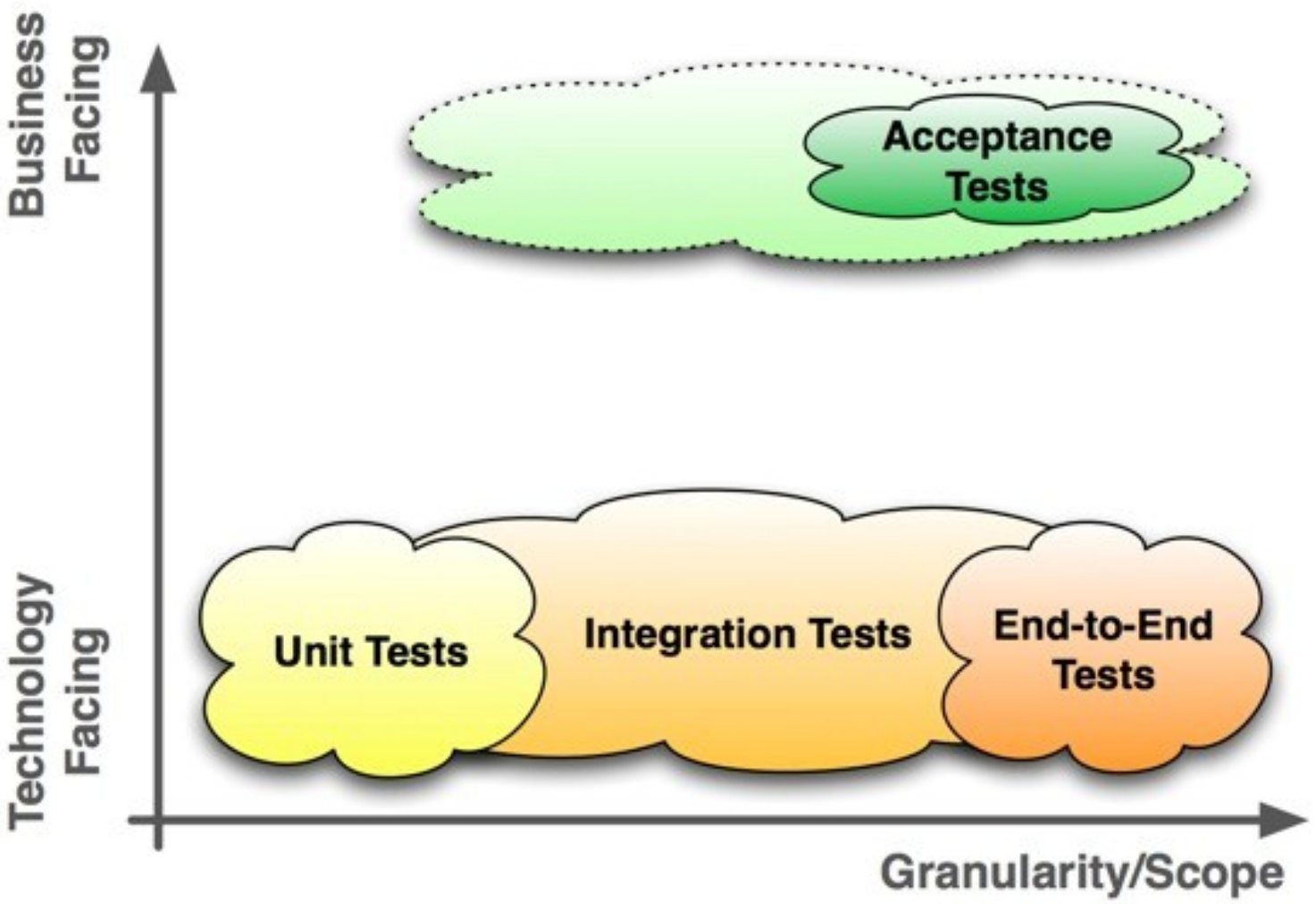
Debugging by diffing

“Integration tests and unit tests are fundamentally different”

Sample Inputs,

Stimulation,

Expected Outputs



Source: http://blog.jonasbandi.net/2010/09/acceptance-vs-integration-tests.html

“Integration tests incur less overhead”

Wide scope => Existence

Narrow scope => Location

“The more unit testing, the better”

Sensitivity to existence,
Sensitivity to location,
Only one test failing at a time

(Speed, Volume of code)

“Test-writing takes time”

Reality

Less bugs are introduced

Reality

Fixes do not introduce new bugs

Reality

Tests =>
Refactoring =>
Cleaner Code =>
Less Bugs

The Quadratic Theory – I

Testing work (per feature) is proportional to #features

The Quadratic Theory – II

Projects slow down as they evolve

The Quadratic Theory – III

Unit Testing => multiply by 10^{-6}

The Quadratic Theory – IV

With unit test, work (per feature) is effectively proportional to complexity of the feature

“Tests often temper with the correct design of the class”

Testability <=> (Good) Design

```
// Non-testable
public class Reporter {

  private PrintWriter pw;

  public Reporter(File f) { pw = new PrintWriter(new FileWriter(f)); }

  public void writePerson(String firstName,
    String lastName)
  {
    writePerson(firstName, null, lastName);
  }

  public void writePerson(String firstName, String middleName,
    String lastName)
  {
    middleName = middleName == null || middleName.length() == 0 ? ""
      : " " + middleName.charAt(0) + ".";
    pw.println(lastName + ", " + firstName + middleName);
  }
}
```

```
// Testable
public class Reporter {

  private PrintWriter pw;

  public Reporter(PrintWriter pw) { this.pw = pw; }

  public void writePerson(String firstName,
    String lastName)
  {
    writePerson(firstName, null, lastName);
  }

  public void writePerson(String firstName, String middleName,
    String lastName)
  {
    middleName = middleName == null || middleName.length() == 0 ? ""
      : " " + middleName.charAt(0) + ".";
    pw.println(lastName + ", " + firstName + middleName);
  }
}
```

```
public class Reporter_Test {

  private static class WriterSpy extends PrintWriter {
    public WriterSpy() { super(System.out); }

    @Override
    public void println(String s) { last = s; }

    public String last;
  }

  @Test
  public void test1() {
    WriterSpy s = new WriterSpy();
    Reporter r = new Reporter(s);

    r.writePerson("John", "Winston", "Lennon");
    Assert.assertEquals("Lennon, John W.", s.last);

  }
}
```

“I will have twice as much code to maintain”

Reality

Tests enable refactoring of program

Program enable refactoring of tests

Guideline

Testing code should be
(way) simpler
than production code

“What is BDD?”

```
public class BorrowingSteps {

   private Library library;
   private Display display;

   @Given("a library with these books: $books")
   public void libraryWithBooks(List<String> books) {
      display = Mockito.mock(Display.class);
      library = new Library(display, books);
   }

   @When("$user borrows $book")
   public void userBorrowsBook(String user, String book) {
      library.borrow(user, book);
   }

   @When("$user returns $book")
   public void userReturnsBook(String user, String book) {
      library.returnBook(user, book);
   }

   @Then("$book should not be borrowed")
   public void bookShouldNotBeBorrowed(String book) {
      Assert.assertFalse(library.isBorrowed(book));
   }
   ...
```

```
   ...

   @Then("$book should be borrowed")
   public void bookShouldBeBorrowed(String book) {
      Assert.assertTrue(library.isBorrowed(book));
   }

   @Then("a \"$message\" message should be displayed")
   public void messageShouldBeDisplayed(String message) {
      Mockito.verify(display).showErrorMessage(message);
   }
}
```

Given a library with these books: (b1,b2,b3,b4)
When Alon borrows b2
Then b2 should be borrowed

When Alon borrows bx
Then a "bx is not a legal book name" message should be displayed

When Alon returns b2
Then b2 should not be borrowed

When Alon returns b4
Then a "Alon stole a book" message should be displayed

When Alon borrows b3
And Alon borrows b3
Then a "Can't borrow same book twice" message should be displayed

“Mocking?”

```
public class FolderScanner {
  public FolderScanner(File root) { ... }
  public List<String> getPaths() { ... }
  public List<File> getFiles() { ... }
  public String getContent(String path) { ... }
}

public class Java {
  public void load(String javaSourceCode) { ... }
  public List<String> vertices() { ... }
  public String getBody(String v) { .. }
  public boolean isPackage(String v) { ... }
  public boolean isMethod(String v)  { ... }
  public String getArgs(String v)  { ... }
  public String getModifiers(String v)  { ... }
  public String getType(String v) { ... }
  ...
}
```

```
public interface ProgressListener {
  public void setLimits(int first, int last);
  public void setCurrent(int n);

  public static final ProgressListener NOP
    = new ProgressListener() {
      @Override
      public void setLimits(int first, int last) {}

      @Override
      public void setCurrent(int n) {}
  };
}
```

```
public class ProgramLoader {

   private FolderScanner fs;
   private Java java;

   public ProgramLoader(Java java, FolderScanner fs) {
      this.java = java; this.fs = fs;
   }

   public void run() { run(ProgressListener.NOP,
      new AtomicBoolean(true)); }

   public void run(ProgressListener listener, AtomicBoolean bool) {
      List<String> temp = new ArrayList<String>(fs.getPaths());

      Iterable<String> lst = Iterables.filter(temp,
         new Predicate<String>() {
            @Override
            public boolean apply(String x) {
               return x.endsWith(".java");
            }
         });
      ...
```

```
      ... // Cont. from previous slide
      int index = -1;

      for(String p : lst) {
         if(!bool.get())
            return;
         listener.setCurrent(++index);
         try
         {
            java.load(fs.getContent(p));
         }
         catch(Exception e)
         {
            throw new RuntimeException(e);
         }
      }
   }
}
```

```
@Test
public void shouldLoadFromFolderScanner()
{
   Java java = Mockito.mock(Java.class);
   FolderScanner fs = Mockito.mock(FolderScanner.class);

   Mockito.when(fs.getPaths()).thenReturn(Arrays.asList("x/y/C1.java",
      "x/y/C2.txt", "x/z/C3.java"));

   Mockito.when(fs.getContent("x/y/C1.java")).thenReturn("content of C1");
   Mockito.when(fs.getContent("x/y/C2.txt")).thenReturn("content of C2");
   Mockito.when(fs.getContent("x/z/C3.java")).thenReturn("content of C3");

   ProgramLoader pl = new ProgramLoader(java, fs);
   pl.run();

   Mockito.verify(java).load("content of C1");
   Mockito.verify(java).load("content of C3");
}
```

“Isn't TDD just another name for testing?”

Source: http://www.natpryce.com/articles/000780.html

“With TDD there's no planning”

Warning:

If you're saying the following, you're not doing testing yet

... or

... you're doing it wrong

“Tests should be written by QA engineers”

“Unit testing is great, but some aspects of my program are too complicated to test”

Pitfalls

“Legacy Code?”

(you want to)

Refactor to make it maintainable,
w/o tests, you can't refactor.

To make it refactorable you need to write tests,

to make it testable you need to refactor

“We're writing less tests 'cus the deadline is just around the corner”

“UI”

Practices

- Always verify that you start from green
- Make sure a test fails the first time you write it
- Before fixing a bug, capture it with a test
- Tests should be simpler than subject
- In a suite place shorter tests first

- One assertion per test method
- Improved TDD cycle
  - Write a failing test
  - ***Verify it fails for the right reason***
  - ***Fix the error message***
  - Make it pass
  - Refactor
- Builders for generating test data
- Meaningful data

-Thank You-