SOFTWARE TESTING

JUNIT, TDD, AND WHITE-BOX TECHNIQUES

Hakam Alomari

Miami University Software Technology & Analysis Group (MUSTANG)
Computer Science & Software Engineering
Miami University, Oxford, Ohio, USA

AGENDA

- Project's Due Dates
- White-Box Techniques
  - Code Coverage
- Unit testing

Project's Due Dates ?

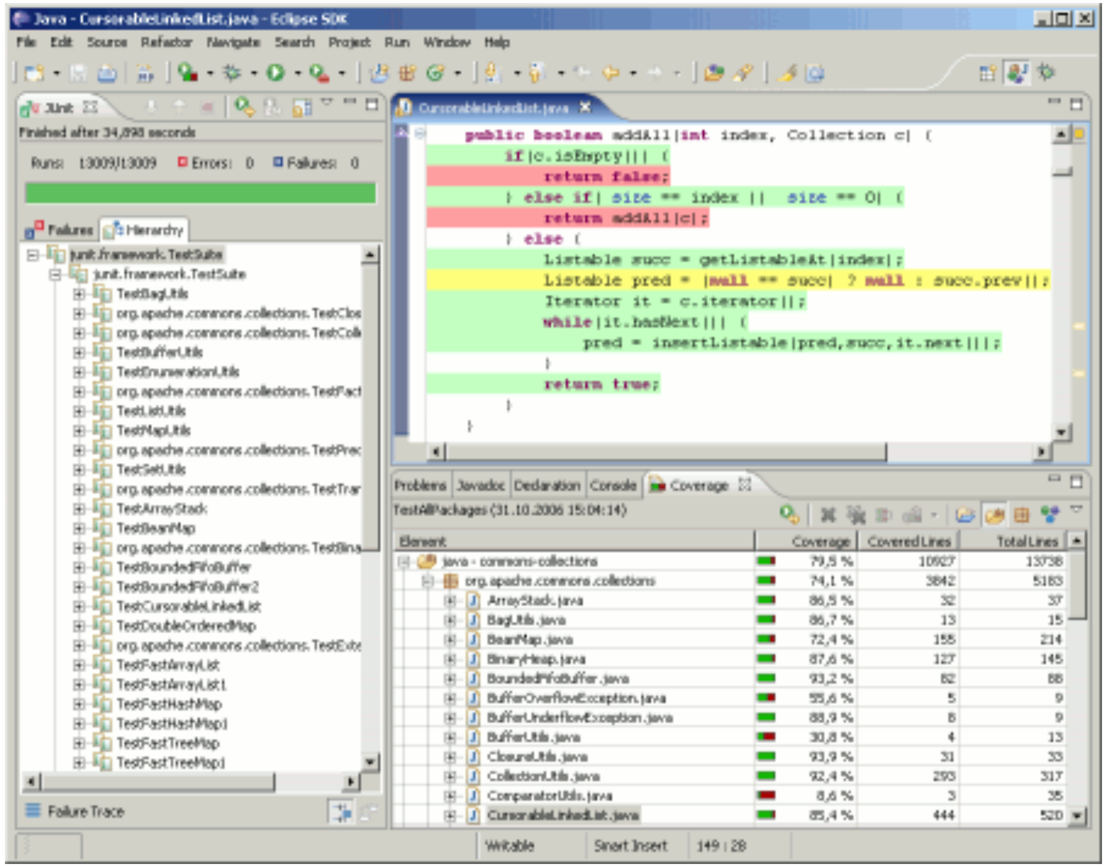
- ~~April 2nd → 5th week deliverables~~
- ~~April 6th → Iteration 1 presentation → weeks 3 + 4 → 90 mints~~
- ~~April 16th → 6th week deliverables~~
- ~~April 20th → Iteration 2 presentation → weeks 5 + 6 → 90 mints~~
- April 23rd → 7th week deliverables
- April 30th → 8th week deliverables
- May 4th → Iteration 3 (final) presentation → weeks 7 + 8 → 90 mints

LEVELS OF WHITE-BOX CODE COVERAGE

- A measurement to evaluate the percentage of code tested
- Options are:
  - Statement coverage:
    - Has each statement in the program been executed?
  - Decision coverage (aka branch coverage)
    - Has each branch of each control structure (such as in if and case statements) been executed? Another way of saying this is, has every edge in the CFG been executed
  - Condition coverage (or predicate coverage)
    - Has each Boolean sub-expression evaluated both to true and false?
  - Path coverage
    - Is every possible combination of branches – every path through the program – taken by some test case?

EclEmma is a free Java code coverage tool for Eclipse

http://www.eclemma.org



Industry Practice

- All-statements is common goal, rarely achieved (due to unreachable code)
- All branches if possible:
  - Safety critical industry has more arduous criteria (e.g., “MCDC”, modified condition/decision coverage)
- All-paths is infeasible

Statement Coverage

- Each line of code is executed.

```
if (a)
    stmt1;
if (b)
    stmt2;
```

- a = t; b = t gives statement coverage
- a = t; b = f doesn't give statement coverage

Decision Coverage

- Decision coverage is also known as branch coverage
- The Boolean condition at every branch point (if, while, etc.) has been evaluated to both T and F.

```
if (a and b)
    stmt1;
if (c)
    stmt2;
```

- a = t; b = t; c = t and a = f; b = ?; c = f gives decision coverage

EXAMPLE

- The decision has taken all possible outcomes at least once.

```
If ((a) & (b) | (c)) then…
```

true ***false***

- We could also say we cover both the true and the false branch (Branch Testing).

DECISION VS STATEMENT

- Does Decision Coverage Guarantee Statement Coverage?
  - Yes assuming the program has at least one decision, there is only one entry point, and you ignore exception handling code.

- Does Statement Coverage Guarantee Decision Coverage?

```
if (a)
    stmt1;
```

  - If no, give an example of input that gives statement coverage but not decision coverage.
  - No. a = t. If this is the only test case, you don't have decision coverage because you haven't tested the decision where the if statement evaluates to false.

Condition Coverage

- Each Boolean sub-expression at a branch point has been evaluated to true and false.

```
if (a and b)
    stmt1;
```

- a = t, b = t and a = f; b = f gives condition coverage

EXAMPLE

- Every condition in the decision has taken all possible outcomes at least once.

```
If  ((a)  &  (b)  |  (c))  then…
```

true *false* *true* *false* *true* *false*

Decision vs Condition

- Does condition coverage guarantee decision coverage?

```
if (a and b)
    stmt1;
```

- If no, give example input that gives condition coverage but not decision coverage.
- No. a = t, b = f and a = f, b = t. This gives condition coverage but doesn't test the case where the branch is taken.

CONDITION/DECISION COVERAGE

- Consider the following code:
  - if (**a** or **b**) and **c** then

- The condition/decision criteria will be satisfied by the following set of tests:
  - a=true, b=true, c=true
  - a=false, b=false, c=false

Modified Condition/Decision Coverage (MC/DC)

- Every condition in the decision independently affects the decision's outcome.

```
If ((a)   &   (b)   |   (c)) then…
```

true *false* *true* *false*

Change the value of each condition individually while keeping all other conditions constant.

- This criterion extends condition/decision criteria with requirements that each condition should affect the decision outcome independently.

MODIFIED CONDITION/DECISION COVERAGE (MC/DC)

If (A and B) then…

- (1) Create truth table for conditions.
- (2) Extend truth table so that it indicated which test cases can be used to show the independence of each condition.

A B	result
T T	T
T F	F
F T	F
F F	F

number	A B	result	A	B
1	T T	T	3	2
2	T F	F		1
3	F T	F	1	
4	F F	F		

CREATING TEST CASES CONT'D

number	A B	result	A	B
1	T T	T	3	2
2	T F	F		1
3	F T	F	1	
4	F F	F		

- Show independence of **A**:
  - Take 1 + 3
- Show independence of **B**:
  - Take 1 + 2
- Resulting test cases are
  - 1 + 2 + 3
  - (T , T) + (T , F) + (F , T)

- **(T,T)** test is required as it is the only one that returns true
- **(F,T)** test is required as it is the only test that changes the value of only ***A*** and also changes the decision's outcome, thereby establishing the independence of ***A*** In similar fashion
- **(T,T)** and (T,F) tests are required to show the independence of ***B*****.**
- Test set {(T,T), (T,F), (F,T)} satisfies MC/DC for the expression ***A*** and ***B*****.**

More Advanced Example

If (A and (B or C)) then…

number	ABC	result	A	B	C
1	TTT	T	5		
2	TTF	T	6	4	
3	TFT	T	7		4
4	TFF	F		2	3
5	FTT	F	1		
6	FTF	F	2		
7	FFT	F	3		
8	FFF	F			

Ta = {(1, 5) (2, 6) (3, 7)}

Tb = {(2, 4)}

Tc = {(4, 3)}

T T F

T F T

T F F

One of {6 or 7} to cover A

Another MC/DC Example

- Consider the following code:
  - if (**a** or **b**) and **c** then
- The condition/decision criteria will be satisfied by the following set of tests:
  - a=true, b=true, c=true
  - a=false, b=false, c=false
- However, the above tests set will not satisfy modified condition/decision coverage, since in the first test, the value of 'b' and in the second test the value of 'c' would not influence the output.
- So, the following test set is needed to satisfy MC/DC:
  - a = false, b = false, c = true F
  - a = true, b = false, c = true T
  - a = false, b = true, c = true T
  - a = false, b = true, c = false F

Path Coverage

- In order to achieve path coverage you need a set of test cases that executes every possible route through a unit of code.
  - Sometimes, path coverage is impractical for all
- The objective of path testing is to ensure that the set of test cases is such that each path through the program is executed at least once.
  - The starting point for path testing is a program flow graph that shows nodes representing program decisions and arcs representing the flow of control

PATH COVERAGE

- How many paths are there in the following unit of code?

```
if (a)
    stmt1;
if (b)
    stmt2;
if (c)
    stmt3;
```

Answer: 2^3 or 8

a=f,b=f,c=f
a=f,b=f,c=t
a=f,b=t,c=f
a=f,b=t,c=t
a=t,b=f,c=f
a=t,b=f,c=t
a=t,b=t,c=f
a=t,b=t,c=t

- If there is a loop in your flow chart and you can't "unroll" the loop (calculate an upper bound on the number of times through the loop) its impossible to achieve path coverage.

Path Coverage

- What inputs (test cases) are needed to achieve path coverage on the following code fragment?

```
procedure AddTwoNumbers()
    top: print "Enter two numbers";
        read a;
        read b;
        print a+b;
        if (a != -1) goto top;
```

- There is no finite number of test cases that will achieve path coverage.

Control Flow Testing

- Control flow testing uses the control structure of a program to develop the test cases for the program.
- The test cases are developed to sufficiently cover the whole control structure of the program.
- The control structure of a program can be represented by the control flow graph of the program.
  - A test case is a complete path from the entry node to the exit node of a control flow graph.

Control Flow Graph

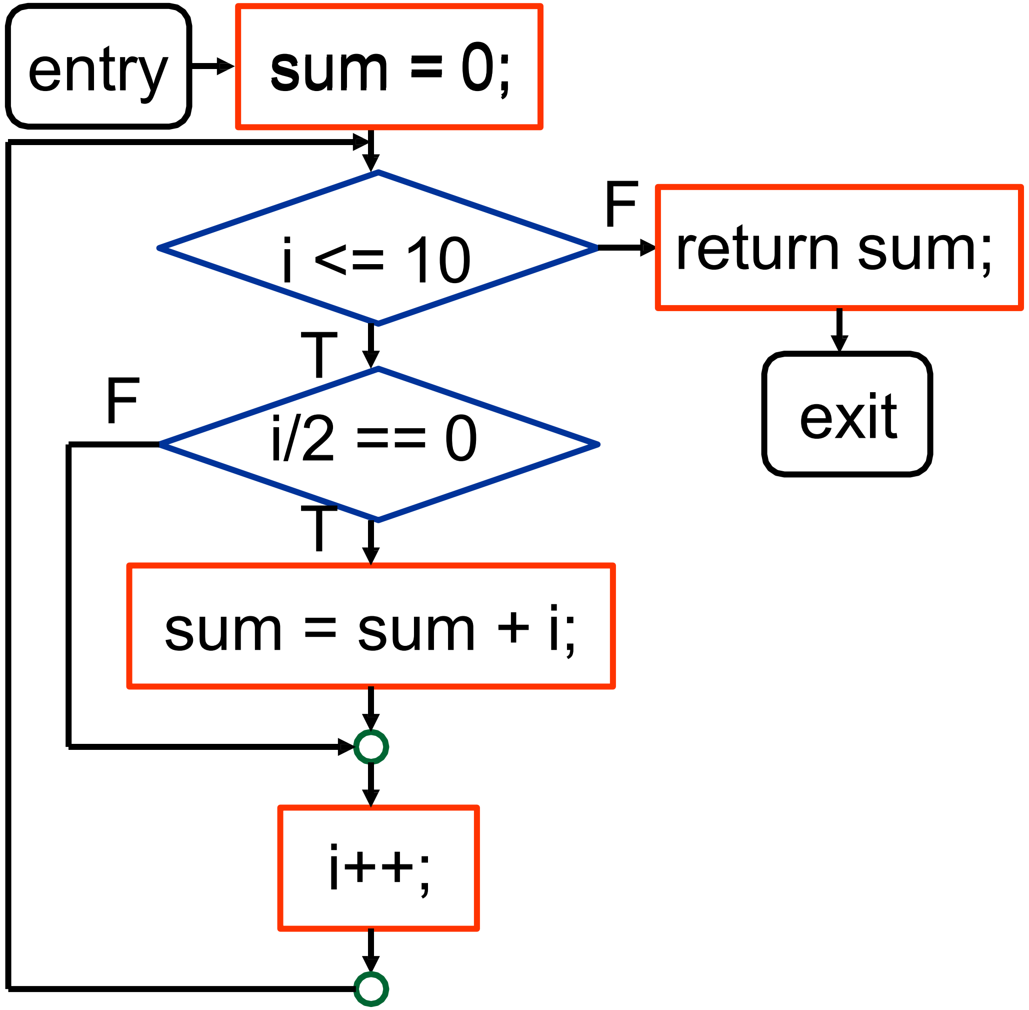
- The control flow graph G = (N, E) of a program consists of a set of nodes N and a set of edges E.
  - Each node represents a set of program statements. There are five types of nodes.
  - There is an edge from node n_1 to node n_2 if the control may flow from the last statement in n_1 to the first statement in n_2.

Control Flow Graph: Nodes

- There is a unique entry node and a unique exit node.
- A decision node contains a conditional statement that creates 2 or more control branches (e.g. if or switch statements).
- A merge node usually does not contain any statement and is used to represent a program point where multiple control branches merge.
- A statement node contains a sequence of statements. The control must enter from the first statement and exit from the last statement.

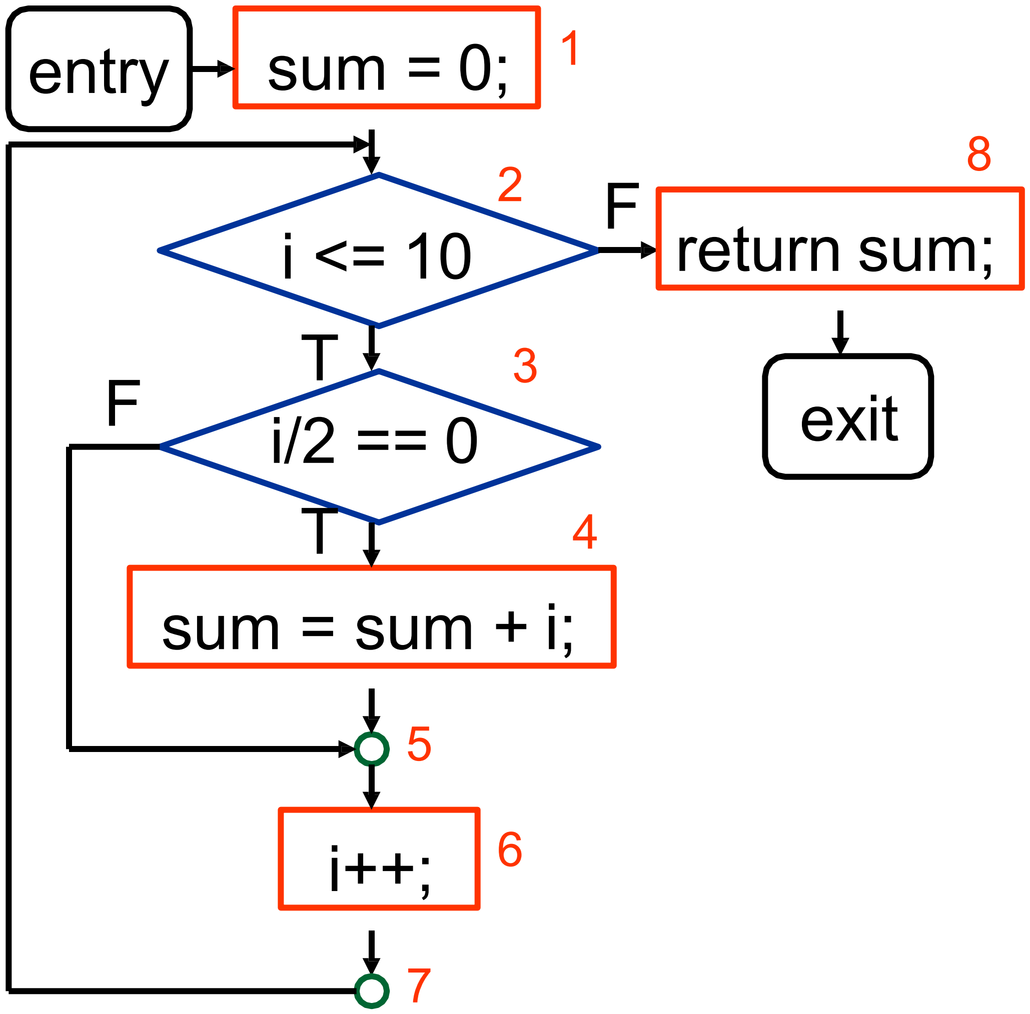
Control Flow Graph: An Example

```
int evensum(int i) {
  int sum = 0;
  while (i <= 10) {
      if (i/2 == 0)
         sum = sum + i;
      i++;
  }
return sum;
}
```



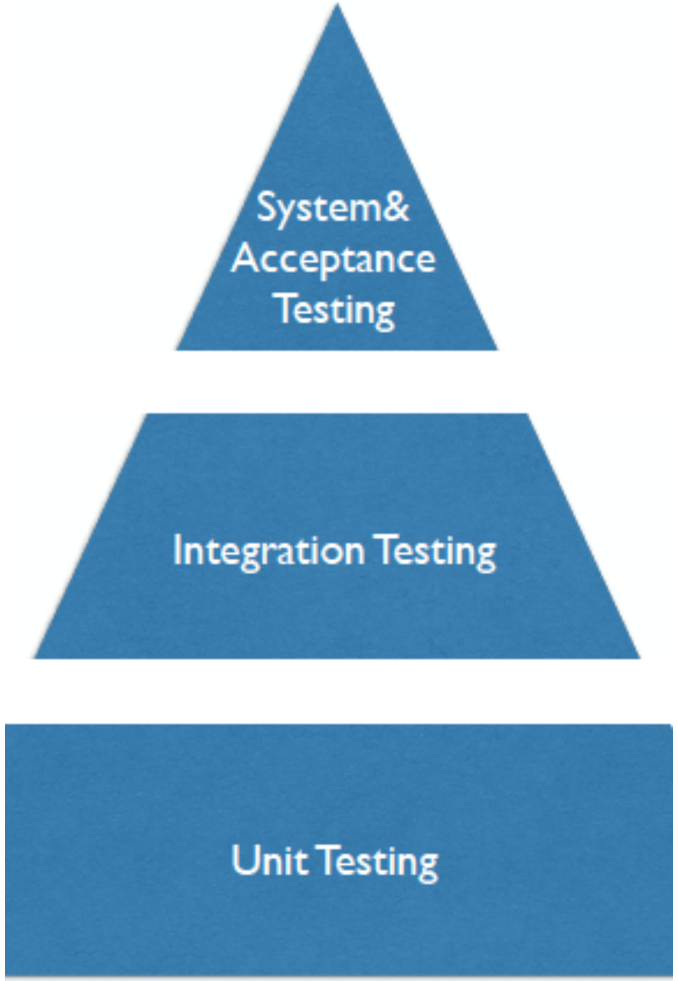
Coverages

- Statement Coverage
  - Every statement in the program has been executed at least once.
  - 1 → 2 → 3 → 4 → 5 → 6 → 7 → 2 → 8
- Decision Coverage
  - Every statement in the program has been executed at least once, and every decision in the program has taken all possible outcomes at least once.
  - 1 → 2 → 3 → 5 → 6 → 7 → 2 → 3 → 4 → 5 → 6 → 7 → 2 → 8



GRANULARITY OF TESTING

- Unit testing
  - Test for each single module
- Integration testing
  - Test the interaction between modules
- System testing
  - Test the system as a whole, by developers
- Acceptance testing
  - Validate the system against user requirements by customers with formal test cases



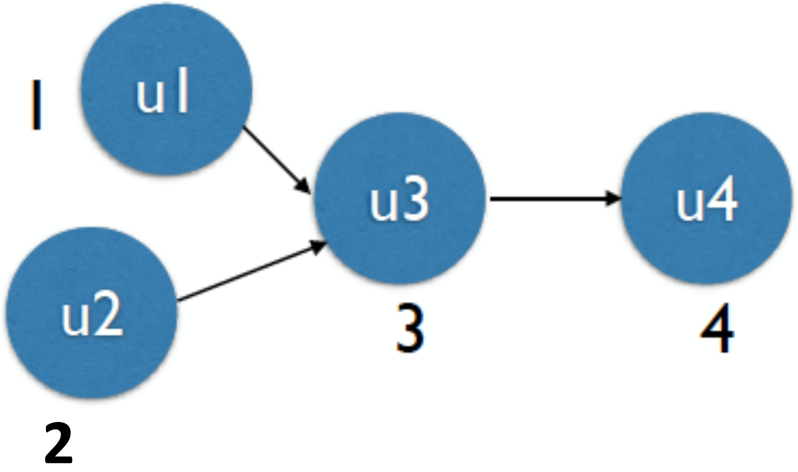
UNIT TESTING

- Testing of basic module of the software
  - A function, a class, etc.

- Typical problems revealed
  - Local data structures
  - Algorithms
  - Boundary conditions
  - Error handling

Why & How

- Why Unit testing
  - Divide-and-conquer approach
    - Split system into units
    - Debug unit individually
    - Narrow down places where bugs can be



- How to do Unit testing
  - Build systems in layers
    - Starts with classes that don't depend on others.
    - Continue testing building on already tested classes.

UNIT TEST FRAMEWORK

- xUnit
  - Created by Kent Beck in 1989
  - This is the same guy who invented TDD
  - The first one was sUnit (for smalltalk)

- JUnit
  - The most popular xUnit framework
  - There are about 70 xUnit frameworks for corresponding languages

PROGRAM TO TEST

```
public class IMath {

    /**
     * Returns an integer to the square root of x (discarding the fractional parts)
     */
    public int isqrt(int x) {
        int guess = 1;
        while (guess * guess < x) {
            guess++;
        }
        return guess;
    }
}
```

CONVENTIONAL TESTING

```
/** A class to test the class IMath. */
public class IMathTestNoJUnit {
   /** Runs the tests. */
   public static void main(String[] args) {
       printTestResult(0);
       printTestResult(1);
       printTestResult(2);
       printTestResult(3);
       printTestResult(100);
  }
 private static void printTestResult(int arg) {
       IMath tester=new IMath();
       System.out.print("isqrt(" + arg + ") ==>  ");
       System.out.println(tester.isqrt(arg));
  }
}
```

Conventional Test Output

- What does this say about the code? Is it right?
- What's the problem with this kind of test output?

```
Isqrt(0) ==> 1
Isqrt(1) ==> 1
Isqrt(2) ==> 2
Isqrt(3) ==> 2
Isqrt(100) ==> 10
```

SOLUTION?

- Automatic verification by testing program
  - Can write such a test program by yourself, or
  - Use testing tool supports, such as JUnit

- JUnit
  - A simple, flexible, easy-to-use, open-source, and practical unit testing framework for Java.
  - Can deal with a large and extensive set of test cases.
  - Refer to www.junit.org

TESTING WITH JUnit (1)

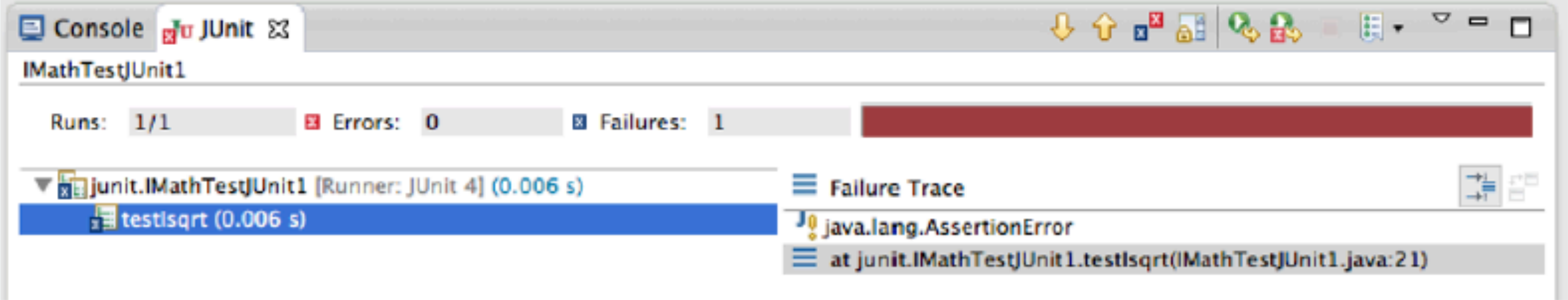
```
import org.junit.Test;
import static org.junit.Assert.*;

/** A JUnit test class to test the class IMath. */
public class IMathTestJUnit1 {

    /** A JUnit test method to test isqrt. */
    @Test
    public void testIsqrt() {
        IMath tester = new IMath();
        assertTrue(0 == tester.isqrt(0));
        assertTrue(1 == tester.isqrt(1));
        assertTrue(1 == tester.isqrt(2));
        assertTrue(1 == tester.isqrt(3));
        assertTrue(10 == tester.isqrt(100));
    }

    /** Other JUnit test methods*/
}
```

Test driver

Test case

Test oracle

JUnit Execution (1)



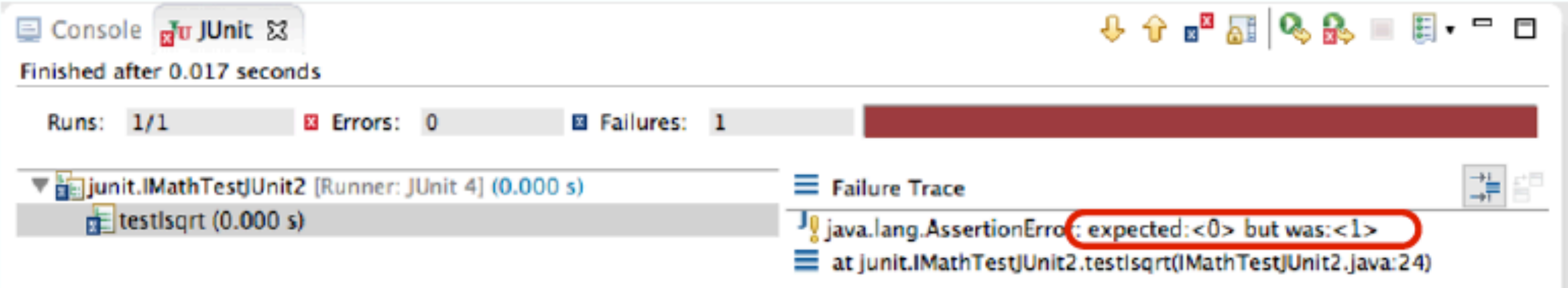
Not so good, why?

TESTING WITH JUnit (2)

```
import org.junit.Test;
import static org.junit.Assert.*;

/** A JUnit test class to test the class IMath. */
public class IMathTestJUnit2 {

    /** A JUnit test method to test isqrt. */
    @Test
    public void testIsqrt() {
        IMath tester = new IMath();
        assertEquals(0, tester.isqrt(0));
        assertEquals(1, tester.isqrt(1));
        assertEquals(1, tester.isqrt(2));
        assertEquals(1, tester.isqrt(3));
        assertEquals(10, tester.isqrt(100));
    }

    /** Other JUnit test methods*/
}
```

```
assertTrue(0 == tester.isqrt(0));
assertTrue(1 == tester.isqrt(1));
assertTrue(1 == tester.isqrt(2));
assertTrue(1 == tester.isqrt(3));
assertTrue(10 == tester.isqrt(100));
```

JUnit EXECUTION (2)

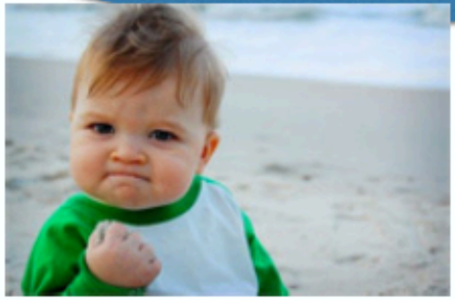


- Why now better error info?
  - assertTrue(0==tester.isqrt(0))
  - assertEquals(0, tester.isqrt(0))

detailed result is abstracted into boolean before passed to JUnit

the detailed result is passed to JUnit

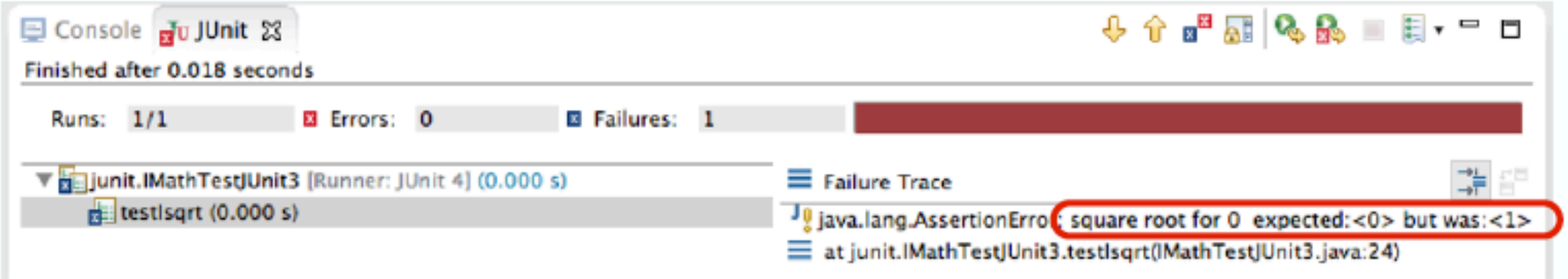
Can we make it better?



Testing with JUnit (3)

```
import org.junit.Test;
import static org.junit.Assert.*;

/** A JUnit test class to test the class IMath. */
public class IMathTestJUnit3 {

    /** A JUnit test method to test isqrt. */
    @Test
    public void testIsqrt() {
        IMath tester = new IMath();
        assertEquals("square root for 0 ", 0, tester.isqrt(0));
        assertEquals("square root for 1 ", 1, tester.isqrt(1));
        assertEquals("square root for 2 ", 1, tester.isqrt(2));
        assertEquals("square root for 3 ", 1, tester.isqrt(3));
        assertEquals("square root for 100 ", 10, tester.isqrt(100));
    }

    /** Other JUnit test methods*/
}
```

JUnit EXECUTION (3)



Still have problems, why?

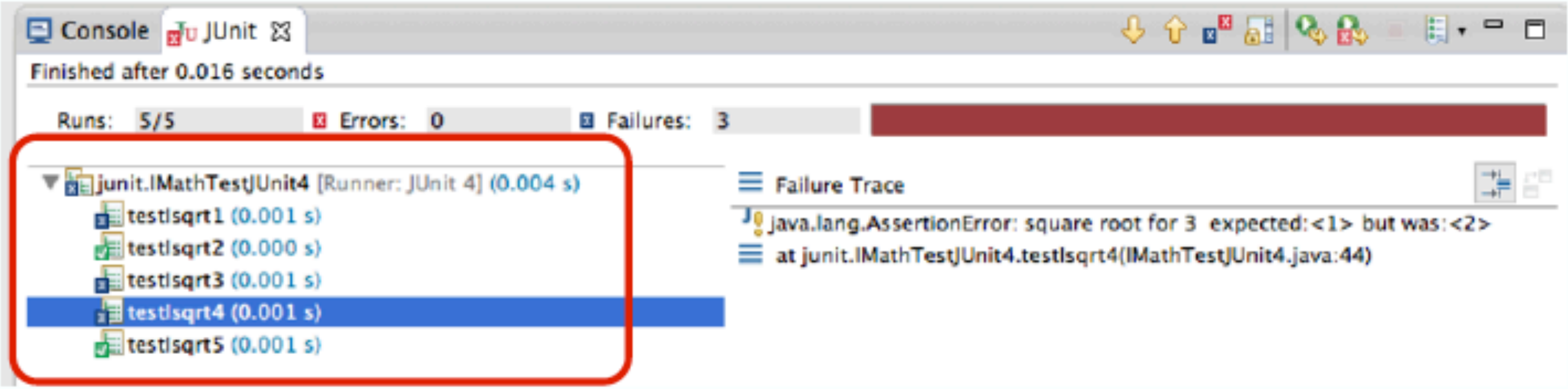
We only see the error info for the first input...

TESTING WITH JUnit (4)

```
public class IMathTestJUnit4 {
    private IMath tester;
    @Before /** Setup method executed before each test */
    public void setup(){
        tester=new IMath();
    }

    @Test /** JUnit test methods to test isqrt. */
    public void testIsqrt1() {
        assertEquals("square root for 0 ", 0, tester.isqrt(0));
    }
    @Test
    public void testIsqrt2() {
        assertEquals("square root for 1 ", 1, tester.isqrt(1));
    }
    @Test
    public void testIsqrt3() {
        assertEquals("square root for 2 ", 1, tester.isqrt(2));
    }
    ...
```

Test fixture

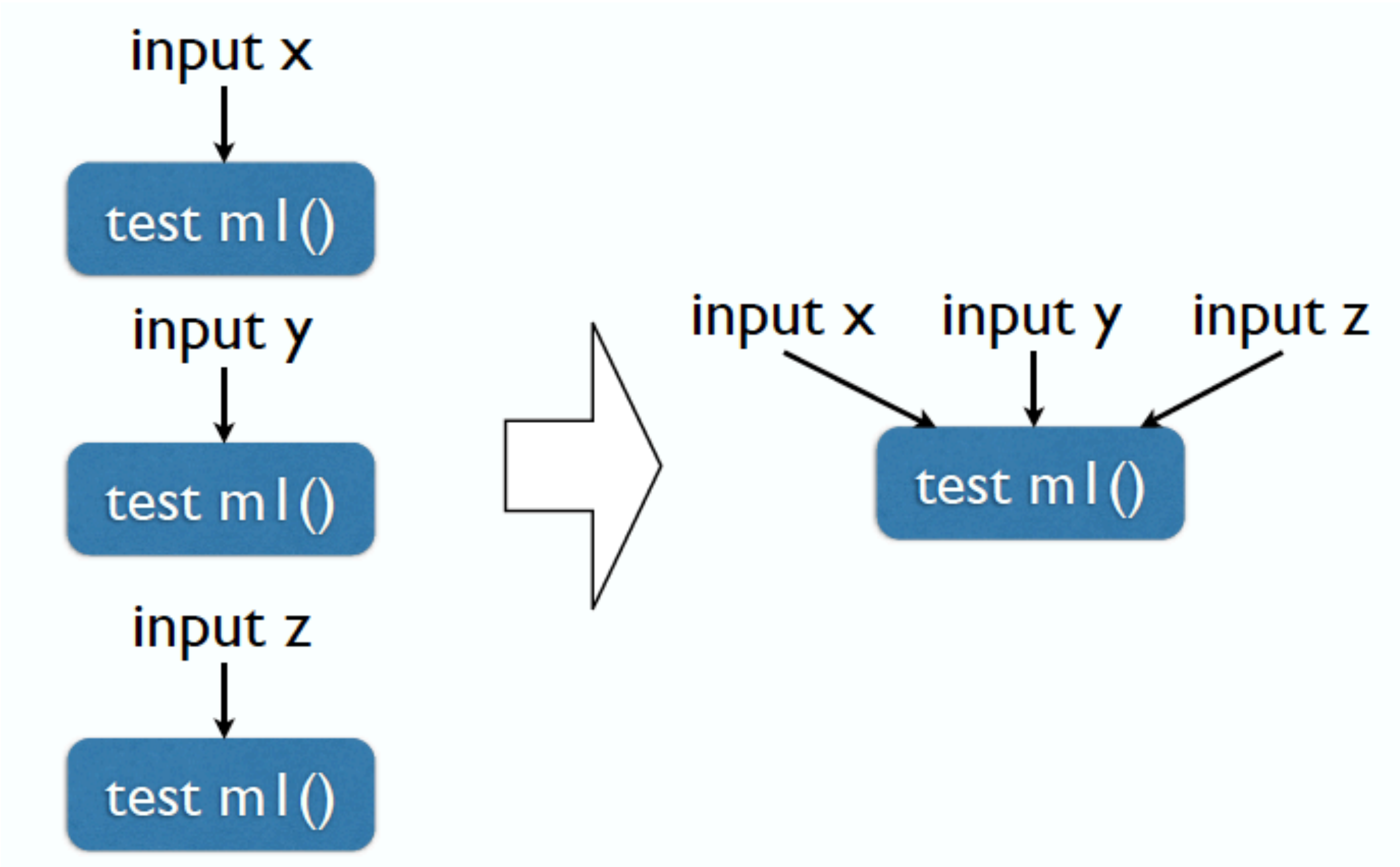
JUnit EXECUTION (4)



Still may have trouble, why?

We need to write so many similar test methods...

Parameterized Tests: Illustration



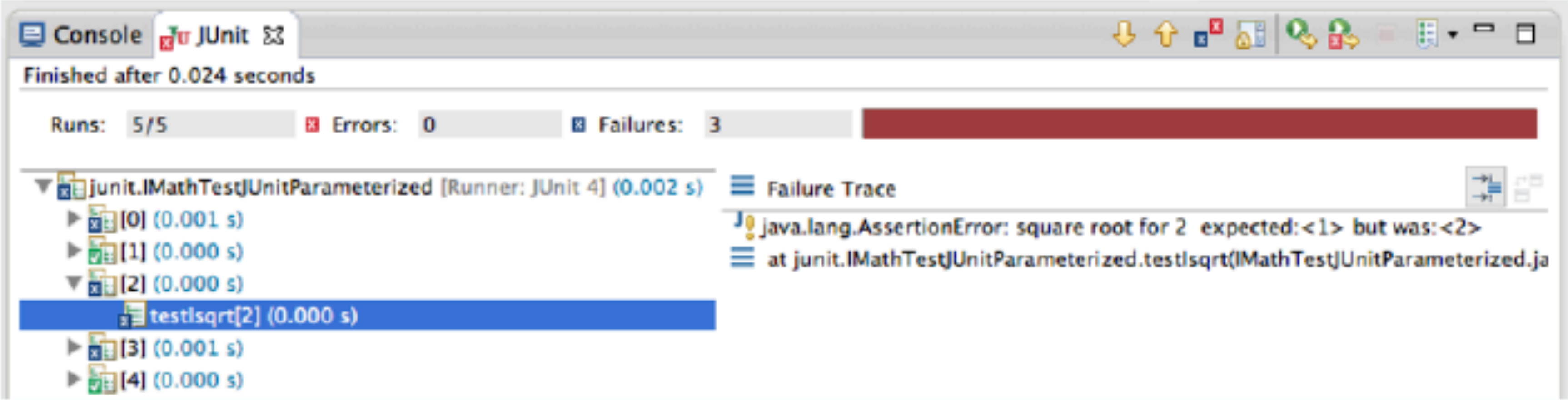
Testing with JUnit: Parameterized Tests

Indicate this is a parameterized test class

To store input-output pairs

```
@RunWith(Parameterized.class)
public class IMathTestJUnitParameterized {
    private IMath tester;
    private int input;
    private int expectedOutput;
    /** Constructor method to accept each input-output pair*/
    public IMathTestJUnitParameterized(int input, int expectedOutput) {
        this.input = input;
        this.expectedOutput = expectedOutput;
    }
    @Before /** Set up method to create the test fixture */
    public void initialize() {tester = new IMath(); }
    @Parameterized.Parameters /** Store input-output pairs, i.e., the test data */
    public static Collection<Object[]> valuePairs() {
        return Arrays.asList(new Object[][] { { 0, 0 }, { 1, 1 }, { 2, 1 }, { 3, 1 }, { 100, 10 } });
    }
    @Test /** Parameterized JUnit test method*/
    public void testIsqrt() {
        assertEquals("square root for " + input + " ", expectedOutput, tester.isqrt(input));
    }
```

JUnit Execution: Parameterized Tests



Note that not all tests can be abstract into parameterized tests

A Counter Example

```
public class ArrayList {
    ...
    /** Return the size of current list */
    public int size() {
        ...
    }
    /** Add an element to the list */
    public void add(Object o) {
        ...
    }
    /** Remove an element from the list */
    public void remove(int i) {
        ...
    }
}
```

```
public class ListTestJUnit {
    List list;
    @Before /** Set up method to create the test fixture */
    public void initialize() {
        list = new ArrayList();
    }
    /** JUnit test methods*/
    @Test
    public void test1() {
        list.add(1);
        list.remove(0);
        assertEquals(0, list.size());
    }
    @Test
    public void test2() {
```

These tests cannot be abstract into parameterized tests, because the tests contains different method invocations

JUnit Annotations

Annotation	Description
@Test	Identify test methods
@Test (timeout=100)	Fail if the test takes more than 100ms
@Before	Execute before each test method
@After	Execute after each test method
@BeforeClass	Execute before each test class
@AfterClass	Execute after each test class
@Ignore	Ignore the test method

JUnit Assertions

Assertion	Description
fail([msg])	Let the test method fail, optional msg
assertTrue([msg], bool)	Check that the boolean condition is true
assertFalse([msg], bool)	Check that the boolean condition is false
assertEquals([msg], expected, actual)	Check that the two values are equal
assertNull([msg], obj)	Check that the object is null
assertNotNull([msg], obj)	Check that the object is not null
assertSame([msg], expected, actual)	Check that both variables refer to the same object
assertNotSame([msg], expected, actual)	Check that variables refer to different objects

MORE ON JUNIT?

- Homepage:
  - www.junit.org
- Tutorials
  - http://www.vogella.com/tutorials/JUnit/article.html
  - http://www.tutorialspoint.com/junit/
  - https://courses.cs.washington.edu/courses/cse143/11wi/eclipse-tutorial/junit.shtml

MANTRA

- Develop test cases before you code!
- Test as you go!
- Test always and often!