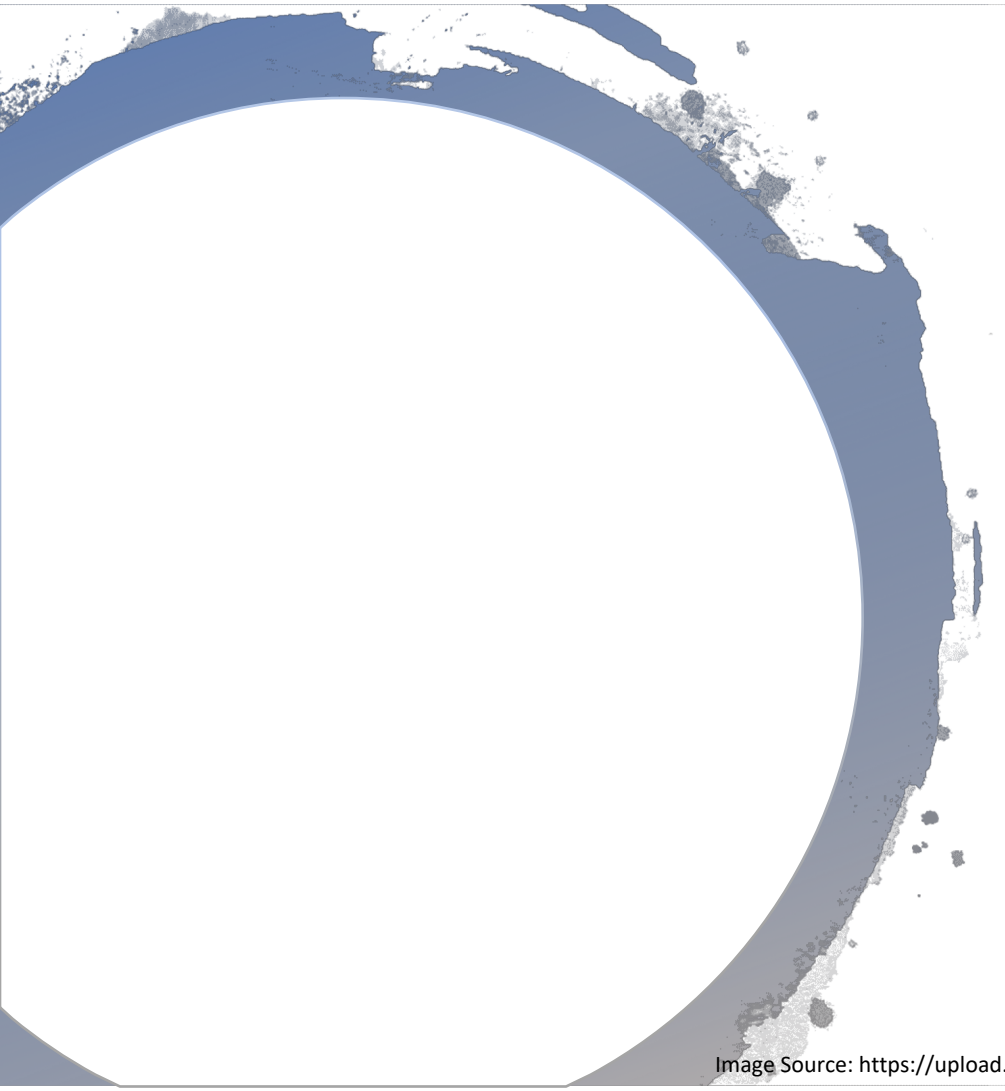
REMINDER: Lab Test Session 2

- Week of Feb 11th - 22nd
- Will cover M2-b,c, and M3-a,b
- Please do arrive on time. Lab test slots are filling up and if there are no more empty slots left, it will not be possible to reschedule if you miss your test.
- Please do come prepared with the lecture materials and exercises. Though we will do our best to guide you, this is a test not an exercise.



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

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)

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 method

```
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

```
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

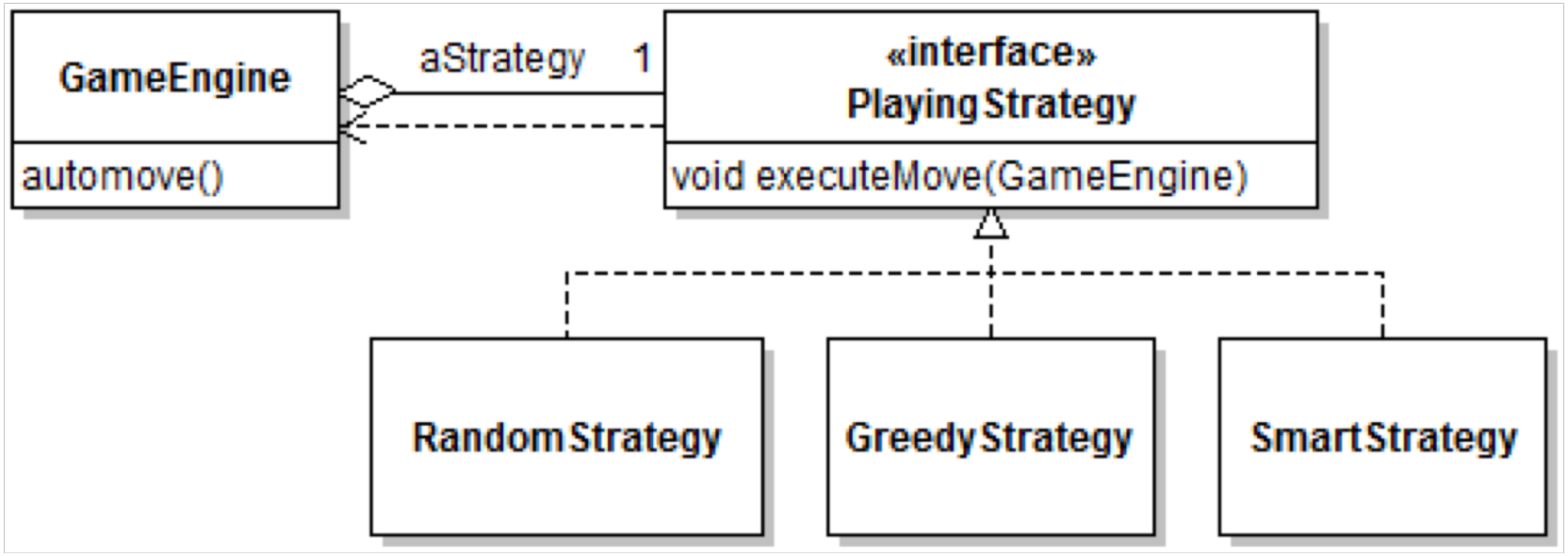
```
public class FlashLightTest {
    private FlashLight flashLight;

    @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.

```
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***

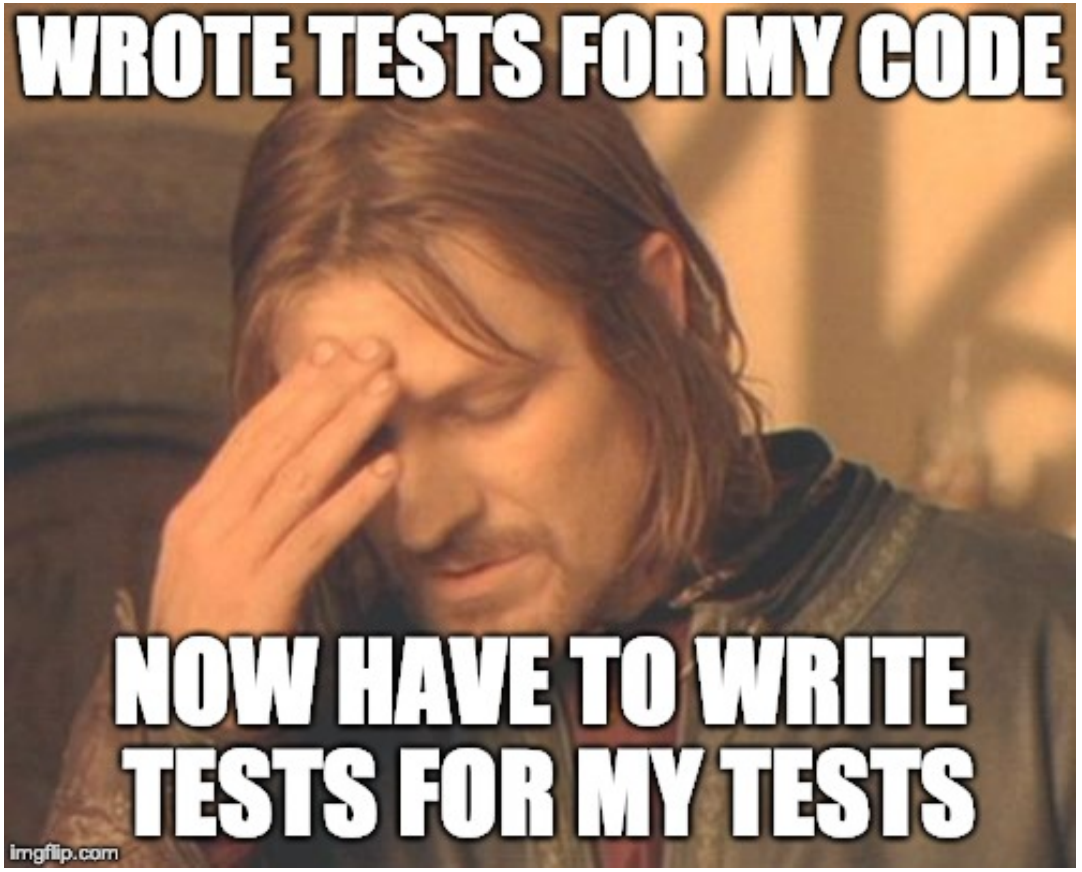
Wrote Code

Wrote our tests

Test Adequacy

Are our tests sufficient?

How do we check?



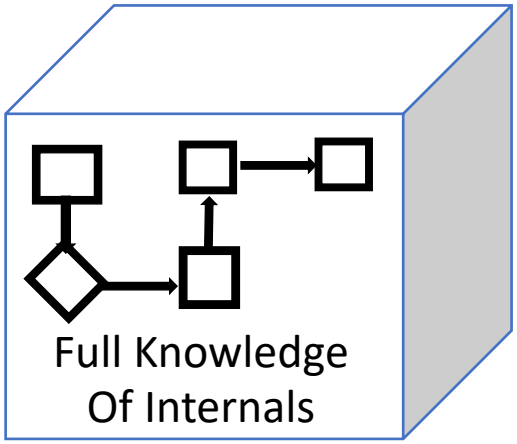
First of all, what are we testing for..?

Recall:

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

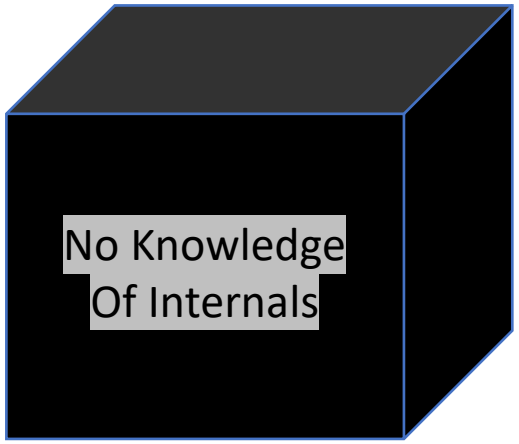
Structural Testing

- White-box Testing
- Checks the *implemented behaviour*
- Based on *how it works*



Functional Testing

- Black-box Testing
- Checks the *specified behaviour*
- Based on *what it should 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.

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!"

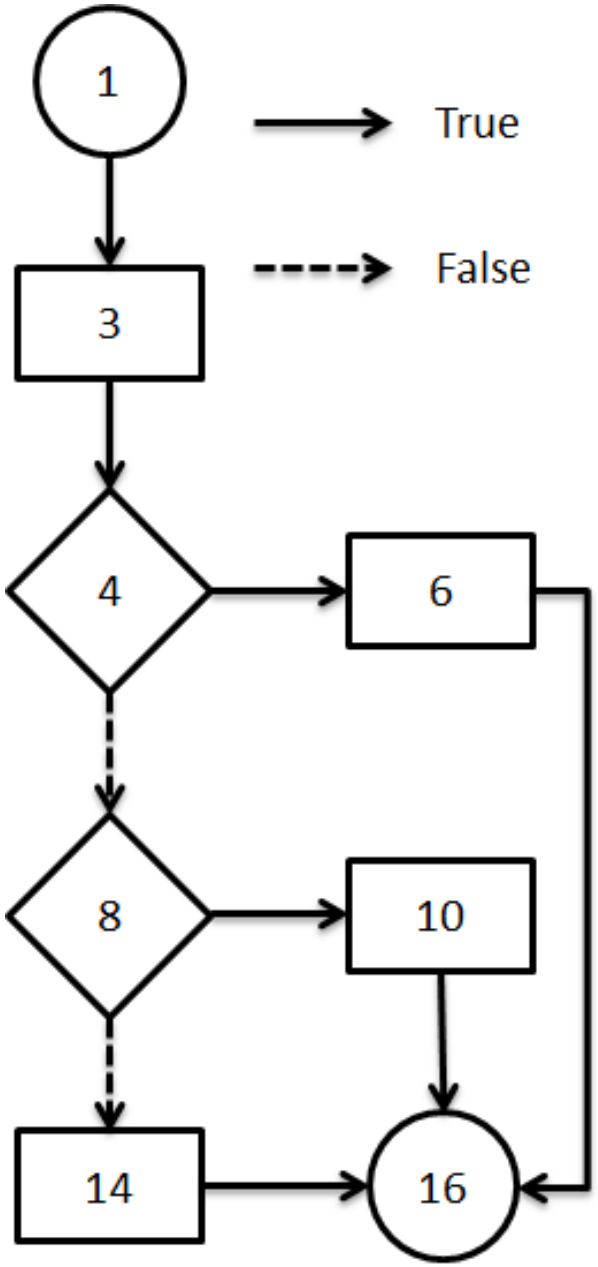
"If our test suite satisfies Y% of the rules or obligations, it’s doing well!”

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?
- Requirement vs Cost Trade-Off
  - As long as the *important* parts work, and in general, everything is fine, we'll take it!

Measuring Coverage

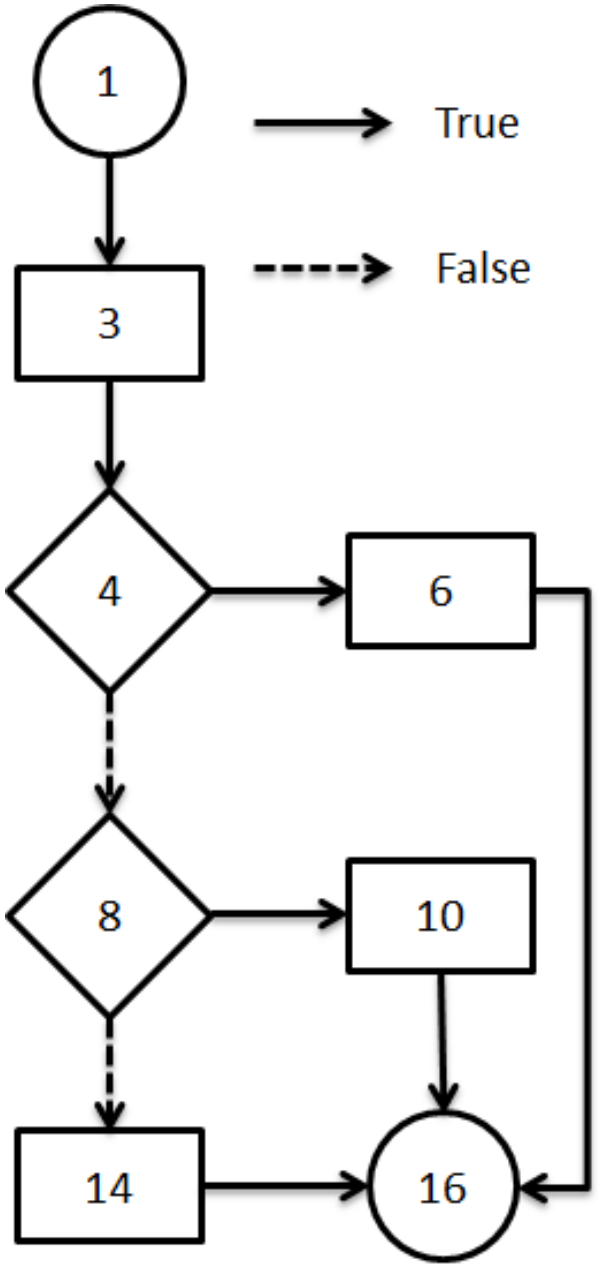
The Control Flow Graph (CFG)

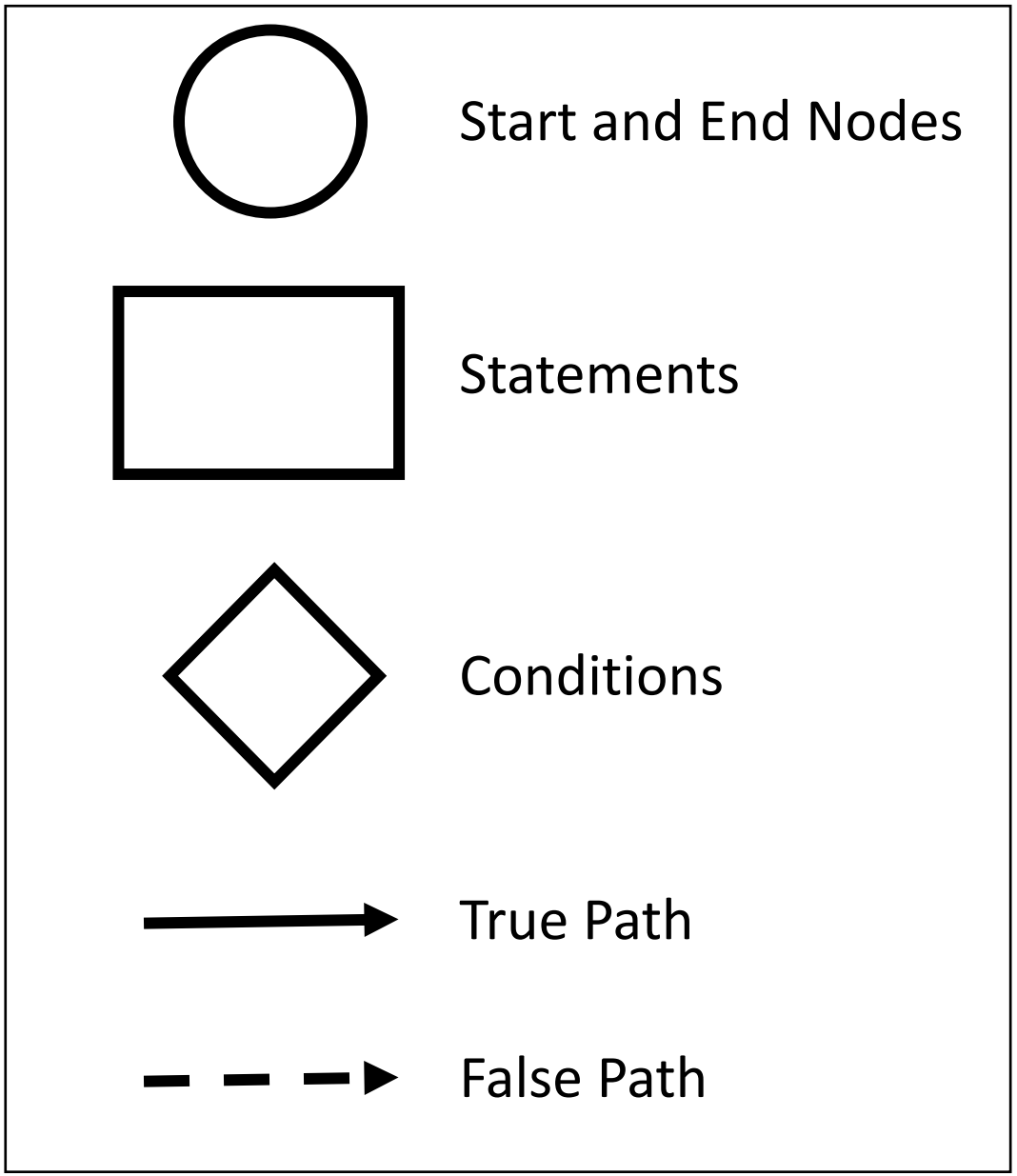


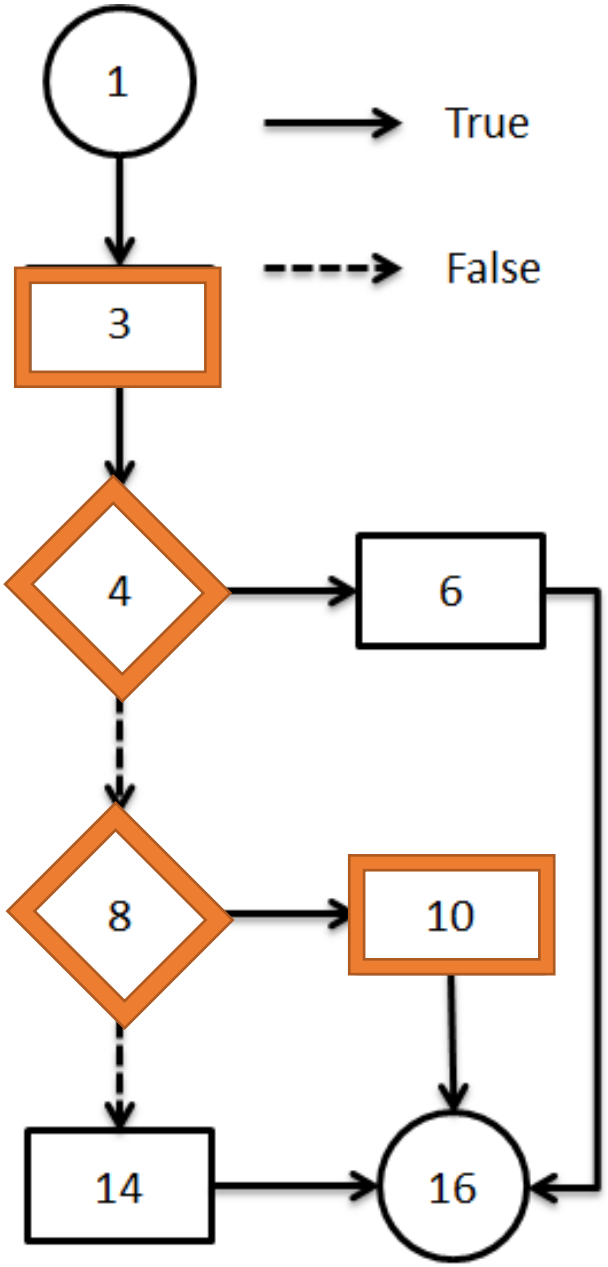

```
1  public static double[] roots(double a, double b, double c)
2  {
3     double q = b*b - 4*a*c;
4     if( q > 0 && a != 0 ) // Two roots
5     {
6        return new double[]{(-b+q)/2*a, (-b-q)/2*a};
7     }
8     else if( q == 0 ) // One root
9     {
10      return new double[]{-b/2*a};
11    }
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

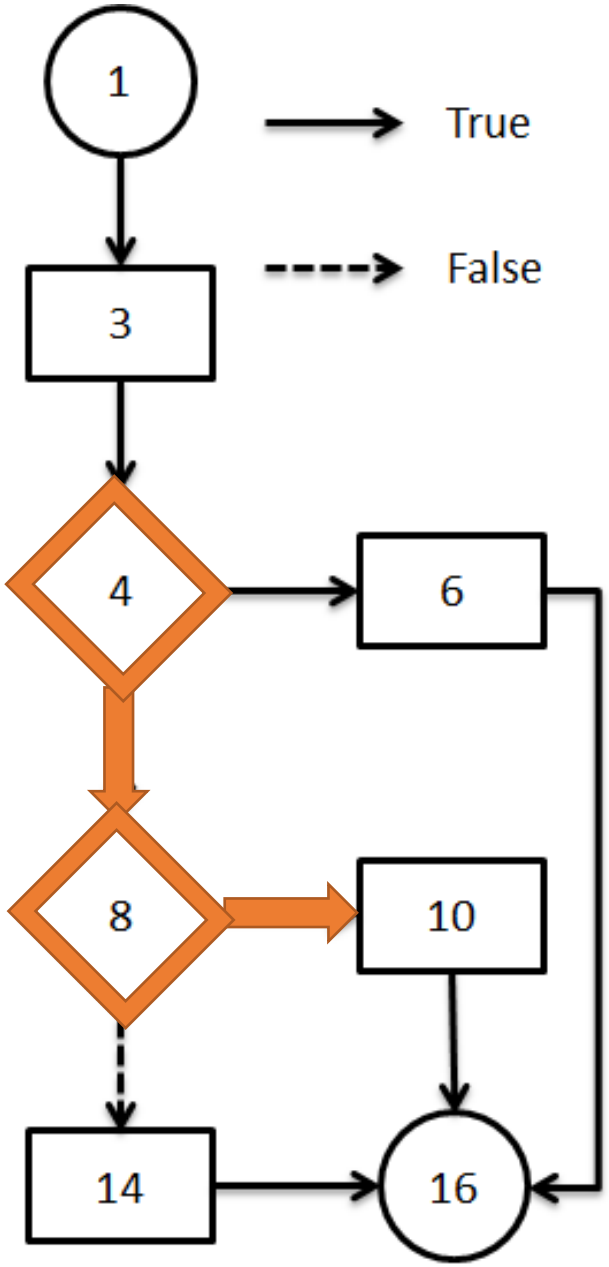
- Input: (1,2,1)

= (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

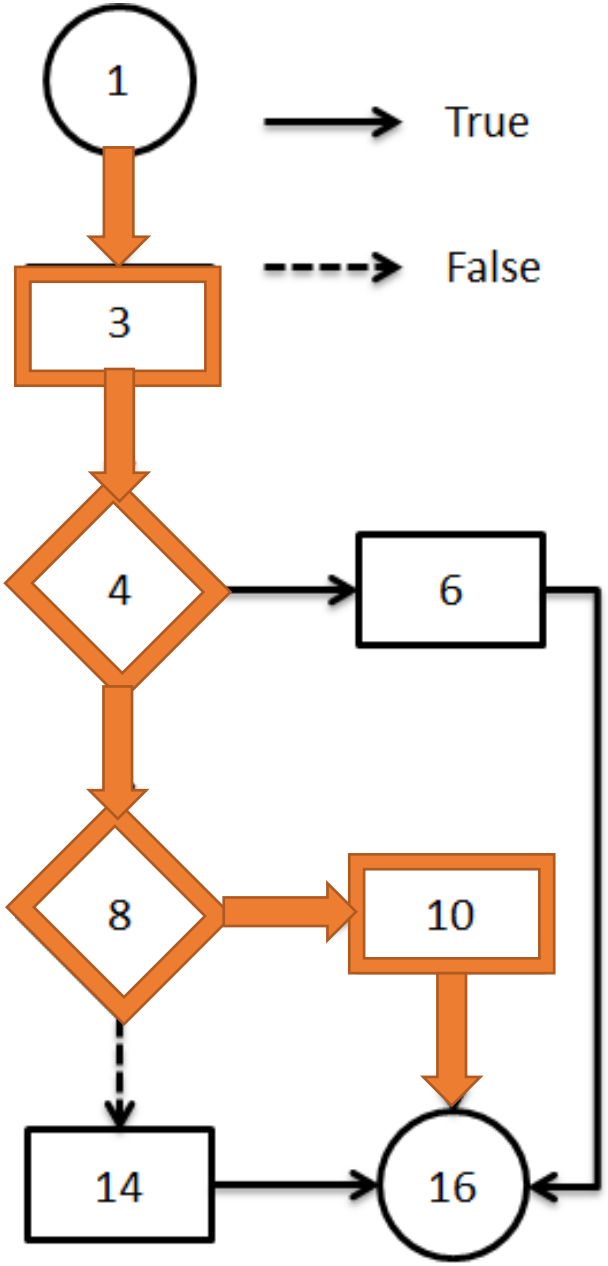
- Input: (1,2,1)

= (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)

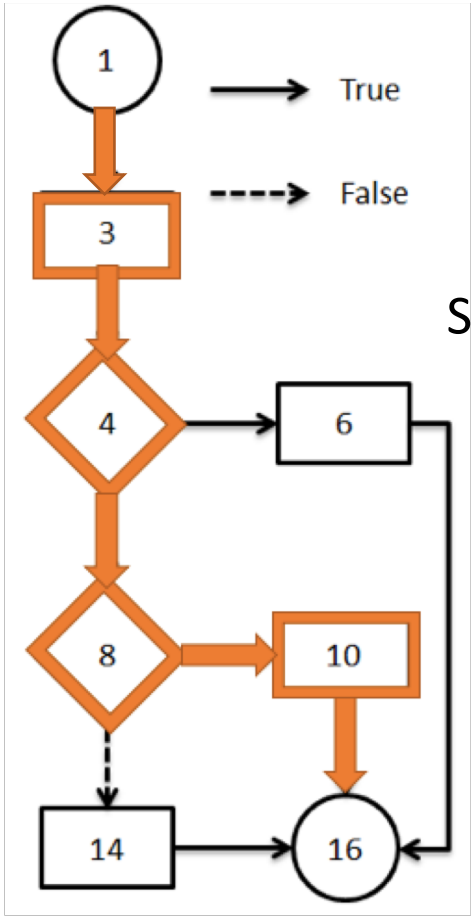
= 1/3 = 33%

"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 }
```

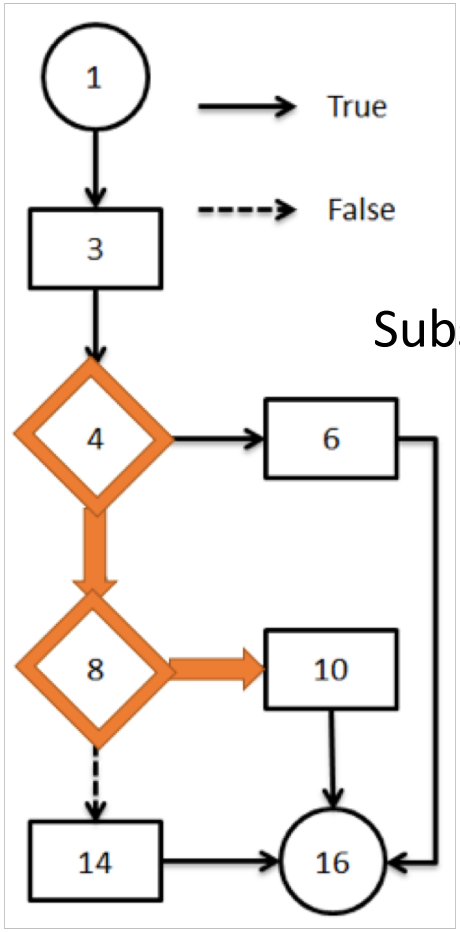
The Subsumes Relationship

Path Coverage



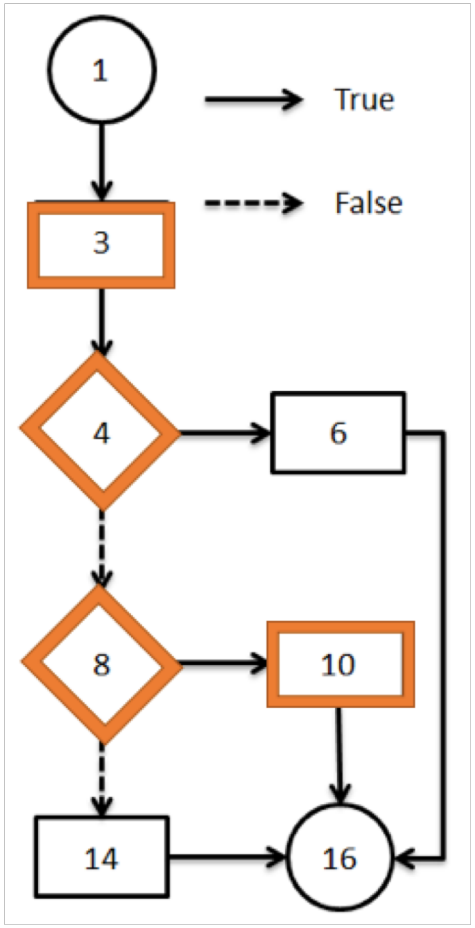
Subsumes

Branch Coverage



Subsumes

Statement Coverage



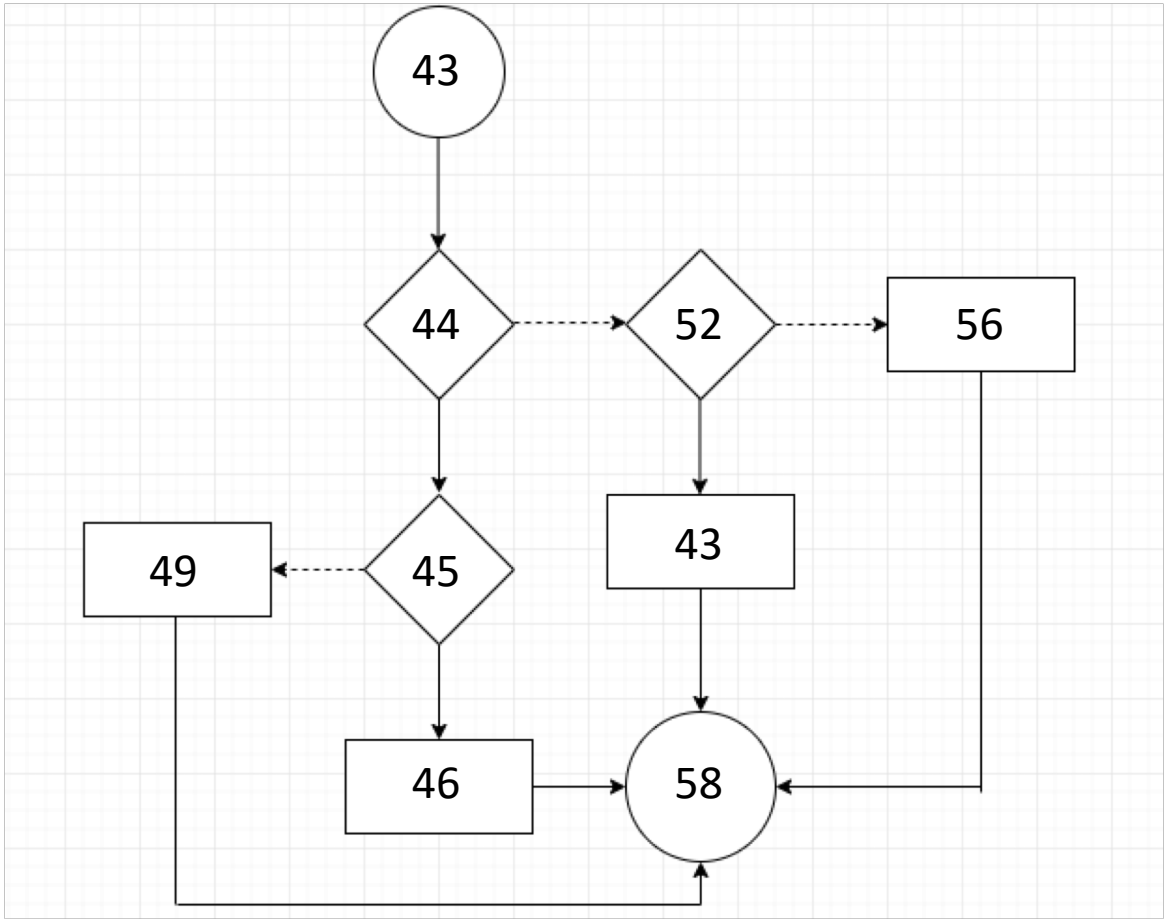
Activity:

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

```
36 /*
37  * Say we have a function to recommend brightness level based on different parameters
38  * The function returns 1 or 5 if the light is not already on but is needed,
39  *          depending on the darkness.
40  * It returns 0 if light is not needed but the light is currently on.
41  * It returns -1 in all other scenarios.
42  */
43 public int recommend_brightness(boolean dark, boolean need_light, boolean is_on) {
44     if (need_light == true && is_on == false) {
45         if(dark == true) {
46             return 5;
47         }
48         else {
49             return 1;
50         }
51     }
52     else if (need_light == false && is_on == true){
53         return 0;
54     }
55     else{
56         return -1;
57     }
58 }
```

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 Coverage!

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)?

```
36 /*
37  * Say we have a function to recommend brightness level based on different parameters
38  * The function returns 1 or 5 if the light is not already on but is needed,
39  *          depending on the darkness.
40  * It returns 0 if light is not needed but the light is currently on.
41  * It returns -1 in all other scenarios.
42  */
43 public int recommend_brightness(boolean dark, boolean need_light, boolean is_on) {
44     if (need_light == true && is_on == false) {
45         if(dark == true) {
46             return 5;
47         }
48         else {
49             return 1;
50         }
51     }
52     else if (need_light == false && is_on == true){
53         return 0;
54     }
55     else{
56         return -1;
57     }
```

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

```
36 /*
37  * Say we have a function to recommend brightness level based on different parameters
38  * The function returns 1 or 5 if the light is not already on but is needed,
39  *          depending on the darkness.
40  * It returns 0 if light is not needed but the light is currently on.
41  * It returns -1 in all other scenarios.
42  */
43 public int recommend_brightness(boolean dark, boolean need_light, boolean is_on) {
44     if (need_light == true && is_on == false) {
45         if(dark == true) {
46             return 5;
47         }
48         else {
49             return 1;
50         }
51     }
52     else if (need_light == false && is_on == true) {
53         return 0;
54     }
55     else{
56         return -1;
57     }
```

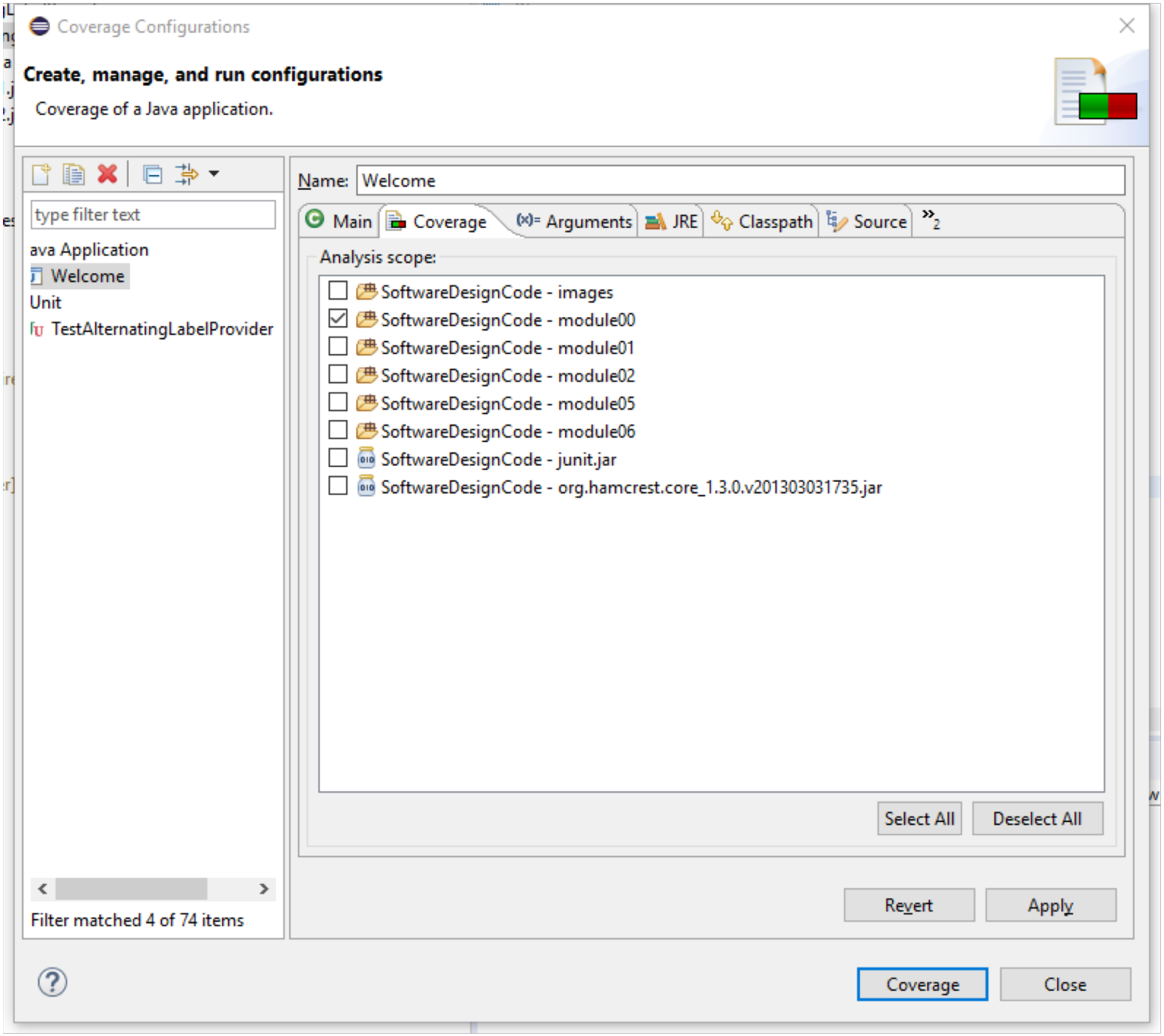
Additional Conditions Coverage

- 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

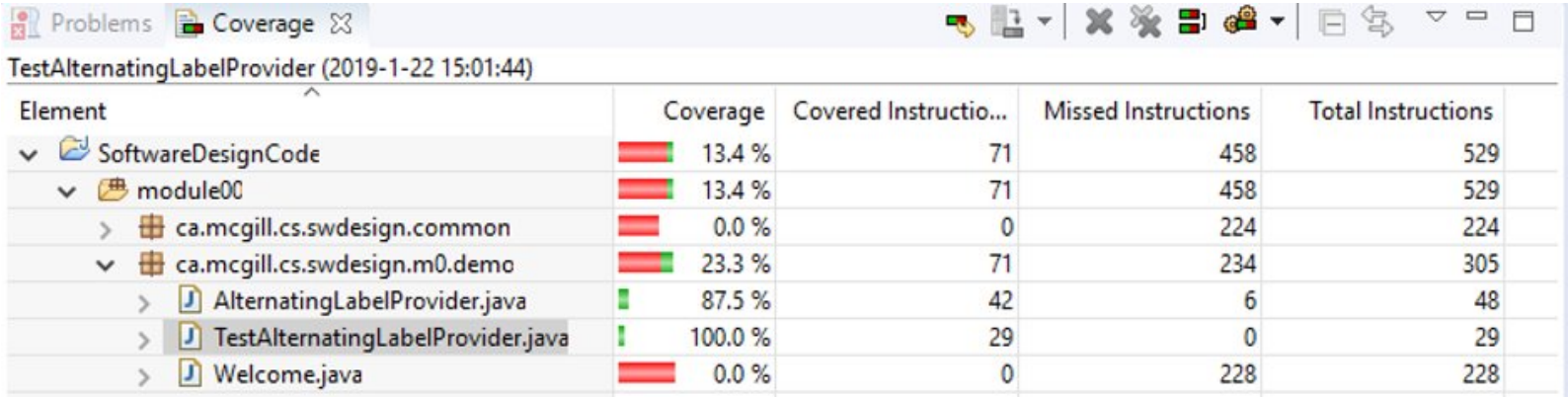
Coverage Tool: EclEmma

- 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
Main
Coverage
Arguments
JRE
Classpath
Source
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
Select All
Deselect All
Revert
Apply
Coverage
Close

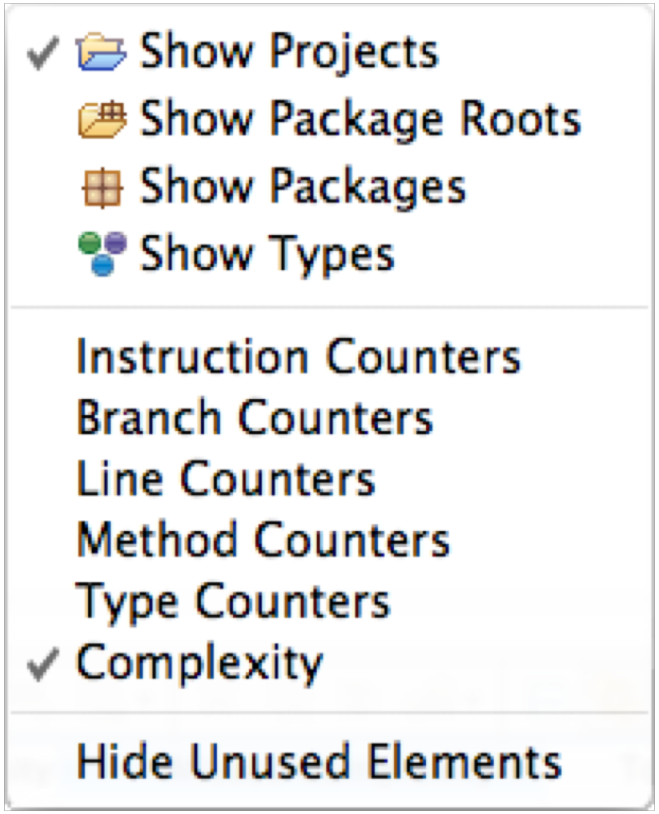
Coverage View

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



Counter Mode

- Select Different Counter Mode From drop-down menu



Source Code Annotation

- Green: Fully Covered
- Yellow: Partially Covered
- Red: Not 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.

Thanks for attending

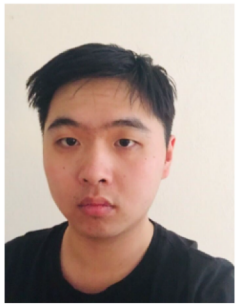
- TA Team

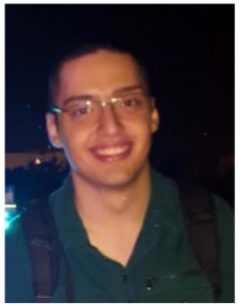

Mathieu Nassif


Deeksha Arya


Alexander Nicholson


Cheryl Wang


Shi Yan Du


Kian Ahrabian