Intro to Software Testing

Chapter 3

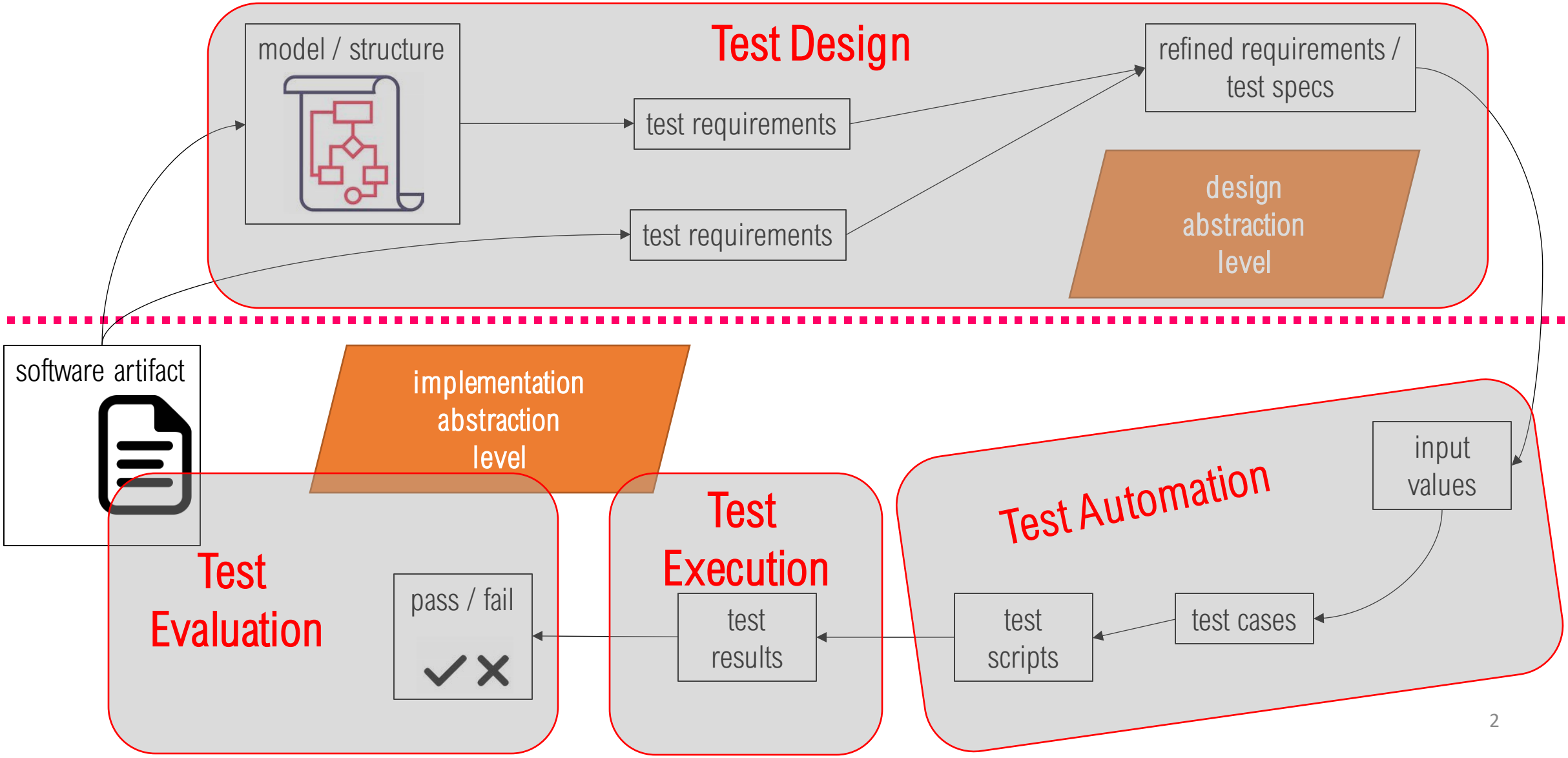
Test Automation

Dr. Brittany Johnson-Matthews
(Dr. B for short)

https://go.gmu.edu/SWE637

Adapted from slides by Jeff Offutt and Bob Kurtz

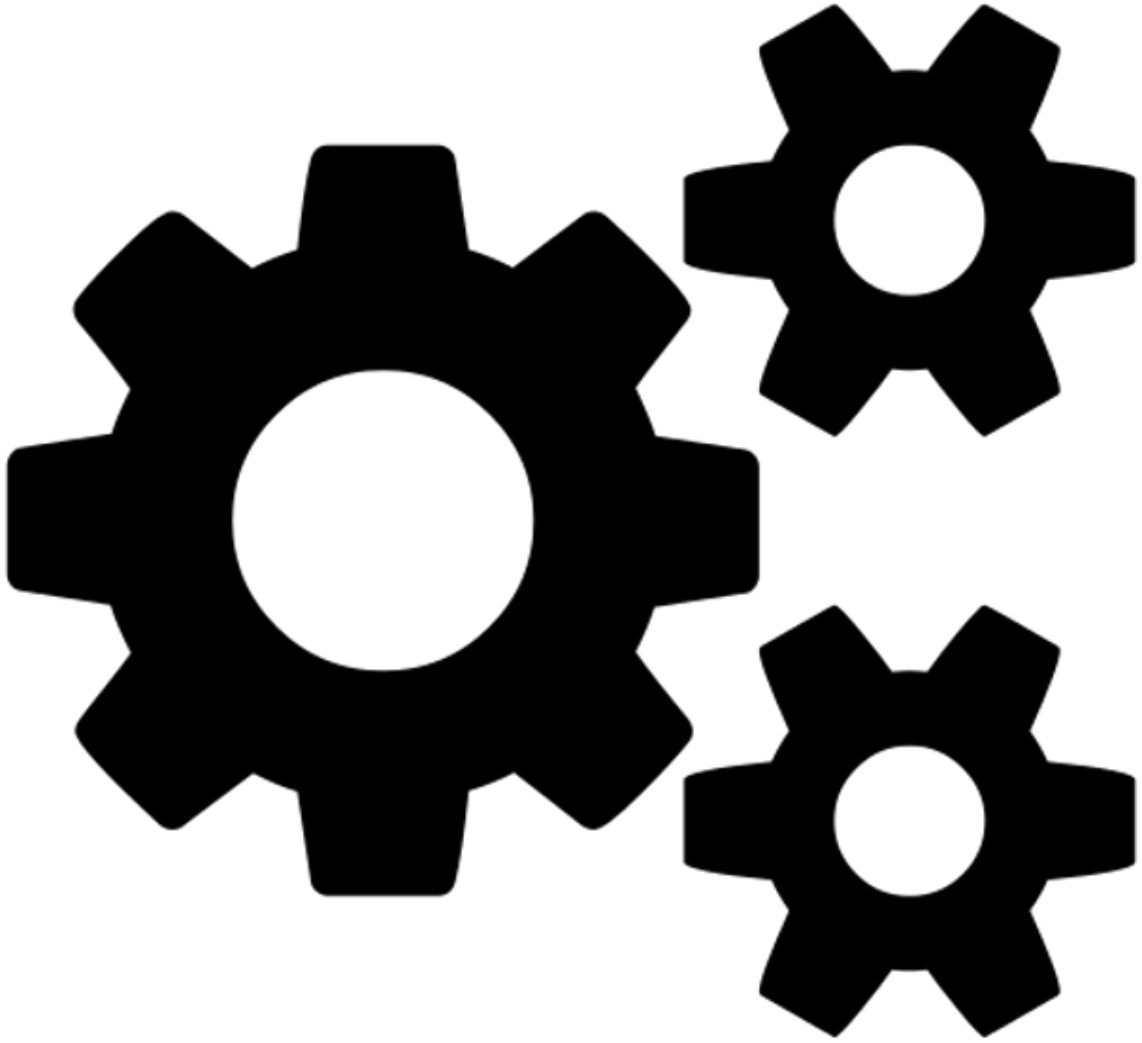
Model-Driven Test Design Activities



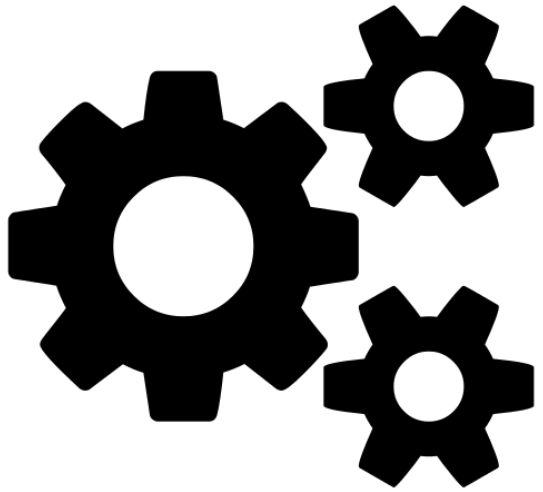
What is Test Automation?

Test automation is the use of software to:

- Control the execution of tests
- Compare actual outcomes to predicted outcomes
- Set up test preconditions
- Other test control and test reporting functions



Why Use Test Automation?



Reduces cost

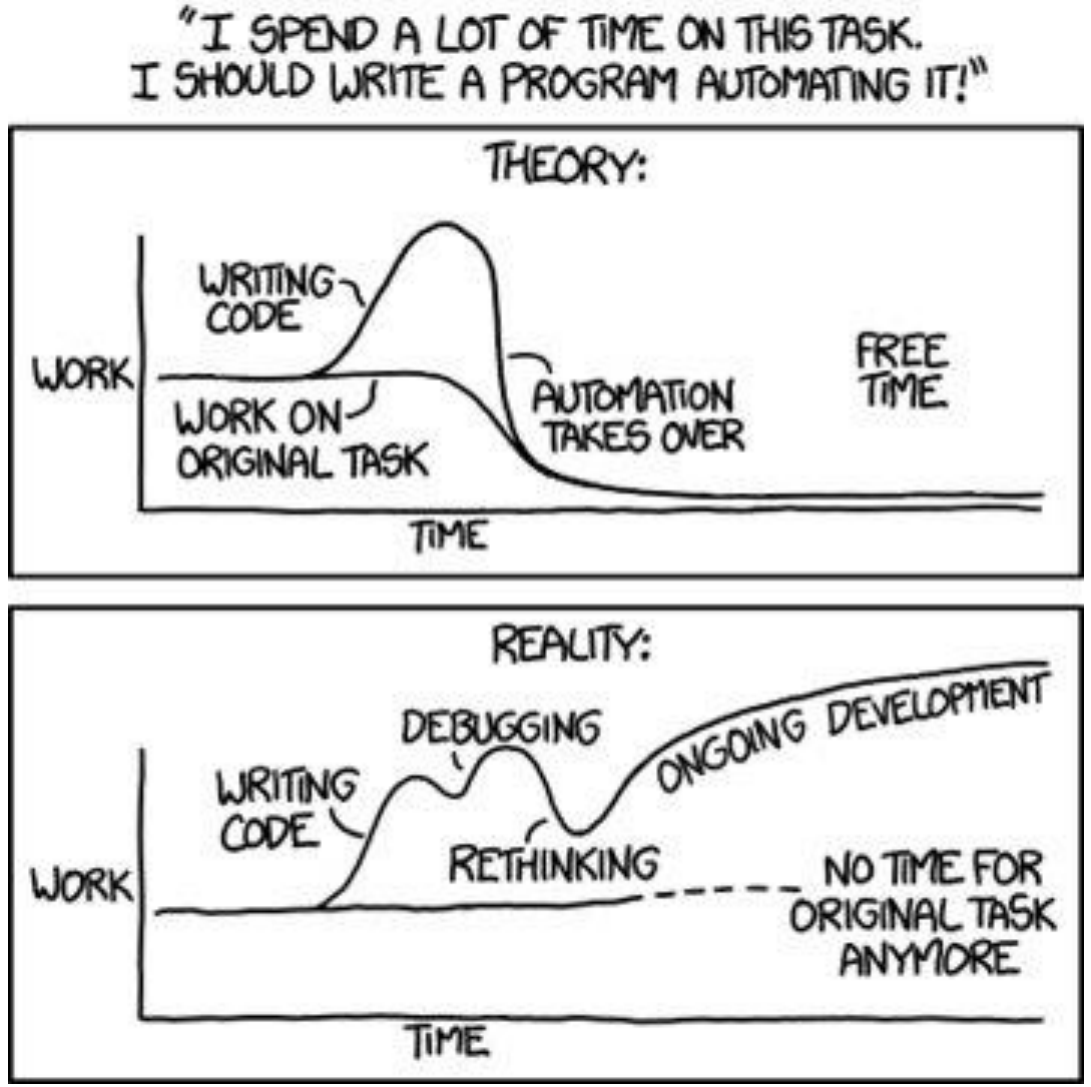
Reduces human error

Reduces variance in test quality from different individuals

Significantly reduces the cost of regression testing

Enables much more frequent testing

Why Use Test Automation?



Software Testability (3.1)

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*

Software Testability

Testability is dominated by two practical problems:

Controllability: how easy it is to provide a program with the needed inputs, in terms of values, operations, and behaviors

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

Abstraction reduces both controllability and observability.

Controllability and Observability

Other inputs can affect controllability and observability

- *Prefix values*: inputs necessary to put the software into the appropriate state to receive the test case values
- *Postfix values*: inputs that need to be sent to the software after the test case values are sent
  - *Verification values*: values needed to see the results of the test case values
  - *Exit values*: values or commands needed to terminate the program or otherwise return it to a stable state

Components of a test case (3.2)

A test case is a *multipart artifact* that includes:

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

Putting Test 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 that can be executed automatically on the test software and produce a report

Test Automation Framework (3.3)

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

Some popular test automation frameworks:

- **JUnit** (Java, open source)
- **Google Test** (C++, open source)
- **MSTest** (C#, .NET, commercial)
- **VectorCAST** (C++, Ada, commercial)
- **Cucumber** (various languages, open source)

What is JUnit?

Open source Java testing framework used to write and run repeatable automated tests

A structure for writing test drivers

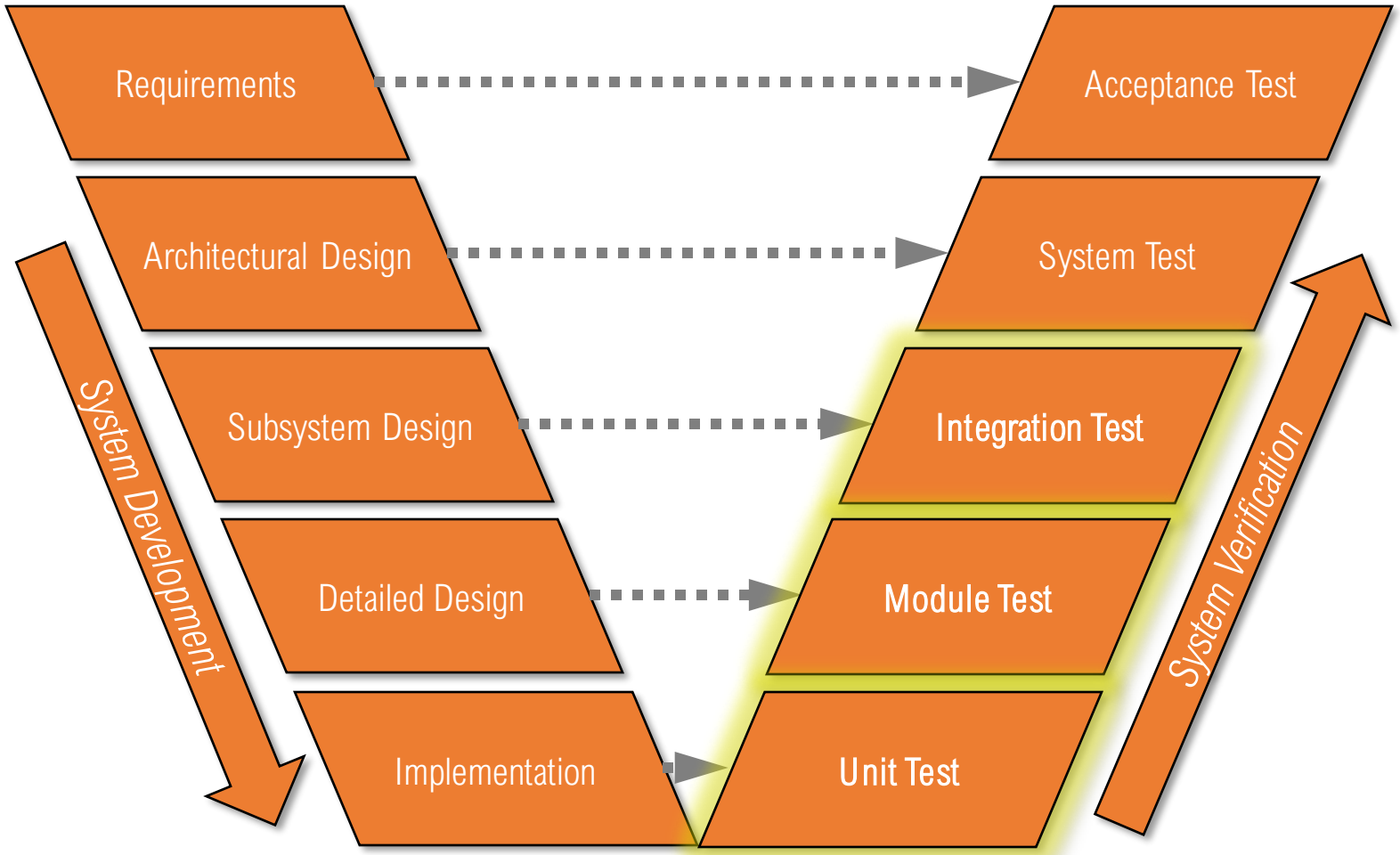
JUnit features include:

- 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

When is JUnit useful?



Writing tests with JUnit

Use the methods of the org.junit.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_expression>)
- assertFalse (<boolean_expression>)
- assertEquals (<expected_value>,<expression>)
- assertNear (<expected_value>, <expression>, <delta>)
- assertNull (<expression>)
- fail (String)

Assertions can take a string as an optional first argument; that string will be displayed if the assertion fails

JUnit Tests

Open source Java testing framework used to write and run repeatable automated tests

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

Simple JUnit example

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

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

public class CalcTest
{
   @Test public void testAdd()
   {
       assertEquals (“Calc sum incorrect”,
             5, Calc.add (2, 3));
   }
}
```

Displayed if assertion fails (optional)

Expected result

Test case values

JUnit Test Fixtures

A *test fixture* controls the state of the test

- Objects and variables that are used by more than one test
- Initializations (prefix values)
- Reset values (postfix values)

Different tests can use the objects without sharing the state

Objects used in test fixtures should be declared as instance variables

- They should be initialized in a `@Before` method
- Can be deallocated or reset in an `@After` method
- You can choose to initialize/reset them before/after *all* tests, or before/after *each* test

Example: Testing the Min Class

```
import java.util.*;

public class Min
{
  /**
    * Returns the mininum element in a list
    * @param list Comparable list of elements to search
    * @return the minimum element in the list
    * @throws NullPointerException if list is null or
    *         if any list elements are null
    * @throws ClassCastException if list elements are not mutually comparable
    * @throws IllegalArgumentException if list is empty
    */
    public static <T extends Comparable<? super T>> T min (List<? extends T> list)
    {
       // See next slide for method details
    }
}
```

Example: Testing the Min Class

```
...
public static <T extends Comparable<? super T>>
  T min (List<? extends T> list)
{
   if (list.size() == 0)
   {
      throw new IllegalArgumentException ("Min.min");
   }

   Iterator<? extends T> itr = list.iterator();
   T result = itr.next();

   if (result == null)
   {
       throw new NullPointerException ("Min.min");
   }

   while (itr.hasNext())
   {   // throws NPE, CCE as needed
       T comp = itr.next();
       if (comp.compareTo (result) < 0)
       {
           result = comp;
       }
   }
   return result;
}
...
```

Example: Testing the Min Class

Standard imports for JUnit classes

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

Test data and pre-test setup (prefix)

```
private List<String> list;   // test data

// Set up - Called before every test method.
@Before
public void setUp()
{
   list = new ArrayList<String>();
}
```

Post-test teardown (postfix)

- Not really useful in this case

```
// Tear down - Called after every test method.
@After
public void tearDown()
{
   list = null;   // redundant in this example
}
```

Intuitive Test Case Development

```
...
public static <T extends Comparable<? super T>>
  T min (List<? extends T> list)
{
   if (list.size() == 0)
   {
      throw new IllegalArgumentException ("Min.min");
   }

   Iterator<? extends T> itr = list.iterator();
   T result = itr.next();

   if (result == null)
   {
       throw new NullPointerException ("Min.min");
   }

   while (itr.hasNext())
   {   // throws NPE, CCE as needed
       T comp = itr.next();
       if (comp.compareTo (result) < 0)
       {
           result = comp;
       }
   }
   return result;
}
...
```

1. Null list?
2. Empty list?
3. Null value in first list element?
4. Only one element?
5. Incompatible type of list element?
6. Null value in later list element?
7. Keep previous result?
8. Replace previous result?

Intuitive Test Case Development

1. Test a null list

```
@Test public void testNullList()
{
    list = null; // reset list from @Before
    try {
        Min.min (list);
    }
    catch (NullPointerException e) {
        return;
    }
    fail ("NullPointerException expected");
}
```

2. Test an empty list

```
@Test (expected=IllegalArgumentException.class)
public void testEmptyList()
{
    Min.min (list);
}
```

Explicit try/catch or "expected" annotation are two different ways to assert that an exception is the expected result of the test

Intuitive Test Case Development

3. Test a **null** value in the first element of the list.

```
@Test (expected=NullPointerException.class)
public void testFirstNullElement()
{
    list.add (null);
    list.add (“foo”);
    Min.min (list);
}
```

4. Test a list with one element

```
@Test public void testOneElement()
{
    list.add (“foo”);
    Object obj = Min.min (list);
    assertEquals(“foo”, obj);
}
```

Intuitive Test Case Development

5. Test a list with more than one element that has an incompatible element type

```
@Test (expected=ClassCastException.class)
@SuppressWarnings(“unchecked”)
public void testIncompatibleType()
{
    List list = new ArrayList(); // generic type
    list.add (“foo”);
    list.add (“bar”);
    list.add (1);
    Min.min (list);
}
```

Requires redefining the list to an unspecified generic type

Intuitive Test Case Development

6. Test a list with multiple elements that has a null value in a later element of the list

```
@Test (expected=NullPointerException.class)
public void testLaterNullElement()
{
    list.add (“foo”);
    list.add (null);
    Min.min (list);
}
```

Intuitive Test Case Development

7. Test a list with more than one element that has the minimum element first (keep previous result)

```
@Test (expected=NullPointerException.class)
public void testLaterNullElement()
{
    list.add (“foo”);
    list.add (null);
    Min.min (list);
}
```

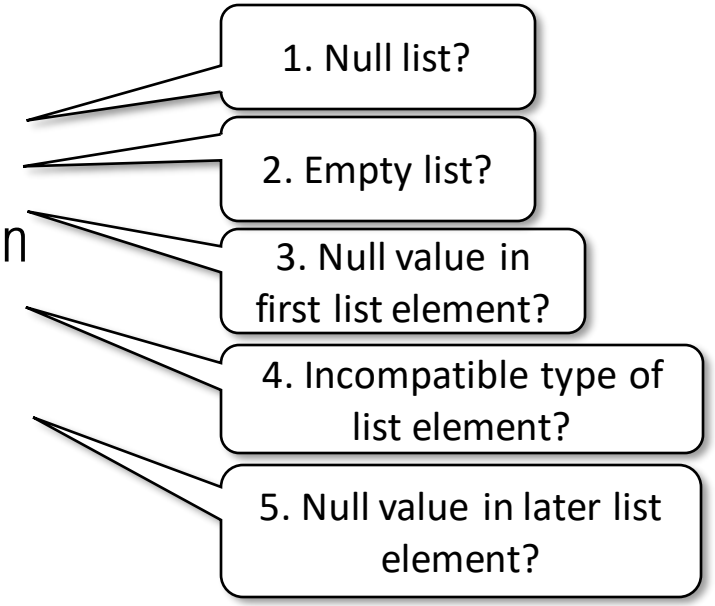
8. Test a list with more than one element and has the minimum element later (replace previous result).

```
@Test public void testLaterValueIsMin()
{
    list.add (“foo”);
    list.add (“bar”);
    list.add (“foobar”);
    Object obj = Min.min (list);
    assertEquals (“bar”, obj);
}
```

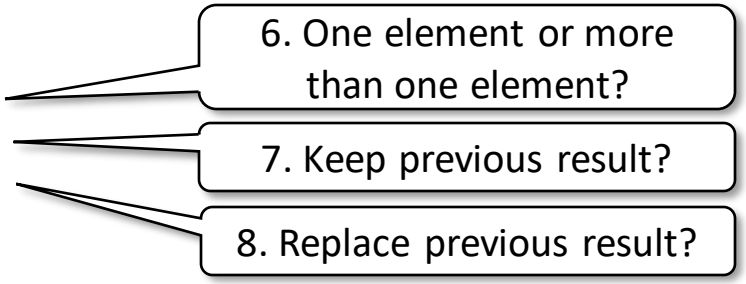
Intuitive Test Case Summary

Eight tests identified

Five tests expect an exception to be thrown



Three tests without exceptions that test “normal” execution



It’s typical to have more exception cases than happy-path cases.

Parameterized Tests

We often want to run tests where we make the same function call many times with different input data

- Imagine testing a method that adds two integers – the mechanics of the test is the same every time, but we might provide many pairs of test inputs

```
public class Adder
{
    public static int add (int i, int j)
    {
        return i+j;
    }
}
```

Parameterized Tests

```
@RunWith(Parameterized.class)
public class AdderTest
{
    // Define test inputs and expected output
    public int i, j, sum;

    // Create a constructor to set up the parameterized data
    public AdderTest(int i, int j, int sum)
    {
        this.i = i;
        this.j = j;
        this.sum = sum;
    }

    @Parameters
    public static Collection<Object[]> parameters()
    {
        return Arrays.asList(new Object[][] { { 1, 1, 2 }, { 2, 3, 5 } });
    }

    @Test
    public void testManyValues()
    {
        assertEquals (sum, Adder.add(i, j));
    }
}
```

Identifies this as a parameterized test

Parameterized Tests

```
@RunWith(Parameterized.class)
public class AdderTest
{
    // Define test inputs and expected output
    public int i, j, sum;

    // Create a constructor to set up the parameterized data
    public AdderTest(int i, int j, int sum)
    {
        this.i = i;
        this.j = j;
        this.sum = sum;
    }

    @Parameters
    public static Collection<Object[]> parameters()
    {
        return Arrays.asList(new Object[][] { { 1, 1, 2 }, { 2, 3, 5 } });
    }

    @Test
    public void testManyValues()
    {
        assertEquals (sum, Adder.add(i, j));
    }
}
```

Member variables to hold each instance of test inputs and expected output

Parameterized Tests

```
@RunWith(Parameterized.class)
public class AdderTest
{
    // Define test inputs and expected output
    public int i, j, sum;

    // Create a constructor to set up the parameterized data
    public AdderTest(int i, int j, int sum)
    {
        this.i = i;
        this.j = j;
        this.sum = sum;
    }

    @Parameters
    public static Collection<Object[]> parameters()
    {
        return Arrays.asList(new Object[][] { { 1, 1, 2 }, { 2, 3, 5 } });
    }

    @Test
    public void testManyValues()
    {
        assertEquals (sum, Adder.add(i, j));
    }
}
```

Constructor that initializes the member variables for inputs and expected output

Parameterized Tests

```
@RunWith(Parameterized.class)
public class AdderTest
{
    // Define test inputs and expected output
    public int i, j, sum;

    // Create a constructor to set up the parameterized data
    public AdderTest(int i, int j, int sum)
    {
        this.i = i;
        this.j = j;
        this.sum = sum;
    }

    @Parameters
    public static Collection<Object[]> parameters()
    {
        return Arrays.asList(new Object[][] { { 1, 1, 2 }, { 2, 3, 5 } });
    }

    @Test
    public void testManyValues()
    {
        assertEquals (sum, Adder.add(i, j));
    }
}
```

Definition of test parameters, including a tuple containing the inputs and output for each individual test (the constructor will be called once for each tuple)

Parameterized Tests

```
@RunWith(Parameterized.class)
public class AdderTest
{
    // Define test inputs and expected output
    public int i, j, sum;

    // Create a constructor to set up the parameterized data
    public AdderTest(int i, int j, int sum)
    {
        this.i = i;
        this.j = j;
        this.sum = sum;
    }

    @Parameters
    public static Collection<Object[]> parameters()
    {
        return Arrays.asList(new Object[][] { { 1, 1, 2 }, { 2, 3, 5 } });
    }

    @Test
    public void testManyValues()
    {
        assertEquals (sum, Adder.add(i, j));
    }
}
```

Test method that is *automatically* run once for each tuple in the test parameters

JUnit Theories

Theories in JUnit are an extension of parameterized tests

Theories are built on the Assume-Act-Assert model

- *Assumptions* (preconditions) limit values appropriately
- *Action* performs activity under test
- *Assertions* (postconditions) check result

Note – theories as shown are from JUnit4, property-based testing tools (like jqwik) are becoming the new way to build such tests

JUnit Theory Example

If we think about our `Adder` class, we might theorize that:

1. The sum of two positive numbers is greater than either number
2. The sum of two negative numbers is less than either number

Note that we're not specifying a specific expected output value

We can write test theories to verify that `Adder.add()` behaves this way

JUnit Theory Example

```
@RunWith(Theories.class)
public class AdderTheoryTest
{
    @DataPoints
    public static int[] testValues = { 0, 1, -2, 10, -11, 12345, -12346};

    @Theory public void sumOfPositivesIsGreaterThanEither
      (int i, int j)
    {
        assumeTrue (i > 0);           // Assume
        assumeTrue (j > 0);           // Assume
        int sum = Adder.add(i, j);    // Act
        assertTrue (sum > i);         // Assert
        assertTrue (sum > j);         // Assert
    }

    @Theory public void sumOfNegativesIsLessThanEither
      (int i, int j)
    {
        assumeTrue (i < 0);           // Assume
        assumeTrue (j < 0);           // Assume
        int sum = Adder.add(i, j);    // Act
        assertTrue (sum < i);         // Assert
        assertTrue (sum < j);         // Assert
    }
}
```

Identifies this as a theory-based test

JUnit Theory Example

```
@RunWith(Theories.class)
public class AdderTheoryTest
{
    @DataPoints
    public static int[] testValues = { 0, 1, -2, 10, -11, 12345, -12346};

    @Theory public void sumOfPositivesIsGreaterThanEither
      (int i, int j)
    {
        assumeTrue (i > 0);           // Assume
        assumeTrue (j > 0);           // Assume
        int sum = Adder.add(i, j);    // Act
        assertTrue (sum > i);         // Assert
        assertTrue (sum > j);         // Assert
    }

    @Theory public void sumOfNegativesIsLessThanEither
      (int i, int j)
    {
        assumeTrue (i < 0);           // Assume
        assumeTrue (j < 0);           // Assume
        int sum = Adder.add(i, j);    // Act
        assertTrue (sum < i);         // Assert
        assertTrue (sum < j);         // Assert
    }
}
```

Identifies possible test inputs; the theories will be executed with every combination of test inputs

JUnit Theory Example

```
@RunWith(Theories.class)
public class AdderTheoryTest
{
    @DataPoints
    public static int[] testValues = { 0, 1, -2, 10, -11, 12345, -12346};

    @Theory public void sumOfPositivesIsGreaterThanEither
      (int i, int j)
    {
        assumeTrue (i > 0);           // Assume
        assumeTrue (j > 0);           // Assume
        int sum = Adder.add(i, j);    // Act
        assertTrue (sum > i);         // Assert
        assertTrue (sum > j);         // Assert
    }

    @Theory public void sumOfNegativesIsLessThanEither
      (int i, int j)
    {
        assumeTrue (i < 0);           // Assume
        assumeTrue (j < 0);           // Assume
        int sum = Adder.add(i, j);    // Act
        assertTrue (sum < i);         // Assert
        assertTrue (sum < j);         // Assert
    }
}
```

Specifies the theory (test) name and parameters

JUnit Theory Example

```
@RunWith(Theories.class)
public class AdderTheoryTest
{
    @DataPoints
    public static int[] testValues = { 0, 1, -2, 10, -11, 12345, -12346};

    @Theory public void sumOfPositivesIsGreaterThanEither
      (int i, int j)
    {
        assumeTrue (i > 0);           // Assume
        assumeTrue (j > 0);           // Assume
        int sum = Adder.add(i, j);    // Act
        assertTrue (sum > i);         // Assert
        assertTrue (sum > j);         // Assert
    }

    @Theory public void sumOfNegativesIsLessThanEither
      (int i, int j)
    {
        assumeTrue (i < 0);           // Assume
        assumeTrue (j < 0);           // Assume
        int sum = Adder.add(i, j);    // Act
        assertTrue (sum < i);         // Assert
        assertTrue (sum < j);         // Assert
    }
}
```

Sets **assumptions** (preconditions) for this theory; only test inputs that satisfy the assumptions will be used to execute the theory. Valid test inputs are (1, 1), (1, 10), (1, 12245), (10, 1), (10, 10), (10,12345), (12345, 1), (12345, 10), (12345, 12345)

The **assumptions** for this theory allow test inputs (-2, -2), (-2, -11), (-2, -12246), (-11, -2), (-11, -11), (-11, -12346), (-12346, -2), (-12346, -11), (-12346, -12346)

JUnit Theory Example

```
@RunWith(Theories.class)
public class AdderTheoryTest
{
    @DataPoints
    public static int[] testValues = { 0, 1, -2, 10, -11, 12345, -12346};

    @Theory public void sumOfPositivesIsGreaterThanEither
      (int i, int j)
    {
        assumeTrue (i > 0);           // Assume
        assumeTrue (j > 0);           // Assume
        int sum = Adder.add(i, j);    // Act
        assertTrue (sum > i);         // Assert
        assertTrue (sum > j);         // Assert
    }

    @Theory public void sumOfNegativesIsLessThanEither
      (int i, int j)
    {
        assumeTrue (i < 0);           // Assume
        assumeTrue (j < 0);           // Assume
        int sum = Adder.add(i, j);    // Act
        assertTrue (sum < i);         // Assert
        assertTrue (sum < j);         // Assert
    }
}
```

Specifies **actions** to test

JUnit Theory Example

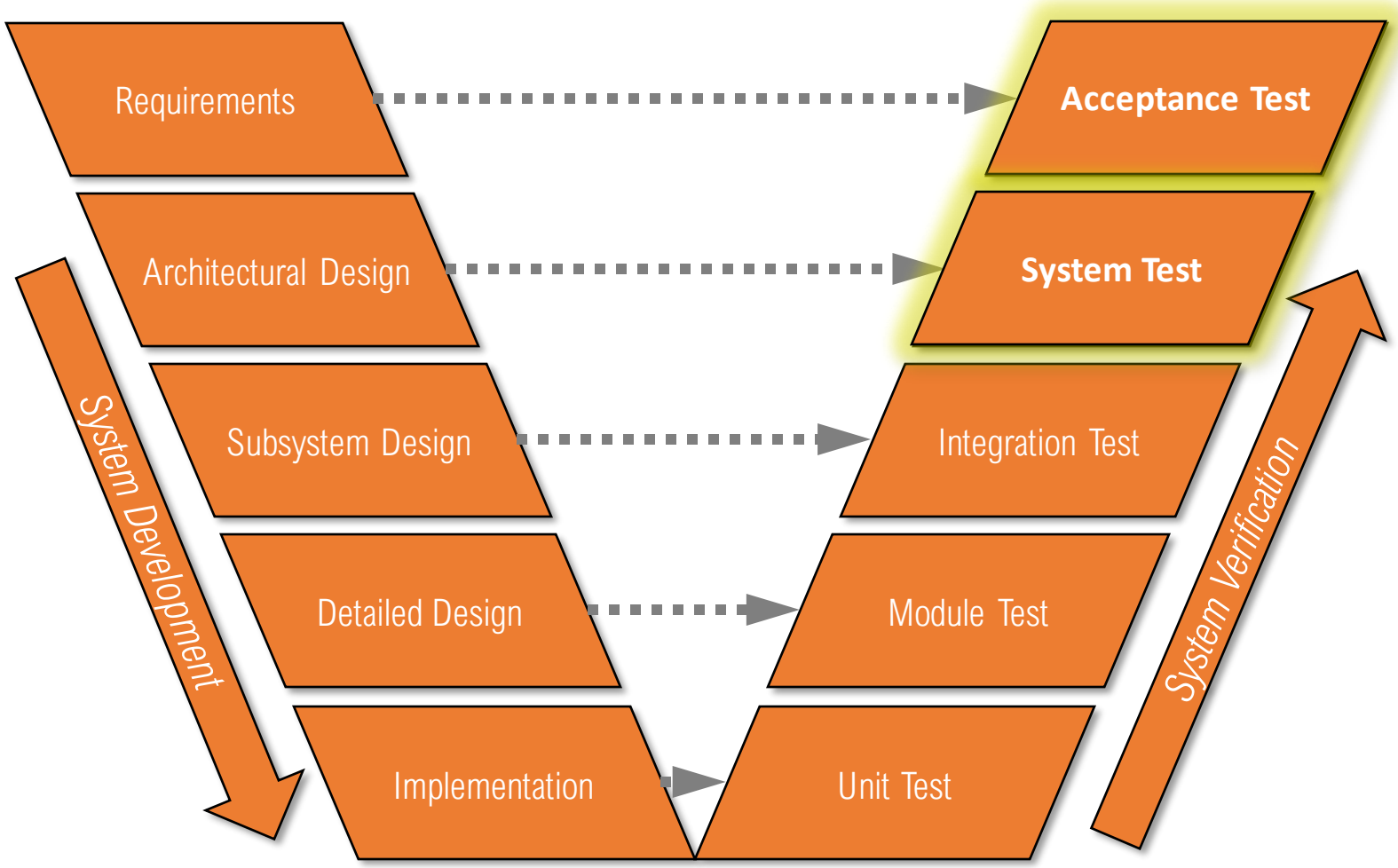
```
@RunWith(Theories.class)
public class AdderTheoryTest
{
    @DataPoints
    public static int[] testValues = { 0, 1, -2, 10, -11, 12345, -12346};

    @Theory public void sumOfPositivesIsGreaterThanEither
      (int i, int j)
    {
        assumeTrue (i > 0);           // Assume
        assumeTrue (j > 0);           // Assume
        int sum = Adder.add(i, j);    // Act
        assertTrue (sum > i);         // Assert
        assertTrue (sum > j);         // Assert
    }

    @Theory public void sumOfNegativesIsLessThanEither
      (int i, int j)
    {
        assumeTrue (i < 0);           // Assume
        assumeTrue (j < 0);           // Assume
        int sum = Adder.add(i, j);    // Act
        assertTrue (sum < i);         // Assert
        assertTrue (sum < j);         // Assert
    }
}
```

Specifies **assertions** (postconditions) to verify the theory

Automated system testing / AT



Automated System Testing / AT

JUnit is not a good fit at the system and acceptance testing level.

- Too technical; not easily understood by system engineers, architects, and most importantly customers
- Built around invoking classes/methods directly from the test framework, not around interacting with a running system

We need a test automation tool specifically for acceptance testing.

Cucumber

Cucumber is an acceptance test automation tool

- System-level tests are described using **Gherkin**, an English-like test specification language
- Gherkin tests can easily be written by non-development staff
- Customers can understand the tests

Gherkin Test Specifications

A Gherkin test specification consists of a:

1. **Scenario**: description of the functionality under test
2. **Given**: a precondition for an individual test
3. **When**: an input or stimulus to the system that initiates the test
4. **Then**: a postcondition or expected result for an individual test

Gherkin test specifications are written at a level of abstraction that the system engineers, users, and/or customers can understand.

Gherkin Example

Imagine an online ordering system. Two tests might be:

```
Scenario: CustomerLogin
  The customer logs in to the system and is
  directed to his/her account.

Given the ordering system is running
And the customer is on the login page

When the customer enters a correct username
  and password

Then the system validates the login
And the customer is directed to the shopping
  search page
```

```
Scenario: AddItemToShoppingCart
  The customer selects a displayed item
  and adds it to his/her shopping cart.

Given the ordering system is running
And the customer is logged in
And the customer is on item page 44216

When the customer selects add-item

Then item 44216 is added to the customers
  shopping cart
And the displayed order total is increased
  by 14.99
And the customer is given the choice to
  check out or continue shopping
```

Gherkin behind the curtain

Each statement in Gherkin is implemented in a Java ***step definition***

```
public class OrderingSystemStepDefinitions
{
    ...

    @When ("the customer enters a correct username and password")
    public void the_customer_enters_a_correct_username_and_password()
    {
        guiAgent.setTextField("username", "JohnSmith@test.com");
        guiAgent.setTextField("password", "testPassword");
        guiAgent.selectButton("login");
    }

    ...

    @Then ("the system validates the login|the customer is logged in")
    public void the_system_validates_the_login()
    {
        List<String> validatedUsers = authenticationAgent.getValidatedUsers();
        assertTrue (validatedUsers.contains("JohnSmith@test.com");
    }

    ...
}
```

Annotation performs a match to the Gherkin statement (actually a regular expression)

Java method defines what the Gherkin statement really means/does

Step definitions are actually JUnit tests!

Gherkin behind the curtain

Step definitions can take arguments from Gherkin

```
public class OrderingSystemStepDefinitions
{
    ...

    @When ("the customer is on item page (d+)")
    public void the_customer_is_on_item_page(int itemId)
    {
        guiAgent.displayItemPage(itemId);
    }

    ...

    @Then ("Then item (d+) is added to the customers shopping cart")
    public void then_item_is_added_to_the_customers_shopping_cart(int itemId)
    {
        List<Integer> cartContents = shoppingCartAgent.getShoppingCartItems();
        assertTrue (cartContents.contains(itemId));
    }

    ...
}
```

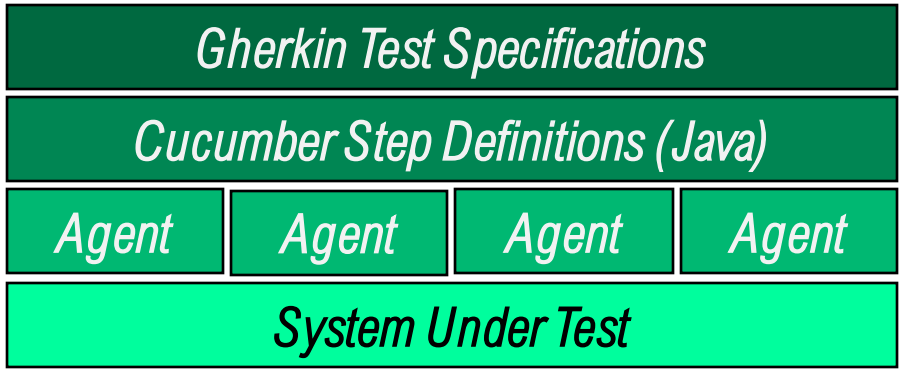
Regex can define variables

Variables appear in arg list

Gherkin Test Architecture

A common architecture for Gherkin tests uses independent agents outside of the step definitions; this keeps the step definitions high-level and generic

```
guiAgent.setTextField("username", "JohnSmith@test.com");
guiAgent.setTextField("password", "testPassword");
guiAgent.selectButton("login");
```



Summary

The only way to make testing efficient and effective is to automate as much as is practical

Test frameworks like JUnit and Cucumber provide simple ways to automate tests

- They let us focus our effort on *high-value test logic*, not *low-value test logistics*

But test frameworks can't tell us what to test!

- That's the purpose of *test design* and *test criteria*