**SOFTENG 254:**
**Quality Assurance**

# Lecture 3b: Testing Practicalities

Paramvir Singh
School of Computer Science

# Potential Assessment Question

Which of the following best describes whether or not it is acceptable to add a test case to a test suite that already has 100% statement coverage?

(a) It is always acceptable because is not always possible to get 100% statement coverage.

(b) It is not acceptable ever.

(c) It is not acceptable when there are enough test cases.

(d) It is sometimes acceptable because 100% statement coverage does not mean the test suite is the best possible.

(e) None of the above.

# Agenda

- Testing practicalities
  - What is a fault? (a.k.a “how many faults are there in the code”?)
  - How to organise tests (a.k.a “what is a test”?)
  - Testing exceptions (a.k.a “exceptions usually indicate an error so to test that the correct exception is thrown I need to cause an error which means when the test passes (meaning no failure) there was an error. . . wait — what?!!”)

# Example: Counting test cases & faults

- Consider a `Student` class
- Student name may include family name, given name, middle names, preferred name
- One method might be to represent a students name as “initials familyname”

# What is a fault?

```
public String initialsName() {
  String f ="";
  if (_firstName != "") {
    f = ""+ _firstName.charAt(0);
    if (_firstName.length() > 1) {
      f +=". ";
    } else {
      f +=" ";
    }
  }
  String middle ="";
  for (Iterator iter =  _middleNames.iterator();
       iter.hasNext();) {
    String m=(String)iter.next();
    if (m.length() > 1) {
      middle +=m.charAt(0) +". ";
    } else {
      middle +=m.charAt(0) +" ";
    }
  }
  return  f + middle + _lastName;
}
```

# What is a fault?

```
public String initialsName() {
  String f ="";
  if (_firstName != "") {
    f = ""+ _firstName.charAt(0);
    if (_firstName.length() < 1) { // 1-char change
      f +=". ";
    } else {
      f +=" ";
    }
  }
  String middle ="";
  for (Iterator iter =  _middleNames.iterator();
       iter.hasNext();) {
    String m=(String)iter.next();
    if (m.length() > 1) {
      middle +=m.charAt(0) +". ";
    } else {
      middle +=m.charAt(0) +" ";
    }
  }
  return  f + middle + _lastName;
}
```

# Running the tests (JUnit 3.8)



```
.F.F.F.F.F.F.F.F..F.......
Time: 0.025
There were 10 failures:
1) test3Names(TestStudent)junit.framework.ComparisonFailure:
 expected:< ..... > but was:< ..... >

[... stuff deleted ...]

FAILURES!!!
Tests run: 18,    Failures: 10,   Errors: 0
```

# What is a fault?



```
public String initialsName() {
  String f ="";
  if (_firstName != "") {
    f = ""+ _firstName.charAt(0);
    if (_firstName.length() > 1) {
      f +=". ";
    } else {
      f +=" ";
    }
  }
  String middle ="";
  for (Iterator iter =  _middleNames.iterator();
       iter.hasNext();) {
    String m=(String)iter.next();
    if (m.length() > 1) {
      middle +=m.charAt(0) +". ";
    } else {
      middle +=m.charAt(0) +" ";
    }
  }

  return f +middle +   _lastName;
}
```

# Running the tests



```
.............F.....
Time: 0.022
There was 1 failure:
1) test1LetterNames(TestStudent)junit.framework.ComparisonFailure:
expected:<... B...> but was:<.... B....>

[... stuff deleted ...]

FAILURES!!!
Tests run: 18,    Failures: 1,    Errors: 0
```

# What is a fault?

- Two failed tests — what (or where) is the fault?

```
public void test3Names() {
  Student student = new Student("John", "Paul",  "Jones");

  assertEquals("John Paul Jones", student.fullName());
  assertEquals("John Jones", student.preferredName());
  assertEquals("Jones, John Paul", student.lastNameFirst());
  assertEquals("J. P. Jones", student.initialsName());
  assertEquals("jjon", student.upi());
}

public void test3NamesPreferredNameDifferent() {
  Student student = new Student("Farrokh", "Pluto", "Bulsara");
  student.setPreferredName("Freddie  Mercury");

  assertEquals("Farrokh Pluto Bulsara", student.fullName());
  assertEquals("Freddie Mercury", student.preferredName());
  assertEquals("Bulsara, Farrokh Pluto", student.lastNameFirst());
  assertEquals("F. P. Bulsara", student.initialsName());
  assertEquals("fbul", student.upi());
}
```

# What is a test?



```
// One test or 3?
public void testMean() {
  Stats stats = new Stats(new int[]{0});
  assertEquals("Mean of list {0}",
               0, stats.getMean());

  stats = new Stats(new int[]{2});
  assertEquals("Mean of list {2}",
               2, stats.getMean());

  Stats stats = new Stats(new int[]{4, 2});
  assertEquals(3, stats.getMean());
}
```

# What is a test?



```
// One test or 2?
public void testPoint() {
  Point p = new Point(1,2);
  assertEquals("X coordinate", 1, p.getX());
  assertEquals("Y coordinate", 2, p.getY());
}
```

# Organising tests



```
public void test3NamesAsList() {
    List<String> middle = new Vector<String>();
    middle.add("Paul");
    Student student = new Student("John", middle, "Jones");

    assertEquals("John Paul Jones", student.fullName());
    assertEquals("John Jones", student.preferredName());
    assertEquals("Jones, John Paul", student.lastNameFirst());
    assertEquals("J. P. Jones", student.initialsName());
    assertEquals("jjon", student.upi());
}
```

# Organising tests



```
public void testFullname() {
    List<String> middle = new Vector<String>();
    middle.add("Paul");
    Student student = new Student("John", middle, "Jones");
    assertEquals("John Paul Jones", student.fullName());

    middle = new Vector<String>();
    middle.add("Dennis");
    middle.add("Blandford");
    student = new Student("Peter", middle, "Townshend");
    assertEquals("Peter Dennis Blandford Townshend",
                 student.fullName());

    student = new Student("Brian", "", "Johnson");
    assertEquals("Brian Johnson", student.fullName());
}
```

# Organising tests



```
public void testFullnameNoMiddle() {
    Student student = new Student("Brian", "", "Johnson");
    assertEquals("Brian Johnson", student.fullName());
}

public void testFullnameSingleMiddle() {
    List<String> middle = new Vector<String>();
    middle.add("Paul");
    Student student = new Student("John", middle, "Jones");
    assertEquals("John Paul Jones", student.fullName());
}

public void testFullnameMultipleMiddle() {
    List<String> middle = new Vector<String>();
    middle.add("Dennis");
    middle.add("Blandford");
    Student student = new Student("Peter", middle, "Townshend");
    assertEquals("Peter Dennis Blandford Townshend",
                 student.fullName());
}
```

# Failures, Errors, and failed tests

- JUnit 3.8 distinguishes between **errors** and **failures**
  - Both represent **failed tests**
  - Both apply to what the IEEE standard refers to as a “failure”
- The JUnit terms represent different kinds of problems detected by JUnit

  **Failure** The expected result specified by the (human) tester was not what JUnit observed. Typically a failed assertion.

  **Error** JUnit was unable to complete the test. Typically some kind of exception being thrown and not caught
- JUnit tests are only useful if, when they do not pass, it is due to a *failure in the implementation under test*

# JUnit Errors due to Bad Tests

```
public void testA(){
    Student student = null;

    assertEquals("John Paul Jones", student.fullName());
    assertEquals("John Jones", student.preferredName());
    assertEquals("Jones, John Paul", student.lastNameFirst());
    assertEquals("J. P. Jones", student.initialsName());
    assertEquals("jjon", student.upi());
}
```

```
.E
Time: 0.001
There was 1 error:
1) testA(se254.BadTests)java.lang.NullPointerException
        at se254.BadTests.testA(BadTests.java:9)
        <...omitted...>
FAILURES!!!
Tests run: 1,  Failures: 0,  Errors: 1
```

# JUnit Failures due to Bad Tests

```
public void testB(){
    Student student = new Student("John", "Paul",  "Jones");

    assertEquals("John Henry Bonham", student.fullName());
    assertEquals("John Jones", student.preferredName());
    assertEquals("Jones, John Paul", student.lastNameFirst());
    assertEquals("J. P. Jones", student.initialsName());
    assertEquals("jjon", student.upi());
}
```

```
.F
Time:  0.002
There was 1 failure:
1) testB(se254.BadTests)junit.framework.ComparisonFailure:
    expected:<John [Henry Bonham]> but was:<John [Paul Jones]>
    at se254.BadTests.testB(BadTests.java:19)
    <...omitted...>

FAILURES!!!
Tests run: 1,    Failures: 1,   Errors: 0
```

# Example: JUnit Error



Stats

```
private double mean() {
  if (_numbers.length == 0) {
    throw new RuntimeException("Empty input");
  }
  double sum = 0;

  for (int i = 0; i <= _numbers.length; i++) {
    sum = sum + _numbers[i];
  }

  return sum /  _numbers.length;
}
```

TestStatsMean

```
public void testZero() {
  Stats stats = new Stats(new int[]{0});
  assertEquals("Mean of list {0}", 0, stats.getMean());
}
```

# Example: JUnit Error



```
Stats
private double mean() {
  if (_numbers.length == 0) {
    throw new RuntimeException("Empty input");
  }
  double sum = 0;

  for (int i = 0; i <= _numbers.length; i++) {
    sum = sum + _numbers[i];
  }

  return sum /  _numbers.length;
}

TestStatsMean
public void testZero() {
  Stats stats = new Stats(new int[]{0});
  assertEquals("Mean of list {0}", 0, stats.getMean());
}
```

# Example: JUnit Error



Stats

```
private double mean() {
  if (_numbers.length == 0) {
    throw new RuntimeException("Empty input");
  }
  double sum = 0;

  for (int i = 0; i <= _numbers.length; i++) {
    sum = sum + _numbers[i];
  }

  return sum /  _numbers.length;
}
```

TestStatsMean

```
public void testZero() {
  Stats stats = new Stats(new int[]{0});
  assertEquals("Mean of list {0}", 0, stats.getMean());
}
```

# Example: JUnit Error



Stats

```
private double mean() {
  if (_numbers.length == 0) {
    throw new RuntimeException("Empty input");
  }
  double sum = 0;

  for (int i = 0; i <= _numbers.length; i++) {
    sum = sum + _numbers[i];
  }

  return sum / _numbers.length;
}
```

_numbers:{ 0 }

TestStatsMean

```
public void testZero() {
  Stats stats = new Stats(new int[]{0});
  assertEquals("Mean of list {0}", 0, stats.getMean());
}
```

# Example: JUnit Error



```
Stats
private double mean() {                                  _numbers:{ 0 }
  if (_numbers.length ==0){
    throw new RuntimeException("Empty input");
  }
  double sum =0;                                         sum: 0

  for (int i = 0; i <= _numbers.length; i++) {
    sum = sum + _numbers[i];
  }

  return sum/  _numbers.length;
}

TestStatsMean
public void testZero() {
  Stats stats =new Stats(new int[]{0});
  assertEquals("Mean of list {0}", 0, stats.getMean());
}
```

# Example: JUnit Error



Stats

```
private double mean() {
  if (_numbers.length == 0) {
    throw new RuntimeException("Empty input");
  }
  double sum = 0;

  for (int i = 0; i <= _numbers.length; i++) {
    sum = sum + _numbers[i];
  }

  return sum /  _numbers.length;
}
```

_numbers:{ 0 }

sum: 0

i: 0

TestStatsMean

```
public void testZero() {
  Stats stats = new Stats(new int[]{0});
  assertEquals("Mean of list {0}", 0, stats.getMean());
}
```

# Example: JUnit Error



```
Stats
private double mean() {                                  _numbers:{ 0 }
  if (_numbers.length ==0){
    throw new RuntimeException("Empty input");
  }
  double sum =0;                                         sum: 0

  for (int i = 0; i <= _numbers.length; i++) {           i: 0
    sum = sum + _numbers[i];
  }

  return sum/  _numbers.length;
}

TestStatsMean
public void testZero() {
  Stats stats =new Stats(new int[]{0});
  assertEquals("Mean of list {0}", 0, stats.getMean());
}
```

# Example: JUnit Error



Stats

```
private double mean() {
  if (_numbers.length ==0){
    throw new RuntimeException("Empty input");
  }
  double sum =0;

  for (int i = 0; i <= _numbers.length; i++) {
    sum = sum + _numbers[i];
  }

  return sum/  _numbers.length;
}
```

_numbers:{ 0 }

sum: 0

i: 1

TestStatsMean

```
public void testZero() {
  Stats stats =new Stats(new int[]{0});
  assertEquals("Mean of list {0}", 0, stats.getMean());
}
```

# Example: JUnit Error

```
Stats
private double mean() {                                   _numbers:{ 0 }
  if (_numbers.length ==0){
    throw new RuntimeException("Empty input");
  }
  double sum =0;                                          sum: 0

  for (int i = 0; i <= _numbers.length; i++) {            i: 1
    sum = sum + _numbers[i];   ArrayIndexOutOfBoundsException
  }

  return sum/  _numbers.length;
}

TestStatsMean
public void testZero() {
  Stats stats =new Stats(new int[]{0});
  assertEquals("Mean of list {0}", 0, stats.getMean());
}
```

# Example: JUnit Error



```
java -classpath junit.jar:. TestStatsAll
.E
Time: 0.021
There was 1 error:
1) testZero(TestStatsMean)java.lang.ArrayIndexOutOfBoundsException: 1
        at Stats.mean(Stats.java:37)
        at Stats.getMean(Stats.java:50)
        at TestStatsMean.testZero(TestStatsMean.java:10)
        <...omitted...>

FAILURES!!!
Tests run: 1,    Failures: 0,   Errors: 1
```

# Testing error checking

- Many methods only work correctly if their parameters have certain values *e.g., List must be non-empty, numbers must be non-negative, parameter must be non-null*
- Such methods will often check that their expectations are met, and signal an error if they aren't
- ⇒ how to test this behaviour?

# Exceptions review

- Printing a message isn't always appropriate
- Returning a value indicating a problem isn't always safe
- ⇒ **exceptions** — a general purpose mechanism for signalling "unexpected situations"
- In Java, an exception is an object
- When an unexpected situation occurs, an exception is **thrown**
- A thrown exception may be **caught**, or otherwise the program terminates

# Example: Throwing an exception



```
private double mean() {

    if (_numbers.length == 0){
        throw new RuntimeException("Empty input");
    }
    ....
}
```

# Example: Testing for exceptions

```
public void testEmptyInputBroken() {
    Stats stats =new Stats(new int[]);
    assertEquals(0, stats.getMean()); // Correctly throws exception
}
```

```
prompt> java -cp .... TestStatsMean

JUnit version 4.11
...E.........
Time: 0.006
There was 1 failure:
1) testEmptyInputBroken(se254.lab.junitreview.TestStatsMean)
java.lang.RuntimeException: Empty input
        at se254.lab.junitreview.Stats.mean(Stats.java:35)
        at se254.lab.junitreview.Stats.getMean(Stats.java:53)
        at se254.lab.junitreview.TestStatsMean.testEmptyInputBroken....
....
```

# Example: Testing for exceptions



```
public void testEmptyInput() {
    Stats stats = new Stats(new int[]{});
    try {
        assertEquals(0, stats.getMean());
        fail("Should have failed: empty list");
    } catch (Exception e) {
        // Nothing here - just ignore the fact that the
        // exception occurred (since that's the expected
        // behaviour in this case).
    }
}
```

# A successful test



```
public void testEmptyInput() {
    Stats stats = new Stats(new int[]{});
    try {
        assertEquals(0, stats.getMean());
        fail("Should have failed: empty list");
    } catch (Exception e) {
        // Nothing here - just ignore the fact that the
        // exception occurred (since that's the expected
        // behaviour in this case).
    }
}
```

# A successful test



```
public void testEmptyInput() {
    Stats stats = new Stats(new int[]{});
    try {
        assertEquals(0, stats.getMean());
        fail("Should have failed: empty list");
    } catch (Exception e) {
        // Nothing here - just ignore the fact that the
        // exception occurred (since that's the expected
        // behaviour in this case).
    }
}
```

# A successful test



```
public void testEmptyInput() {
    Stats stats = new Stats(new int[]{});
    try {
        assertEquals(0, stats.getMean());
        fail("Should have failed: empty list");
    } catch (Exception e) {
        // Nothing here - just ignore the fact that the
        // exception occurred (since that's the expected
        // behaviour in this case).
    }
}
```

# A successful test



```
class Stats {
    private double mean() {

        if (_numbers.length == 0) {
            throw new RuntimeException("Empty input");
        }
        ....
    }

    public int getMean() {
        return (int) (mean() + 0.5);
    }
    ....
}
```

# A successful test



```
class Stats {
    private double mean() {

        if (_numbers.length == 0) {
            throw new RuntimeException("Empty input");
        }
        ....
    }

    public int getMean() {
        return (int) (mean() + 0.5);
    }
    ....
}
```

# A successful test

```
class Stats {
    private double mean() {

        if (_numbers.length == 0) {
            throw new RuntimeException("Empty input");
        }
        ....
    }

    public int getMean() {
        return (int) (mean() + 0.5);
    }
    ....
}
```

# A successful test



```
public void testEmptyInput() {
    Stats stats = new Stats(new int[]{});
    try {
        assertEquals(0, stats.getMean());
        fail("Should have failed: empty list");
    } catch (Exception e) {
        // Nothing here - just ignore the fact that the
        // exception occurred (since that's the expected
        // behaviour in this case).
    }
}
```

# A successful test



```
public void testEmptyInput() {
    Stats stats = new Stats(new int[]{});
    try {
        assertEquals(0, stats.getMean());
        fail("Should have failed: empty list");
    } catch (Exception e) {
        // Nothing here - just ignore the fact that the
        // exception occurred (since that's the expected
        // behaviour in this case).
    }
}
```

# A failed test



```
public void testEmptyInput() {
    Stats stats = new Stats(new int[]{});
    try {
        assertEquals(0, stats.getMean());
        fail("Should have failed: empty list");
    } catch (Exception e) {
        // Nothing here - just ignore the fact that the
        // exception occurred (since that's the expected
        // behaviour in this case).
    }
}
```

# A failed test



```
public void testEmptyInput() {
    Stats stats = new Stats(new int[]{});
    try {
        assertEquals(0, stats.getMean());
        fail("Should have failed: empty list");
    } catch (Exception e) {
        // Nothing here - just ignore the fact that the
        // exception occurred (since that's the expected
        // behaviour in this case).
    }
}
```

# A failed test



```
public void testEmptyInput() {
    Stats stats = new Stats(new int[]{});
    try {
        assertEquals(0, stats.getMean());
        fail("Should have failed: empty list");
    } catch (Exception e) {
        // Nothing here - just ignore the fact that the
        // exception occurred (since that's the expected
        // behaviour in this case).
    }
}
```

# A failed test



```
public void testEmptyInput() {
    Stats stats = new Stats(new int[]{});
    try {
        assertEquals(0, stats.getMean());
        fail("Should have failed: empty list");
    } catch (Exception e) {
        // Nothing here - just ignore the fact that the
        // exception occurred (since that's the expected
        // behaviour in this case).
    }
}
```

# A failed test

```
There was 1 failure:
1) testEmptyInput(TestIntRectangle)junit.framework.AssertionFailedError:
   Should have failed: empty list
   at TestIntRectangle.testEmptyInput(TestIntRectangle.java:8)
   at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
   at sun.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAccessorIm
      .....
```

# Bad Practice

- Catching exceptions more general than expected
- Won't detect when the wrong exception is thrown

```
public void testEmptyInput() {
    Stats stats = new Stats(new int[]{});
    try {
        assertEquals(0, stats.getMean());
        fail("Should have failed: empty list");
    } catch (Exception e) {
        // Nothing here - just ignore the fact that the
        // exception occurred (since that's the expected
        // behaviour in this case).
    }
}
```

# Good Practice

- Make the test as specific as possible

```
public void testEmptyInput() {
    Stats stats = new Stats(new int[]{});
    try {
        assertEquals(0, stats.getMean());
        fail("Should have failed: empty list");
    } catch (RuntimeException e) {
        assertEquals("Empty input", e.getMessage());
    }
}
```

# Bad Practice

- Don’t catch exceptions unless you are testing that they are being thrown

```
public void testSomeScenario() {
    IUT iut = new IUT();
    try {
        iut.callSomeMethod();
        assertEquals(...);
    } catch (Exception e) {
        System.out.println("Unexpected Exception thrown");
    }
}
```

# Better Practice

- "But callSomeMethod throws a checked exception so I have to catch it!"

```
public void testSomeScenario() throws Exception {
    IUT iut = new IUT();
    iut.callSomeMethod();
    assertEquals(...);
}
```

# Really Bad Practice

- DO NOT turn a JUnit error into a JUnit failure
- This loses stack trace information and doesn't change the fact that the test has failed.

```
public void testSomeScenario() {
    IUT iut = new IUT();
    try {
        iut.callSomeMethod();
        assertEquals(...);
    } catch (Exception e) {
        fail("Unexpected Exception thrown");
    }
}
```

# JUnit 4 Testing Exception

- @Test takes a parameter `expected`, indicating the `Exception` that is expected to be thrown
- Is not useful if the same exception can be thrown for multiple reasons
- A catch clause will catch all exceptions of the stated type and all its descendants

# JUnit 4 Testing Exception



```
public void throwsRuntimeException(boolean flag, String msg) {
  if (flag) {
    throw new RuntimeException("Didn't like flag");
  } else {
    System.out.println("Message is of length " + msg.length());
  }
}
```

# Testing Exceptions (Junit 4)

```
@Test(expected=RuntimeException.class)
public void testExceptionShouldPass() {
  throwsRuntimeException(true, "nothing");
}
@Test(expected=RuntimeException.class)
public void testExceptionShouldFail() {
  throwsRuntimeException(false, null);
}
```

- So not recommended except in situations where the expected exception is very specific.

# Key Points

- “number of failed (JUnit) tests” does not necessarily relate to “number of faults” or “how much work has to be done”
- “number of tests” is difficult to define
- Test all code, including exception handlers
- The more specific the test, the more information its failure will provide about the fault that caused the failure (e.g. its location)