CSc 179 - Software Test Quality Assurance

Test Automation: Introduction to Junit

Credits:

http://www.junit.org

http://www.tutorialspoint.com/junit/

https://www.guru99.com/junit-tutorial.html

http://www.utdallas.edu/~lxz144130/ (Slides adapted)

Some earlier materials are from CSc 179/234 (Fall 2017)

Reading : AO – Test Automation - Chapter 3

CSc Dept, CSUS

Agenda

- Black-Box vs White-Box Testing (Review)
- Testing concepts
- Unit Testing/Unit Test Frame Work
- Junit Installation
- Program to test: `isqrt`
- Junit Functionalities / Demonstrations
- References

- In Class Attendance Quiz

Black-Box and White-Box Testing (Review)

- Black Box (aka Functional , aka Spec-Based)
  - Tests derived from functional requirements
  - Input/Output Driven
  - Internal source code of software is not relevant to design the tests

- White Box (aka Code-Based, aka Structural)
  - Tests derived from source code structure
  - Tests are evaluated in terms of coverage of the source code

Many others in between (Gray Box)

Testing: Concepts

- Test case (or, simply test)
  - An execution of the software with a given test input, including:
    - Input values
    - Sometimes include execution steps
    - Expected outputs

```
int actual_output=sum(1,2)
assertTrue(actual_output==3);
```

Example JUnit test case for testing "sum(int a,int b)"

Testing: Concepts

- Test oracle
  - The **expected outputs** of software for given input. A part of test cases
  - Hardest problem in auto-testing: test oracle generation

```
int actual_output=sum(1,2)
assertTrue(actual_output==3);
```

Example JUnit test case for testing "sum(int a,int b)"

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

- **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

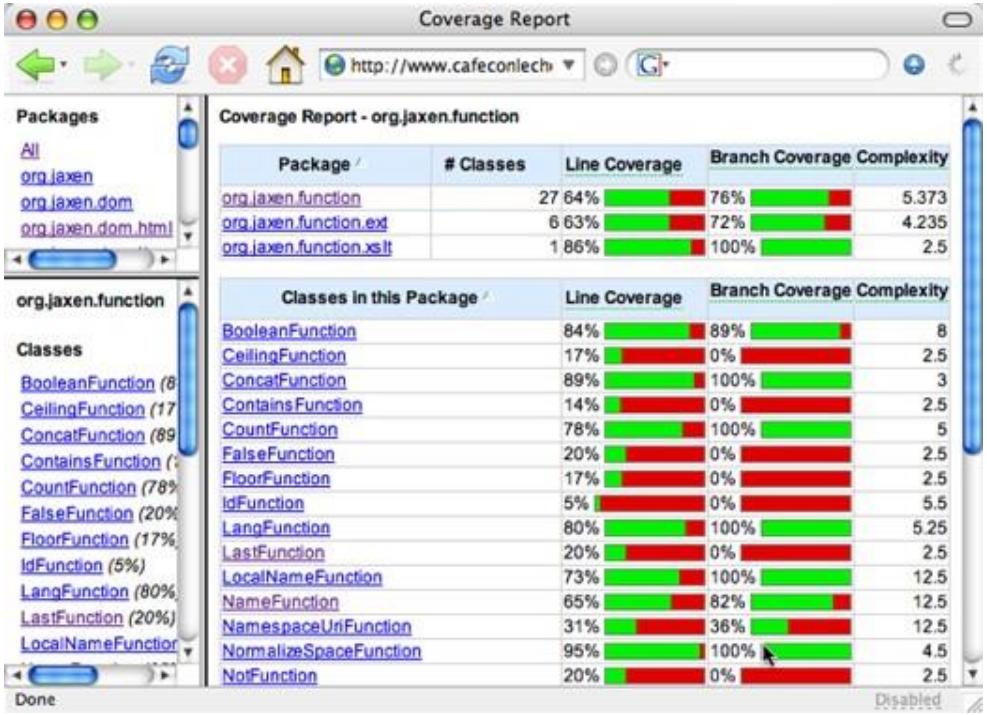
Testing: Concepts

- **Test adequacy**

- We can't always use all test inputs, so which do we use and when do we stop?
- We need a strategy to determine when we have done enough.
- Adequacy criterion: A rule that lets us judge the sufficiency of a set of test data for a piece of software
  - i.e. 95% of statements executed when running a test suite.

Testing: Concepts

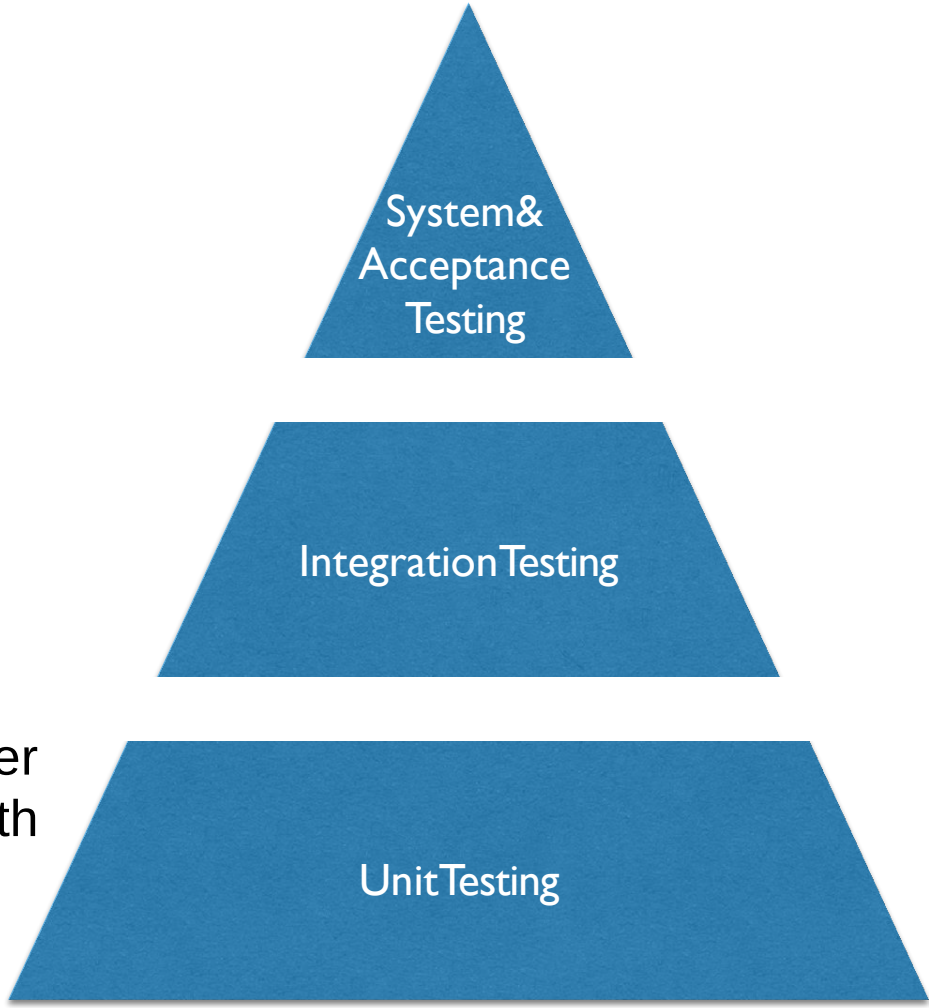
- Test adequacy example: test coverage
  - A measurement to evaluate the percentage of code tested
    - Statement coverage
    - Branch coverage, ...
    - Eclemma



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
- Acceptance Testing
  - Validate the system against user requirements, by customers with no formal test cases

Starting from here



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?

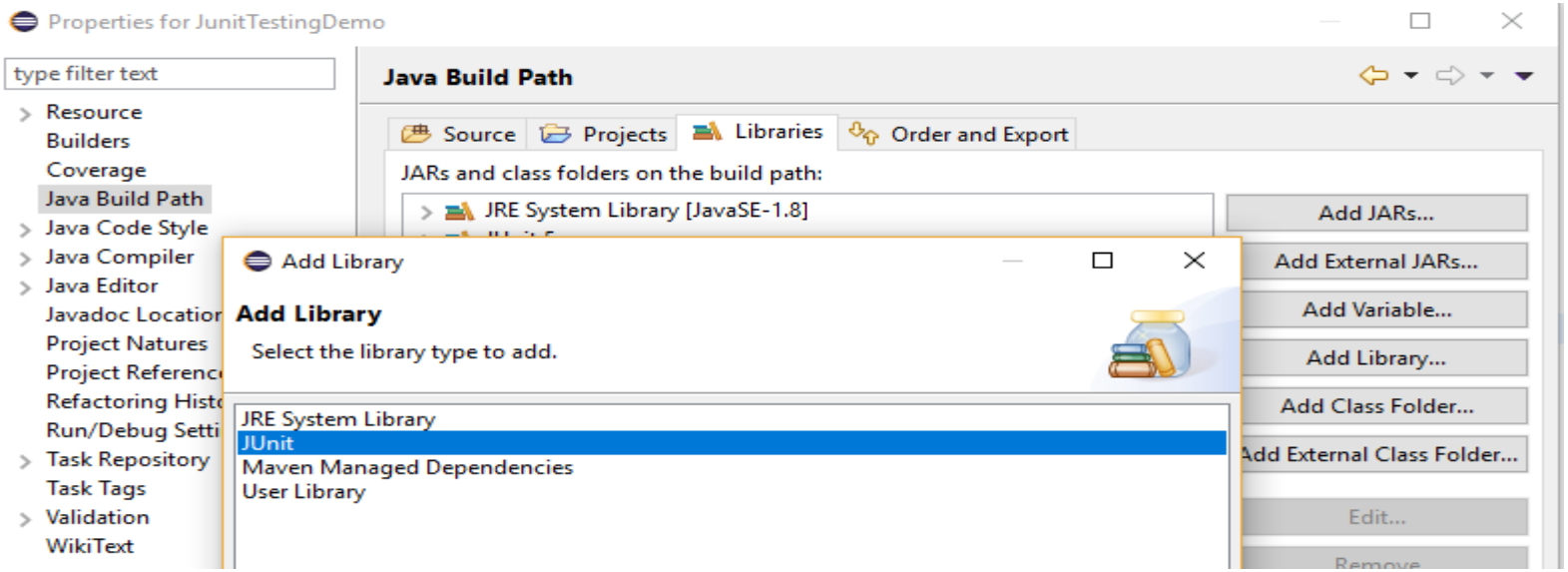
- Divide-and-conquer approach
  - Split system into units
  - Debug unit individually
  - Narrow down places where bugs can be
  - Don’t want to chase down bugs in other units

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

Installation Junit on Eclipse IDE

- Right click on project folder
- View Property
- Java build path
- Add library
- Choose Junit
- Click Next



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;
   }
}
```

We will test this program using various features/functions of Junit in later slides!

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
```

Tester has to manually judge the result to confirm if a test case is passing or not.

Scaling: What is happening if you want to test another set of values?

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) – Use assertTrue

```
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 // Identify test method
    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*/
}
```

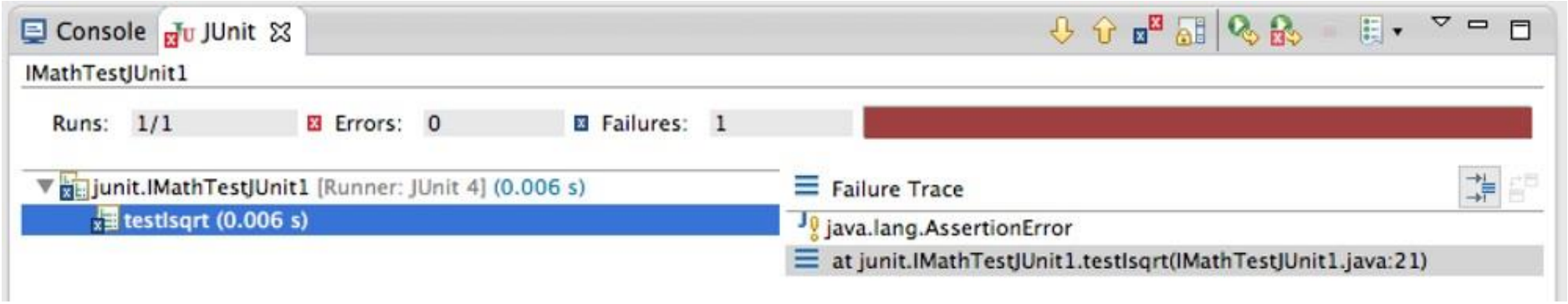
Test driver

Test case

Test oracle

JUnit Execution (1)

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



Not so good, why?

Why assertion error? Result was not so friendly. It is hard to know what is going on!
Result is abstracted into Boolean before sending to Junit.

Testing with JUnit (2) – Use assertEquals

```
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));
```

New test code!

Testing with JUnit (2) – Use assertEquals

- 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

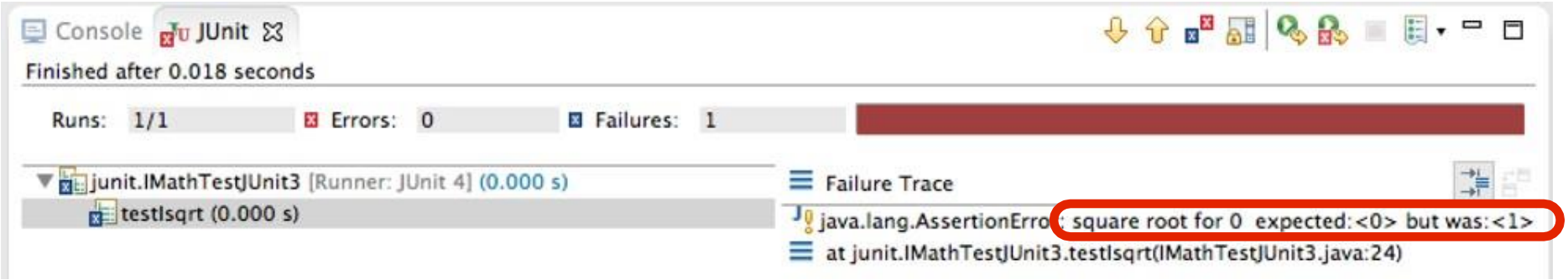
Can we make it better?

Testing with JUnit (3) - Use assertEquals + Meaningful Text associated with each assertion

```
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*/
}
```

Put more meaningful text!
(i.e. location or test #

Testing with JUnit (3) - Use assertEquals + Meaningful Text associated with each assertion



Still have problems, why?

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

Other test cases ?

Testing with JUnit (4) – Use Test fixture with multiple @Test

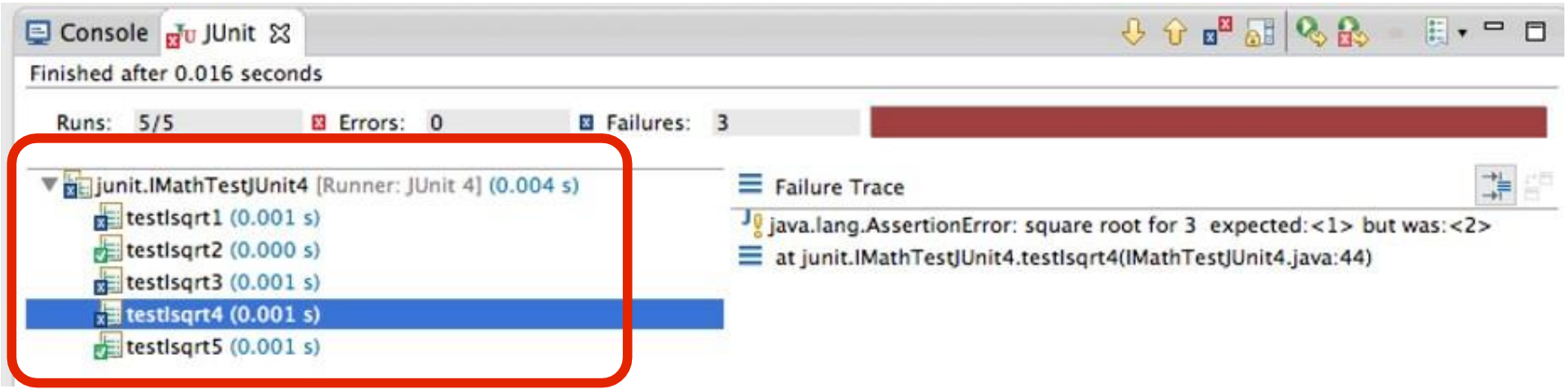
```
public class IMathTestJUnit4 {
    private IMath tester;
    @Before /** Setup method executed before each test */
    public void setup(){
        tester=new IMath();
    }

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

Note:

Test fixture

Do once!

Testing with JUnit (4) – Use Test fixture with multiple @Test

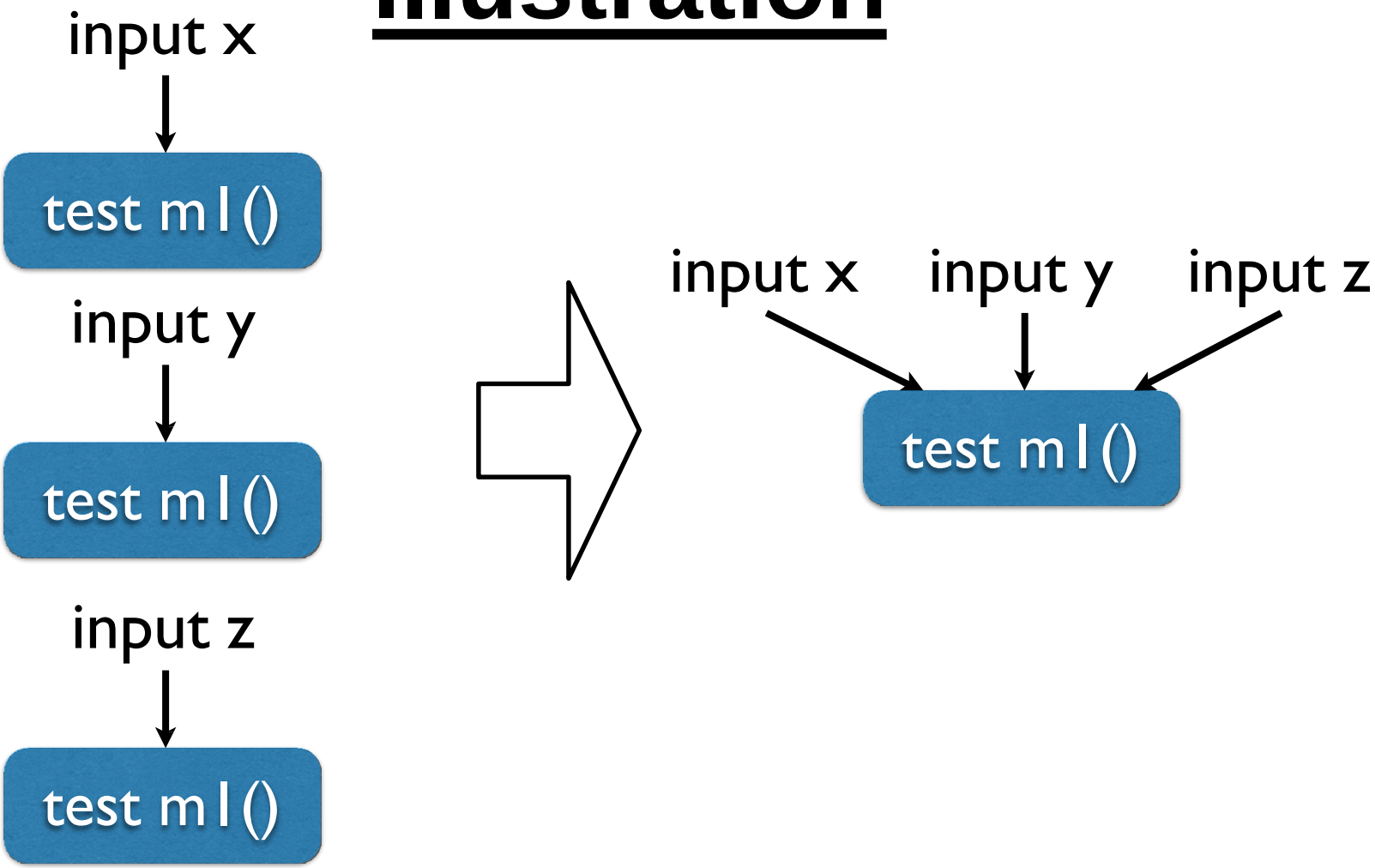


Still may have trouble, why?

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

Can we combined them ?

Parameterized Tests: Illustration



Testing with JUnit: Parameterized Test

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() {
        returnArrays.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));
    }
}
```

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 = newArrayList();
        }
        /** 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, <u>because the tests contains different method invocations</u>

JUnit Test Suite

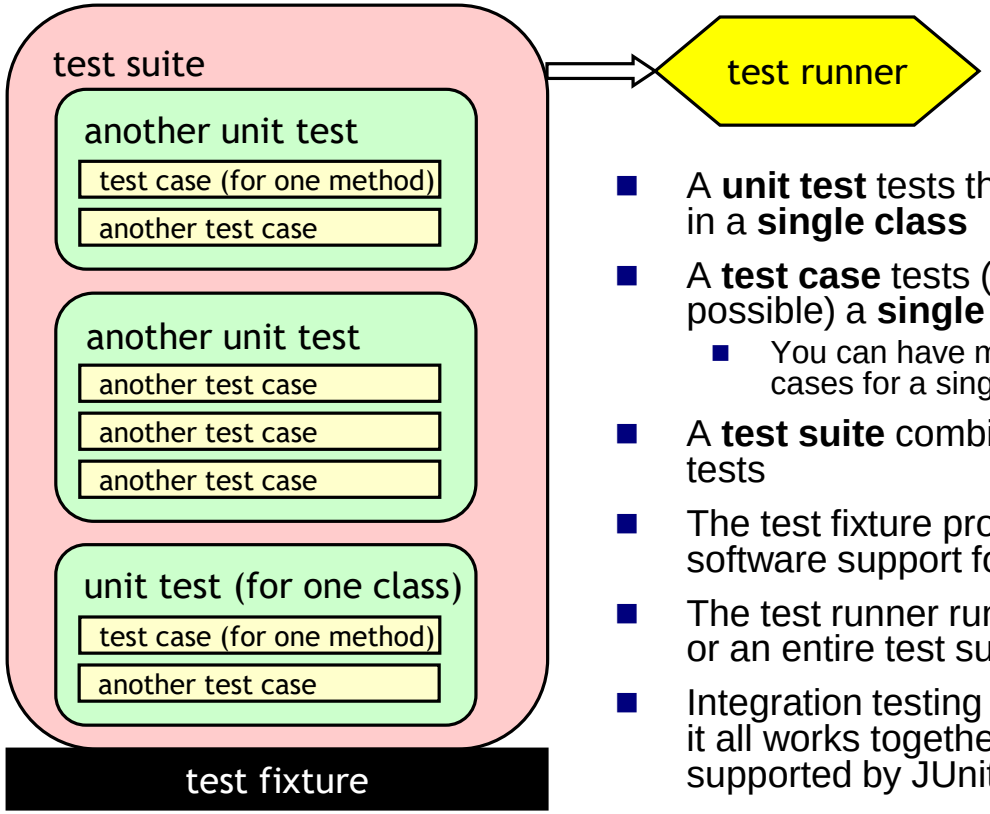
- **Test Suite**: a set of tests (or other test suites)
  - Organize tests into a larger test set.
  - Help with automation of testing
- Consider the following case, how can I organize all the tests to make testing easier?
  - I need to test the List data structure
  - I also need to test the Set data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

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

}
```

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

}
```

Note: www.guru99.com/create-junit-test-suite.html

In a picture



- A **unit test** tests the methods in a **single class**
- A **test case** tests (insofar as possible) a **single method**
  - You can have multiple test cases for a single method
- A **test suite** combines unit tests
- The test fixture provides software support for all this
- The test runner runs unit tests or an entire test suite
- Integration testing (testing that it all works together) is not well supported by JUnit

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
@RunWith*	JUnit will invoke the class it references to run the tests in that class

* is the annotation we covered today.

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

* is the assertion we covered today.

Terminology

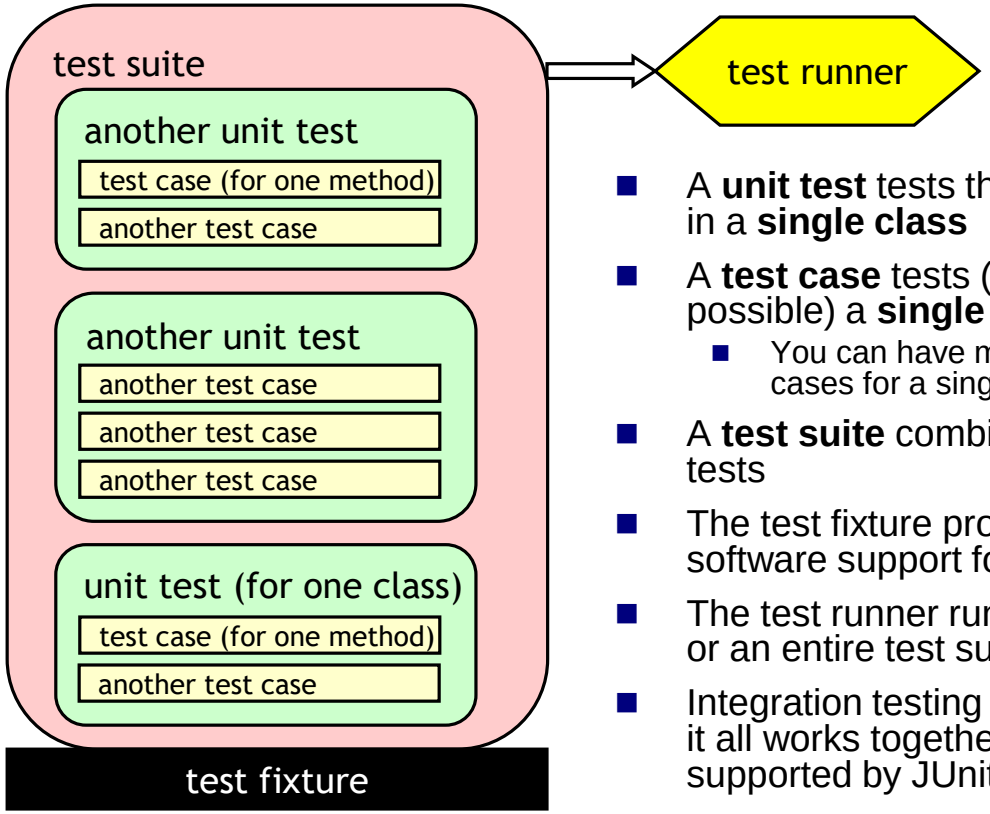
- A **test fixture** sets up the data (both objects and primitives) that are needed to run tests

  Example: If you are testing code that updates an employee record, you need an employee record to test it on
- A unit test is a test of a ***single* class**
- A test case tests the response of a **single method** to a particular set of inputs
- A test suite is a collection of **test cases**
- A **test runner** is software that runs tests and **reports results**

- An **integration test** is a test of how well classes work together
- How well Junit support integration testing ? (class project)

JUnit provides some limited support for integration tests

Once more, in a picture



- A **unit test** tests the methods in a **single class**
- A **test case** tests (insofar as possible) a **single method**
  - You can have multiple test cases for a single method
- A **test suite** combines unit tests
- The test fixture provides software support for all this
- The test runner runs unit tests or an entire test suite
- Integration testing (testing that it all works together) is not well supported by JUnit

More on JUnit?

- Hompage:
  - www.junit.org
- Tutorials
  - http://www.vogella.com/tutorials/JUnit/article.html
  - http://www.tutorialspoint.com/junit/
  - https://www.guru99.com/junit-tutorial.html