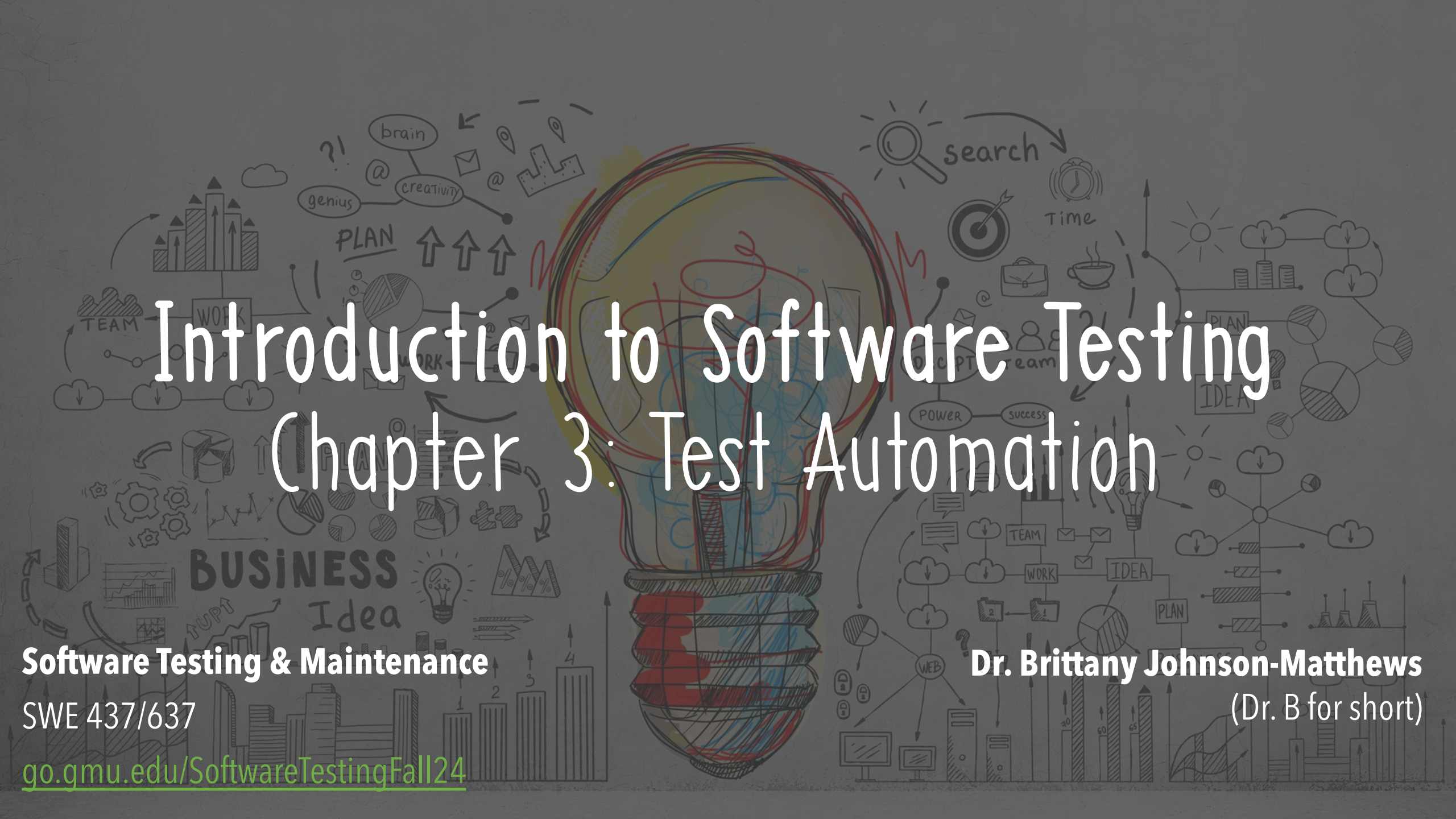


Introduction to Software Testing
Chapter 3: Test Automation

Software Testing & Maintenance

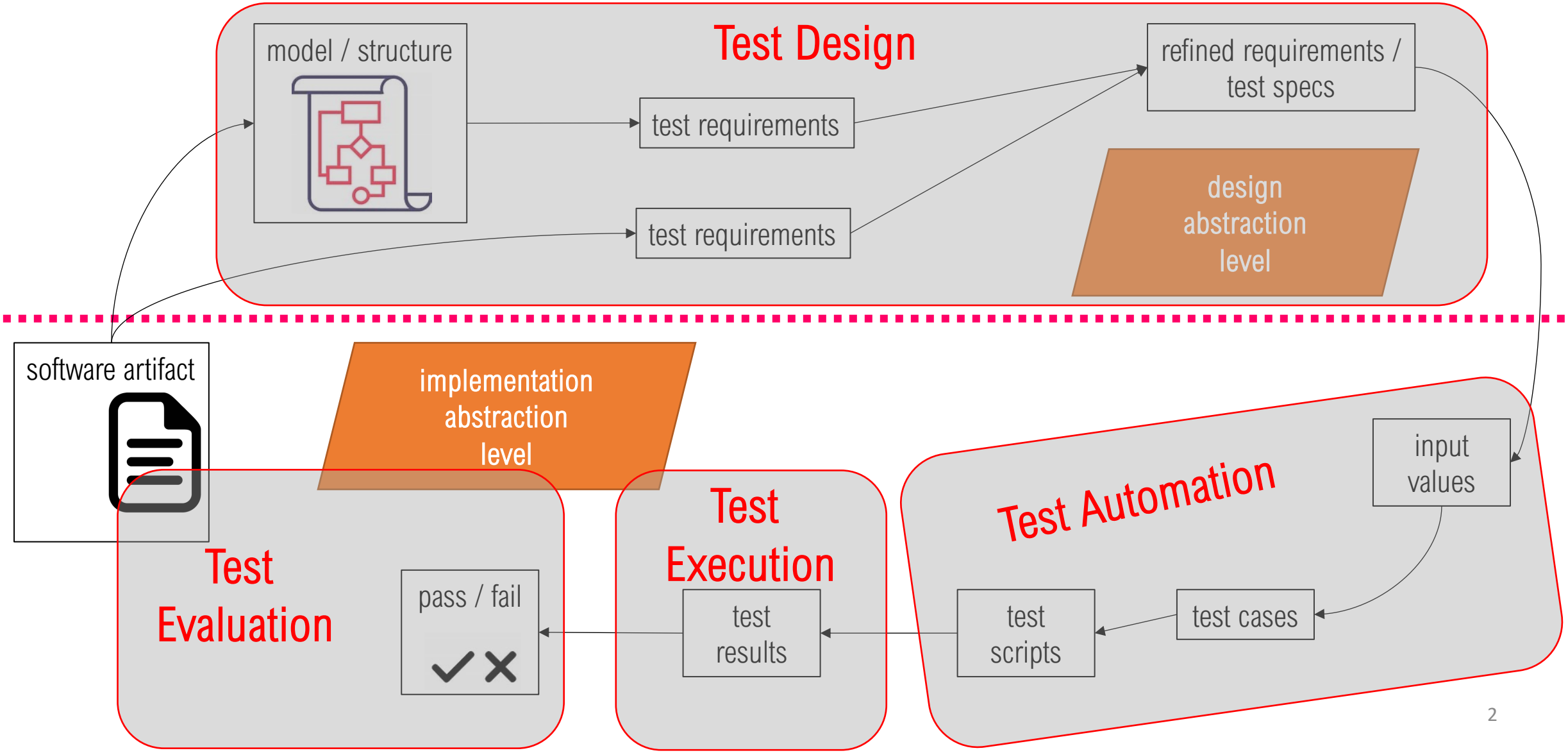
SWE 437/637

go.gmu.edu/SoftwareTestingFall24

Dr. Brittany Johnson-Matthews

(Dr. B for short)

MDTD Activities



Testing activities

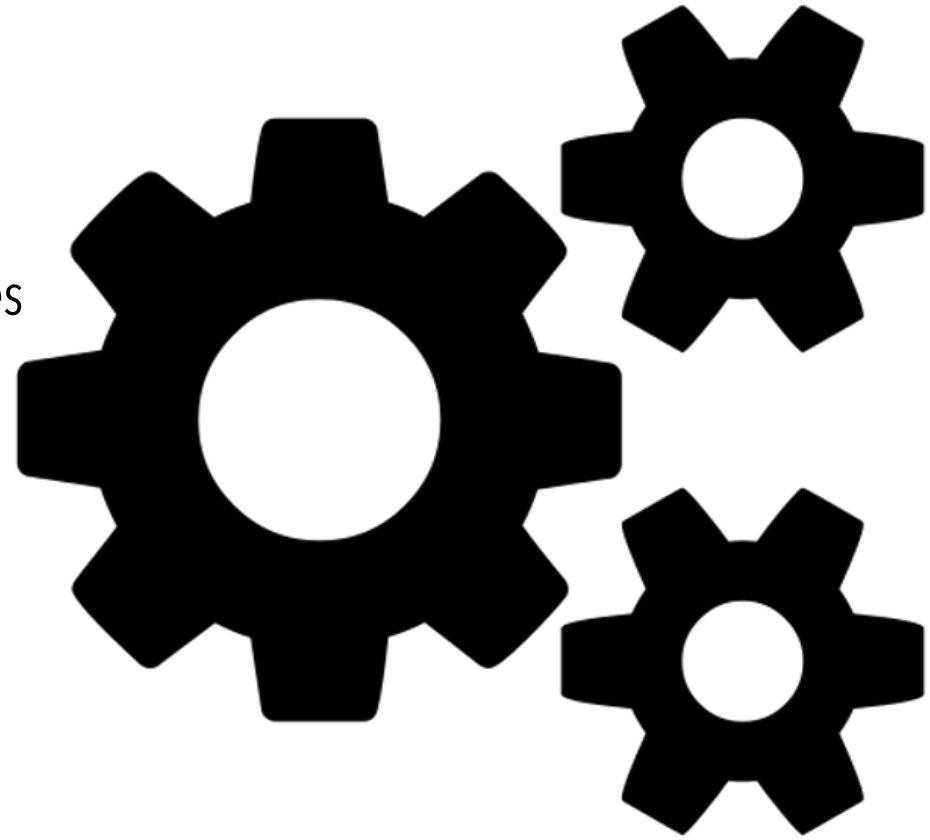
Testing can be broken up into **four** general types of activities

1. **Test design**
    - 1.a. **Criteria based**
    - 1.b. **Human-based**
2. **Test automation**
3. **Test execution**
4. **Test evaluation**

What is test automation?

Using software to control the testing

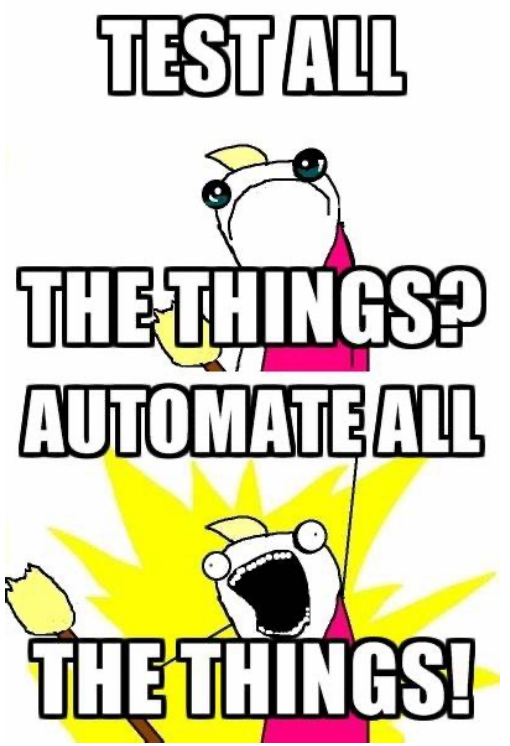
- **Setting up** test preconditions
- Test **execution**
- **Comparing** actual outcomes to predicted outcomes
- Test **reporting**



What is test automation?

Using software to control the testing

- **Setting up** test preconditions
- Test **execution**
- **Comparing** actual results to test results
- Test **reporting**



Reduces **cost**

Reduces **human error**

Reduces **variance** in test quality from different individuals

Significantly reduces the cost of **regression** testing

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.

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.

In other words, **how hard is it to find faults** in the software

Testability is dominated by **two** practical problems:

-How to *observe the results* of test execution (***observability***)

-How to *provide test values* to the software (***controllability***)

Observability & Controllability

Observability

How easy it is to observe the behavior of a program in term 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

Observability & Controllability

Observability

How easy it is to observe the behavior of a program in term 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

Observability & Controllability

Observability

How easy it is to observe the behavior of a program in term 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

Abstraction reduces controllability and observability

Observability & Controllability

In automated testing, we use ***prefix*** and ***postfix*** values to support 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** with a definite structure

Components of a test case (3.2)

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

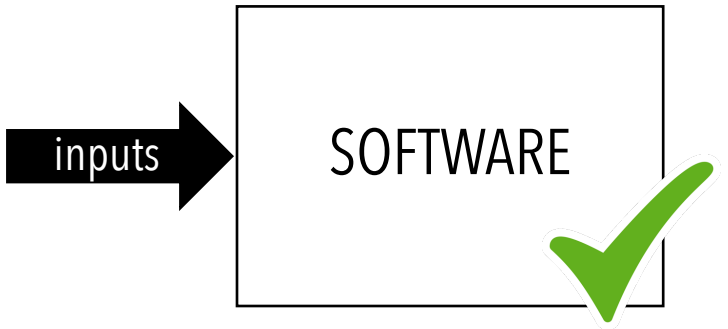
The result that will be produced by the test if the software behaves as expected

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

Affecting controllability & observability

Prefix values

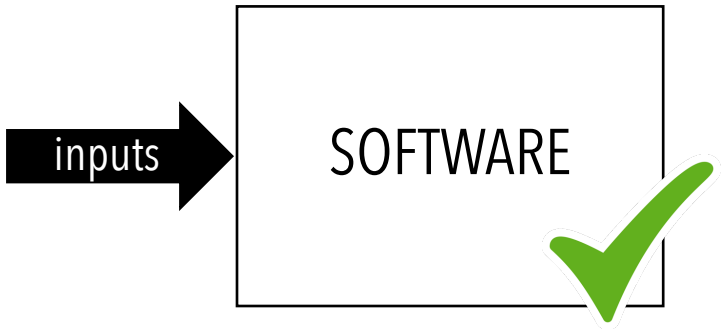
Inputs to put the software into the correct state to receive the test case values



Affecting controllability & observability

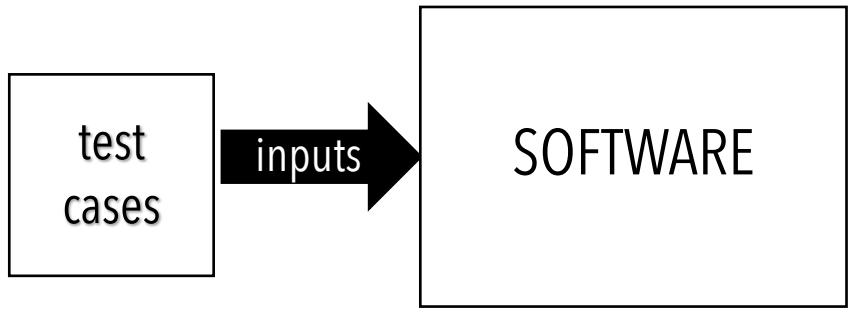
Prefix values

Inputs to put the software into the correct state to receive the test case values



Postfix values

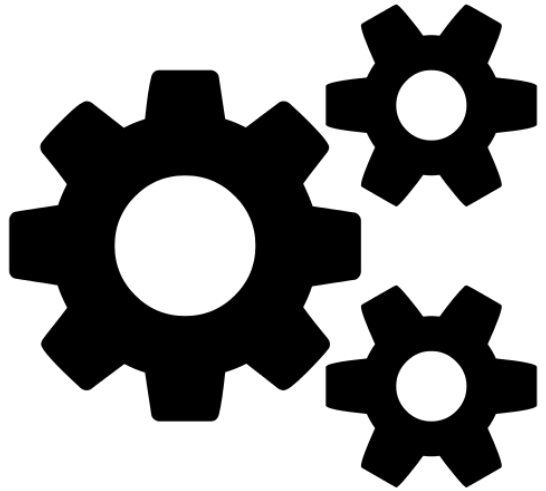
Inputs that must be sent to the software after the test case values



Putting it all 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 (or suite)

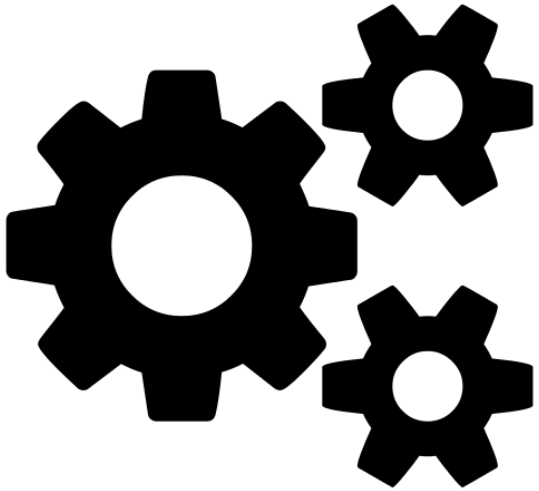
A set of test cases

Executable test script

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

Test automation framework (3.3)

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



MSTest

JUnit

JUnit for Java test automation

JUnit is a widely used, open source Java **automated testing framework**.

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 intended for **unit and integration testing**, not systems testing

JUnit for Java test automation

JUnit is a widely used, open source Java **automated testing framework**.

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 intended for **unit and integration testing**, not systems testing

Each test is embedded into one **test method**

A **test class** contains one or more test methods

Test classes **include**:

-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

Writing tests with JUnit

Use methods in 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)`

JUnit test fixtures

A **test fixture** is the **state** of the test

-Objects and variables that are used by more than one test

-Initializations (*prefix* values)

-Reset values (*postfix* values)

JUnit test fixtures

A **test fixture** is 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

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()
   {
       assertTrue("testAdd incorrect",
             5 == Calc.add(2, 3));
   }
}
```

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()
   {
       assertTrue("testAdd incorrect",
              5 == Calc.add(2, 3));
   }
}
```

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()
   {
       assertTrue("testAdd incorrect",
              5 == Calc.add(2, 3));
   }
}
```

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()
   {
        assertTrue("testAdd incorrect",
              5 == Calc.add(2, 3));
   }
}
```

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()
   {
        assertTrue("testAdd incorrect",
                5 == Calc.add(2, 3));
   }
}
```

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()
   {
        assertTrue("testAdd incorrect",
                   5 == Calc.add(2, 3));
   }
}
```

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()
   {
        assertTrue("testAdd incorrect",
                   5 == Calc.add(2, 3));
   }
}
```

Test values

Printed if assert fails

Expected output

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
    */
```

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;
}

}
```

In-class Exercise

Write **test inputs for the Min class**

Be sure to *include expected outputs*

Once you have enough tests, **write one in JUnit**.

If you're not sure how, *ask for help.*
If you have written JUnit tests, *help somebody who has not.*

You do not need to execute the tests.

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 Resources

Some JUnit tutorials

- https://www.tutorialspoint.com/junit/index.htm
- https://www.vogella.com/tutorials/JUnit/article.html
- https://www.parasoft.com/blog/junit-tutorial-setting-up-writing-and-running-java-unit-tests/

JUnit: download and documentation

- http://www.junit.org

Summary

The only way to make testing **efficient** as well as **effective** is to automate as much as possible

Test frameworks provide very simple ways to automate our tests

It is no "**silver bullet**" however…it does not solve the hard problem of testing:

What test values to use?

This is test design – the purpose of **test criteria**