ERIK JONSSON SCHOOL OF ENGINEERING AND COMPUTER SCIENCE
The University of Texas at Dallas

CE/CS/SE 3354
Software Engineering

Software Testing
JUnit

This Class

- Software Testing
  - Motivation
  - Concepts
  - Granularity
  - Unit Testing
- JUnit

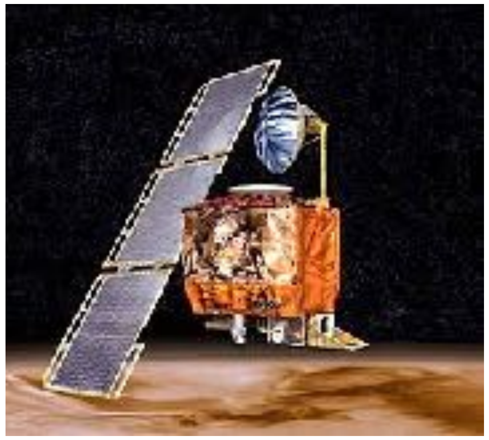
Why Testing?

- Errors can happen in any engineering discipline
- Software is one of the most error-prone products of all engineering areas
  - Requirements are often vague
  - Software can be really complex, undecidable problems are everywhere
  - Result

    Almost all software in the market has some number of bugs (we will see that later)

Why Testing? Examples

- Mars Climate Orbiter ($165M, 1998)
  - Sent to Mars to relay signal from Mars Lander
  - Smashed to the planet: failing to convert between different metric standards



- Shooting down of A300 (290 death, 1988)
  - US CG-49 shoot down a Airbus A300
  - Misleading output of the tracking software

- THERAC-25 Radiation Therapy (1985)
  - 2 cancer patients received fatal overdoses
  - Miss-handling of race condition of the software in the equipment

Why Testing? Numbers

- On average, 1-5 bugs per KLOC (thousand lines of code)
  - In mature software (more than 10 bugs in prototypes)

- 35MLOC
- 63K known bugs at the time of release
- 2 bugs per KLOC

- $59.5B loss due to bugs in US 2002 (estimation by NIST)
- It is not feasible to remove all bugs
  - But try to reduce critical bugs

Approaches to Reduce Bugs

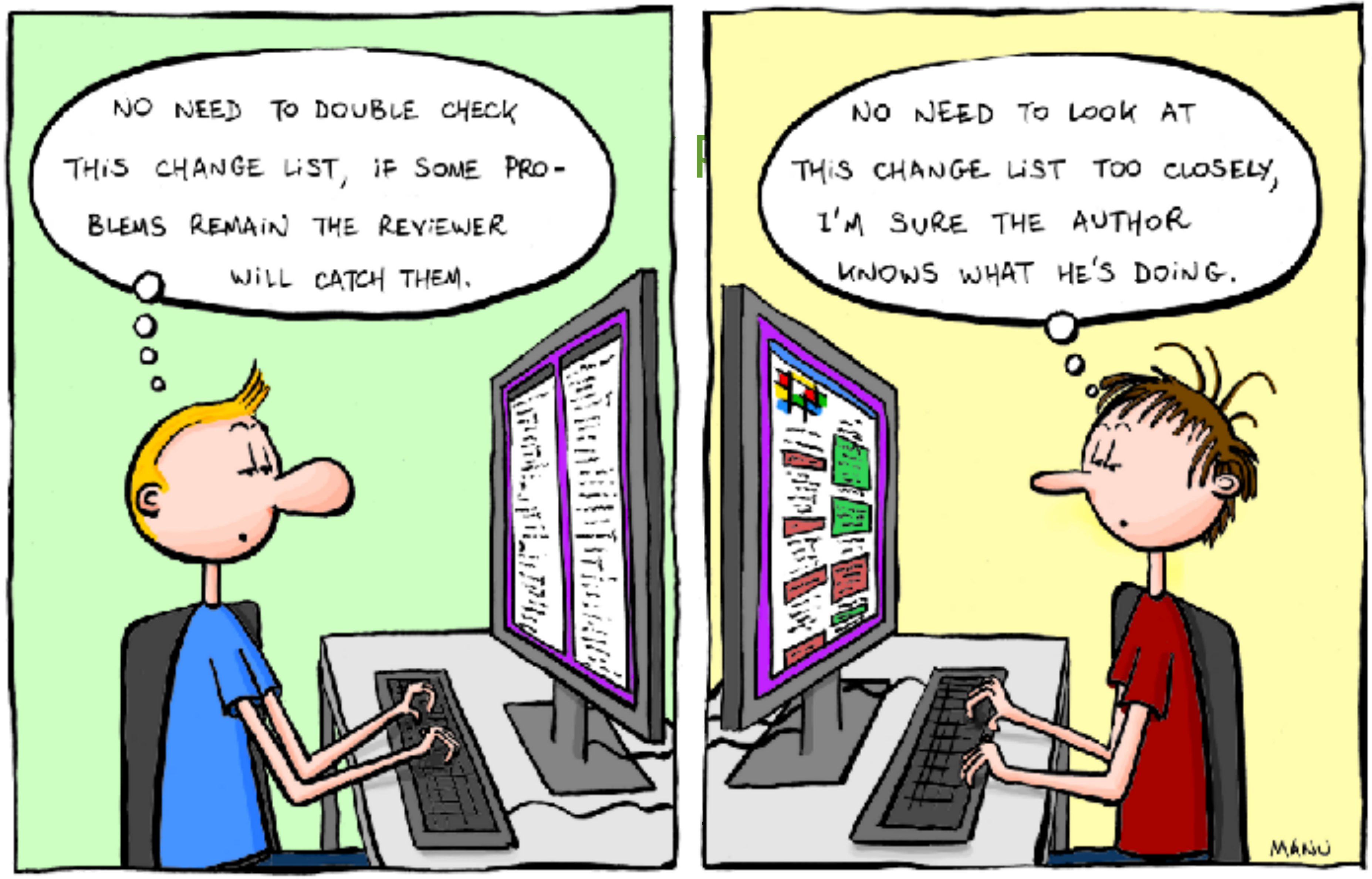
- Manual code review
  - Manually review the code to detect faults
  - Limitations:

    Hard to evaluate your progress

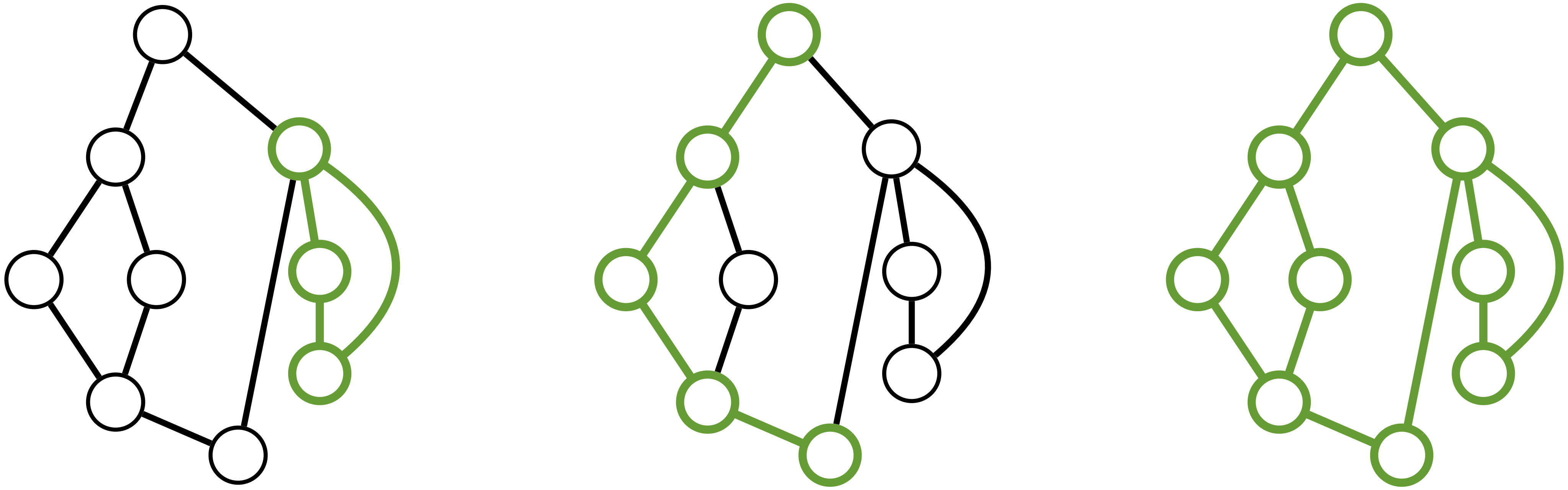
    Can miss many bugs

Approaches to Reduce Bugs

- Manual code review
  - Manually review the code to detect faults


NO NEED TO DOUBLE CHECK THIS CHANGE LIST, IF SOME PROBLEMS REMAIN THE REVIEWER WILL CATCH THEM.
NO NEED TO LOOK AT THIS CHANGE LIST TOO CLOSELY, I'M SURE THE AUTHOR KNOWS WHAT HE'S DOING.
MANU

Automated Approaches to Reduce Bugs

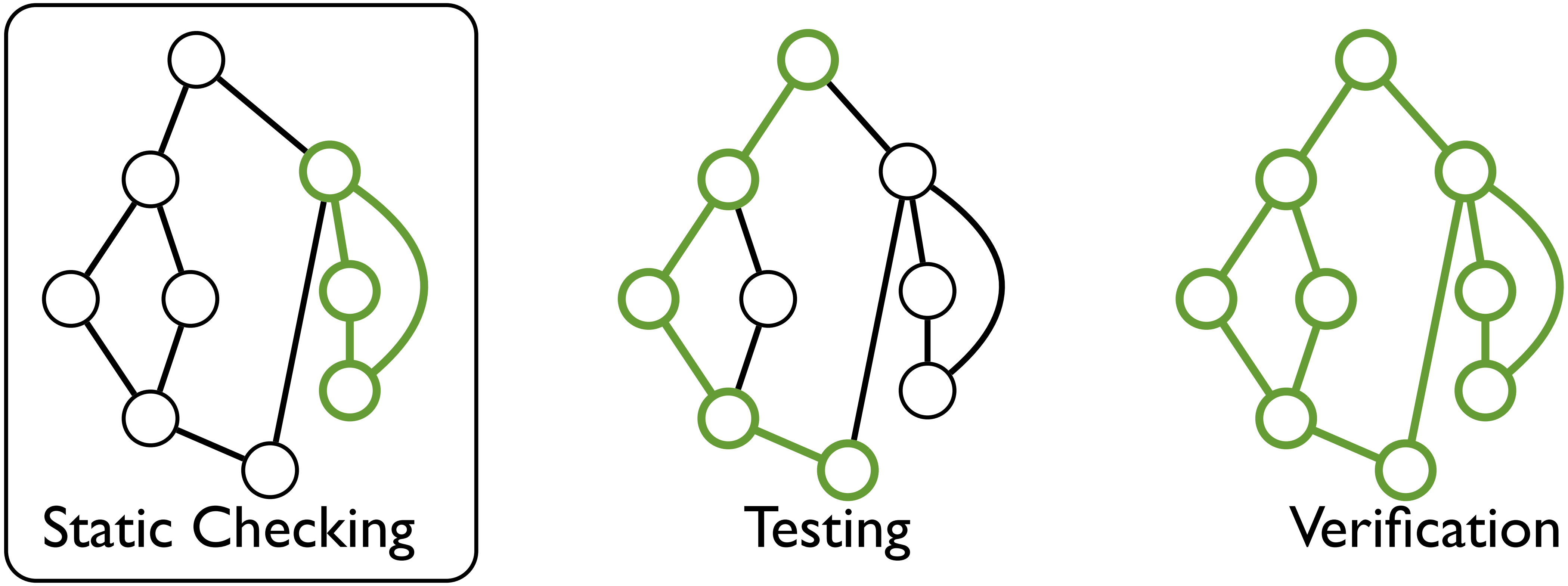


Static Checking

Testing

Verification

Automated Approaches to Reduce Bugs



Static Checking

Testing

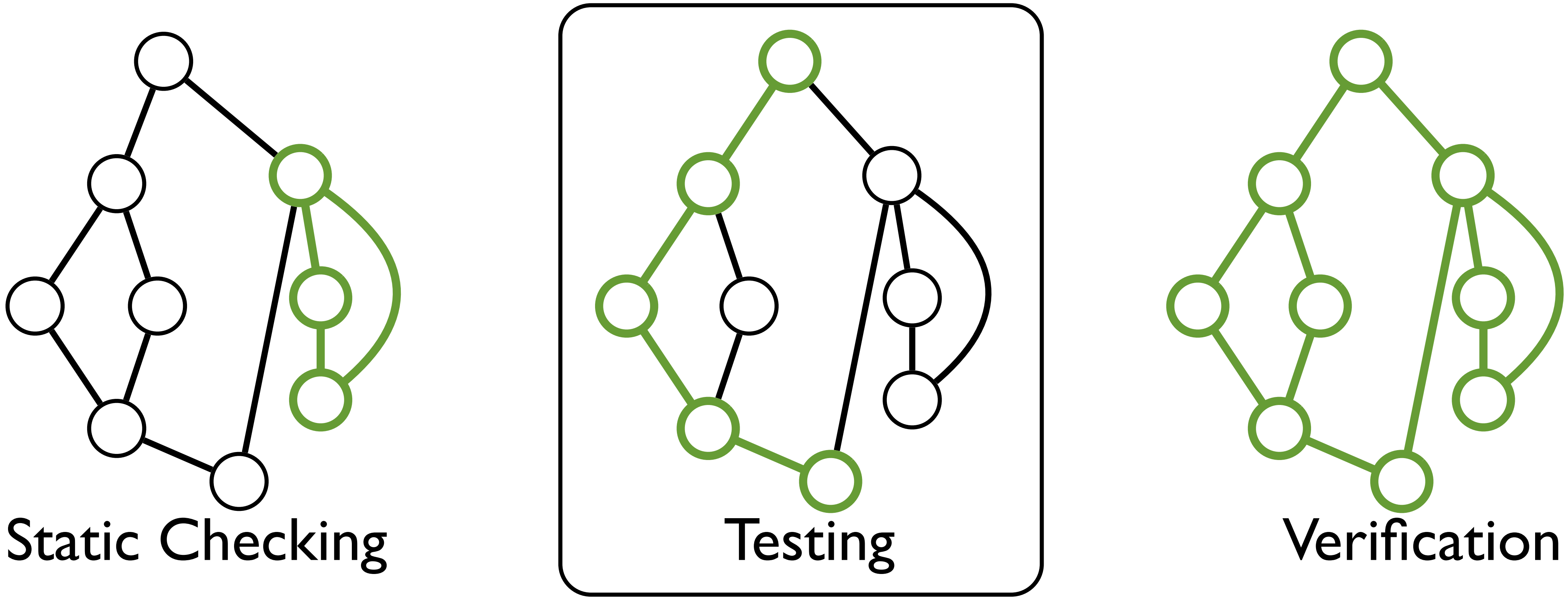
Verification

- Static checking
  - Identify specific problems (e.g., memory leak) in the software by scanning suspicious patterns from the code
  - Limitations

    Limited problem types

    False positive

Automated Approaches to Reduce Bugs



Static Checking

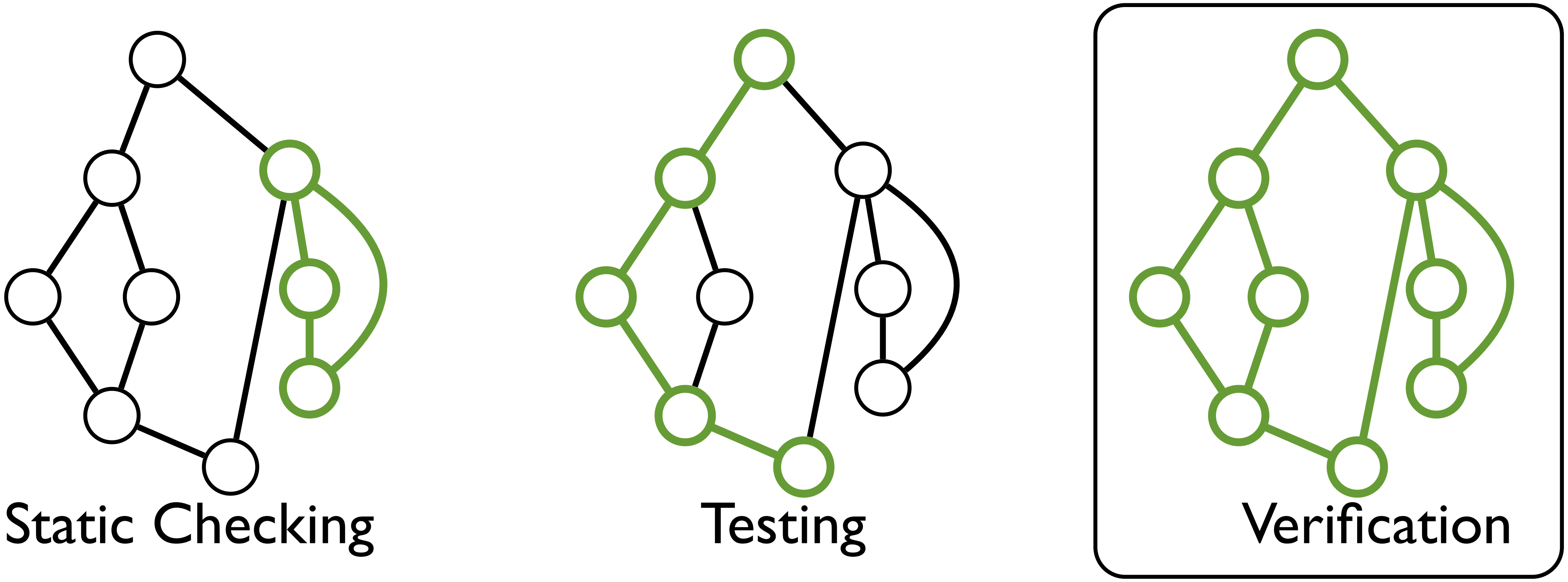
Testing

Verification

- Testing
  - Feed input to software and run it to see whether its behavior is as expected
  - Limitations

    Impossible to cover all possible execution

Automated Approaches to Reduce Bugs



Static Checking

Testing

Verification

- ◉ Formal Verification
  - Consider all the possible program executions, and formally prove that the program is correct or not
  - Limitations

    Difficult to have a formal specification

    Most real-world programs are too expensive to prove

The Most Widely Used Approach

Testing!

"50% of my employees are testers, and the rest spends 50% of their time testing"

Why Testing?

- Testing vs. code review:
  - More reliable than code review
- Testing vs. static checking:
  - Less false positive and applicable to more problems
- Testing vs. formal verification:
  - More scalable and applicable to more programs
- You get what you pay (linear rewards)
  - While the others are not!

Testing: Concepts

- Test case
- Test fixture
- Test suite
- Test script
- Test driver
- Test result
- Test coverage

Testing: Concepts

- Test case (or, simply test)
  - An execution of the software with a given test input
  - Include:

    Input values

    Sometimes include execution steps

    Expected outputs

Testing: Concepts

- Test fixture: a fixed state of the software under test used as a baseline for running tests; also known as the test context, e.g.,
  - Loading a database with a specific, known set of data
  - Preparation of input data and set-up/creation of fake or mock objects

Testing: Concepts

- Test suite
  - A collection of test cases
  - Usually these test cases share similar pre-requisites and configuration
  - Usually can be run together in sequence
  - Different test suites for different purposes

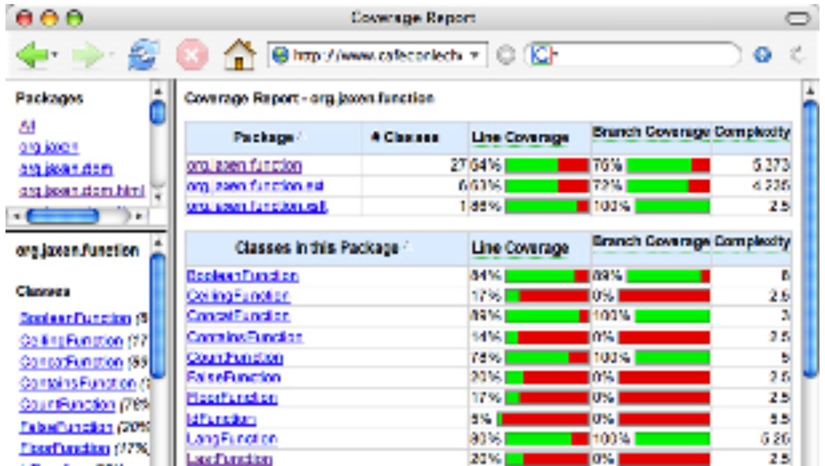
    Certain platforms, Certain feature, performance, …
- Test Script
  - A script to run a sequence of test cases or a test suite automatically

Testing: Concepts

- Test Driver
  - A software framework that can load a collection of test cases or a test suite
  - It can also handle the configuration and comparison between expected outputs and actual outputs
- Test Coverage
  - A measurement to evaluate the percentage of code tested

    Statement coverage

    Branch coverage, ...



Coverage Report

http://www.cafeconleche

Packages
All
org.jaxen
org.jaxen.dom
org.jaxen.dom.html

org.jaxen.function

Classes
BooleanFunction (8
CeilingFunction (17
ConcatFunction (89
ContainsFunction (1
CountFunction (78%
FalseFunction (20%
FloorFunction (17%

Coverage Report - org.jaxen.function

Package	# Classes	Line Coverage	Branch Coverage	Complexity
org.jaxen.function	27	64%	70%	5.273
org.jaxen.function.ext	6	63%	72%	4.235
org.jaxen.function.xslt	1	88%	100%	2.5

Classes in this Package	Line Coverage	Branch Coverage	Complexity
BooleanFunction	84%	89%	6
CeilingFunction	17%	0%	2.5
ConcatFunction	89%	100%	3
ContainsFunction	14%	0%	2.5
CountFunction	78%	100%	5
FalseFunction	20%	0%	2.5
FloorFunction	17%	0%	2.5
IdFunction	5%	0%	5.5
LangFunction	90%	100%	6.26
LastFunction	20%	0%	2.5

Granularity of Testing

- Unit Testing
    - Test of each single module
- Integration Testing
    - Test the interaction between modules
- System Testing
    - Test the system as a whole, by developers on test cases
- Acceptance Testing
    - Validate the system against user requirements, by customers with no formal test cases
- Regression Testing
    - Test a new version with old test cases

Unit Testing

- Testing of an basic module of the software
  - A function, a class, a component
- Typical problems revealed
  - Local data structures
  - Algorithms
  - Boundary conditions
  - Error handling

Why Unit Testing?

- Code isn't right if it's not tested.
- Practical
  - Most programmers rely on testing, e.g., Microsoft has 1 tester per developer
  - You could get work as a tester
- Divide-and-conquer approach
  - Split system into units
  - Debug unit individually
  - Narrow down places where bugs can be
  - Don't want to chase down bugs in other units

Why Unit Testing? (Cont.)

- Support regression testing
  - So can make changes to lots of code and know if you broke something.
  - Can make big changes with confidence.

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.
- Benefits
  - Avoid having to write mock classes
  - When testing a module, ones it depends on are reliable.

Unit Test Framework

- xUnit
  - Created by Kent Beck in 1989

    This is the same guy who invented XP and TDD

    The first one was sUnit (for smalltalk)
  - JUnit

    The most popular xUnit framework

    There are about 70 xUnit frameworks for corresponding languages

> Never in the annals of software engineering was so much owed by so many to so few lines of code
>
> --Martin Fowler

This class

- Software Testing
  - Motivation
  - Concepts
  - Granularity
  - Unit Testing
- JUnit

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

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

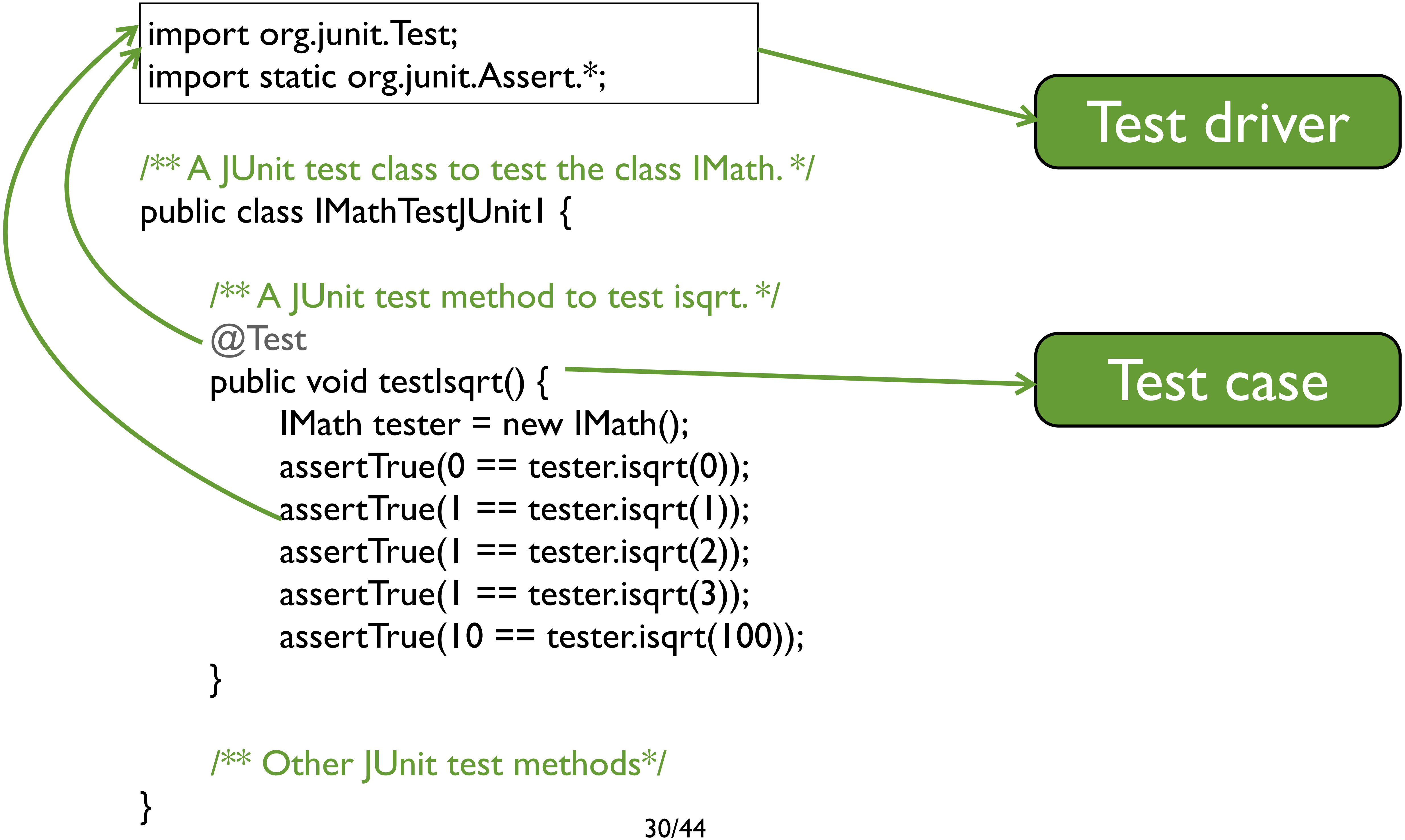
- What does this say about the code? Is it right?
- What’s the problem with this kind of test output?

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.

JUnit

Testing with JUnit (1)



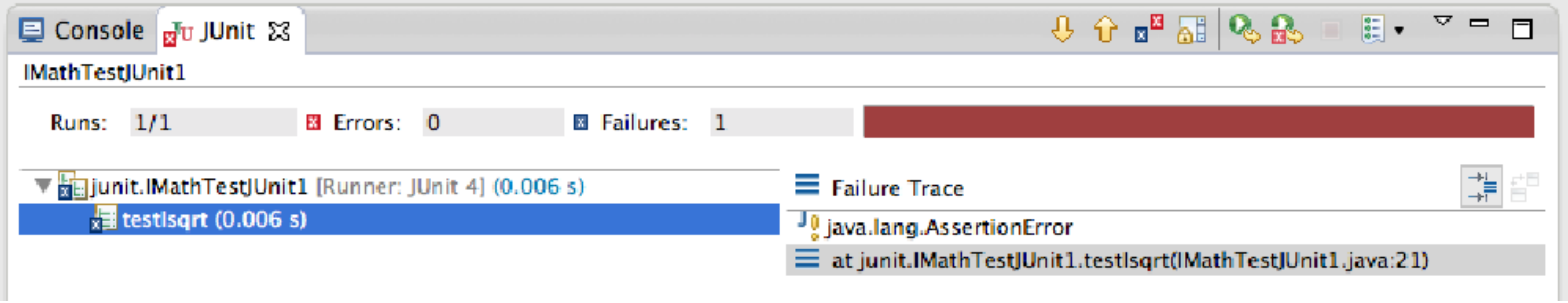
```
import org.junit.Test;
import static org.junit.Assert.*;

/** 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

JUnit Execution (1)

- Right click the JUnit class, and select "Run As" => "JUnit Test"



IMathTestJUnit1

Runs: 1/1 ☒ Errors: 0 ☒ Failures: 1

```
junit.IMathTestJUnit1 [Runner: JUnit 4] (0.006 s)
    testIsqrt (0.006 s)
```

Failure Trace

```
java.lang.AssertionError
at junit.IMathTestJUnit1.testIsqrt(IMathTestJUnit1.java:21)
```

Not so good, why?

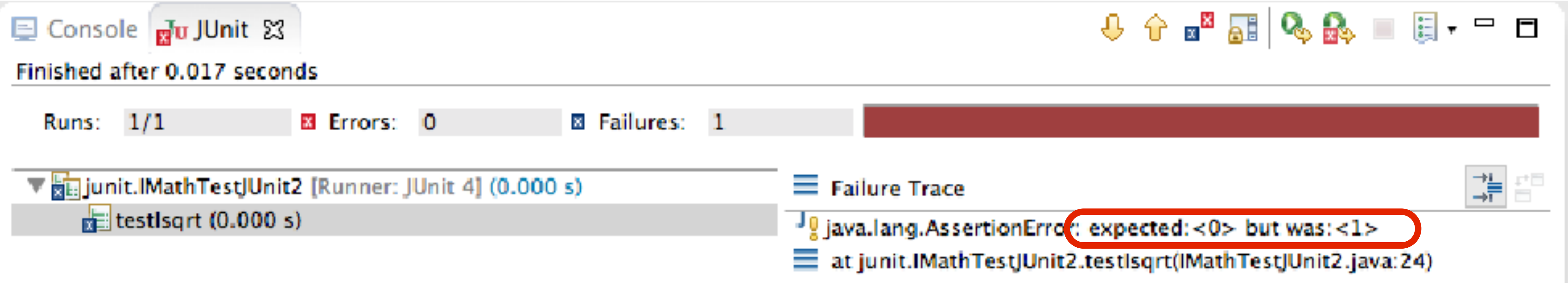
Testing with JUnit (2)

```
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)



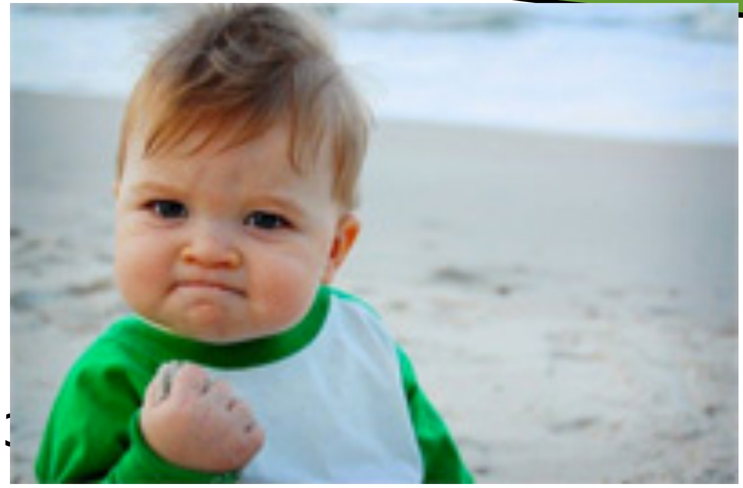
```
Finished after 0.017 seconds

Runs: 1/1        Errors: 0        Failures: 1

junit.IMathTestJUnit2 [Runner: JUnit 4] (0.000 s)
    testIsqrt (0.000 s)

Failure Trace
java.lang.AssertionError: expected:<0> but was:<1>
at junit.IMathTestJUnit2.testIsqrt(IMathTestJUnit2.java:24)
```

- 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?

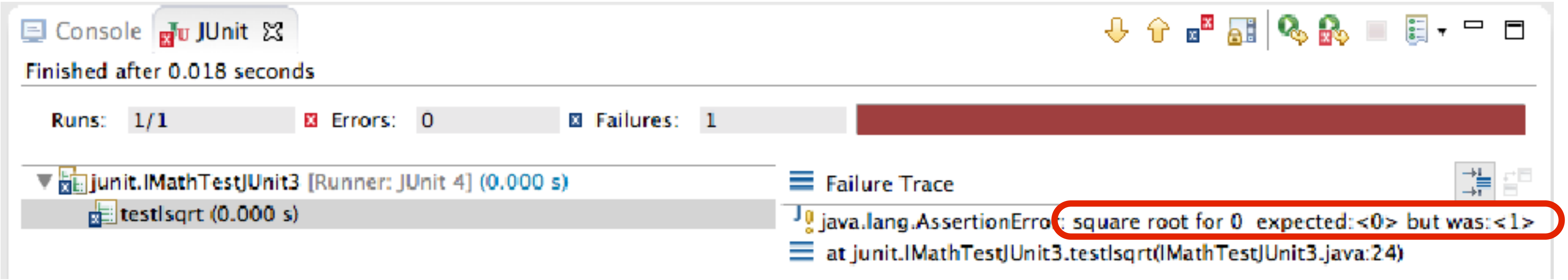


Testing with JUnit (3)

```
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)



```
Finished after 0.018 seconds

Runs:  1/1     Errors:  0     Failures:  1

▼ junit.IMathTestJUnit3 [Runner: JUnit 4] (0.000 s)
      testIsqrt (0.000 s)

java.lang.AssertionError: square root for 0  expected:<0> but was:<1>
at junit.IMathTestJUnit3.testIsqrt(IMathTestJUnit3.java:24)
```

Still have problems, why?

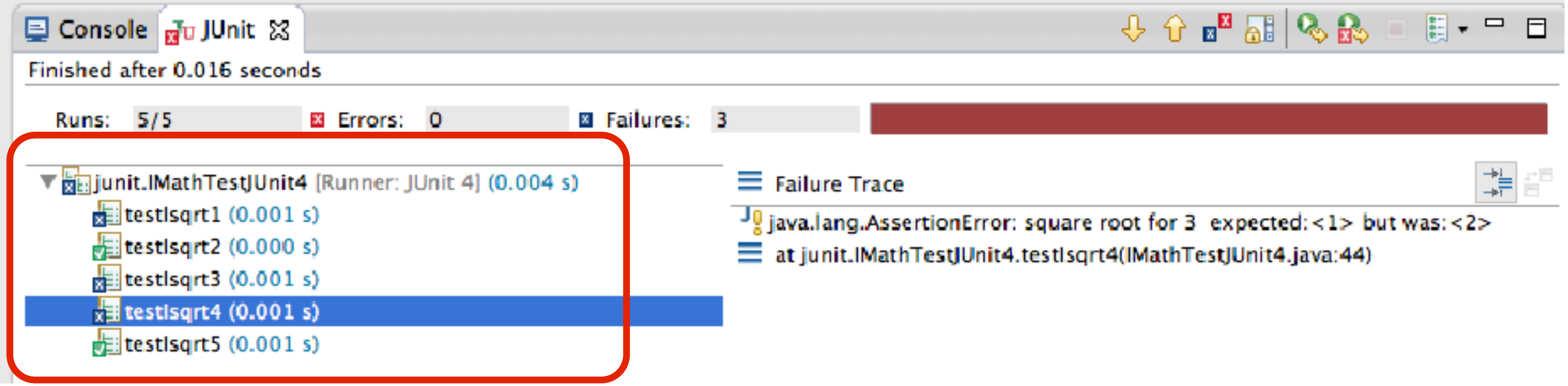
We only see the error info for the first input...

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)

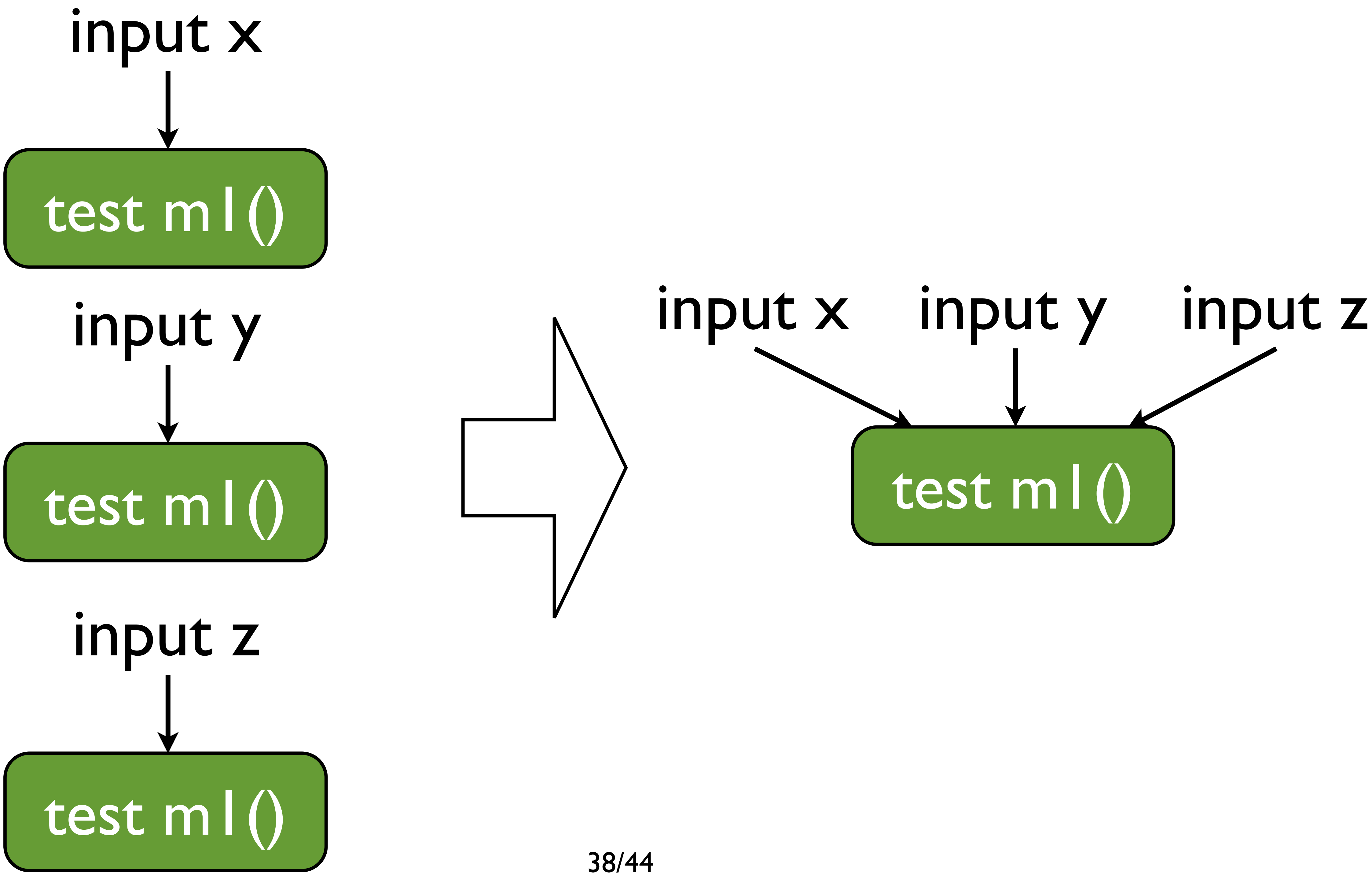


```
Finished after 0.016 seconds

Runs: 5/5        Errors: 0        Failures: 3

▼ junit.IMathTestJUnit4 [Runner: JUnit 4] (0.004 s)
    testIsqrt1 (0.001 s)
    testIsqrt2 (0.000 s)
    testIsqrt3 (0.001 s)
    testIsqrt4 (0.001 s)
    testIsqrt5 (0.001 s)

Failure Trace
java.lang.AssertionError: square root for 3  expected:<1> but was:<2>
at junit.IMathTestJUnit4.testIsqrt4(IMathTestJUnit4.java:44)
```

Still may have trouble, why?

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

Parameterized Tests: Illustration



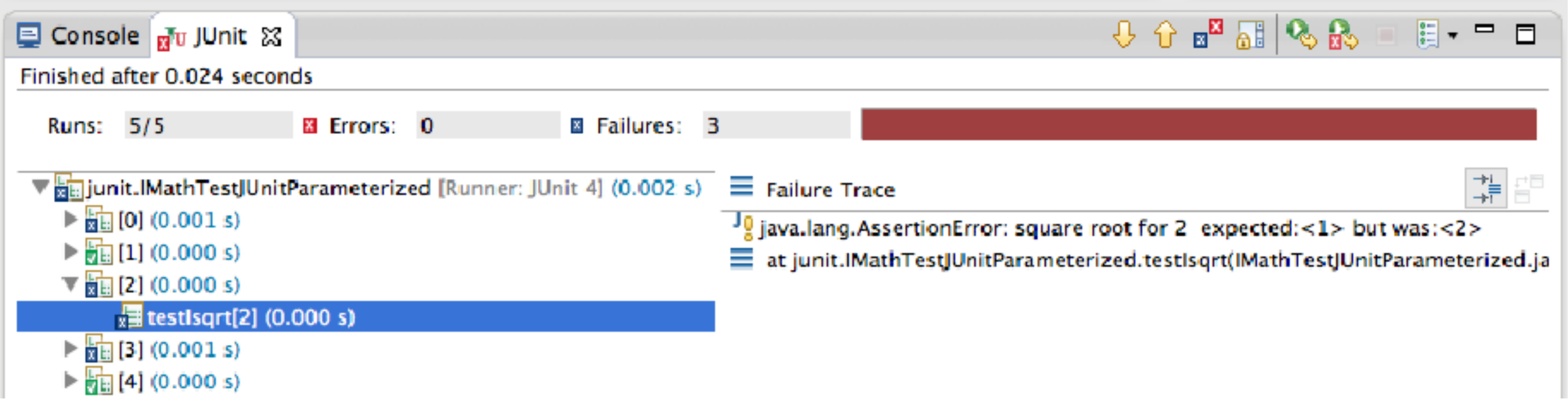
Testing with JUnit: Parameterized Tests

```
@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));
    }
}
```

Indicate this is a parameterized test class

To store input-output pairs

JUnit Execution: Parameterized Tests



```
Finished after 0.024 seconds

Runs: 5/5    Errors: 0    Failures: 3

junit.IMathTestJUnitParameterized [Runner: JUnit 4] (0.002 s)
  [0] (0.001 s)
  [1] (0.000 s)
  [2] (0.000 s)
      testIsqrt[2] (0.000 s)
  [3] (0.001 s)
  [4] (0.000 s)
```

```
java.lang.AssertionError: square root for 2  expected:<1> but was:<2>
at junit.IMathTestJUnitParameterized.testIsqrt(IMathTestJUnitParameterized.ja
```

Note that not all tests can be abstract into parameterized tests

Another Example

```
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.add(1);
                assertEquals(2, list.size());
        }
        @Test
        public void test2() {
                list.add(1);
                list.add(2);
                list.add(3);
                assertEquals(3, list.size());
        }
        @Test
        public void test3() {
                list.add(1);
                list.add(2);
                list.remove(0);
                list.remove(0);
                assertEquals(0, list.size());
        }
}
```

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

JUnit Test Suite

- Test Suite: a set of tests (or other test suites)
  - Organize tests into a larger test set.
  - Help with automation of testing
- Consider the following case, how can I organize all the tests to make testing easier?
  - I need to test the List data structure
  - I also need to test the Set data structure

```
@RunWith(Suite.class)
@SuiteClasses({ ListTestJUnit.class, SetTestJUnit.class })
public class MyJUnitSuite {

}
```

```
@RunWith(Suite.class)
@SuiteClasses({ MyJUnit.class, ... })
public class MyMainJUnitSuite {

}
```

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