Why Testing?

- Tests Reduce Bugs in New Features
- Tests Reduce Bugs in Existing Features
- Tests Are Good Documentation
- Tests Improve Design
- Tests Constrain Features
- Testing Is Fun
- Testing Forces You to Slow Down and Think
- Testing Makes Development Faster
- Tests Reduce Fear

Terminology

Error - a mistake made by a human (in a software development activity)

Defect (or **fault**) - the result of introducing an error into a software artifact (SRS, SDS, code, etc.)

Failure - a departure from the required behavior for a system

Testing is concerned with establishing the presence of program defects.

Debugging is concerned with finding where defects occur (in code, design or requirements) and removing them. (fault identification and removal)

Philosophy

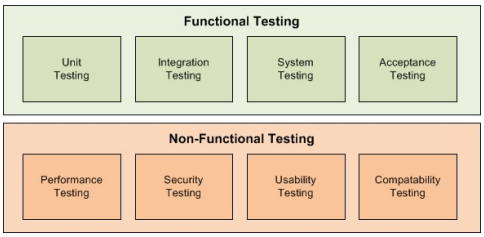
Testing is the one step in software engineering process that could be viewed as destructive rather than constructive.
"A successful test is one that breaks the software."
[McConnell 1993]

A successful test is one that uncovers an as yet undiscovered defect.

Testing can not show the absence of defects, it can only show that software defects are present.

For most software exhaustive testing is not possible.

What to test?



Unit Testing

Unit testing is a software development process in which the smallest testable parts of an application, called units, are individually and independently scrutinized for proper operation. Unit testing can be done manually but is often automated.

There is some debate about what constitutes a "unit". Here some common definitions of a unit:

- the smallest chunk that can be compiled by itself
- a stand-alone procedure of function
- something so small that it would be developed by a single person

Testing by execution in a controlled setting

Black Box Techniques

- tests rely on module *description* to devise test data
- uses inputs, functionality, outputs in the architectural design
- treats module like a "black box"

White Box Techniques

- tests relies on module *source code* to devise test data
- analyze the module algorithm in the detailed design
- treats module like a " white box" or "glass box"

Test First vs. Test Last

- Pick a piece of functionality
- Write a test that expresses a small task that fails
- Write production code until test passes
- Run all tests
- Rework code until all tests pass
- Repeat [1]

- Pick a piece of functionality
- Write production code that implements entire functionality
- Write tests to validate all functionality
- Run all tests
- Rework code until all tests pass [1]

Test Driven Development (TDD)

- Method of developing software not just testing software
- Development in small steps. This will make debugging easier since we will have small code chunks to debug.
- Unit Tests are developed FIRST before the code
- YAGNI principle – "You Ain't Gonna Need It"

Java Unit Testing – JUnit

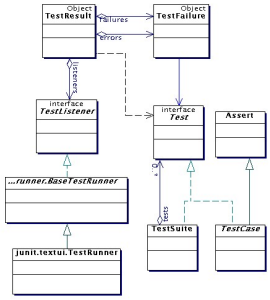
- Beck and Gamma (Gang of Four, more later) developed JUnit on a flight from Zurich to Washington, D.C.

Martin Fowler: "Never in the field of software development was so much owed by so many to so few lines of code."

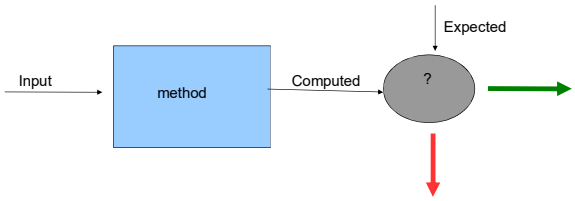
- JUnit has become the standard tool for Test-Driven Development in Java
- JUnit test generators now part of many Java IDEs (Eclipse, BlueJ, Jbuilder, DrJava)

JUnit

- JUnit test framework is a package of classes that lets you write tests for each method, then easily run those tests
- **TestRunner** runs tests and reports **TestResult**s
- You test your class by extending abstract class ***TestCase***
- To write test cases, you need to know and understand the **Assert** class



Expectations are explicit!



Junit – assert* methods

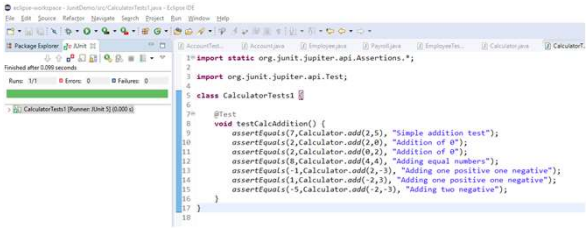
Each assert method has parameters like these:
message, expected-value, actual-value

- **assertTrue**(String *message*, Boolean *test*)
- **assertFalse**(String *message*, Boolean *test*)
- **assertNull**(String *message*, Object *object*)
- **assertNotNull**(String *message*, Object *object*)
- **assertEquals**(String *message*, Object *expected*, Object *actual*) (uses equals method)
- **assertSame**(String *message*, Object *expected*, Object *actual*) (uses == operator)
- **assertNotSame**(String *message*, Object *expected*, Object *actual*)

Junit – Examples

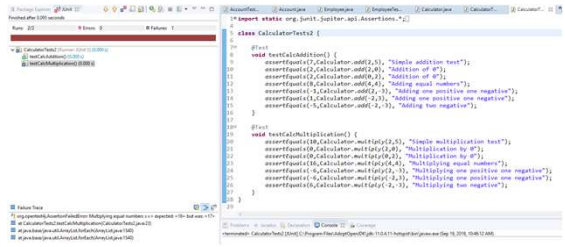
```
public class Calculator { // JUnitDemo project

public static int add(int n1, int n2) {
   return n1 + n2;
}

public static int multiply(int n1, int n2) {
   …
}

…
}
```


Junit – All tests passed!




```
import static org.junit.jupiter.api.Assertions.*;

import org.junit.jupiter.api.Test;


class CalculatorTests {
   @Test
   void testCalcAddition() {
assertEquals(7,Calculator.add(2,5), "Simple addition test");
assertEquals(2,Calculator.add(2,0), "Addition of 0");
assertEquals(2,Calculator.add(0,2), "Addition of 0");
assertEquals(8,Calculator.add(4,4), "Adding equal numbers");
assertEquals(-1,Calculator.add(2,-3), "Adding one positive one negative");
assertEquals(1,Calculator.add(-2,3), "Adding one positive one negative");
assertEquals(-5,Calculator.add(-2,-3), "Adding two negative");
   }

   @Test
   void testCalcMultiplication() {
assertEquals(10,Calculator.multiply(2,5), "Simple multiplication test");
assertEquals(0,Calculator.multiply(2,0), "Multiplication by 0");
assertEquals(0,Calculator.multiply(0,2), "Multiplication by 0");
assertEquals(16,Calculator.multiply(4,4), "Multiplying equal numbers");
assertEquals(-6,Calculator.multiply(2,-3), "Multiplying one positive one negative");
assertEquals(-6,Calculator.multiply(-2,3), "Multiplying one positive one negative");
assertEquals(6,Calculator.multiply(-2,-3), "Multiplying two negative");
   }
}
```


Junit – Oops!!!



org.opentest4j.AssertionFailedError: Multiplying equal numbers ==> expected: <16> but was: <17>

assertEquals(16,Calculator.multiply(4,4), "Multiplying equal numbers");

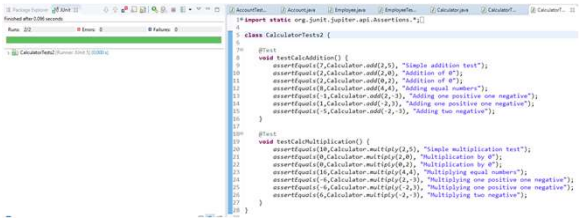
Junit – Check back in the Calculator class!

```
public class Calculator {

public static int add(int n1, int n2) {
   return n1 + n2;
}

public static int multiply(int n1, int n2) {
   int solution = n1*n2;
   if (n1 == n2) {   // YEAH, bit obvious!
      solution++;
   }
   return solution;
}
}
```


Junit – Fix it!

```
public class Calculator {

public static int add(int n1, int n2) {
   return n1 + n2;
}

public static int multiply(int n1, int n2) {
   return n1*n2;
}
}
```


Junit – Now we're good (for the moment…)



```
import static org.junit.jupiter.api.Assertions.*;

class CalculatorTests2 {

   @Test
   void testCalcAddition() {
      assertEquals(7,Calculator.add(2,5), "Simple addition test");
      assertEquals(2,Calculator.add(2,0), "Addition of 0");
      assertEquals(2,Calculator.add(0,2), "Addition of 0");
      assertEquals(8,Calculator.add(4,4), "Adding equal numbers");
      assertEquals(-1,Calculator.add(2,-3), "Adding one positive one negative");
      assertEquals(1,Calculator.add(-2,3), "Adding one positive one negative");
      assertEquals(-5,Calculator.add(-2,-3), "Adding two negative");
   }

   @Test
   void testCalcMultiplication() {
      assertEquals(10,Calculator.multiply(2,5), "Simple multiplication test");
      assertEquals(0,Calculator.multiply(2,0), "Multiplication by 0");
      assertEquals(0,Calculator.multiply(0,2), "Multiplication by 0");
      assertEquals(16,Calculator.multiply(4,4), "Multiplying equal numbers");
      assertEquals(-6,Calculator.multiply(2,-3), "Multiplying one positive one negative");
      assertEquals(-6,Calculator.multiply(-2,3), "Multiplying one positive one negative");
      assertEquals(6,Calculator.multiply(-2,-3), "Multiplying two negative");
   }
}
```


Junit - More

https://junit.org/junit5/docs/current/user-guide/

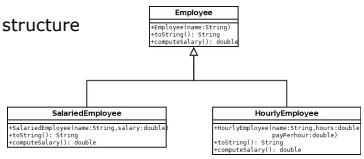
https://www.youtube.com/watch?v=aoX0UTzhx80

http://agiledata.org/essays/tdd.html

https://www.youtube.com/watch?v=O-ZT_dtlrR0

TDD – Small example

1. Determine the structure



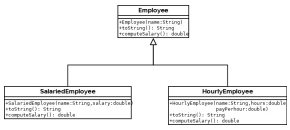
2. Write (all?) tests first!

```
class EmployeeTests {
    @Test
    public void standardTests() {
        Employee ron = new Employee("Ron");
        assertEquals("Employee: Ron",ron.toString());
        assertEquals(0.0,ron.computeSalary(),1e-4);
    }
}
```


More tests

```
class HourlyEmployeeTests {
    @Test
    public void standardTests() {
       HourlyEmployee jim =
        new HourlyEmployee("Jim",150,12);
       assertEquals("Employee: Jim hours: 150.0 pay/h:  +
        12.0",jim.toString());
       assertEquals(1800.0,jim.computeSalary(),1e-4);
    }
}

class SalariedEmployeeTests {
    @Test
    public void standardTests() {
       SalariedEmployee kim = new SalariedEmployee("Kim",2100);
       assertEquals("Employee: Kim salary: 2100.0",kim.toString());
       assertEquals(2100.0,kim.computeSalary(),1e-4);
    }
}
```

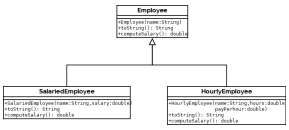

TDD – Small example



Test organization: Not a "one-fits-all" scheme

- Most tests won't even compile at the beginning
- Most tests will fail at the beginning
- My recipe: Build tests per class
  → EmployeeTests, SalariedEmployeeTests. HourlyEmployeeTests
- For comprehensive tests (all three): Build a "Test Suite"
- See Inherit2 project

TDD – Small example



3. Make all tests succeed

Easier said than done, since none of these three classes are present!

→ Options to get started

- Stubs
- Mock-ups

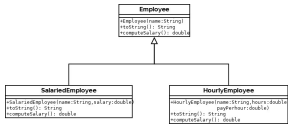
Testing Vocabulary

- *Dummy* objects are passed around but never actually used. Usually they are just used to fill parameter lists.
- *Fake* objects actually have working implementations, but usually take some shortcut which makes them not suitable for production (…)
- *Stubs* provide canned answers to calls made during the test, usually not responding at all to anything outside what's programmed in for the test.
- *Mocks* are (…) objects pre-programmed with expectations which form a specification of the calls they are expected to receive.

https://martinfowler.com/articles/mocksArentStubs.html#TheDifferenceBetweenMocksAndStubs

TDD – Small example

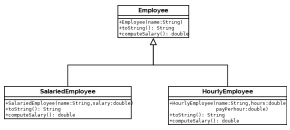
Employee "stub"




```
public class Employee {

    public Employee(String name) {
    }

    public String toString() {
        return "";
    }

    public double computeSalary() {
        return 0.0;
    }
}
```


TDD – Small example

Another stub




```
public class HourlyEmployee extends Employee {

public HourlyEmployee(String name, double hours,
                      double payPerHour) {
    super(name);
}
}
```


TDD – Test/Coding strategy

```
@Test
public void standardTests() {
    Employee ron = new Employee("Ron");
    assertEquals("Employee: Ron",ron.toString());
    assertEquals(0.0,ron.computeSalary(),1e-4);
}
```

Principle (a little annoying):

If an assert fails in a test, the test is abandoned!

→ You can only see one failure at a time
→ The sequence of fixes depends on the sequence of asserts
→ Maybe just have assert per test?
Not really practical, so we'll stick with this issue for the time being