# Software Testing Technique Chapter 3

# Unit Testing and Test Coverage

Junjie Chen（陈俊洁）
Office: 55-A317
Email: junjiechen@tju.edu.cn

*2022*

## Outline

- Test Automation
- Introduction to Junit
- Test Coverage

## TEST AUTOMATION

## What is Test Automation?

**The use of software to control the execution of tests, the comparison of actual outcomes to expected outcomes, the setting up of test preconditions, and other test control and test reporting functions**

- **Reduces cost**
- **Reduces human error**
- **Reduces variance in test quality from different individuals**
- **Significantly reduces the cost of regression testing**

## Software Testability

The degree to which a system or component facilitates the establishment of test criteria and the performance of tests to determine whether those criteria have been met

- Plainly speaking – how hard it is to find faults in the software
- Testability is dominated by two practical problems
  - How to provide the test values to the software
  - How to observe the results of test execution

## Observability and Controllability

- **Observability**

**How easy it is to observe the behavior of a program in terms of its outputs, effects on the environment and other hardware and software components**

  - Software that affects hardware devices, databases, or remote files have low observability

- **Controllability**

**How easy it is to provide a program with the needed inputs, in terms of values, operations, and behaviors**

  - Easy to control software with inputs from keyboards
  - Inputs from hardware sensors or distributed software is harder

# Components of a Test Case

- **A test case is a multipart artifact with a definite structure**
- **Test case values**

**The input values needed to complete an execution of the software under test**

- **Expected results**

**The result that will be produced by the test if the software behaves as expected**

  - **A *test oracle* uses expected results to decide whether a test passed or failed**

## Affecting Controllability and Observability

- **Prefix values**

Inputs necessary to put the software into the appropriate state to receive the test case values

- **Postfix values**

Any inputs that need to be sent to the software after the test case values are sent

1. ***Verification Values*** **: Values needed to see the results of the test case values**
2. ***Exit Values*** **: Values or commands needed to terminate the program or otherwise return it to a stable state**

## Putting Tests Together

- **Test case**

The test case values, prefix values, postfix values, and expected results necessary for a complete execution and evaluation of the software under test

- **Test set**

A set of test cases

- **Executable test script**

A test case that is prepared in a form to be executed automatically on the test software and produce a report

## INTRODUCTION TO JUNIT

## Test Automation Framework

A set of assumptions, concepts, and tools that support test automation

## Example: Old way vs. New way

-
  ```
  int max(int a, int b) {
     if (a > b) {
        return a;
     } else {
        return b;
     }
  }
  ```

- **Traditional way:**

```
void testMax() {
   int x = max(3, 7);
   if (x != 7) {
      System.out.println("max(3, 7) gives " + x);
   }
   x = max(3, -7);
   if (x != 3) {
      System.out.println("max(3, -7) gives " + x);
   }
}
public static void main(String[] args) {
   new MyClass().testMax();
}
```

- **New way:**

```
@Test
void testMax() {
   assertEquals(7, max(3, 7));
   assertEquals(3, max(3, -7));
}
```

## What is JUnit?

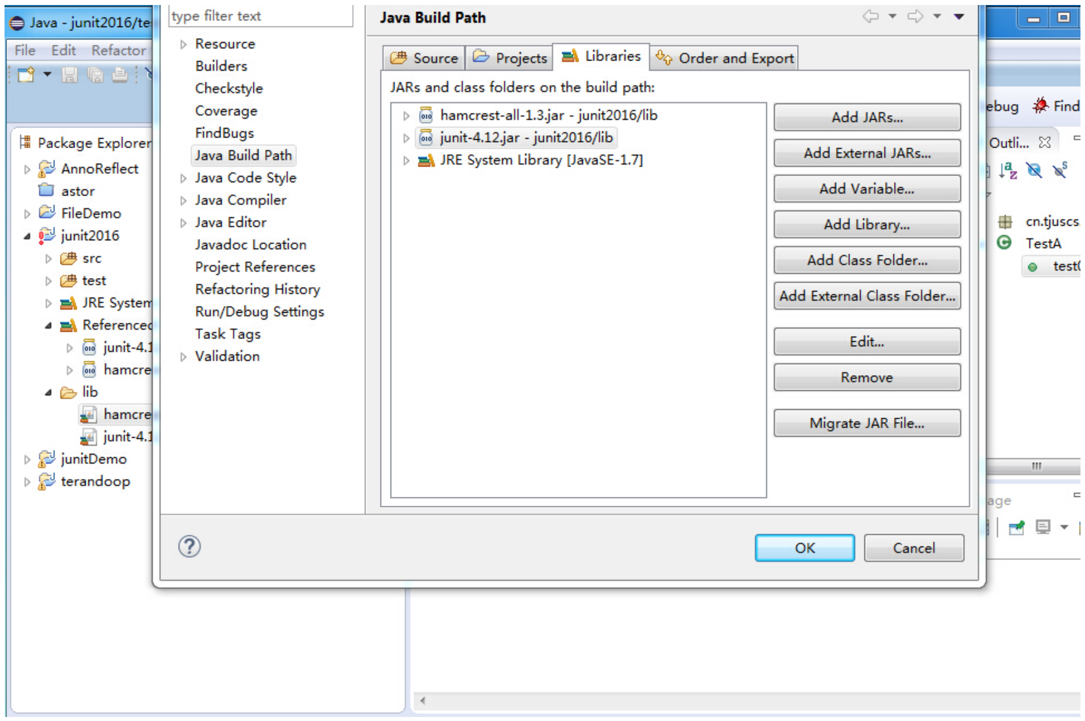
- **Open source Java testing framework used to write and run repeatable automated tests**
- **JUnit is open source (junit.org)**
- **A structure for writing test drivers**
- **JUnit features include:**
  - **Assertions for testing expected results**
  - **Test features for sharing common test data**
  - **Test suites for easily organizing and running tests**
  - **Graphical and textual test runners**
- **JUnit is widely used in industry**
- **JUnit can be used as stand alone Java programs (from the command line) or within an IDE such as Eclipse**

## JUnit Tests

- JUnit can be used to test …
  - … an entire object
  - … part of an object – a method or some interacting methods
  - … interaction between several objects
- It is primarily for unit and integration testing, not system testing
- Each test is embedded into one test method
- A test class contains one or more test methods
- Test classes include :
  - A test runner to run the tests (main())
  - A collection of test methods
  - Methods to set up the state before and update the state after each test and before and after all tests (Junit3: setUp() and teardown())
- Get started at junit.org

# How to install/use Junit with Eclipse

- **Install Junit with Eclipse**
  - **Download junit***.jar and hamcrest from junit.org**
  - **Add these two .jar into the build path**


Java Build Path

JARs and class folders on the build path:

hamcrest-all-1.3.jar - junit2016/lib
junit-4.12.jar - junit2016/lib
JRE System Library [JavaSE-1.7]

# Writing Tests for JUnit

- **Need to use the methods of the junit.framework.assert class**
  - **javadoc gives a complete description of its capabilities**
- **Each test method checks a condition (assertion) and reports to the test runner whether the test failed or succeeded**
- **The test runner uses the result to report to the user (in command line mode) or update the display (in an IDE)**
- **All of the methods return void**
- **A few representative methods of junit.framework.assert**
  - ***assertTrue (boolean)***
  - ***assertTrue (String, boolean)***
  - ***assertEquals (Object, Object)***
  - ***assertNull (Object)***
  - ***Fail (String)***

# Example JUnit Test Case

```
public class Calc
{
public long add (int a, int b)
   {
      return a + b;
   }
}
```

```
import org.junit.Test
import static org.junit.Assert.*;
public class calcTest
{
   private Calc calc;
   @Test
   public void testAdd()
   {
      calc = new Calc ();
      assertEquals ((long) 5, calc.add (2, 3));
   }
}
```

Expected result

The test

## Assertions

- A set of assertion methods useful for writing tests. Only failed assertions are recorded. These methods can be used directly: Assert.assertEquals(...), however, they read better if they are referenced through static import:

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

...

assertEquals(...);
```

- “Keep the bar green to keep the code clean”

## Sample Assertions

- **static void assertEquals (boolean expected, boolean actual)**
  **Asserts that two booleans are equal**
- **static void assertEquals (byte expected, byte actual)**
  **Asserts that two bytes are equal**
- **static void assertEquals (char expected, char actual)**
  **Asserts that two chars are equal**
- **static void assertEquals (double expected, double actual, double delta)**
  **Asserts that two doubles are equal, within a delta**
- **static void assertEquals (float expected, float actual, float delta)**
  **Asserts that two floats are equal, within a delta**
- **static void assertEquals (int expected, int actual)**
  **Asserts that two ints are equal**
- **For a complete list, see**
  - ***http://junit.sourceforge.net/javadoc/org/junit/Assert.html***

# Annotations in Junit

- **@Test**
  - **(expected)**
  - **(timeout)**
- **@Before and @After**
  - **@Before and @After annotated method will be invoked before and after each test case**
- **@BeforeClass and @AfterClass**
  - **@BeforeClass and @AfterClass annotated method will be invoked before and after each test class.**
- **@Ignore**
- **@Runwith**
  - **Suite.class**
  - **Parameterized.class**

## Workflow of Junit

- @Before
  - Initialization
- @After
  - Release some resources
- @BeforeClass and @AfterClass
  - Must be “static”
  - Be executed only once.
  - Initialize and release some expensive resouces.
- The workflow of a test class
  - @BeforeClass→@Before→@Test→@After→@AfterClass
- The workflow of a test method
  - @Before→@Test→@After

## Test Exception

- @Test(expected=ArithmeticException.class)

```
        public void testDivideException(){
                int result = cal.divide(1, 0);
                /*
                 * If there is an exception
                 */
        }
```

## Simple performance test

- @Test(timeout=300)

```
public void testTime(){
        try {
                Thread.sleep(500);
        } catch (InterruptedException e) {
        // TODO Auto-generated catch block
                e.printStackTrace();
        }
}
```

## @Runwith(Suite.class)

- If there are many test cases, how to execute them?

```
public class TestA {
        @Test
        public void test01(){
                System.out.println("TestA --01");
        }
}
```

```
public class TestB {
        @Test
        public void test02(){
                System.out.println("TestB --02");
        }
}
```

```
@RunWith(Suite.class)
@SuiteClasses({TestA.class,
TestB.class,
TestCalculate.class})
public class TestSuite {

}
```

## @Runwith(Parameterized.class)

- **Allow developer to run the same test over and over again using different values.**
- **Five steps:**
  - **Annotate test class with @RunWith(Parameterized.class)**
  - **Create a public static method annotated with @Parameters that returns a Collection of Objects (as Array) as test data set.**
  - **Create a public constructor that takes in what is equivalent to one "row" of test data.**
  - **Create an instance variable for each "column" of test data.**
  - **Create your tests case(s) using the instance variables as the source of the test data.**

```
@RunWith(Parameterized.class)
public class TestPara {
    private int input1;
    private int input2;
    private int expected;
    private Calculate cal = null;

    public TestPara(int input1,int input2,int expected){
        this.input1 = input1;
        this.input2 = input2;
        this.expected = expected;
    }

    @Before
    public void setUp(){
        cal = new Calculate();
    }
    @Parameters
    public static Collection<Object[]> getData(){
                return Arrays.asList(new Object[][]{
                {1,1,2},
                {2,3,5},
                {3,5,8}
                });
    }
    @Test
    public void testAdd() {
                assertEquals(this.expected,cal.add(input1, input2));
    }

}
```

Annotation

# Junit and Hamcrest

- **Please write the assertion for the following case:**
  - **40 is between 30 and 50**

```
assertTrue((40>30) && (40<50))
```

```
assertThat(40,allOf(greaterThan(30),lessThan(50)))
```

  - **A string “abcdefg” starts with “abc” and ends with “efg”.**

```
assertThat(“abcdefg”,allOf(startsWith(“abc”),endsWith(“efg”)))
```

  - **More flexible, more readable and more useful**

## Hamcrest

- Hamcrest is a framework that assists writing software tests in the Java programming language. It supports creating customized assertion matchers ('Hamcrest' is an anagram of 'matchers'), allowing match rules to be defined declaratively . (From Wikipedia)
- Adding Hamcrest directly to the classpath in Eclipse.
  – The Junit included in Eclipse only contain the core Hamcrest matcher.
  – hamcrest-all-*.jar

# Important Hamcrest matchers

- **allOf - matches if all matchers match (short circuits)**
- **anyOf - matches if any matchers match (short circuits)**
- **not - matches if the wrapped matcher doesn't match and vice versa**
- **equalTo - test object equality using the equals method**
- **is - decorator for equalTo to improve readability**
- **hasToString - test Object.toString**
- **instanceOf, isCompatibleType - test type**
- **notNullValue, nullValue - test for null**
- **sameInstance - test object identity**
- **hasEntry, hasKey, hasValue - test a map contains an entry, key or value**
- **hasItem, hasItems - test a collection contains elements**
- **hasItemInArray - test an array contains an element**
- **closeTo - test floating point values are close to a given value**
- **greaterThan, greaterThanOrEqualTo, lessThan, lessThanOrEqualTo - test ordering**
- **equalToIgnoringCase - test string equality ignoring case**
- **equalToIgnoringWhiteSpace - test string equality ignoring differences in runs of whitespace**
- **containsString, endsWith, startsWith - test string matching**

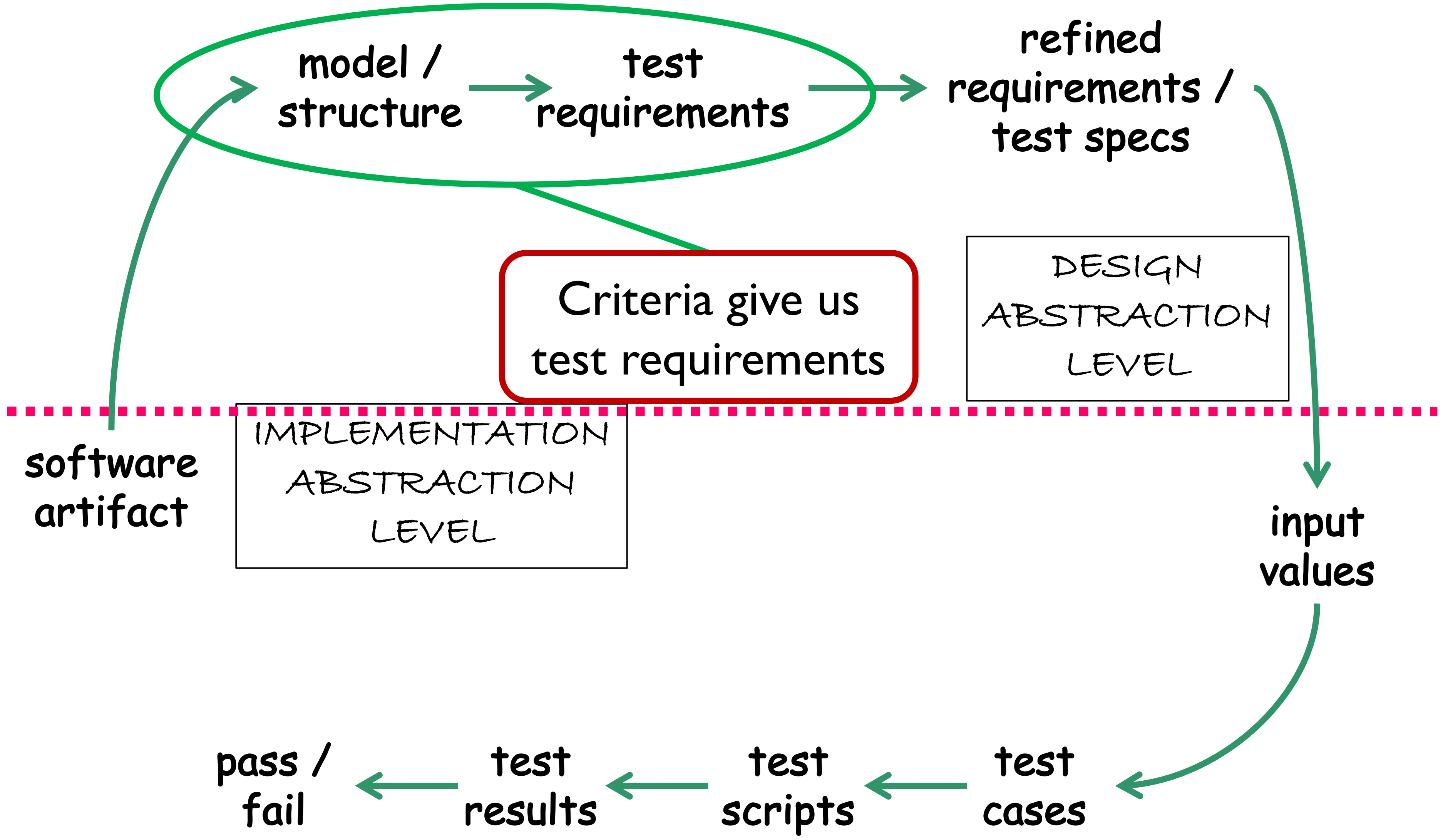
## TEST COVERAGE

# Changing Notions of Testing

- **Old view focused on testing at each software development phase as being very different from other phases**
  - **Unit, module, integration, system …**

- **New view is in terms of structures and criteria**
  - **input space, graphs, logical expressions, syntax**
- **Test design is largely the same at each phase**
  - **Creating the model is different**
  - **Choosing values and automating the tests is different**

# Model-Driven Test Design



## Test Coverage Criteria

A tester's job is simple : Define a model of the software, then find ways to cover it

- **Test Requirements** : **A specific element of a software artifact that a test case must satisfy or cover**

- **Coverage Criterion** : **A rule or collection of rules that impose test requirements on a test set**

**Testing researchers have defined dozens of criteria, but they are all really just a few criteria on four types of structures …**

## Source of Structures

- These structures can be **extracted** from lots of software artifacts
  - **Graphs** can be extracted from UML use cases, finite state machines, source code, …
  - **Logical expressions** can be extracted from decisions in program source, guards on transitions, conditionals in use cases, …

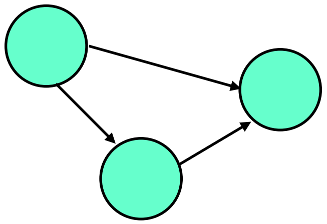
# Criteria Based on Structures

**Structures : Four ways to model software**

1. Input Domain Characterization (sets)

   **A: {0, 1, >1}**
   **B: {600, 700, 800}**
   **C: {swe, cs, isa, infs}**

2. Graphs



3. Logical Expressions

   **(not X or not Y) and A and B**

4. Syntactic Structures (grammars)

```
if (x > y)
    z = x - y;
else
   z = 2 * x;
```

# Example : Jelly Bean Coverage

**Flavors :**

1. **Lemon**
2. **Pistachio**
3. **Cantaloupe**
4. **Pear**
5. **Tangerine**
6. **Apricot**

Colors :

1. Yellow (Lemon, Apricot)
2. Green (Pistachio)
3. Orange (Cantaloupe, Tangerine)
4. White (Pear)

■ Possible coverage criteria :

1. Taste one jelly bean of each flavor
   - Deciding if yellow jelly bean is Lemon or Apricot is a controllability problem
2. Taste one jelly bean of each color

## Coverage

**Given a set of test requirements *TR* for coverage criterion *C*, a test set *T* satisfies *C* coverage if and only if for every test requirement *tr* in *TR*, there is at least one test *t* in *T* such that *t* satisfies *tr***

- **Infeasible test requirements : test requirements that cannot be satisfied**
  - No test case values exist that meet the test requirements
  - Example: Dead code
  - Detection of infeasible test requirements is formally undecidable for most test criteria
- **Thus, 100% coverage is impossible in practice**

## More Jelly Beans

**T1 = { three Lemons, one Pistachio, two Cantaloupes, one Pear, one Tangerine, four Apricots }**

- **Does test set T1 satisfy the flavor criterion ?**

**T2 = { One Lemon, two Pistachios, one Pear, three Tangerines }**

- Does test set T2 satisfy the flavor criterion ?
- Does test set T2 satisfy the color criterion ?

## Coverage Level

**The ratio of the number of test requirements satisfied by *T* to the size of *TR***

- **T2 on the previous slide satisfies 4 of 6 test requirements**

## Two Ways to Use Test Criteria

1. **Directly generate test values to satisfy the criterion**
   - Often assumed by the research community
   - Most obvious way to use criteria
   - Very hard without automated tools
2. **Generate test values externally and measure against the criterion**
   - Usually favored by industry
   - Sometimes misleading
   - If tests do not reach 100% coverage, what does that mean?

**Test criteria are sometimes called metrics**

# Generators and Recognizers

- Generator : A procedure that automatically generates values to satisfy a criterion
- Recognizer : A procedure that decides whether a given set of test values satisfies a criterion

- Both problems are provably undecidable for most criteria
- It is possible to recognize whether test cases satisfy a criterion far more often than it is possible to generate tests that satisfy the criterion

- Coverage analysis tools are quite plentiful

## Comparing Criteria with Subsumption

- **Criteria Subsumption : A test criterion *C1* subsumes *C2* if and only if every set of test cases that satisfies criterion *C1* also satisfies *C2***
- **Must be true for every set of test cases**
- ***Examples* :**
  - **The flavor criterion on jelly beans subsumes the color criterion … if we taste every flavor we taste one of every color**
  - **If a test set has covered every branch in a program (satisfied the branch criterion), then the test set is guaranteed to also have covered every statement**

## Advantages of Criteria-Based Test Design

- Criteria maximize the "bang for the buck"
  - Fewer tests that are more effective at finding faults
- Comprehensive test set with minimal overlap
- Traceability from software artifacts to tests
  - The "why" for each test is answered
  - Built-in support for regression testing
- A "stopping rule" for testing—advance knowledge of how many tests are needed
- Natural to automate

## Characteristics of a Good Coverage Criterion

1. **It should be fairly easy to compute test requirements automatically**
2. **It should be efficient to generate test values**
3. **The resulting tests should reveal as many faults as possible**

- Subsumption is only a rough approximation of fault revealing capability
- Researchers still need to gives us more data on how to compare coverage criteria

## Test Coverage Criteria

- **Traditional software testing is expensive and labor-intensive**
- **Formal coverage criteria are used to decide which test inputs to use**
- **More likely that the tester will find problems**
- **Greater assurance that the software is of high quality and reliability**
- **A goal or stopping rule for testing**
- **Criteria makes testing more efficient and effective**

**How do we start applying these ideas in practice?**

## How to Improve Testing ?

- **Testers need more and better software tools**
- **Testers need to adopt practices and techniques that lead to more efficient and effective testing**
  - **More education**
  - **Different management organizational strategies**
- **Testing & QA teams need more technical expertise**
  - **Developer expertise has been increasing dramatically**
- **Testing & QA teams need to specialize more**
  - **This same trend happened for development in the 1990s**

## Criteria Summary

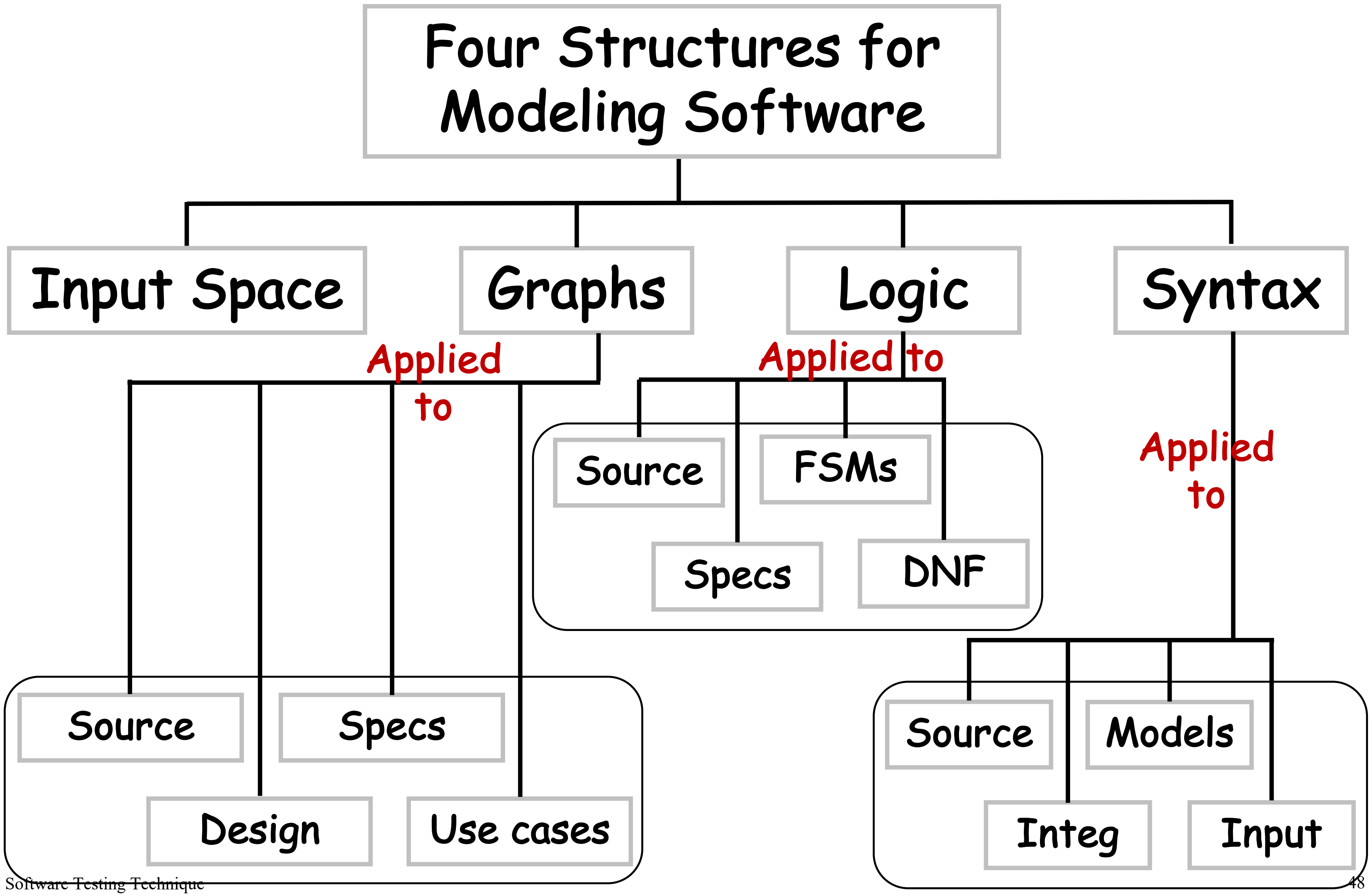
- Many companies still use "monkey testing"
  - A human sits at the keyboard, wiggles the mouse and bangs the keyboard
  - No automation
  - Minimal training required
- Some companies automate human-designed tests
- But companies that use both automation and criteria-based testing

**Save money**

**Find more faults**

**Build better software**

# Structures for Criteria-Based Testing



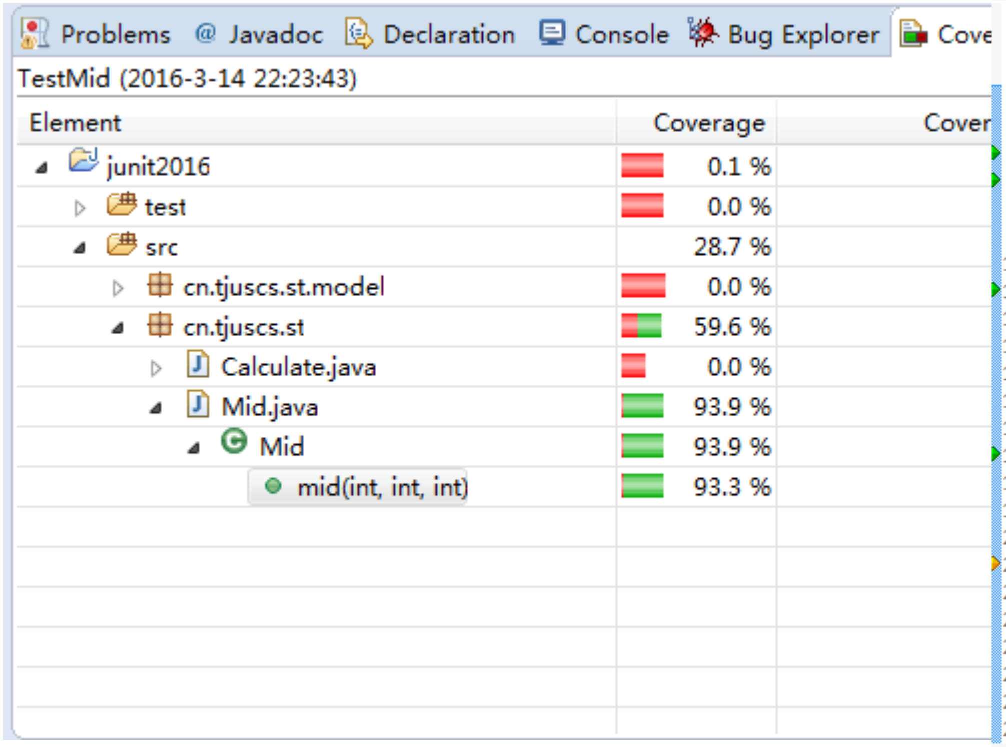
## Summary of Part 1's New Ideas

1. Why do we test – to reduce the risk of using software
   - Faults, failures, the RIPR model
   - Test process maturity levels – level 4 is a mental discipline that improves the quality of the software
2. Model-Driven Test Design
   - Four types of test activities – test design, automation, execution and evaluation
3. Test Automation
   - Testability, observability and controllability, test automation frameworks
4. Criteria-based test design
   - Four structures – test requirements and criteria

**Earlier and better testing empowers test managers**

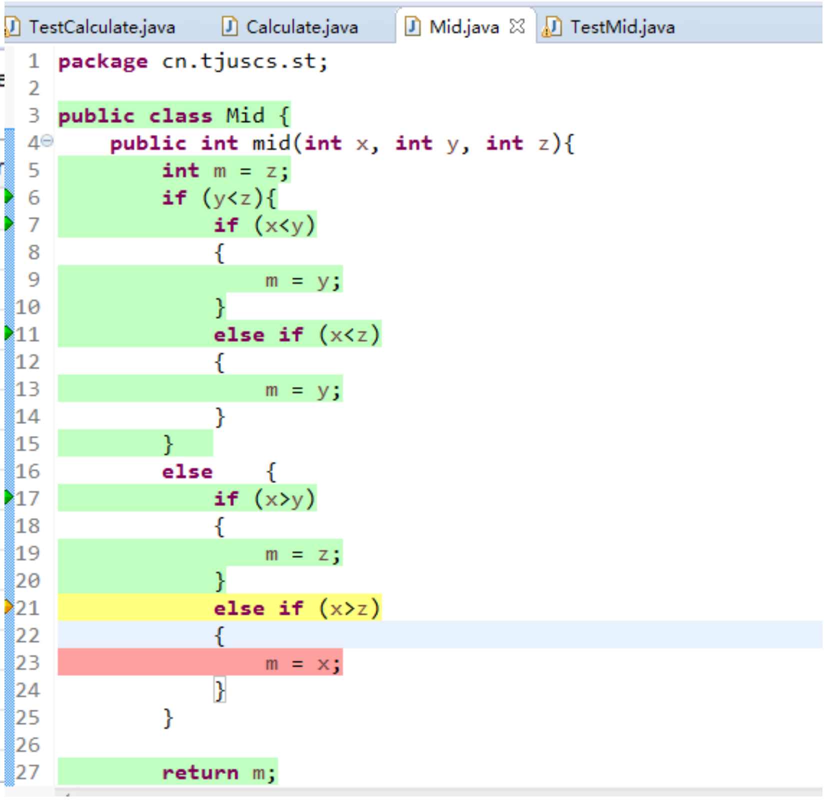
# Testing Coverage Tools in Java

- **Eclemma**
  - http://eclemma.org/
  - Jacoco



TestMid (2016-3-14 22:23:43)

| Element | Coverage | Cover |
|---|---|---|
| junit2016 | 🟥 0.1 % | |
| test | 🟥 0.0 % | |
| src | 28.7 % | |
| cn.tjuscs.st.model | 🟥 0.0 % | |
| cn.tjuscs.st | 🟥🟩 59.6 % | |
| Calculate.java | 🟥 0.0 % | |
| Mid.java | 🟩 93.9 % | |
| Mid | 🟩 93.9 % | |
| mid(int, int, int) | 🟩 93.3 % | |



```java
package cn.tjuscs.st;

public class Mid {
    public int mid(int x, int y, int z){
        int m = z;
        if (y<z){
            if (x<y)
            {
                m = y;
            }
            else if (x<z)
            {
                m = y;
            }
        }
        else     {
            if (x>y)
            {
                m = z;
            }
            else if (x>z)
            {
                m = x;
            }
        }

        return m;
```

- **Cobertura**
  - **https://sourceforge.net/projects/cobertura/**
  - **http://www.importnew.com/13107.html**