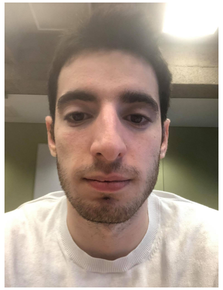
Jin L.C. Guo

M4 – Unit Testing

Image Source: https://upload.wikimedia.org/wikipedia/commons/2/2b/Cepaea_nemoralis_active_pair_on_tree_trunk.jpg

Covered by Aaron Sossin and Eric Liu



Aaron Sossin

Eric Liu

Slides provided by 2019 Winter TAs

Material from:

Introduction to Software Design with Java, by Martin Robillard, lecture notes for COMP 303. (LN)

(Module 4)

The Pragmatic Programmer by Andrew Hunt and David Thomas, Addison-Wesley, 2000. (PP)

(34, 43)

Clean Code by Robert C. Martin, Prentice Hall, 2008

(Chapter 9)

Objectives

- Be able to explain the foundational concepts of testing using the proper terminology;
- Understand type annotations and program reflection and be able to use them effectively;
- Be able to write unit tests with JUnit;
- Be able to approach more advanced testing problems requiring reflection or mock objects;
- Be able to understand the output of a test coverage tool such as EclEmma;
- Be able to understand basic test suite adequacy criteria and the relations between them;

Objectives – This Class (Feb 12.)

- Be able to explain the foundational concepts of testing using the proper terminology;
- Understand type annotations and program reflection and be able to use them effectively;
- Be able to write unit tests with JUnit;
- Be able to approach more advanced testing problems requiring reflection or mock objects;
- Be able to understand the output of a test coverage tool such as EclEmma;
- Be able to understand basic test suite adequacy criteria and the relations between them;

Objectives – Next Class (Feb 14.)

- Be able to explain the foundational concepts of testing using the proper terminology;
- Understand type annotations and program reflection and be able to use them effectively;
- Be able to write unit tests with JUnit;
- Be able to approach more advanced testing problems requiring reflection or mock objects;
- Be able to understand the output of a test coverage tool such as EclEmma;
- Be able to understand basic test suite adequacy criteria and the relations between them;

Objective 1: Foundational Concepts

Me: I don't need to test this functionality...

Functionality: *breaks immediately*

Me:

What is Testing? 🤔

Based on the idea that: software issues arise when code does not do what we expect.

- A fairly intuitive concept.
- You might have done some testing before!

“All software you write *will* be tested—if not by you and your team, then by the eventual users—so you might as well plan on testing it thoroughly … ”

The Pragmatic Programmer by Andrew Hunt and David Thomas, Addison-Wesley, 2000. (PP)

What is Testing? 🤔

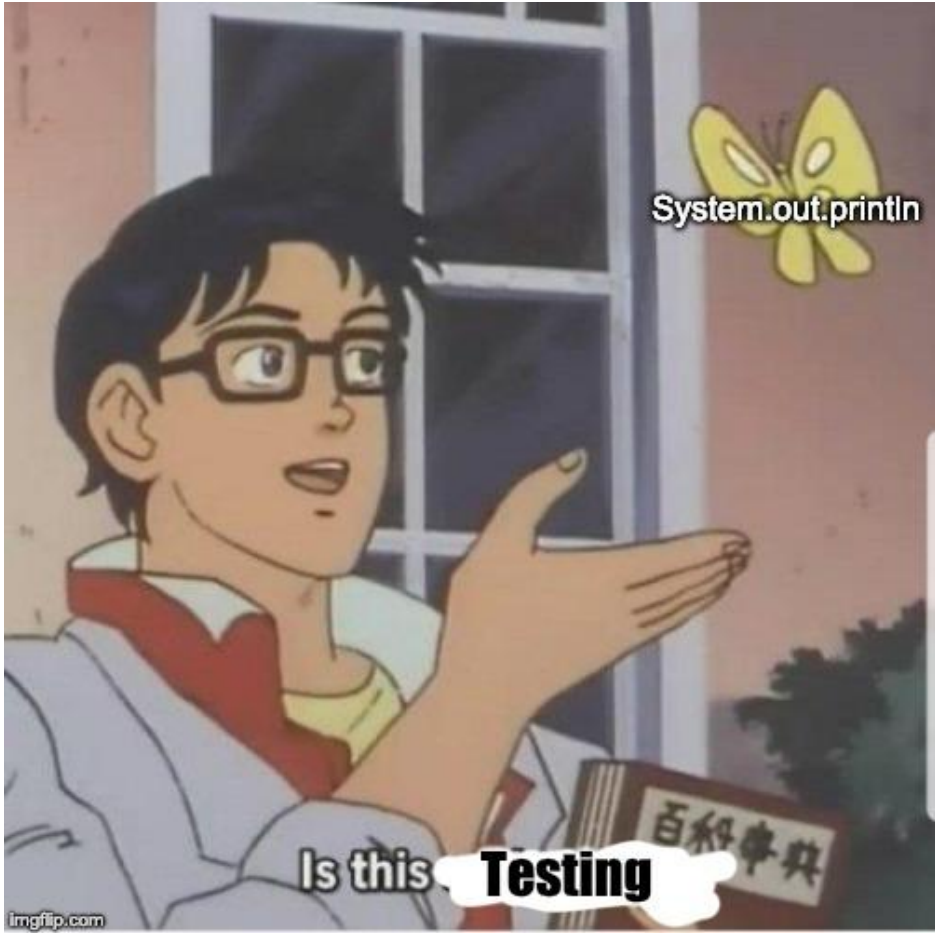
A process of <u>*automatically*</u> running code to verify expectations about it.

Activity: Why automatically?

Automatic: Easy to (re-)run

- Code is evolutionary: when we change code, are we (re-)introducing bugs?

- Code should run in multiple environments, with multiple configurations.
    - Consider web applications, phone apps


System.out.println
Is this Testing
百科事典
imgflip.com

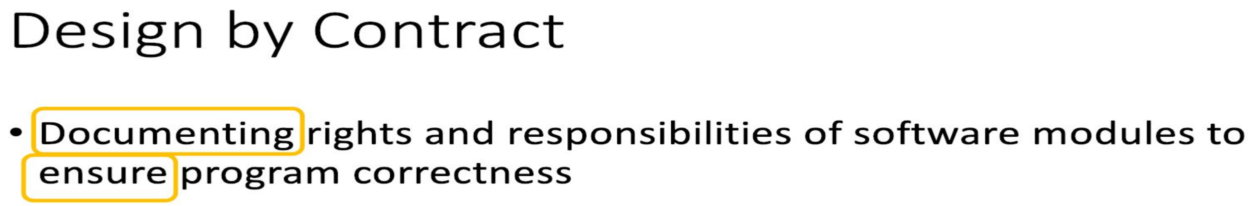
(Automatic) Testing is important!

How do we get started?

What should we be testing?

What should we test?

Recall: Design by Contract



Design by Contract

- Documenting rights and responsibilities of software modules to ensure program correctness

Contracts, a Perspective:

Contracts:

-Help us clarify (and verify) expectations

-Help us write code that is easy to test

What should we test?

Recall: Design by Contract

Problem: no built-in support…
We will see how to address this.

What specifically can we test?

Recall: Separation of concerns.

We have a design that minimizes (eliminates) changes in one place affecting other places.

Let’s test small, self-contained units of functionality.

Unit Testing 💡

Validate that each unit of the software performs as designed

An idea: We can test each unit of functionality in isolation – gives us confidence that when we put them together they won’t break.

Breaking things down into <u>units</u> (of functionality)

Activity:

What are some units of functionality in:

- A Car (Driving)
- Cards (playing Solitaire)
- …

Testing Code

In practice, you should test functions you write as individual units

Try to test every case (combination of inputs)

- Code coverage to be seen later

Objective 3: Unit Testing in Java

Unit Testing in Java

Using: JUnit(1)

- A Unit Testing framework
- Part of the xUnit family of frameworks (PHPUnit, PyUnit)

It’s a framework in that it provides structure for writing your tests.

(1) - https://junit.org/junit4/

Testing Structure

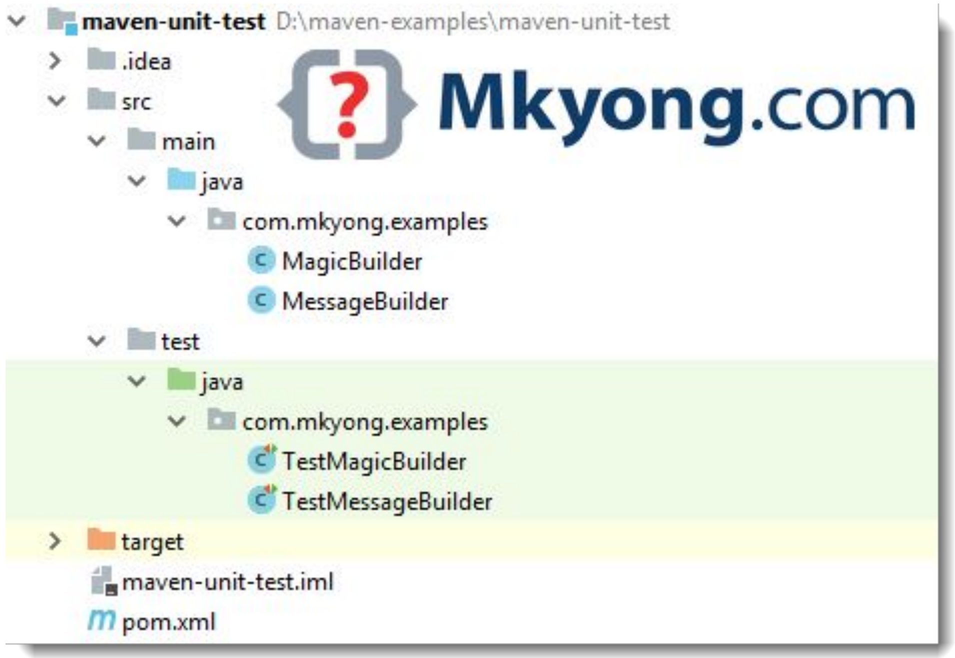
Generally, one “test” class per application class

In each class, test every contained function

Want to make sure no function has potentially undefined behaviour

Testing Structure

Mirror the document structure

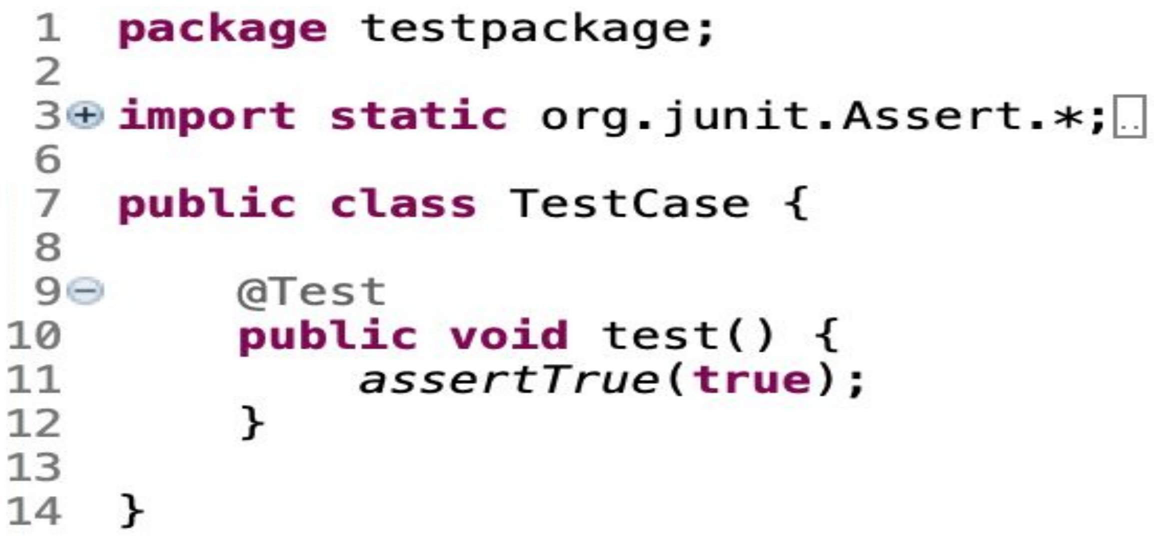


```
maven-unit-test D:\maven-examples\maven-unit-test
  .idea
  src
    main
      java
        com.mkyong.examples
          MagicBuilder
          MessageBuilder
    test
      java
        com.mkyong.examples
          TestMagicBuilder
          TestMessageBuilder
  target
  maven-unit-test.iml
  pom.xml
```

Let’s start with an example:

What does a unit test in JUnit look like? 🤔

A minimal test:



```
package testpackage;

import static org.junit.Assert.*;

public class TestCase {

    @Test
    public void test() {
        assertTrue(true);
    }

}
```

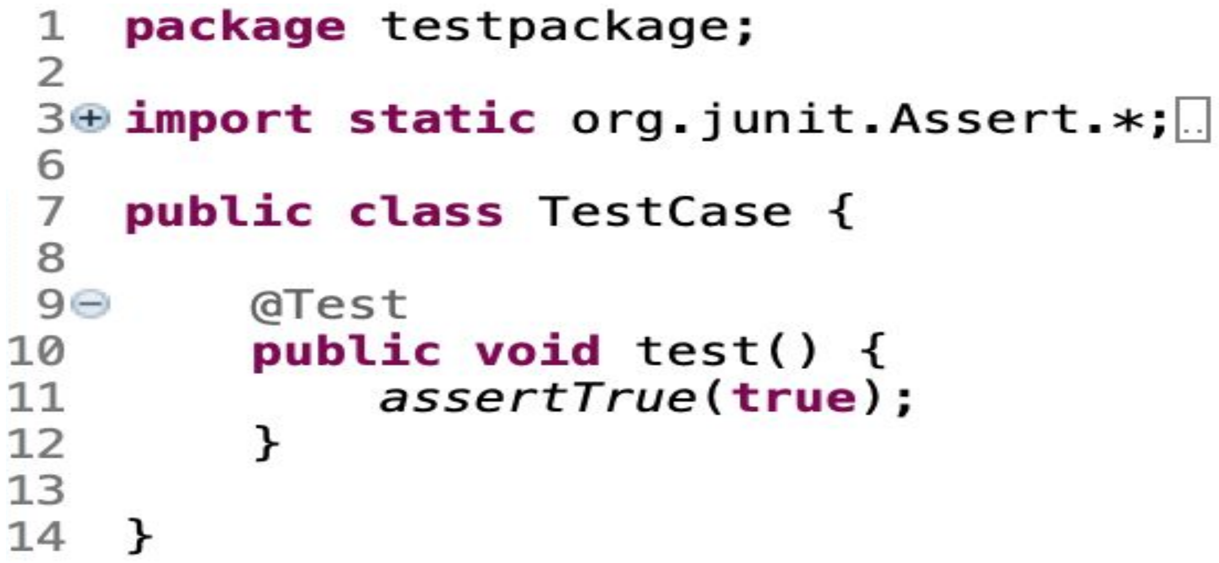
Note this is a fully functional unit test.

Things to Notice:

Still a regular class:

(line 9) This is new ->

No main method?



```
 1  package testpackage;
 2
 3  import static org.junit.Assert.*;
 6
 7  public class TestCase {
 8
 9      @Test
10      public void test() {
11          assertTrue(true);
12      }
13
14  }
```

@Test – An <u>Annotation</u>

Annotations:

- Are metadata, i.e. data about the program.

Recall: @Override - Interfaces, Subclasses

- Many more annotations (seen shortly)

Annotations – Cont’d

The @Test annotation tells JUnit that the following method can be run as a test case.

Annotations can take (optional) arguments:

@Test(expected=…)

@Test(timeout=…)

Annotations – Cont’d

Tutorial on the Oracle documentation website:

https://docs.oracle.com/javase/tutorial/java/annotations/index.html

How does JUnit run tests without a main method?

JUnit runner class

Anatomy of a Unit Test 🔬

Idea: We compare the result of executing a unit under test with an oracle.

Result: Obtained after running some specific code

Unit Under Test: an isolated, usually small part of code

Oracle: The expected result

Anatomy of a Unit Test 🔬

Example: testing the power function in the Math class

Unit Under Test: Math.pow(2, 3)

Result: the output of this function run on inputs 2 and 3

Oracle: the actual mathematical result of 2^3 = 8

Anatomy of a Unit Test 🔬

Check that the result matches the oracle

JUnit assert statements

- assertEquals
- assertTrue
- assertArrayEquals
- assertNull
- etc...

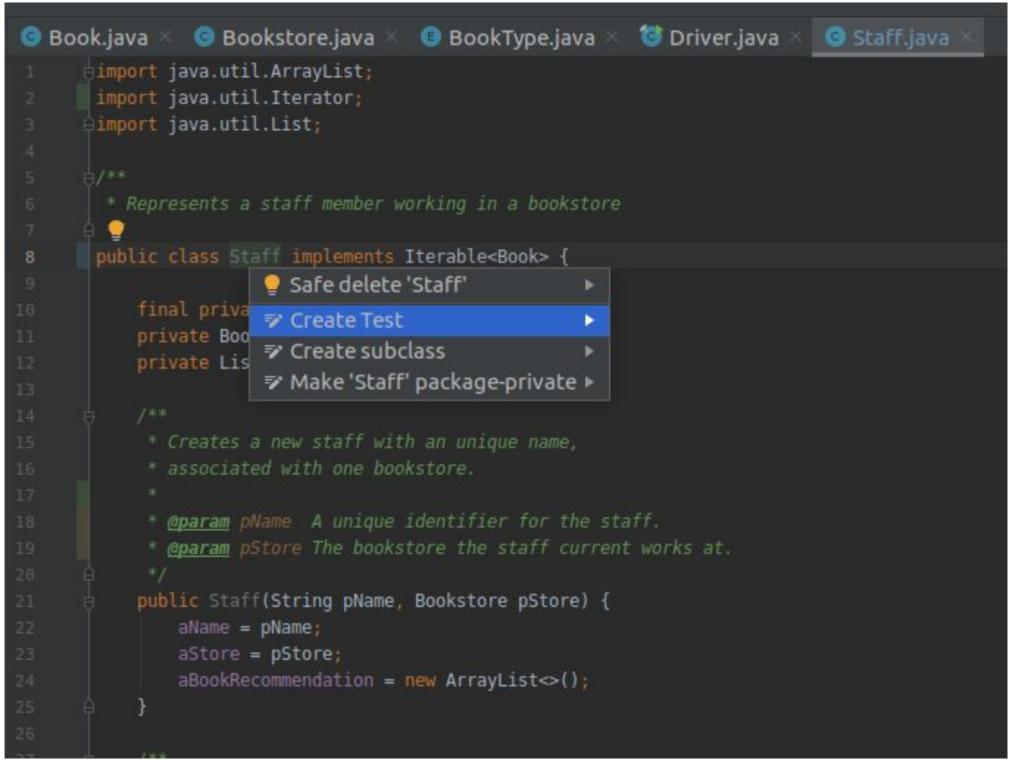
Using JUnit in IntellIJ

JUnit in IntelliJ

How do we create test suites (collection of unit tests) in IntelliJ to test our code?

Built-in and easy integration for JUnit5

JUnit in IntelliJ



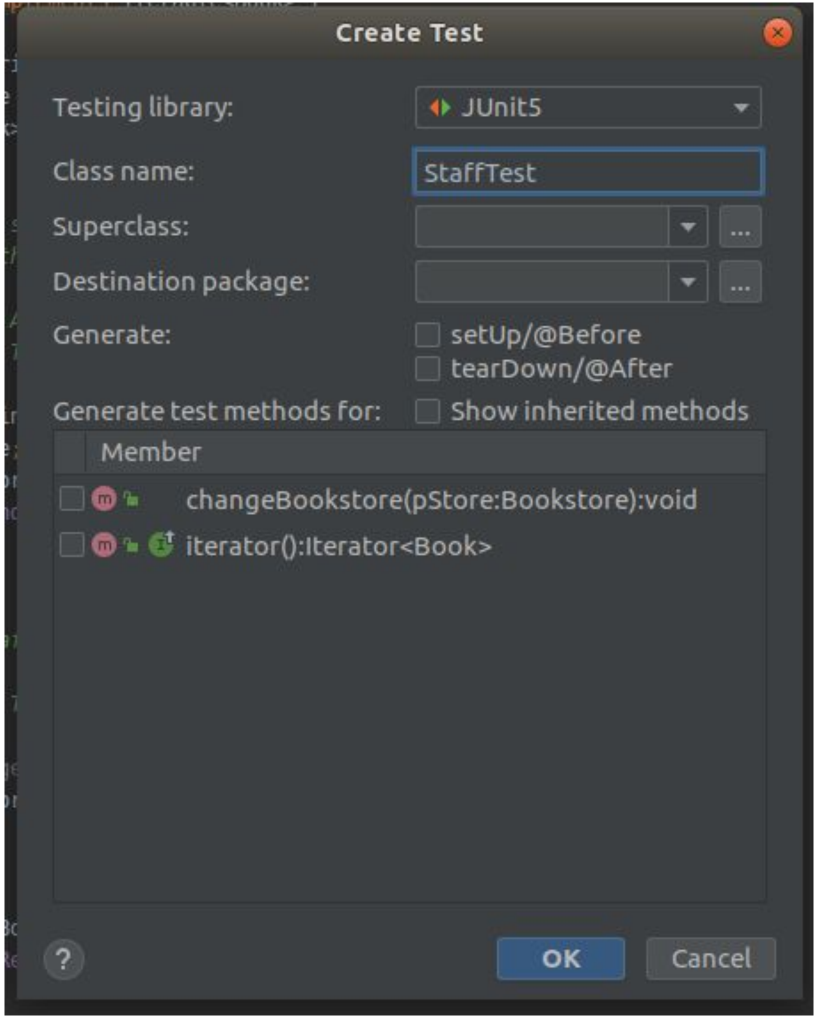
```
import java.util.ArrayList;
import java.util.Iterator;
import java.util.List;

/**
 * Represents a staff member working in a bookstore
 */
public class Staff implements Iterable<Book> {

    final priva
    private Boo
    private Lis

    /**
     * Creates a new staff with an unique name,
     * associated with one bookstore.
     *
     * @param pName  A unique identifier for the staff.
     * @param pStore The bookstore the staff current works at.
     */
    public Staff(String pName, Bookstore pStore) {
        aName = pName;
        aStore = pStore;
        aBookRecommendation = new ArrayList<>();
    }
```

alt-enter when highlighting a line with a class declaration

JUnit in IntelliJ

Very integrated and intuitive setup

Choose which methods to generate test declarations for



Create Test

Testing library: JUnit5

Class name: StaffTest

Superclass:

Destination package:

Generate: setUp/@Before, tearDown/@After

Generate test methods for: Show inherited methods

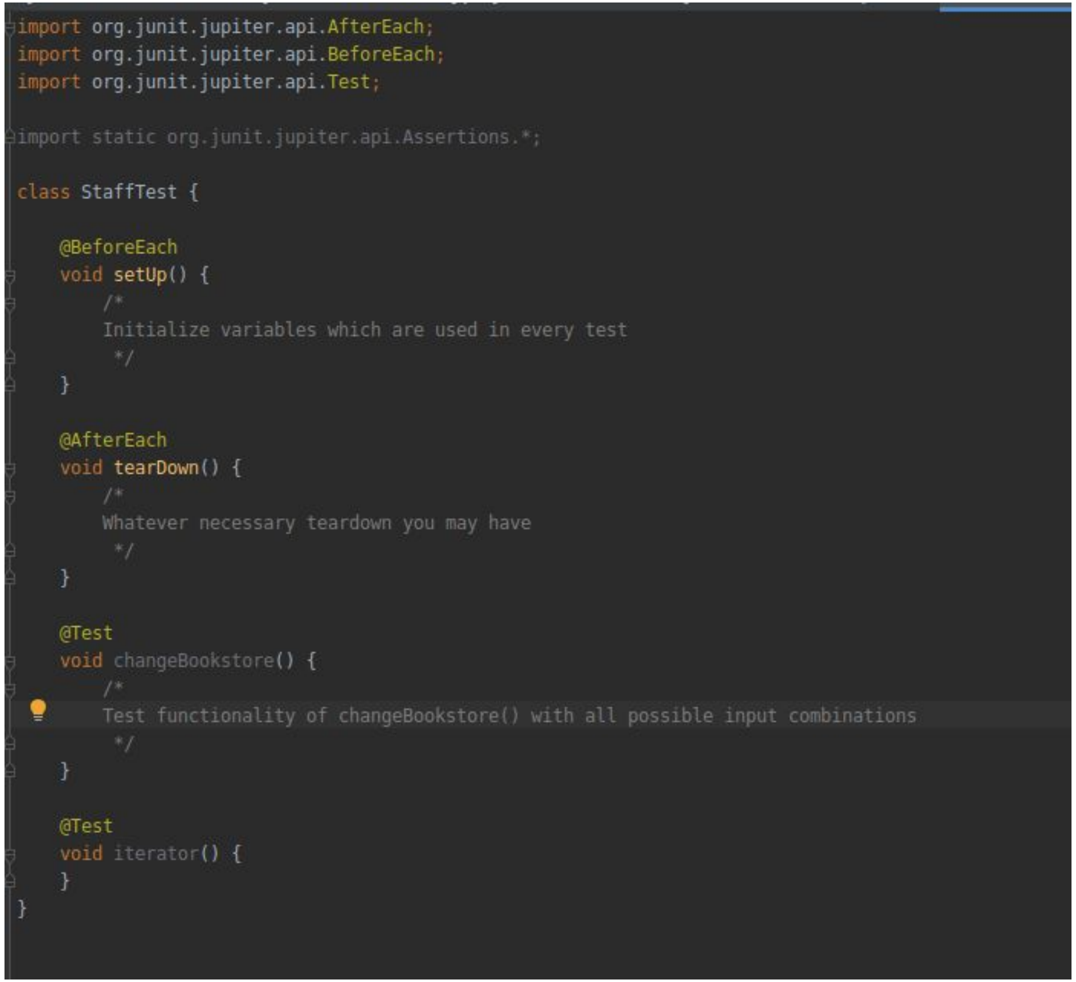
Member
changeBookstore(pStore:Bookstore):void
iterator():Iterator<Book>

JUnit in IntelliJ

Basic structure of a test suite in IntelliJ

More annotations

@BeforeEach, @AfterEach, @BeforeAll



```
import org.junit.jupiter.api.AfterEach;
import org.junit.jupiter.api.BeforeEach;
import org.junit.jupiter.api.Test;

import static org.junit.jupiter.api.Assertions.*;

class StaffTest {

    @BeforeEach
    void setUp() {
        /*
        Initialize variables which are used in every test
         */
    }

    @AfterEach
    void tearDown() {
        /*
        Whatever necessary teardown you may have
         */
    }

    @Test
    void changeBookstore() {
        /*
        Test functionality of changeBookstore() with all possible input combinations
         */
    }

    @Test
    void iterator() {
    }
}
```

Flash Light Example

Let’s model a (simple) Flash Light 🔦

It has one button that toggles it on and off:

CODE DEMO

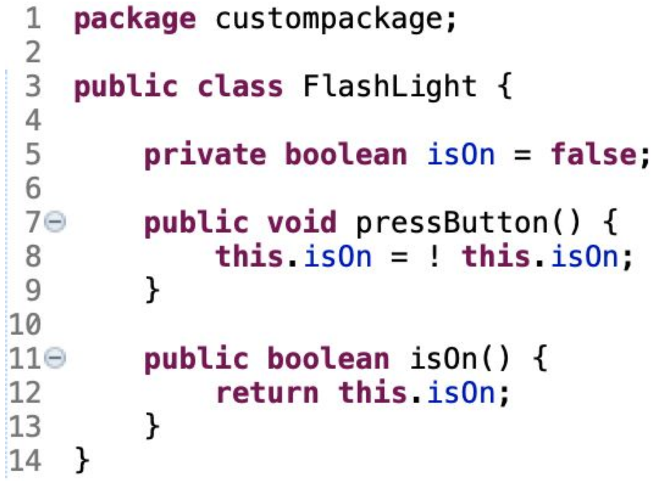
Transition

Activity

1. Identify UUTs

2. Test the UUT against some oracle (The term *oracle* designates the correct or expected result of the execution of a UUT) ie determine the expected behavior of each UUT

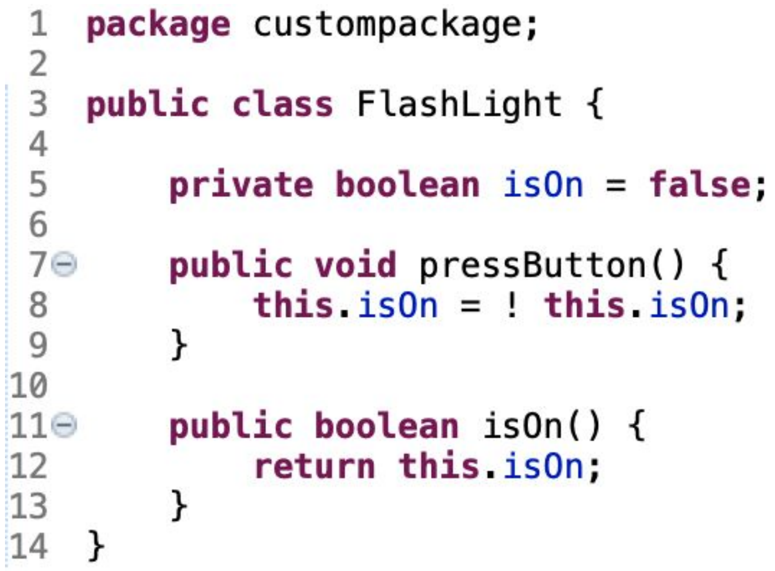
3. Write the tests



```java
package custompackage;

public class FlashLight {

    private boolean isOn = false;

    public void pressButton() {
        this.isOn = ! this.isOn;
    }

    public boolean isOn() {
        return this.isOn;
    }
}
```

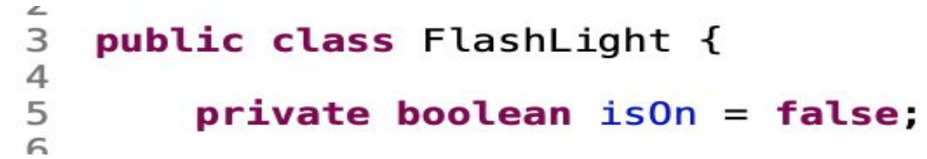
Step 1: Identify <u>UUTs</u>

- pressButton
- The constructor (Technically?) – We'd just like to test the default state of the Flash Light.



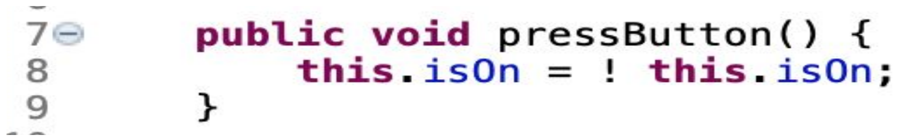
```java
package custompackage;

public class FlashLight {

    private boolean isOn = false;

    public void pressButton() {
        this.isOn = ! this.isOn;
    }

    public boolean isOn() {
        return this.isOn;
    }
}
```

Step 2: Determine expected behavior of UUTs



```
public class FlashLight {

    private boolean isOn = false;
```

Expect it to be off by default



```
    public void pressButton() {
        this.isOn = ! this.isOn;
    }
```

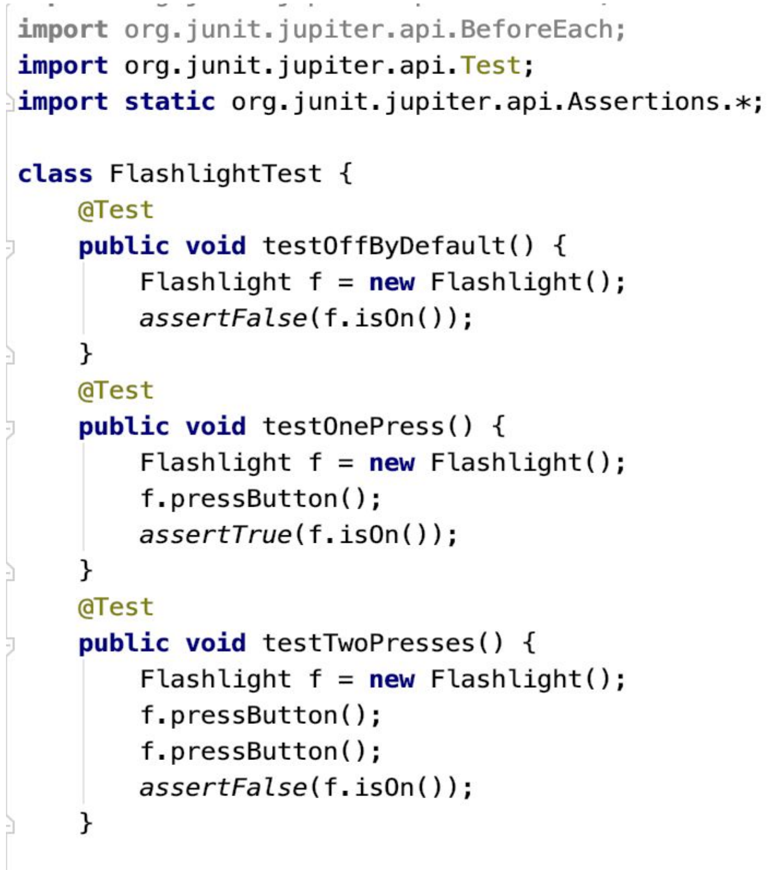
Expect: it to be on after a press

it to be off after two presses

It to be on after three presses

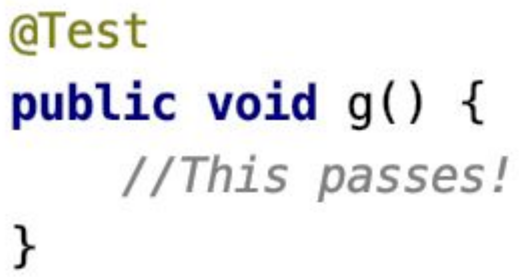
…

Step 3: Let’s write this in JUnit (Demo)



```java
import org.junit.jupiter.api.BeforeEach;
import org.junit.jupiter.api.Test;
import static org.junit.jupiter.api.Assertions.*;

class FlashlightTest {
    @Test
    public void testOffByDefault() {
        Flashlight f = new Flashlight();
        assertFalse(f.isOn());
    }
    @Test
    public void testOnePress() {
        Flashlight f = new Flashlight();
        f.pressButton();
        assertTrue(f.isOn());
    }
    @Test
    public void testTwoPresses() {
        Flashlight f = new Flashlight();
        f.pressButton();
        f.pressButton();
        assertFalse(f.isOn());
    }
```

FYI:



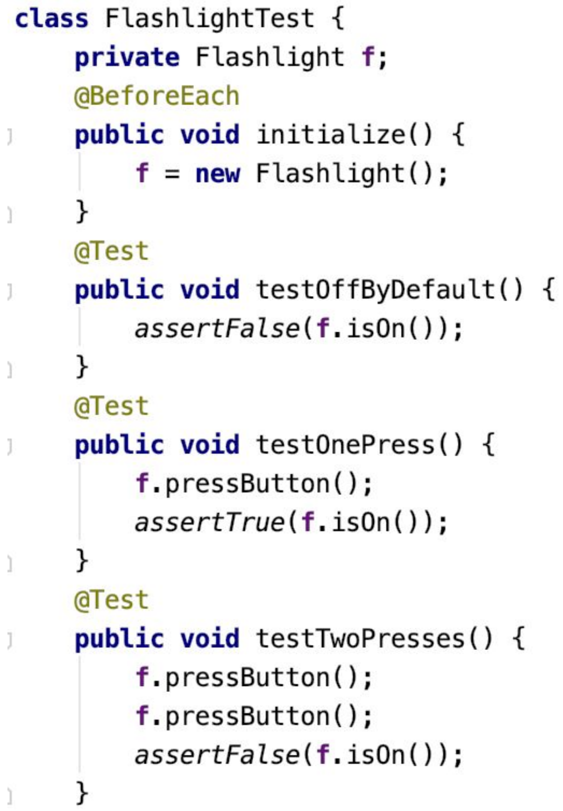
```java
@Test
public void g() {
    //This passes!
}
```

Demo similar to what may be on exam

What did we observe?

- Independency of Tests
  - JUnit provides no ordering guarantee of any kind for the execution of unit tests
  - Tests need to be independent from one another
  - Example: tests on OnePress and TwoPress need to be two independent and complete tests
- Repeat Initialization
  - FlashLight f = new FlashLight()
  - We can improve this...

FlashLightTest Version 2.0



```
class FlashlightTest {
    private Flashlight f;
    @BeforeEach
    public void initialize() {
        f = new Flashlight();
    }
    @Test
    public void testOffByDefault() {
        assertFalse(f.isOn());
    }
    @Test
    public void testOnePress() {
        f.pressButton();
        assertTrue(f.isOn());
    }
    @Test
    public void testTwoPresses() {
        f.pressButton();
        f.pressButton();
        assertFalse(f.isOn());
    }
```

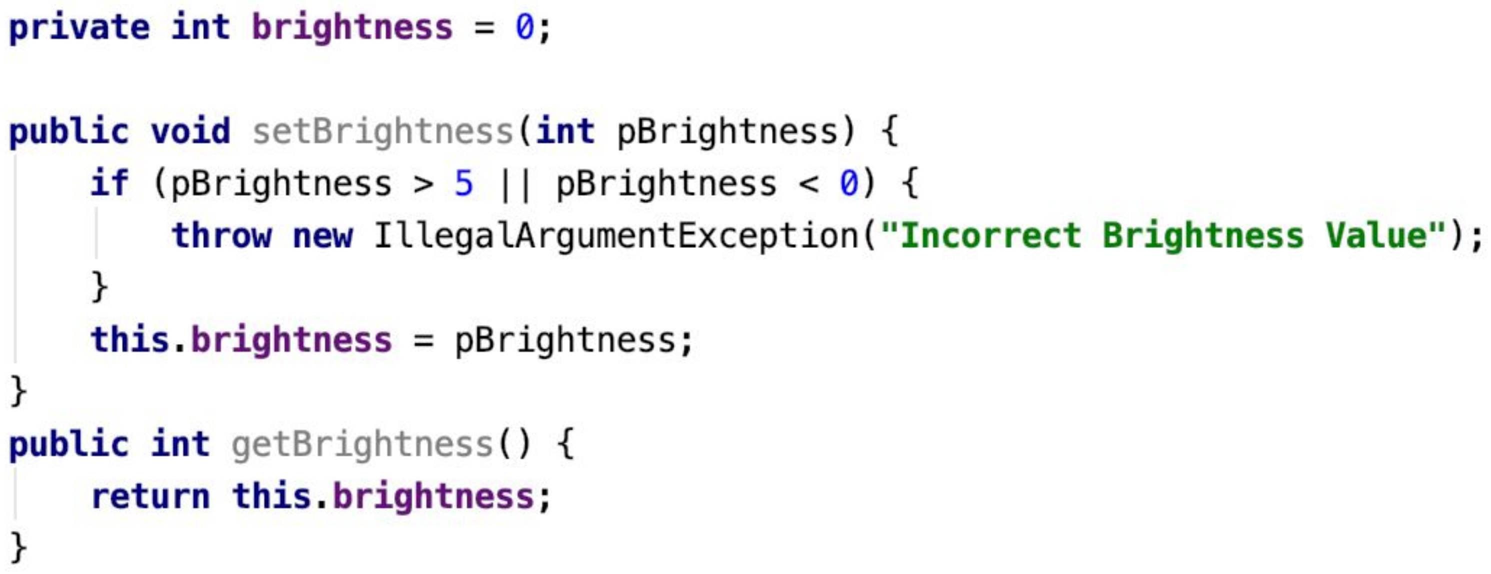
Keypoints in JUnit

- Independency of Tests
  - JUnit provides no ordering guarantee of any kind for the execution of unit tests
  - Tests need to be independent from one another
  - Example: tests on onepress and twopress need to be two independent and complete tests
- @BeforeEach runs before every @Test
  - Other useful annotations includes @BeforeClass, @After, @AfterClass

Tests and Exceptional Conditions

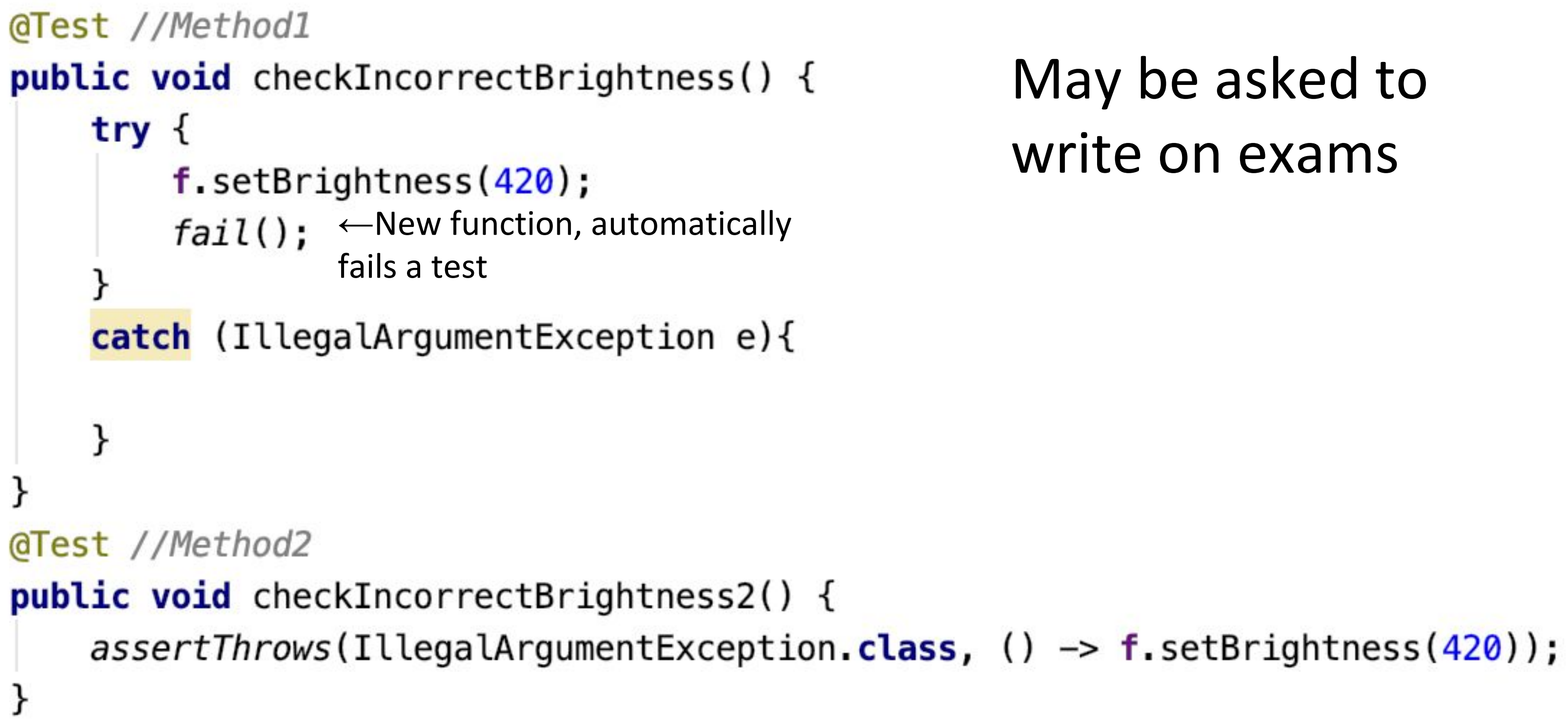
Next, we modify the code for FlashLight Class a little...

To add brightness level (0-5), where 0 means off and 5 means brightest



```java
private int brightness = 0;

public void setBrightness(int pBrightness) {
    if (pBrightness > 5 || pBrightness < 0) {
        throw new IllegalArgumentException("Incorrect Brightness Value");
    }
    this.brightness = pBrightness;
}
public int getBrightness() {
    return this.brightness;
}
```

Tests and Exceptional Conditions



```java
@Test //Method1
public void checkIncorrectBrightness() {
    try {
        f.setBrightness(420);
        fail();
    }
    catch (IllegalArgumentException e){

    }
}
@Test //Method2
public void checkIncorrectBrightness2() {
    assertThrows(IllegalArgumentException.class, () -> f.setBrightness(420));
}
```

←New function, automatically fails a test

May be asked to write on exams

Keypoints in JUnit

- Independency of Tests
  - JUnit provides no ordering guarantee of any kind for the execution of unit tests
  - Tests need to be independent from one another
  - Example: tests on onepress and twopress need to be two independent and complete tests
- @Before runs before every @Test
  - Other useful annotations includes @BeforeClass, @After, @AfterClass
- 2 ways to test exceptions
  - assertThrows
  - Try and catch + fail()

Encapsulation and Unit Testing

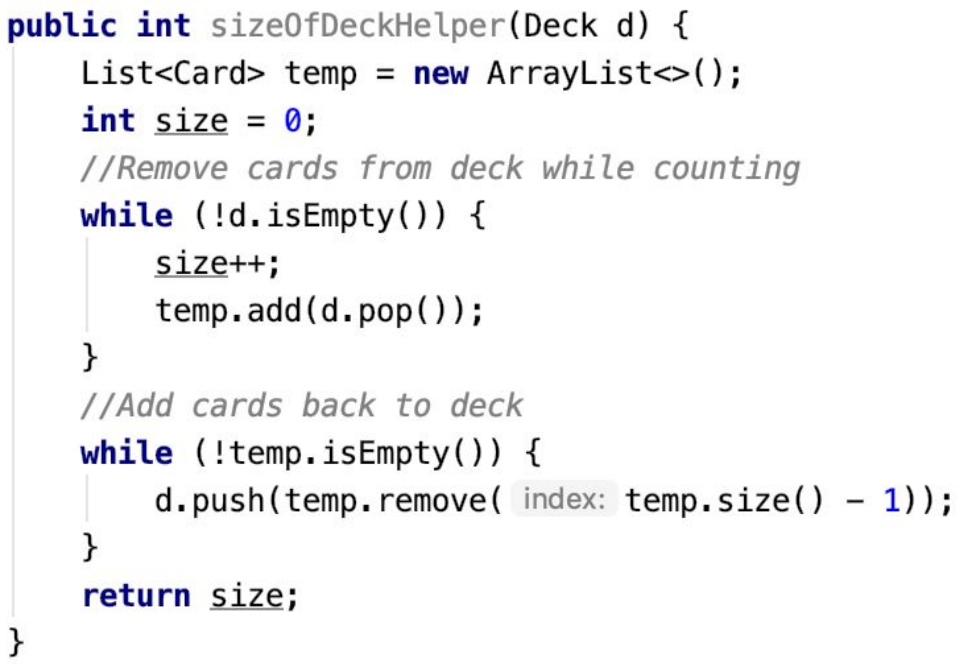
- Tests are in separate classes than the code you are testing (no access to private attributes)
- What if we want to test something private?

RULE OF THUMB: create helper methods in test class opposed to changing code in tested class.

Encapsulation and Unit Testing (cont’d)

Example: You wish to test the size of Deck class after the execution of push() and pop() methods. But, there is no non-private attribute denoting size of deck.

Solution: Create a ‘helper’ method in DeckTester.java that returns the size of the deck.

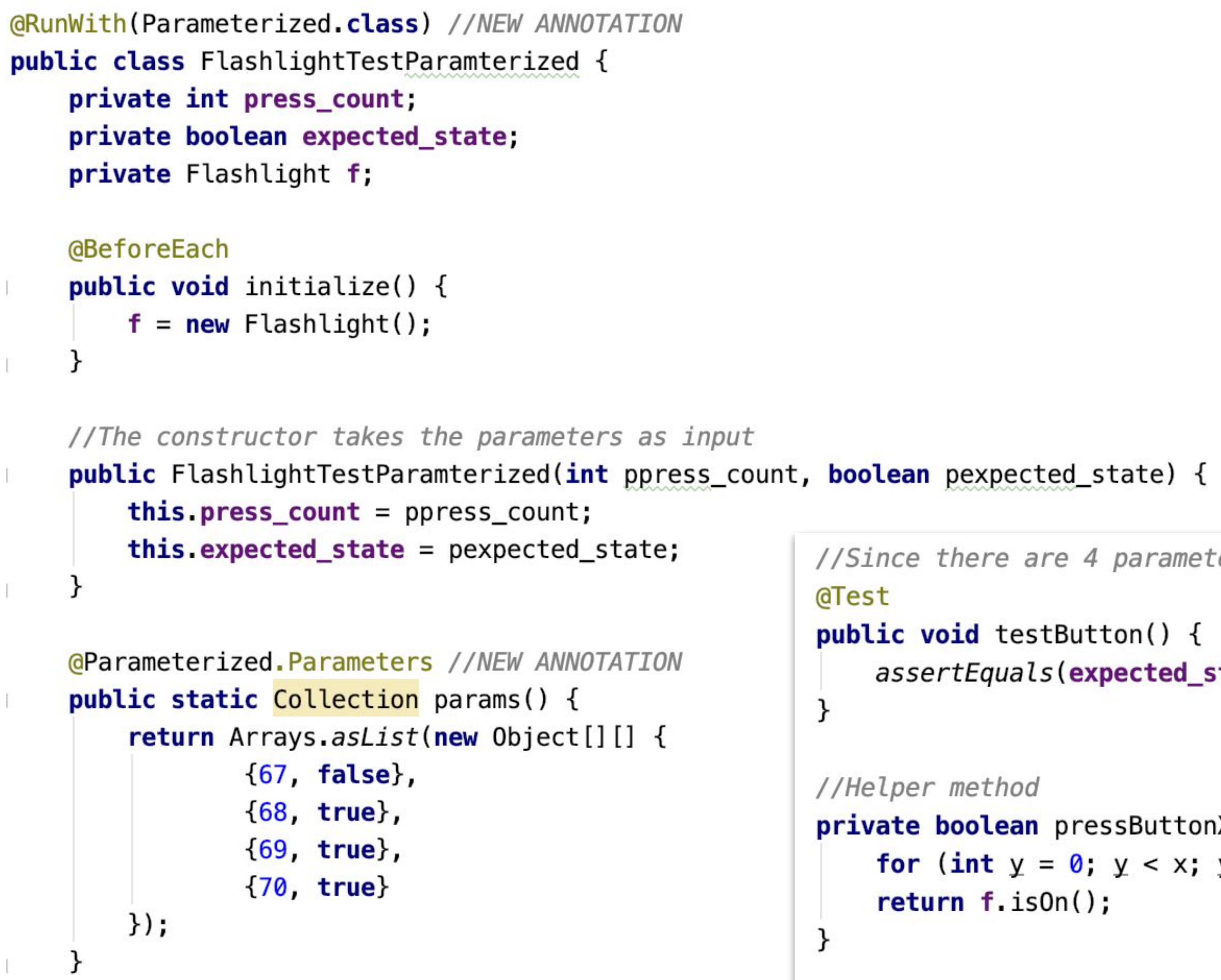


```java
public int sizeOfDeckHelper(Deck d) {
    List<Card> temp = new ArrayList<>();
    int size = 0;
    //Remove cards from deck while counting
    while (!d.isEmpty()) {
        size++;
        temp.add(d.pop());
    }
    //Add cards back to deck
    while (!temp.isEmpty()) {
        d.push(temp.remove( index: temp.size() - 1));
    }
    return size;
}
```

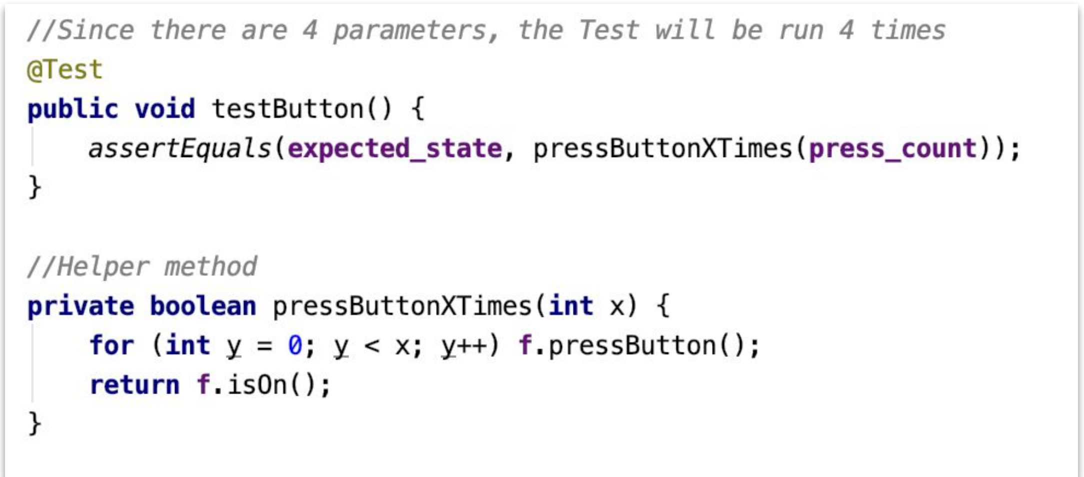
Parameterized Tests

- JUnit allows you to use parameters in a tests class
- Helps developers save time when executing the same tests which differ only in their inputs and expected results

DEMO



```java
@RunWith(Parameterized.class) //NEW ANNOTATION
public class FlashlightTestParamterized {
    private int press_count;
    private boolean expected_state;
    private Flashlight f;

    @BeforeEach
    public void initialize() {
        f = new Flashlight();
    }

    //The constructor takes the parameters as input
    public FlashlightTestParamterized(int ppress_count, boolean pexpected_state) {
        this.press_count = ppress_count;
        this.expected_state = pexpected_state;
    }

    @Parameterized.Parameters //NEW ANNOTATION
    public static Collection params() {
        return Arrays.asList(new Object[][] {
                {67, false},
                {68, true},
                {69, true},
                {70, true}
        });
    }
```



```java
//Since there are 4 parameters, the Test will be run 4 times
@Test
public void testButton() {
    assertEquals(expected_state, pressButtonXTimes(press_count));
}

//Helper method
private boolean pressButtonXTimes(int x) {
    for (int y = 0; y < x; y++) f.pressButton();
    return f.isOn();
}
```

Key points of Parameterized Tests

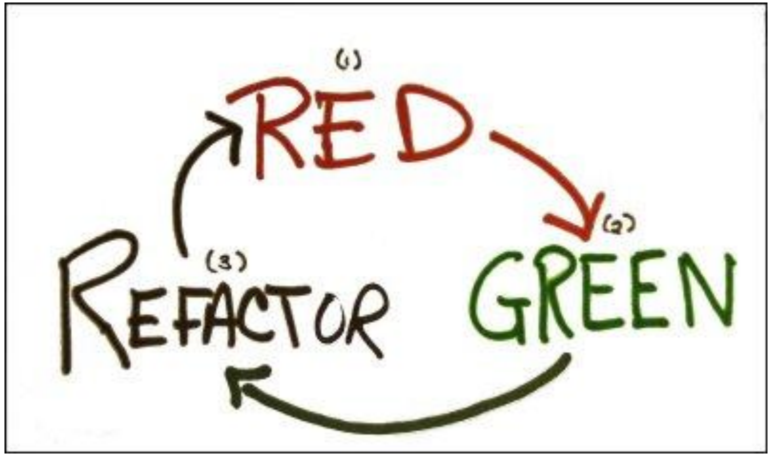
- Constructor that takes as input a single parameter including the input and expected result
- @RunWith(Parameterized.class) header
- @Parameterized.Parameters header denoting the static parameter factory class (that returns a ‘<u>Collection</u>’)
- For every parameter, the ‘runner’ in IntelliJ calls the constructor, runs the @Before and then runs the @Tests

Keypoints in JUnit

- Independency of Tests
  - JUnit provides no ordering guarantee of any kind for the execution of unit tests
  - Tests need to be independent from one another
  - Example: tests on onepress and twopress need to be two independent and complete tests
- @Before runs before every @Test
  - Other useful annotations includes @BeforeClass, @After, @AfterClass
- 2 ways to test exceptions
  - Test annotation parametrs
  - Try and catch + fail()
- Parameterized Tests
  - Annotations
  - Constructor

An aside: Test-Driven Development (TDD)

- Writing tests before you write the implementation.
- Only write new code when a test fails!



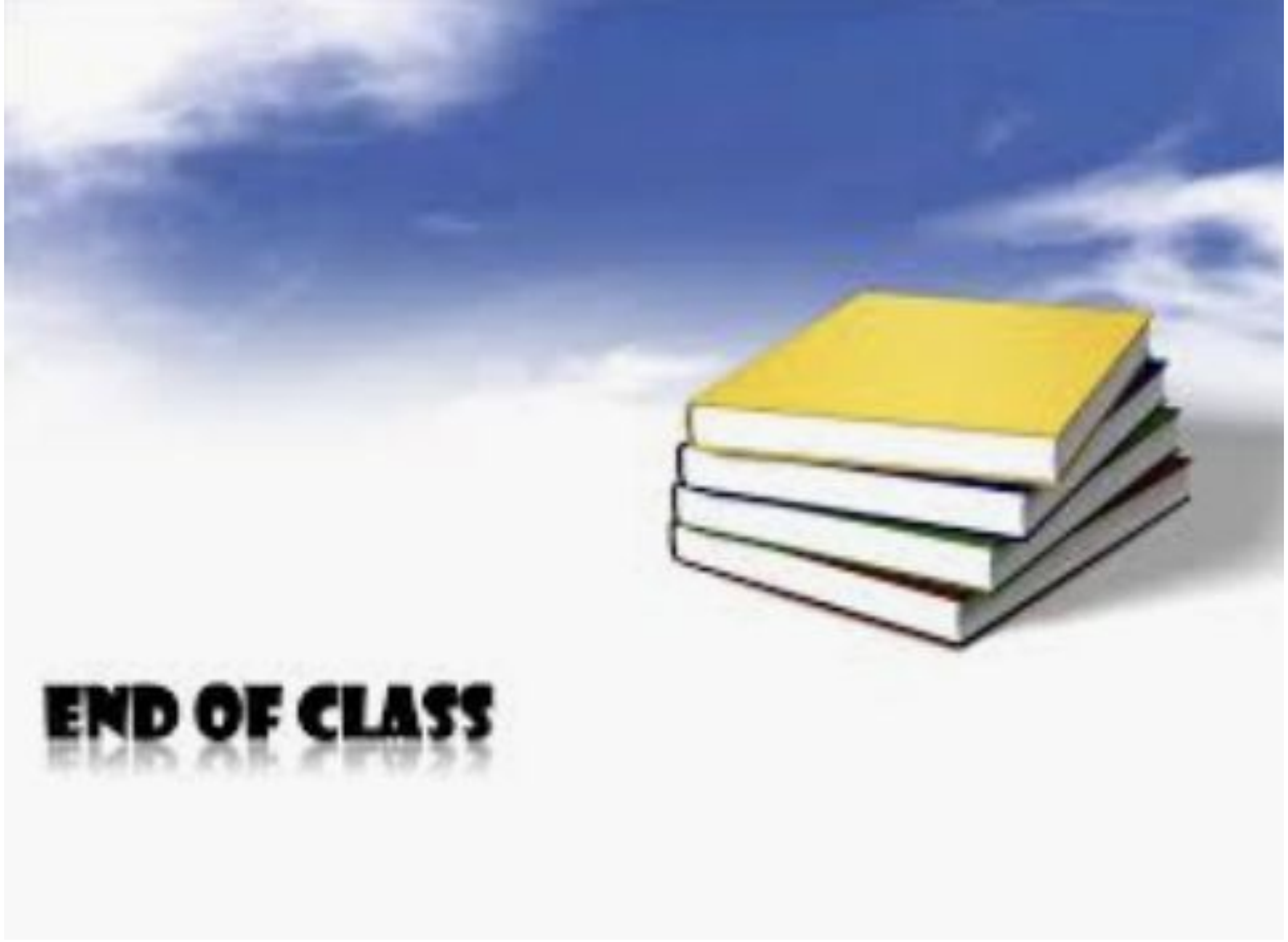
https://blog.cleancoder.com/uncle-bob/2014/12/17/TheCyclesOfTDD.html

An aside: Test-Driven Development (TDD)

Writing a failing unit test: RED

Write the minimum code that makes the test pass: GREEN

Rewrite the code so it follows standards, doesn’t have code smells, etc: REFACTOR


END OF CLASS

Reflection ✨

Recall: @RunWith(…)

Reflection lets you examine/modify runtime behavior of your code.

The following class exists: java.lang.Class

Reflection ✨

Very powerful concept.

Through reflection we can invoke methods at runtime *irrespective of the access specifier* used with them (We will see an example next class).

Tutorial:

https://docs.oracle.com/javase/tutorial/reflect/

Recap and Next Class:

Automated Testing ✓
Testing in JUnit ✓
A little on Annotations and Reflection ✓

Recap and Next Class:

- Be able to explain the foundational concepts of testing using the proper terminology;
- Understand type annotations and program reflection and be able to use them effectively;
- Be able to write unit tests with JUnit;
- Be able to approach more advanced testing problems requiring reflection or mock objects;
- Be able to understand the output of a test coverage tool such as EclEmma;
- Be able to understand basic test suite adequacy criteria and the relations between them;

Recap of Last Class

- Learned the foundational concepts of testing using the proper terminology;
- Understood type annotations and program reflection and be able to use them effectively;
- Looked at unit tests with JUnit;
- Be able to approach more advanced testing problems requiring reflection or mock objects;
- Be able to understand the output of a test coverage tool such as EclEmma;
- Be able to understand basic test suite adequacy criteria and the relations between them;

Overview of This Class

- Be able to explain the foundational concepts of testing using the proper terminology;
- Understand type annotations and program reflection and be able to use them effectively;
- Be able to write unit tests with JUnit;
- Be able to approach more advanced testing problems requiring reflection or mock objects;
- Be able to understand the output of a test coverage tool such as EclEmma;
- Be able to understand basic test suite adequacy criteria and the relations between them;

Encapsulation and Unit Testing

- How can we test private methods? (Two Views)
  - Private methods are not units
  - Private access modifier is a tool to help us structure the project code
- Metaprogramming (Reflection)
  - Should not be used indiscriminately

Reflection ✨

- Reflection (metaprogramming) lets you examine/modify runtime behavior of your code.
- The following class exists: **java.lang.Class**
- Using this you can now access an instance of a *class,* not an *object* of the class.

Reflection ✨

- Very powerful concept
- Through reflection we can invoke methods at runtime *irrespective of the access specifier* used with them
- Tutorial: https://docs.oracle.com/javase/tutorial/reflect/

Uses of Reflection

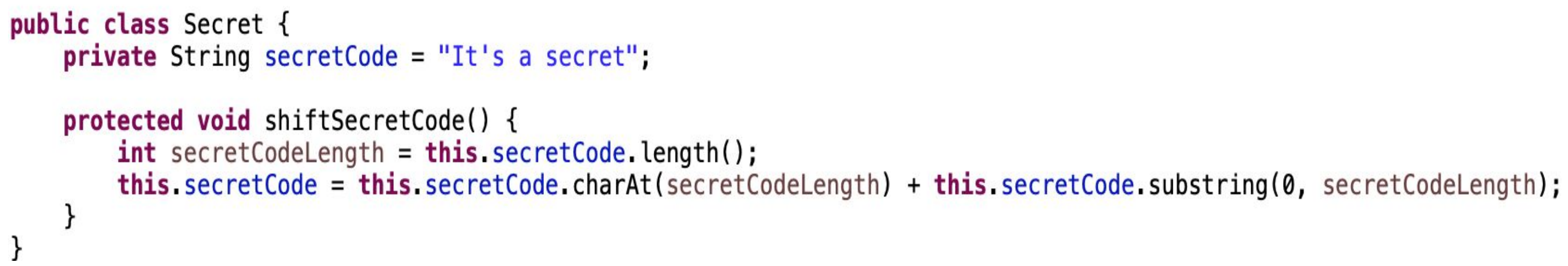
- Extensibility Features
  - Use external, user-defined classes by creating instances of extensibility objects using their fully-qualified names
- Class Browsers and Visual Development Environments
  - Use available information for visual development environments
- Debuggers and Test Tools
  - Examine private members of classes with debuggers
  - Make use in test harness tools

Drawbacks of Reflection

- Performance Overhead
  - Involves types that are dynamically resolved
- Security Restrictions
  - Requires a runtime permission which may not be present when running under a security manager
- Exposure of Internals
  - Might result in unexpected side-effects
  - Breaks abstractions

Example

- Reflection provides an API for accessing fields, methods, and constructors.
- There are distinct methods for accessing members declared directly on the class versus methods which search the super interfaces and super classes for inherited members.
- Let's define a secret class with a private code and a protected



```java
public class Secret {
    private String secretCode = "It's a secret";

    protected void shiftSecretCode() {
        int secretCodeLength = this.secretCode.length();
        this.secretCode = this.secretCode.charAt(secretCodeLength) + this.secretCode.substring(0, secretCodeLength);
    }
}
```

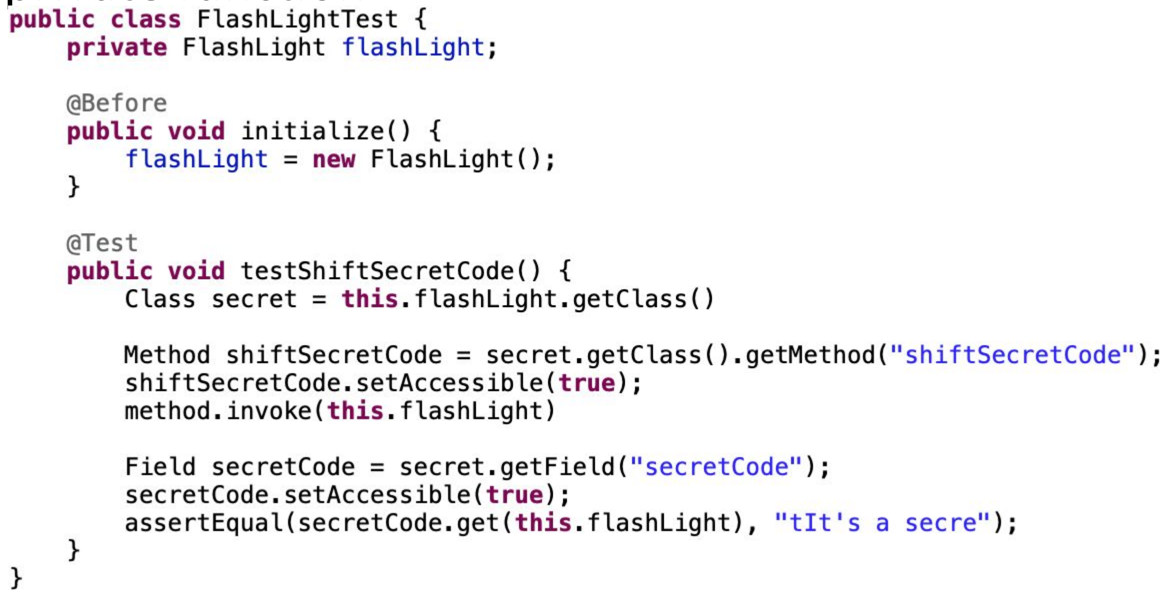
Example (Contd.)

- Now let's inherit from this class and update the code each time we get the brightness level

```java
public class FlashLight extends Secret {
    private int brightnessLevel = 0;

    public void setBrightnessLevel(int pBrightnessLevel) {
        if(pBrightnessLevel > 5 || pBrightnessLevel < 0 ) {
            throw new IllegalArgumentException("BrightnessLevel should in the range of [0-5].");
        }
        this.brightnessLevel = pBrightnessLevel;
    }

    public int getBrightnessLevel() {
        this.shiftSecretCode();
        return this.brightnessLevel;
    }
}
```

Example (Contd.)

- Now we should change the test case and add one test case for the private function



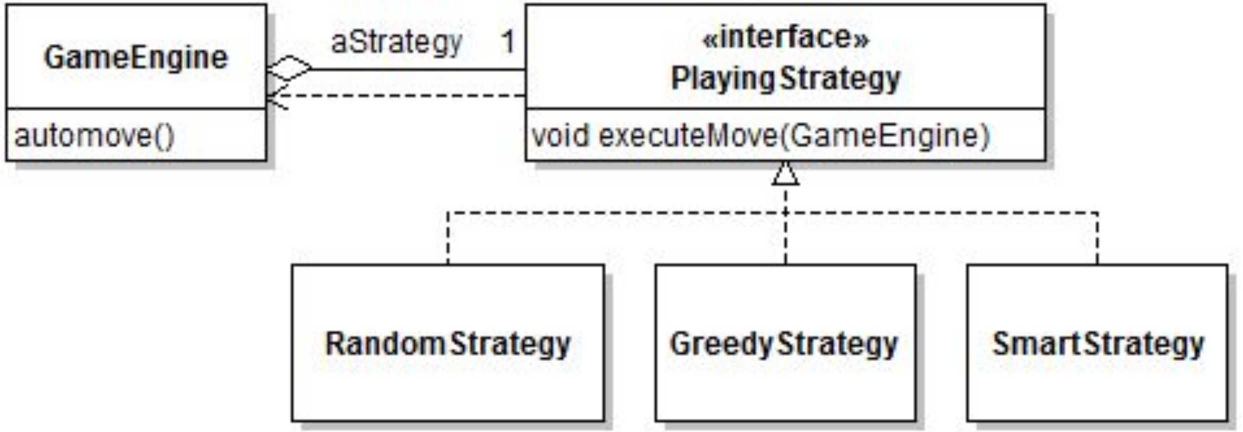
```
public class FlashLightTest {
    private FlashLight flashLight;

    @Before
    public void initialize() {
        flashLight = new FlashLight();
    }

    @Test
    public void testShiftSecretCode() {
        Class secret = this.flashLight.getClass()

        Method shiftSecretCode = secret.getClass().getMethod("shiftSecretCode");
        shiftSecretCode.setAccessible(true);
        method.invoke(this.flashLight)

        Field secretCode = secret.getField("secretCode");
        secretCode.setAccessible(true);
        assertEqual(secretCode.get(this.flashLight), "tIt's a secre");
    }
}
```

Other Use Cases of Reflection

- Obtaining fields and methods types and modifiers
- Obtaining names of method parameters
- Invoking methods
- Finding constructors
- Retrieving and Parsing Constructor Modifiers
- Creating New Class Instances

Testing with Stubs/Mock Object

- The key to unit testing is to test small parts of the program ***in isolation***
- Consider the following example



Testing with Stubs/Mock Object (Contd.)

- Problems with the current approach:
  - Calling the ***executeMove*** method on any strategy will trigger the execution of presumably complex behavior by the strategy
  - Determine the strategy that would be used by the game engine
  - Not different from testing the strategies individually
- An object stub is a greatly simplified version of an object that mimics its behavior sufficiently to support the testing of a UUT that uses this object

Example 1

- Let’s write a test for the Game Engine example in the previous slide.

```java
public class TestGameEngine
{
   @Test
   public void testAutoPlay() throws Exception
   {
      class StubStrategy implements PlayingStrategy
      {
         private boolean aExecuted = false;

         public boolean hasExecuted() { return aExecuted; }

         @Override
         public void executeMove(GameEngine pGameEngine)
         {
            aExecuted = true;
         }
      }
```

Example 1 (Contd.)

- We can then use an instance of this stub instead of a "real" strategy in the rest of the test

```
@Test
public void testAutoPlay() throws Exception
{
    ...
   Field strategyField = GameEngine.class.getDeclaredField("aStrategy");
   strategyField.setAccessible(true);
   StubStrategy strategy = new StubStrategy();
   GameEngine engine = GameEngine.instance();
   strategyField.set(engine, strategy);
```

Example 2

- The use of mock objects in unit testing can get extremely sophisticated, and frameworks exist to support this task (e.g., jMock)
- ***Demo***

Wrote Code ✔️

Wrote our tests ✔️

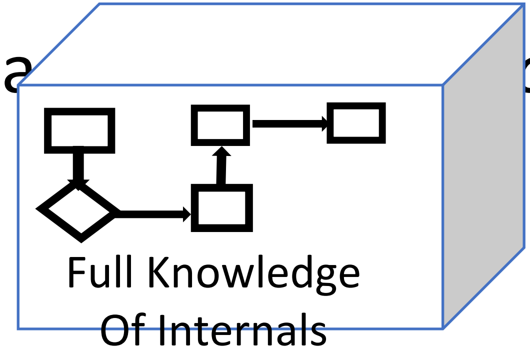
First of all, what are we testing for..?

Recall:

Testing is a process of automatically running code to verify *expectations* about it.

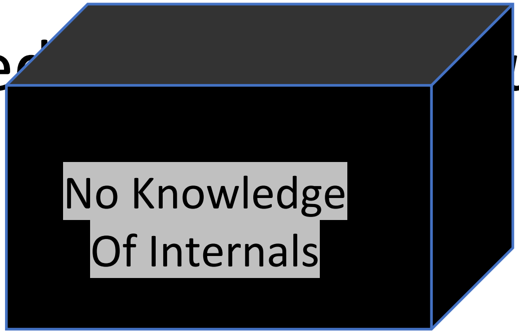
Structural Testing

- White-box Testing
- Checks the *implemented behaviour*
- Ba *orks*



Functional Testing

- Black-box Testing
- Checks the *specified behaviour*
- Base *uld do*



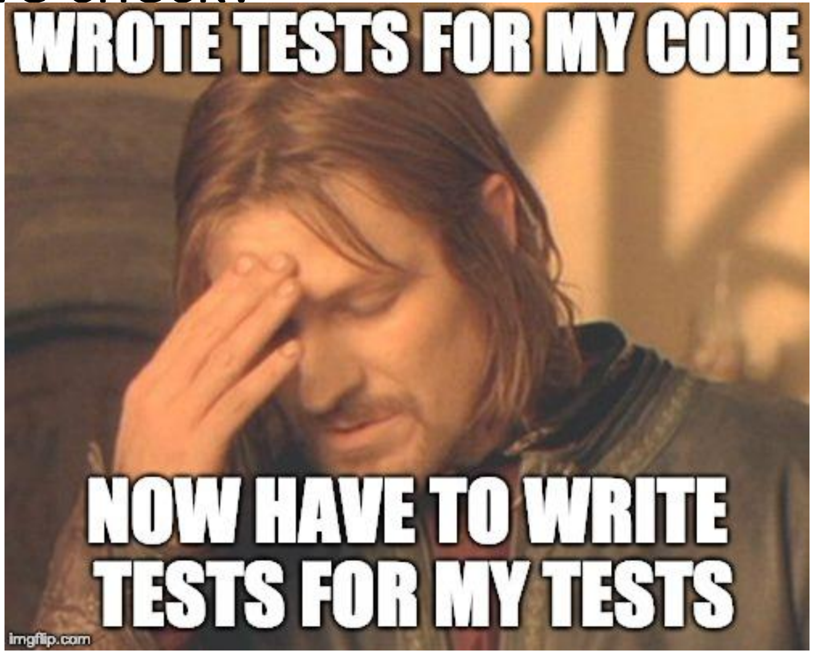
Test Suite Adequacy Criteria – the real MVP

- Measures the *thoroughness* of the Test Suite
- If a test suite covers ALL possible obligations of the software, it is adequate.

Test Adequacy

Are our tests sufficient?

How do we check?



100 % Test Suite Adequacy??

But sometimes there are scenarios that CAN’T be checked or it doesn't make sense to check.

Defensive programming checks edge cases that CAN NOT happen.

The toggle in yesterday's flashlight example – there's no end to testing the number of consecutive clicks!

Coverage

- Measures the *extent of* thoroughness of a test suite.
- What percentage of components does our test suite execute in the program?

The development and testing team get to decide when a test suite is “sufficient” to test their software.

"If our test suite covers X scenarios in our program, we’re good to go!"

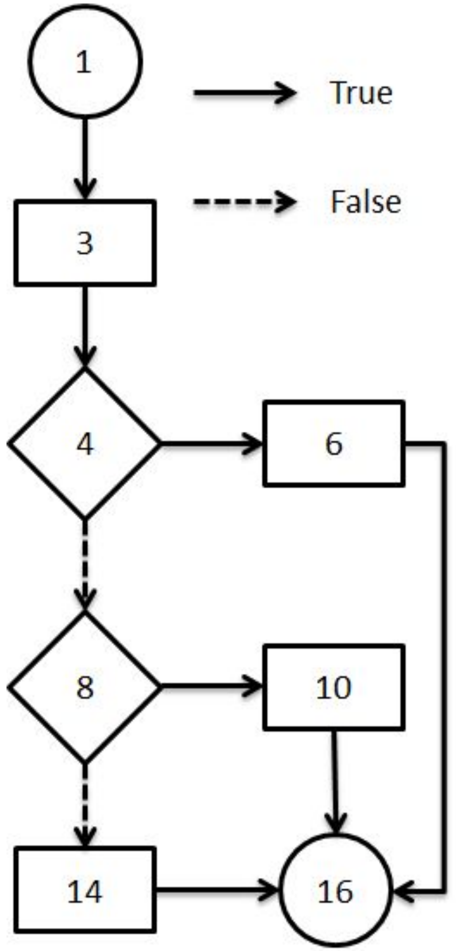
"If our test suite satisfies Y% of the rules or obligations, it’s

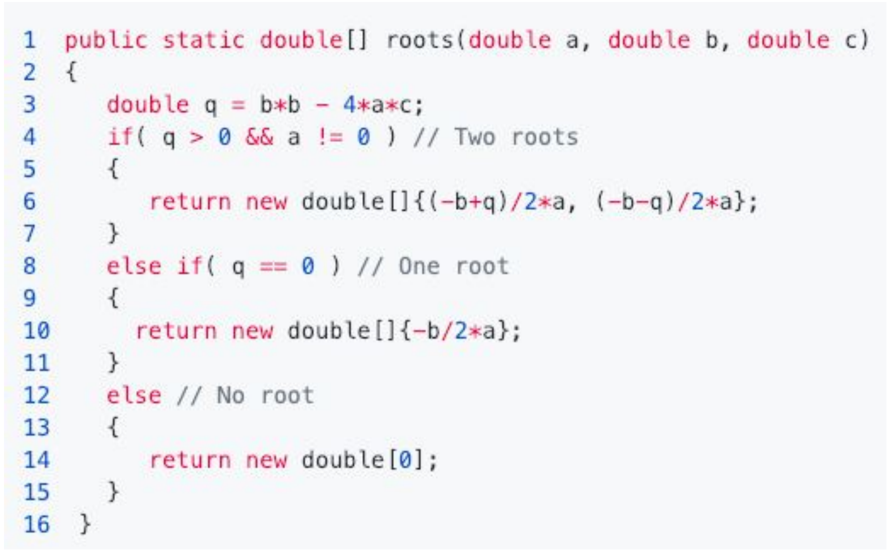
Why is percentage (sometimes) okay?

- Say you’re looking to buy a laptop.
- A used laptop is for sale.
- Though you *expect* all parts to be functioning, it turns out the external memory card reader doesn’t work.
- But you’re getting it at a great price!
- What do you do?

Measuring Coverage

The Control Flow Graph (CFG)

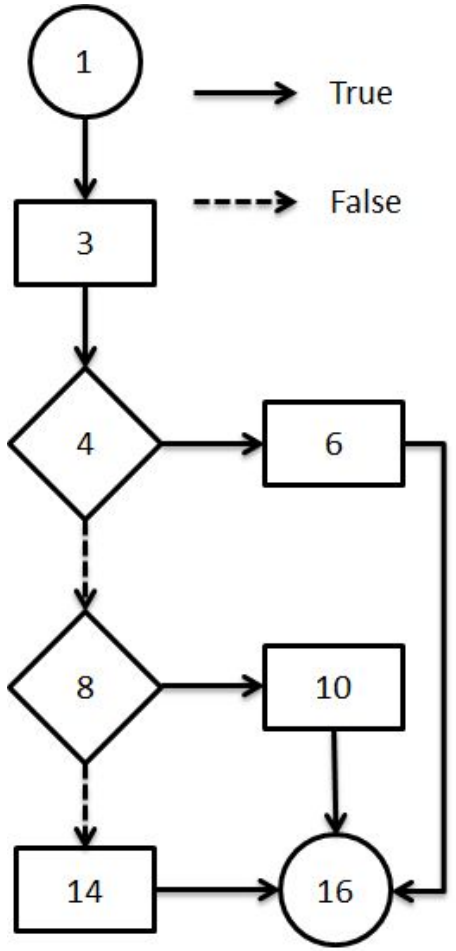


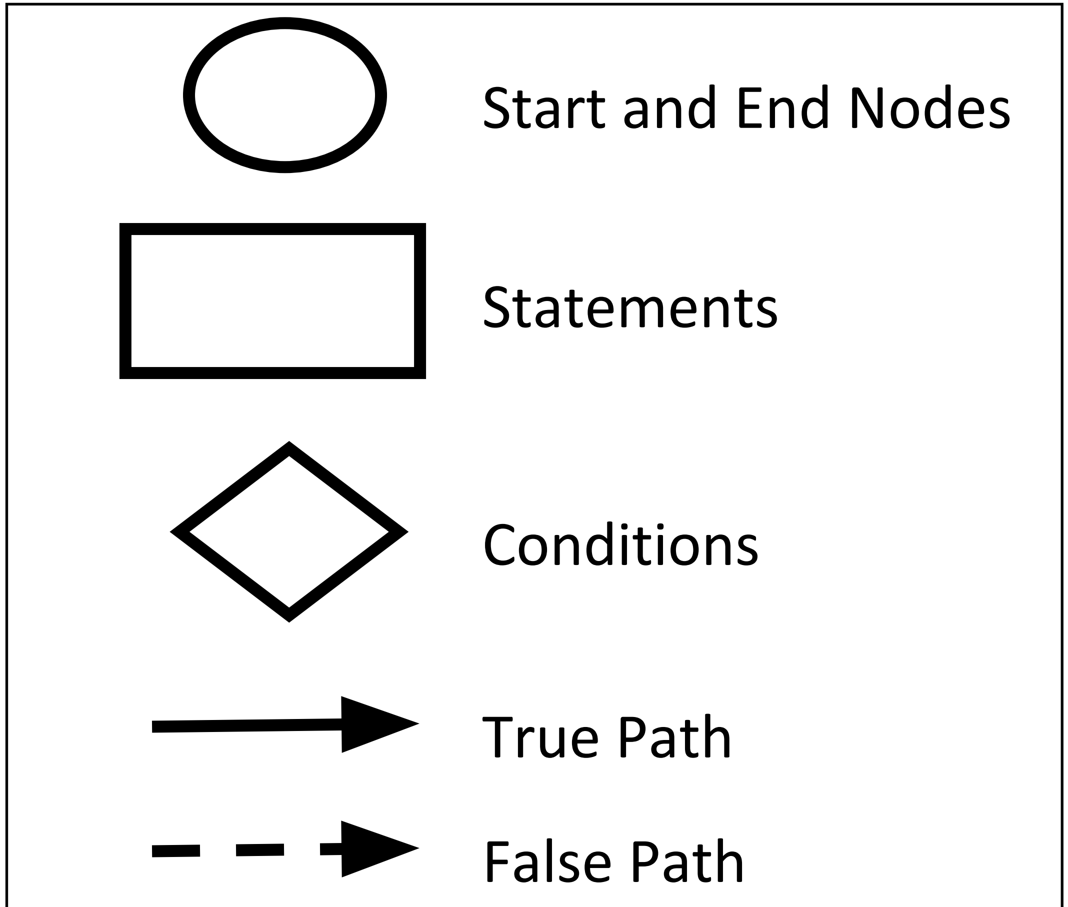


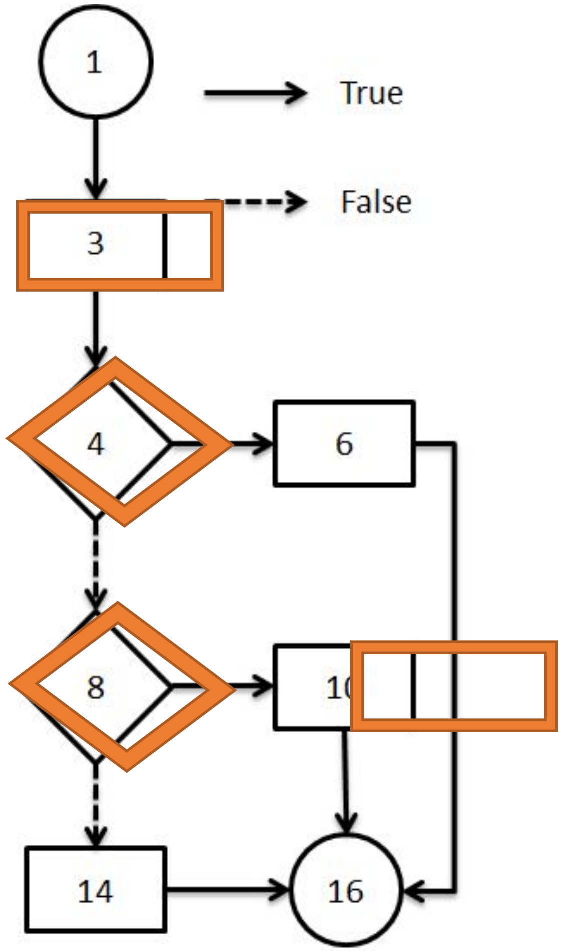
```
1   public static double[] roots(double a, double b, double c)
2   {
3       double q = b*b - 4*a*c;
4       if( q > 0 && a != 0 ) // Two roots
5       {
6           return new double[]{(-b+q)/2*a, (-b-q)/2*a};
7       }
8       else if( q == 0 ) // One root
9       {
10        return new double[]{-b/2*a};
11      }
12      else // No root
13      {
14          return new double[0];
15      }
16  }
```

https://github.com/prmr/SoftwareDesign/blob/master/modules/Module-04.md

Control Flow Graph (CFG)







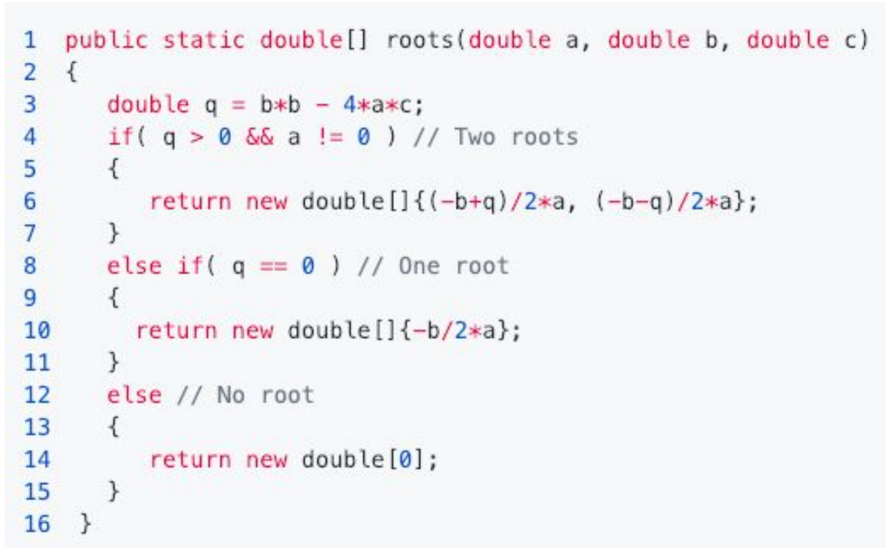
Statement Coverage

- Input: (1,2,1)

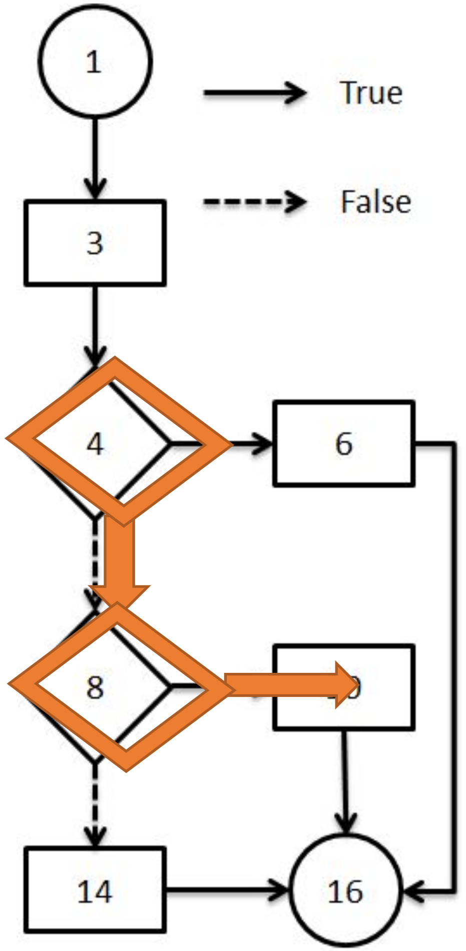
= (Number of statements executed / total number of statements)

= 4/6 = 67%

(start and end nodes are ignored)



```
1  public static double[] roots(double a, double b, double c)
2  {
3      double q = b*b - 4*a*c;
4      if( q > 0 && a != 0 ) // Two roots
5      {
6         return new double[]{(-b+q)/2*a, (-b-q)/2*a};
7      }
8      else if( q == 0 ) // One root
9      {
10       return new double[]{-b/2*a};
11     }
12     else // No root
13     {
14        return new double[0];
15     }
16  }
```



Branch Coverage

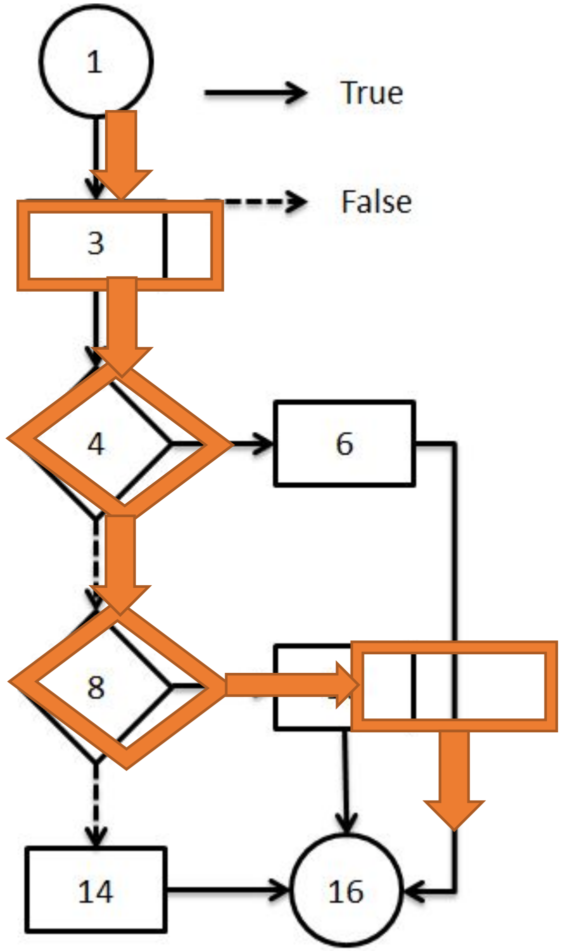
- Input: (1,2,1)

= (Number of branches executed / total number of branches)

= 2/4 = 50%

(start and end nodes are ignored)

```
1  public static double[] roots(double a, double b, double c)
2  {
3      double q = b*b - 4*a*c;
4      if( q > 0 && a != 0 ) // Two roots
5      {
6          return new double[]{(-b+q)/2*a, (-b-q)/2*a};
7      }
8      else if( q == 0 ) // One root
9      {
10        return new double[]{-b/2*a};
11     }
12     else // No root
13     {
14         return new double[0];
15     }
16  }
```



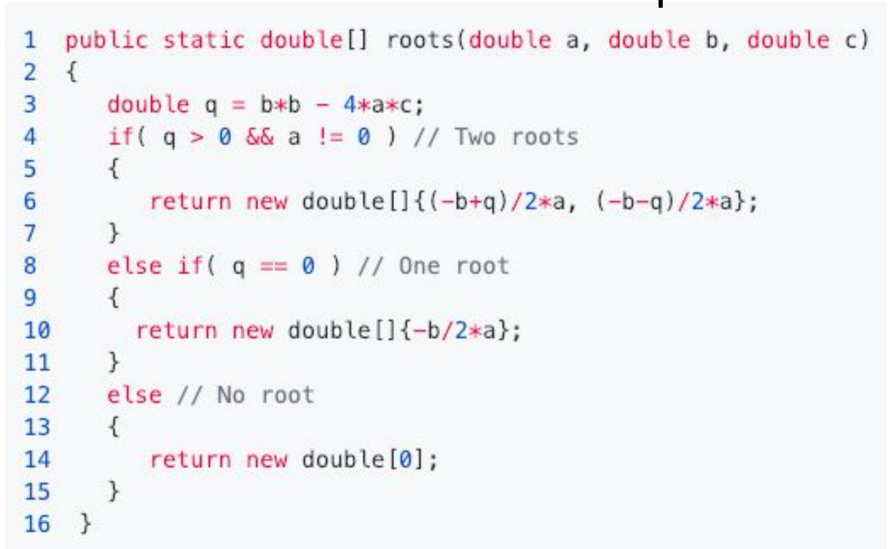
Path Coverage

- Input: (1,2,1)

= (Number of paths executed / total number of paths)

= 1/3 = 33%

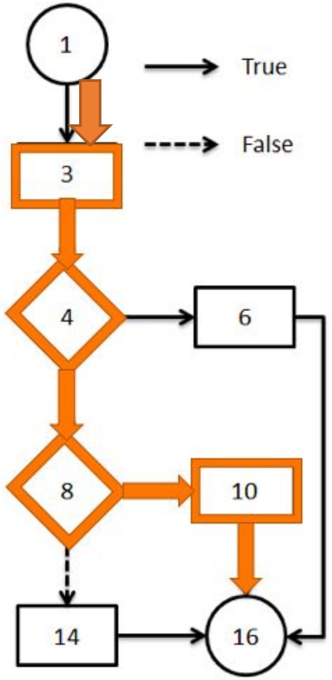
“Theoretical” because how would you calculate it if there were loops?!



```
1   public static double[] roots(double a, double b, double c)
2   {
3       double q = b*b - 4*a*c;
4       if( q > 0 && a != 0 ) // Two roots
5       {
6           return new double[]{(-b+q)/2*a, (-b-q)/2*a};
7       }
8       else if( q == 0 ) // One root
9       {
10         return new double[]{-b/2*a};
11      }
12      else // No root
13      {
14          return new double[0];
15      }
16   }
```

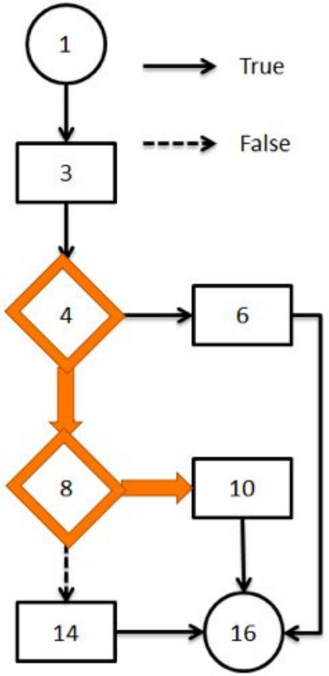
The Subsumes Relationship

Path Coverage



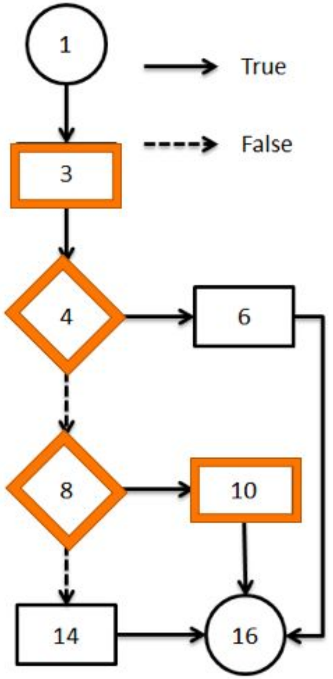
Subsumes

Branch Coverage



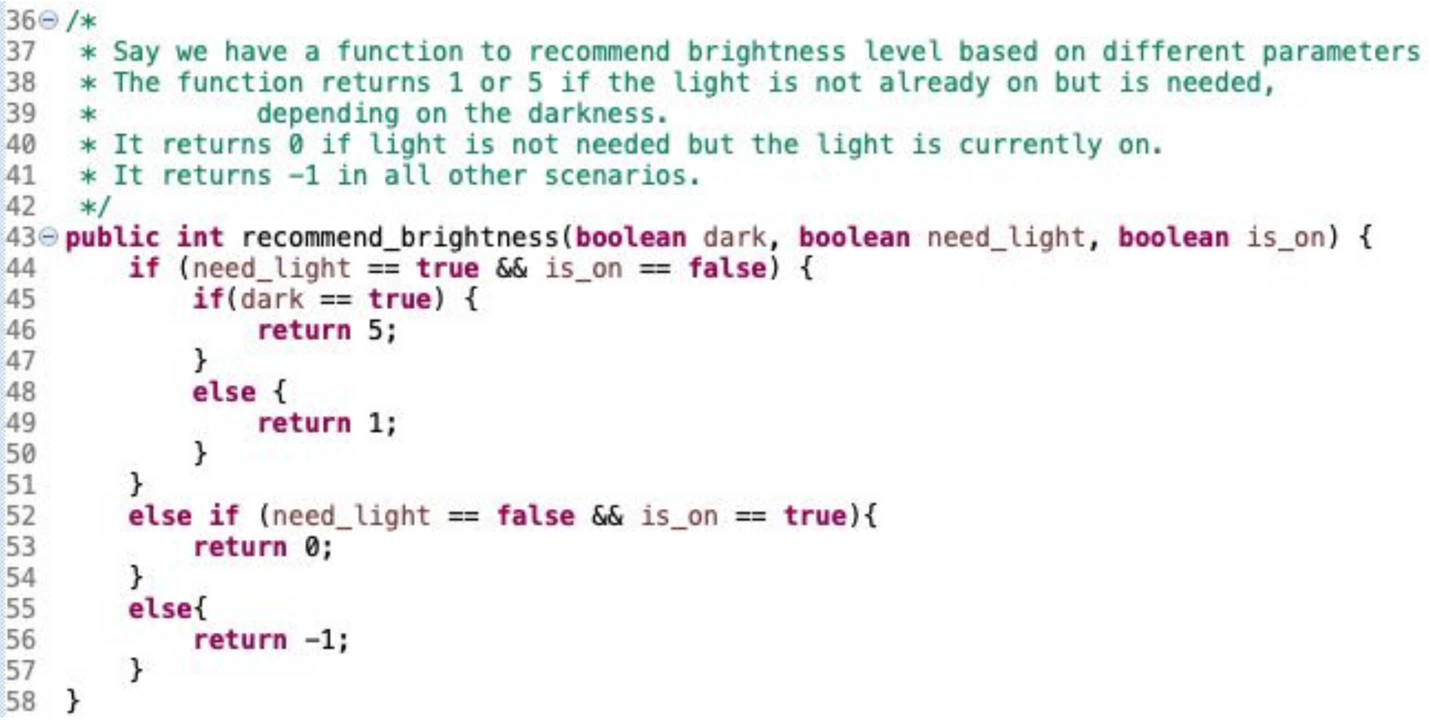
Subsumes

Statement Coverage



Activity:

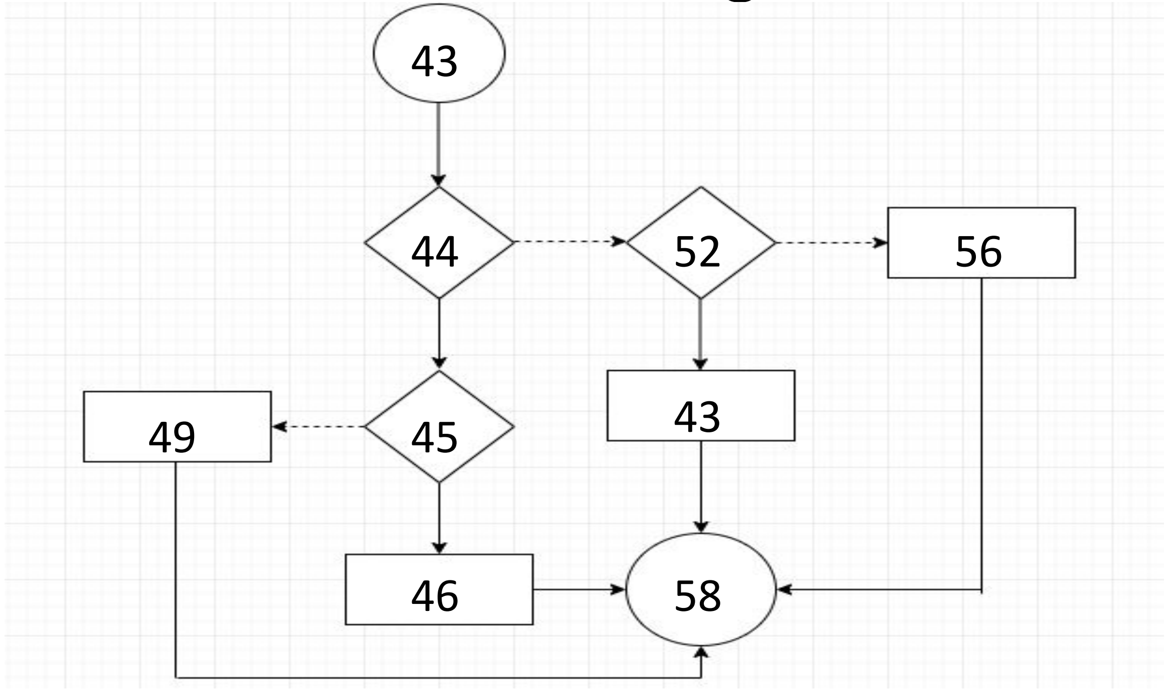
1. Draw the CFG
2. What are the minimum set of test cases that will execute all statements? List them down.



```java
/*
 * Say we have a function to recommend brightness level based on different parameters
 * The function returns 1 or 5 if the light is not already on but is needed,
 *          depending on the darkness.
 * It returns 0 if light is not needed but the light is currently on.
 * It returns -1 in all other scenarios.
 */
public int recommend_brightness(boolean dark, boolean need_light, boolean is_on) {
    if (need_light == true && is_on == false) {
        if(dark == true) {
            return 5;
        }
        else {
            return 1;
        }
    }
    else if (need_light == false && is_on == true){
        return 0;
    }
    else{
        return -1;
    }
}
```

Activity:

Does your CFG look something like this?



Activity:

What are the minimum set of test cases that will execute all statements?

- (True, True, False) - Statements 44, 45, 46
- (False, True, False) - Statements 44, 45, 49
- (True, False, True) - Statements 44, 52, 53
- (True, True True) - Statements 44, 52, 56

This test suite with 4 test cases has 100% Statement

Basic Conditions Coverage

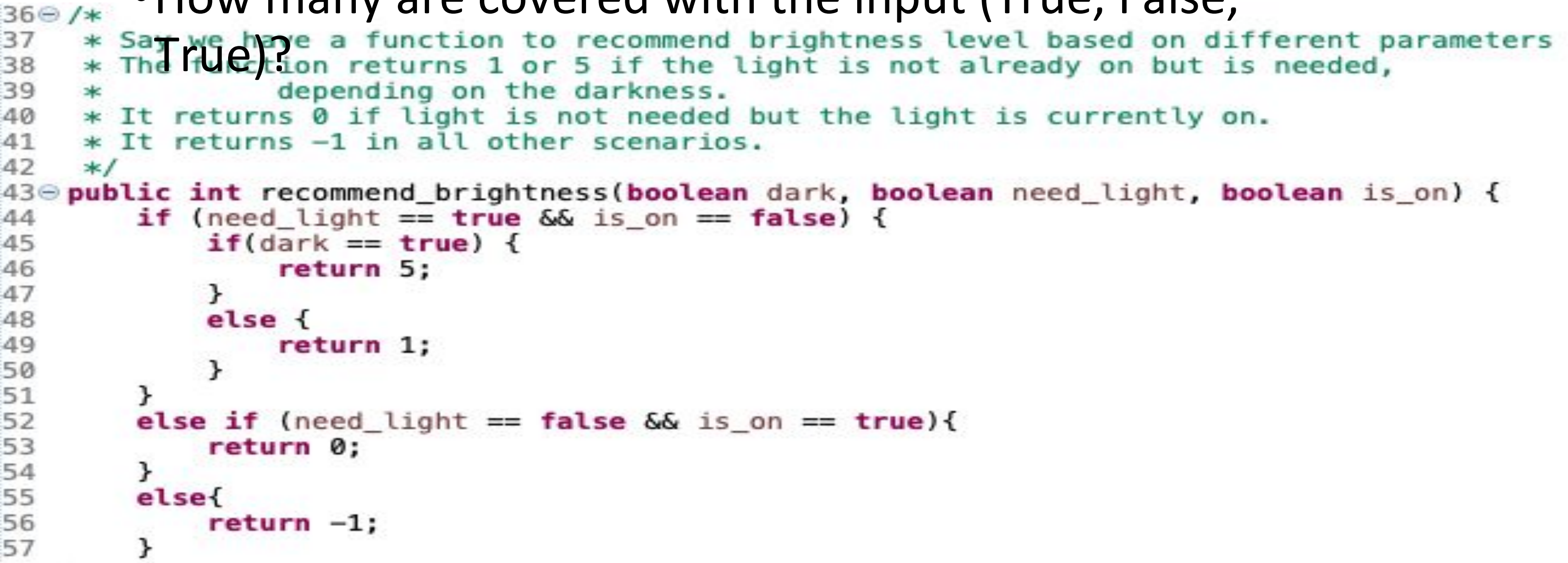
- Every condition can have either a True or False outcome
- Total number of *distinct* condition-outcome pairs = number_of_conditions *2

```
if (need_light == true && is_on == false)
```

What is the percentage of condition-outcome pairs executed?

Basic Conditions Coverage

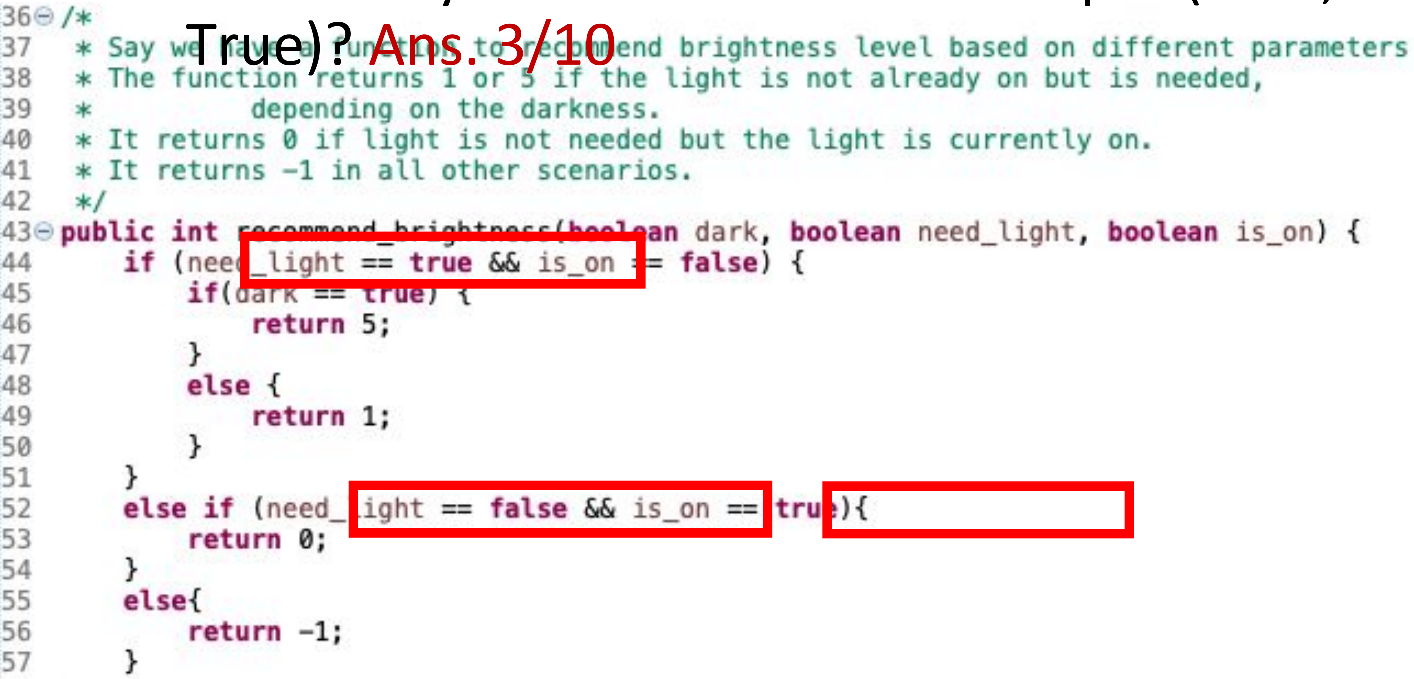
- In this case how many condition-outcome pairs exist?
- How many are covered with the input (True, False, True)?



```
/*
 * Say we have a function to recommend brightness level based on different parameters
 * The function returns 1 or 5 if the light is not already on but is needed,
 *          depending on the darkness.
 * It returns 0 if light is not needed but the light is currently on.
 * It returns -1 in all other scenarios.
 */
public int recommend_brightness(boolean dark, boolean need_light, boolean is_on) {
    if (need_light == true && is_on == false) {
        if(dark == true) {
            return 5;
        }
        else {
            return 1;
        }
    }
    else if (need_light == false && is_on == true){
        return 0;
    }
    else{
        return -1;
    }
```

Basic Conditions Coverage

- In this case how many condition-outcome pairs exist?
  Ans. 10
- How many are covered with the input (True, False, True)? Ans. 3/10



```
/*
 * Say we have a function to recommend brightness level based on different parameters
 * The function returns 1 or 5 if the light is not already on but is needed,
 *           depending on the darkness.
 * It returns 0 if light is not needed but the light is currently on.
 * It returns -1 in all other scenarios.
 */
public int recommend_brightness(boolean dark, boolean need_light, boolean is_on) {
    if (need_light == true && is_on == false) {
        if(dark == true) {
            return 5;
        }
        else {
            return 1;
        }
    }
    else if (need_light == false && is_on == true){
        return 0;
    }
    else{
        return -1;
    }
```

Additional Conditions Coverage

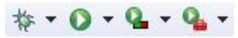
- Branch and Conditions Coverage
    - Satisfying both branch as well as conditions coverage
    - Subsumes Basic Conditions Coverage

- Compound Coverage
    - Satisfies all possible combinations of conditions
    - Usually occurs when multiple conditions exist in a single statement (using && and ||)
    - Subsumes Branch and Conditions Coverage

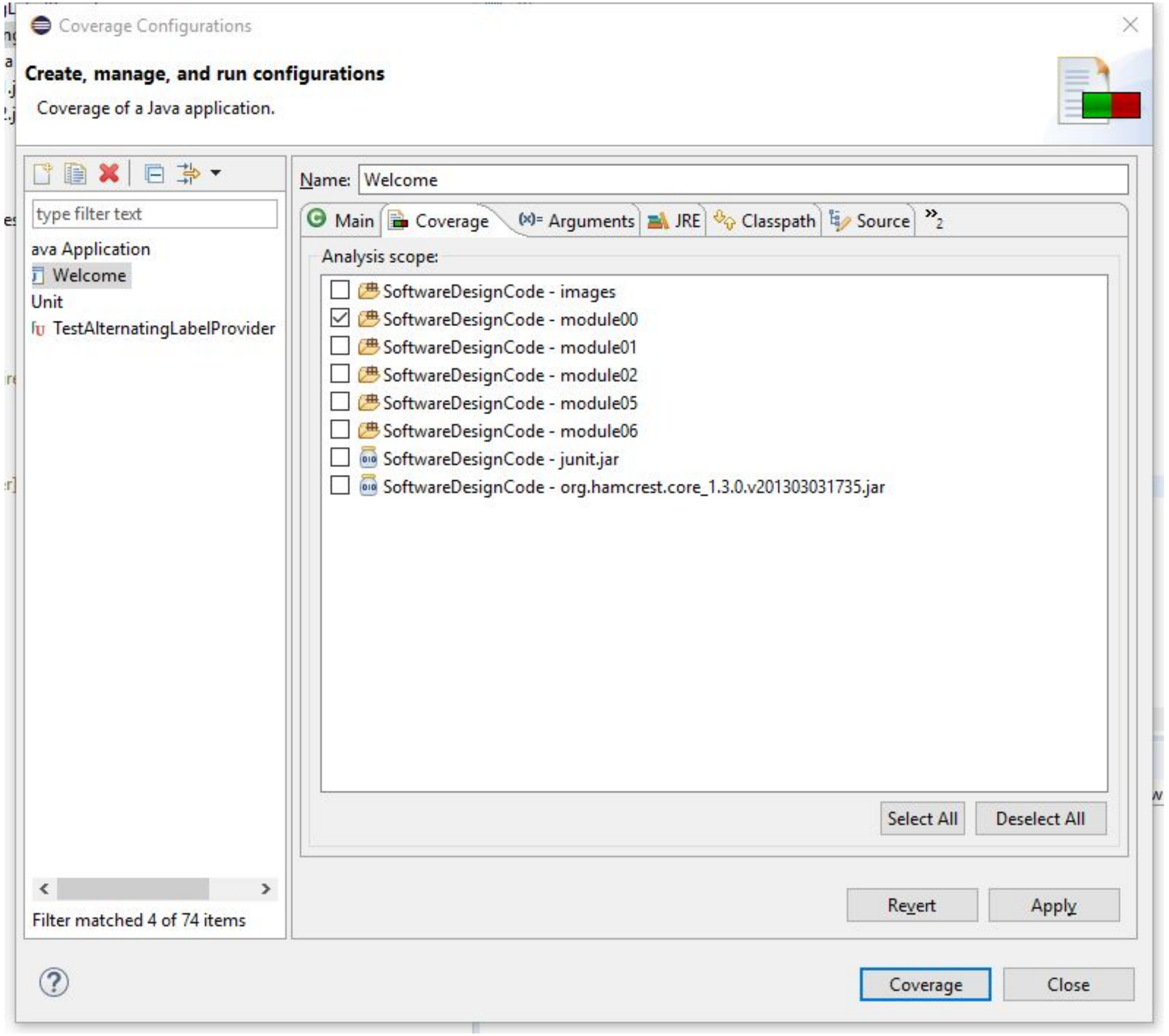
Coverage Tool: EclEmma

- We had brief experience with coverage in exercise M0

- Installation: Search for "EclEmma" in *Help ⟶ Eclipse Marketplace*



- How to use EclEmma?
  - Right click or
  - Modify Coverage Configuration
- How to interpret the result?



Coverage Configurations

Create, manage, and run configurations

Coverage of a Java application.

type filter text

ava Application

Welcome

Unit

TestAlternatingLabelProvider

Filter matched 4 of 74 items

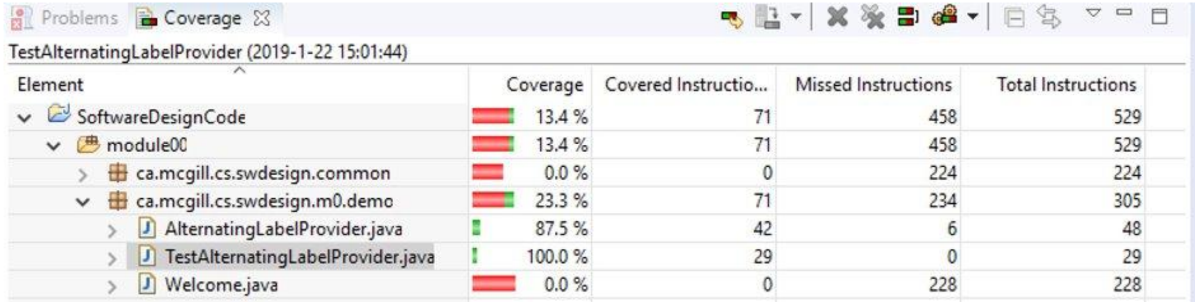
Name: Welcome

Analysis scope:

- [ ] SoftwareDesignCode - images
- [✓] SoftwareDesignCode - module00
- [ ] SoftwareDesignCode - module01
- [ ] SoftwareDesignCode - module02
- [ ] SoftwareDesignCode - module05
- [ ] SoftwareDesignCode - module06
- [ ] SoftwareDesignCode - junit.jar
- [ ] SoftwareDesignCode - org.hamcrest.core_1.3.0.v201303031735.jar

Coverage View

- The Coverage View summarizes the ratio of items coverage to total items

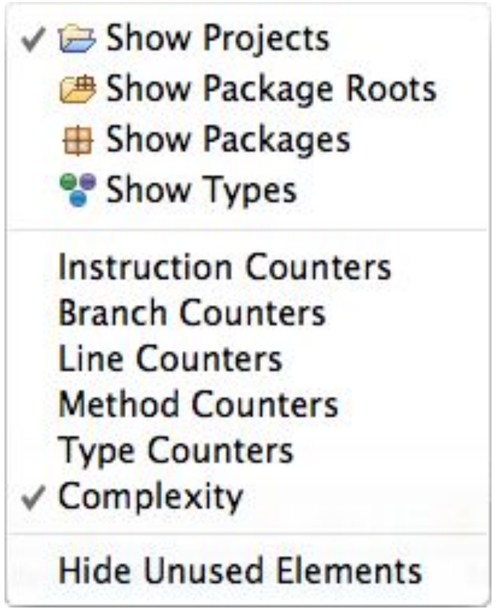


TestAlternatingLabelProvider (2019-1-22 15:01:44)

Element	Coverage	Covered Instructio...	Missed Instructions	Total Instructions
SoftwareDesignCode	13.4 %	71	458	529
module00	13.4 %	71	458	529
ca.mcgill.cs.swdesign.common	0.0 %	0	224	224
ca.mcgill.cs.swdesign.m0.demo	23.3 %	71	234	305
AlternatingLabelProvider.java	87.5 %	42	6	48
TestAlternatingLabelProvider.java	100.0 %	29	0	29
Welcome.java	0.0 %	0	228	228

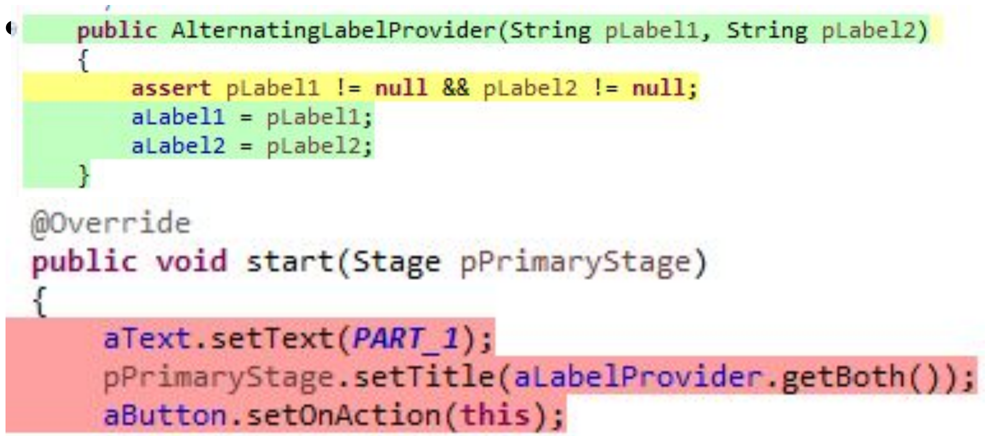
Counter Mode

- Select Different Counter Mode From drop-down menu



Source Code Annotation

- Green: Fully Covered
- Yellow: Partially Covered



```
    public AlternatingLabelProvider(String pLabel1, String pLabel2)
    {
        assert pLabel1 != null && pLabel2 != null;
        aLabel1 = pLabel1;
        aLabel2 = pLabel2;
    }
@Override
public void start(Stage pPrimaryStage)
{
    aText.setText(PART_1);
    pPrimaryStage.setTitle(aLabelProvider.getBoth());
    aButton.setOnAction(this);
```

Counter Mode

- *Instruction coverage* provides information about the amount of code that has been executed or missed.
- *branch coverage* provides information about the amount of branches executed for if/switch statement
- *Line coverage* provides information about the amount of lines that has been executed or missed.A source line is considered executed when at least one instruction that is assigned to this line has been executed.

Counter Mode

- *Method coverage* provides information about the amount of methods that has been executed or missed. A method is considered as executed when at least one instruction has been executed.

- *Complexity* is the minimum number of paths that can, in (linear) combination, generate all possible paths through a method. Thus the complexity value can serve as an indication for the number of unit test cases to fully cover a certain piece of software.