Unit Testing

Jin L.C. Guo

Objectives

- Concepts and Principles:

Unit testing, regression testing, test suites, test coverage;

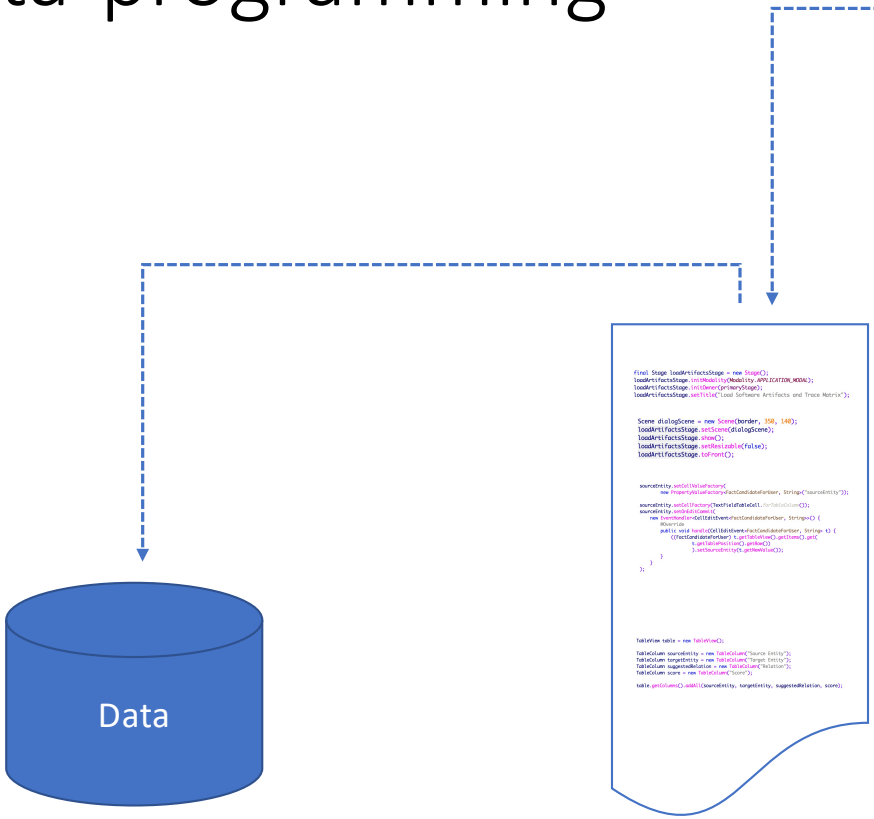
- Programming Mechanisms:

Unit testing frameworks, JUnit, metaprogramming, annotations;

- Design Techniques:

Test suite design and organization

Meta-programming



programming – we manipulate data

Meta-programming – we manipulate source code

@Test Annotation

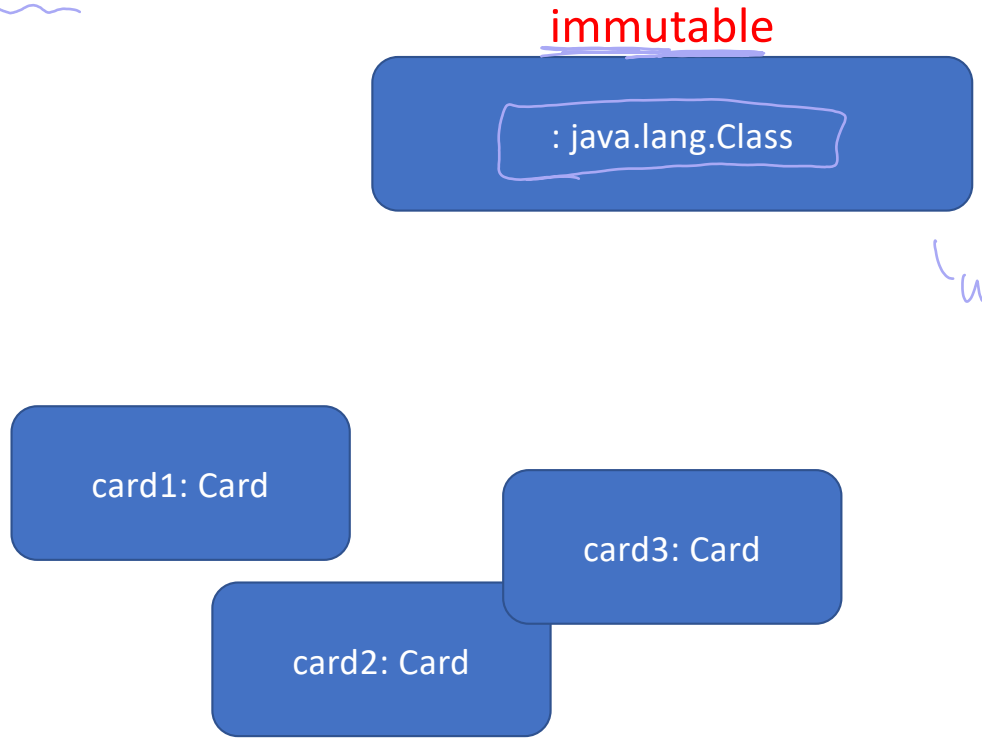
```
import org.junit.jupiter.api.Test;

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


class UndergradTest {

    @Test
    void getFirstName() {
        Student s = new Undergrad("001","Lily", "Joe");
        assertEquals("Lily", s.getFirstName());
    }
}
```

Java Reflection



JVM creates an instance from during runtime

we can't create an object, JVM can

Java Class Class<T>

public final class **Class<T>** extends Object implements Serializable, GenericDeclaration, Type, AnnotatedElement, TypeDescriptor.OfField<Class<?>>, Constable

Class has no public constructor. Instead a Class object is constructed automatically by the Java Virtual Machine … … The methods of class Class expose many characteristics of a class or interface … …

Obtain instances of Class

```
Card card1 = new Card(Card.Rank.ACE, Card.Suit.CLUBS);
Card card2 = new Card(Card.Rank.FIVE, Card.Suit.CLUBS);

Class class1 = card1.getClass();
Class class2 = card2.getClass();
Class class3 = Card.class;
Class class4 = int.class;

try {
    Class class5 = Class.forName("ca.mcgill.cs.swdesign.m2.Card");

} catch (ClassNotFoundException e) {
    e.printStackTrace();
}
```

Access Private Fields

used to test private fields

```
Card card1 = new Card(Card.Rank.ACE, Card.Suit.CLUBS);
Card card2 = new Card(Card.Rank.FIVE, Card.Suit.CLUBS);

Class class1 = card1.getClass();
Class class2 = card2.getClass();
Class class3 = Card.class;
Class class4 = int.class;
```

Card.Rank type

variable name in class file

```
try {
    Field rankOfCard = class1.getDeclaredField("aRank");
    rankOfCard.setAccessible(true);
    rankOfCard.set(card1, Card.Rank.JACK);

} catch (NoSuchFieldException | IllegalAccessException e) {
    e.printStackTrace();
}
```

temporarily accessible

Call constructors through Reflection

```
Constructor<Card>[] constructors = class1.getDeclaredConstructors();

for(Constructor c:constructors) {
    try {
        Object constructedThroughReflection
             = c.newInstance(card1.getRank(), card1.getSuit());

    } catch (InstantiationException e) {
        e.printStackTrace();
    } catch (IllegalAccessException e) {
        e.printStackTrace();
    } catch (InvocationTargetException e) {
        e.printStackTrace();
    }
}
```

getConstructors() - public

all private and public w/in current class file

returns object type
↳ will need to downcast

Call method through Reflection

```
try {
    Method privateMethod =
            BrightnessFlashlight.class.getDeclaredMethod("incrementBrightness");
    privateMethod.setAccessible(true);
    BrightnessFlashlight flashlight =  new BrightnessFlashlight();
    privateMethod.invoke(flashlight);

} catch (NoSuchMethodException | IllegalAccessException |
        InvocationTargetException e) {
    e.printStackTrace();
}
```

Call method through Reflection

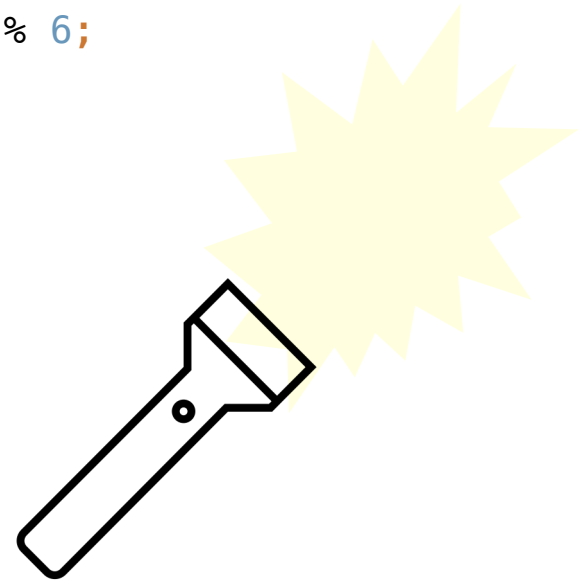
```
Method[] methods = class1.getDeclaredMethods();
for(Method m:methods) {
    out.println(m.toString());
    out.println(Modifier.toString(m.getModifiers()));
}
```

```
Method privateMethod =
BrightnessFlashlight.class.getDeclaredMethod("incrementBrightness");
privateMethod.setAccessible(true);
BrightnessFlashlight flashlight =  new BrightnessFlashlight();
privateMethod.invoke(flashlight);
```

Recall from last lecture about Unit Testing:
What if we have a private method?

```
public class BrightnessFlashlight {

    private void incrementBrightness() {
        this.brightness = (this.brightness + 1) % 6;
    }
}
```



Use Reflection in Unit Testing

declare exceptions – client must handle

```
@Test
void testPrivateIncrementBrightness() throws NoSuchMethodException, InvocationTargetException,
        IllegalAccessException {
    BrightnessFlashlight f = new BrightnessFlashlight();
    Method incrementBrightness = f.getClass().getDeclaredMethod("incrementBrightness");
    incrementBrightness.setAccessible(true);

    incrementBrightness.invoke(f);
    assertEquals(1, f.getBrightness());

    incrementBrightness.invoke(f);
    assertEquals(2, f.getBrightness());

    f.setBrightness(5);
    incrementBrightness.invoke(f);
    assertEquals(0, f.getBrightness());
}
```

must be correct

Testing Private Methods in Practices

1) • Test indirectly by executing the accessible methods that call them

or

2) • Use Java Reflection to change the accessibility of the private methods

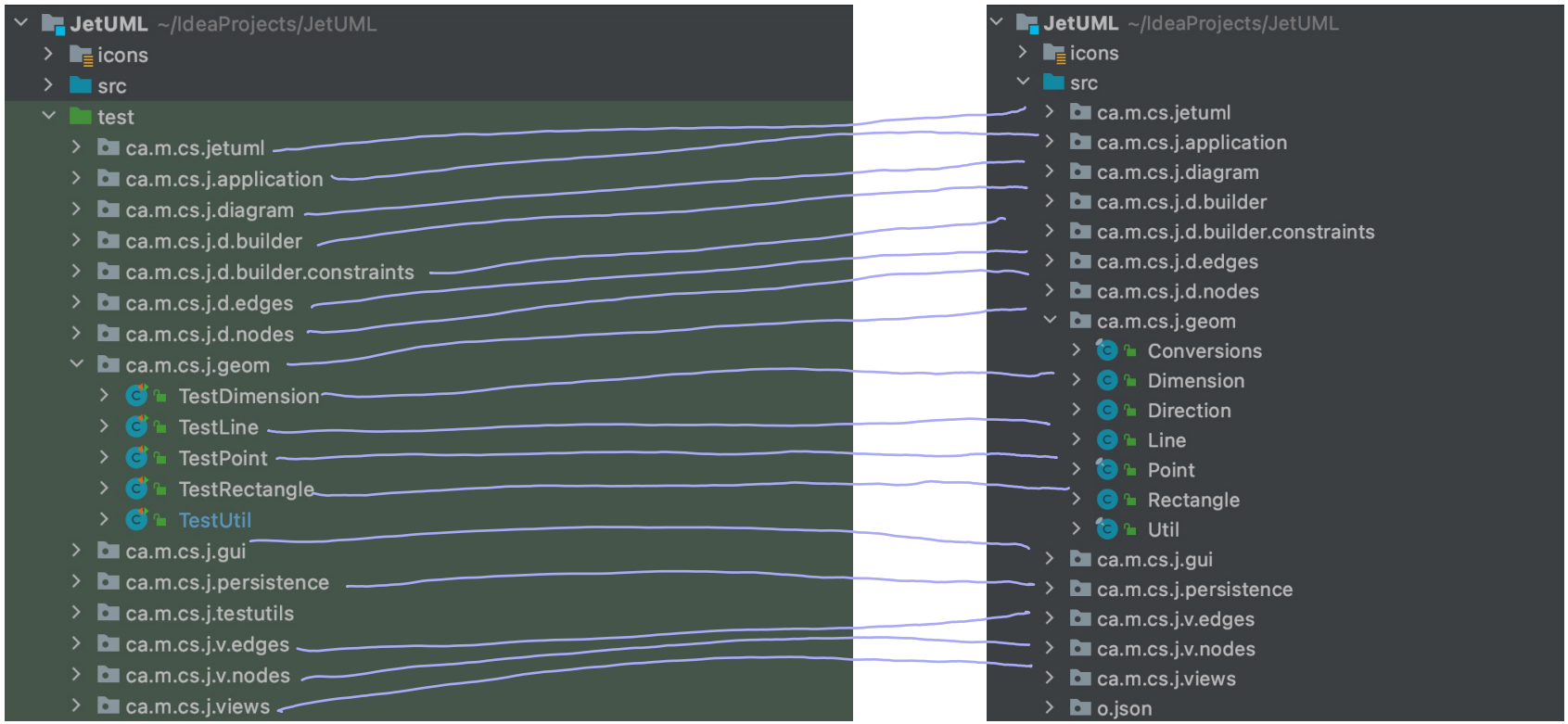
within test class

testCertainBehavior

testCertainBehavior_ExpectedOutcome()

Test Suite

- A collection of tests for a project



Shall concise descriptive names for class

testmethods can have verbose names be JUnit calls not user

Principles for Designing Unit Test

- Fast
- Independent
- Repeatable
- Focused
- Readable

Independent

```
class BrightnessFlashlightTest {
    BrightnessFlashlight f = new BrightnessFlashlight();

    @Test
    void testValidBrightness() {
        f.setBrightness(3);
        assertEquals(3, f.getBrightness());
    }

    @Test
    void testDefaultBrightness1() {
        assertEquals(0, f.getBrightness());
    }

}
```

Junit makes tests independent!

separate/ won't interfere w/ each other

Is this test going to pass? Yes!

Independent

```
class BrightnessFlashlightTest {
    BrightnessFlashlight f;

    @BeforeEach
    void setUp() {
        f = new BrightnessFlashlight();
    }

    @Test
    void testValidBrightness() {
        f.setBrightness(3);
        assertEquals(3, f.getBrightness());
    }

    @Test
    void testDefaultBrightness() {
        assertEquals(0, f.getBrightness());
    }
}
```

for more complex cases
executes this set up
before each test

How to respect the principles in this case:

```
public class BrightnessFlashlight {
    private int brightness = 0;
    private Optional<CloudConfig> config;

    BrightnessFlashlight() {
        config = Optional.empty();
    }


    public void SetCloudConfig(CloudConfig pConfig) {
        config = Optional.of(pConfig);
    }

    public void setBrightnessFromCloud() {
        if (config.isPresent()) {
            setBrightness(config.get().getBrightConfig());
        }
    }
```

Testing with Stubs

dummy classes

```
class BrightnessFlashlightTest {

    static class CloudConfigStub implements CloudConfig {
        int aBrightConfig;

        CloudConfigStub(int pBrightConfig) {
            aBrightConfig = pBrightConfig;
        }

        @Override
        public int getBrightConfig() {
            return aBrightConfig;
        }
    }
            @Test
            void testBrightnessFromCloud(){
                BrightnessFlashlight f = new BrightnessFlashlight();
                CloudConfigStub cloudC = new CloudConfigStub(4);
                f.SetCloudConfig(cloudC);
                f.setBrightnessFromCloud();

                assertEquals(4, f.getBrightness());
            }
```

Use Stubs when

- Triggers the execution of a large chunk of other code;
- Includes sections whose behavior depends on the environment;
- Involves non-deterministic behavior (e.g., randomness).

Test Coverage

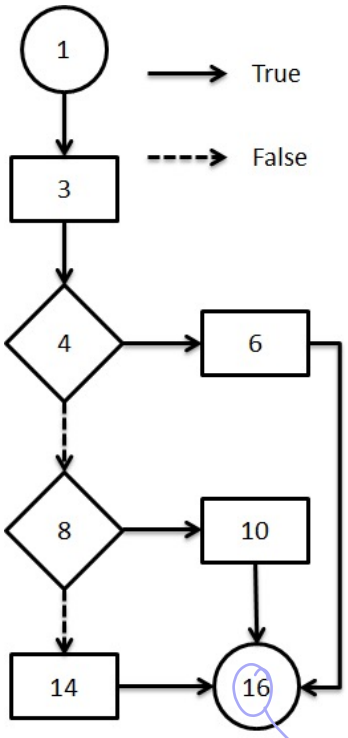
- Measures the *extent of* thoroughness of a test suite.
- What percentage of components does our test suite execute in the program?

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!”

Example:



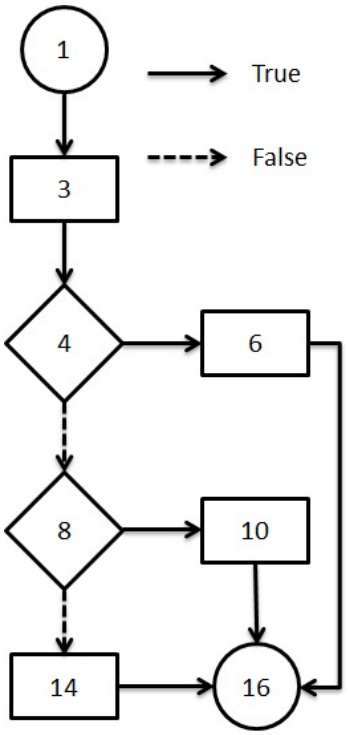
correspond to line

```
/**
 * Finds the roots for ax^2 + bx + c
 */
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 }
```

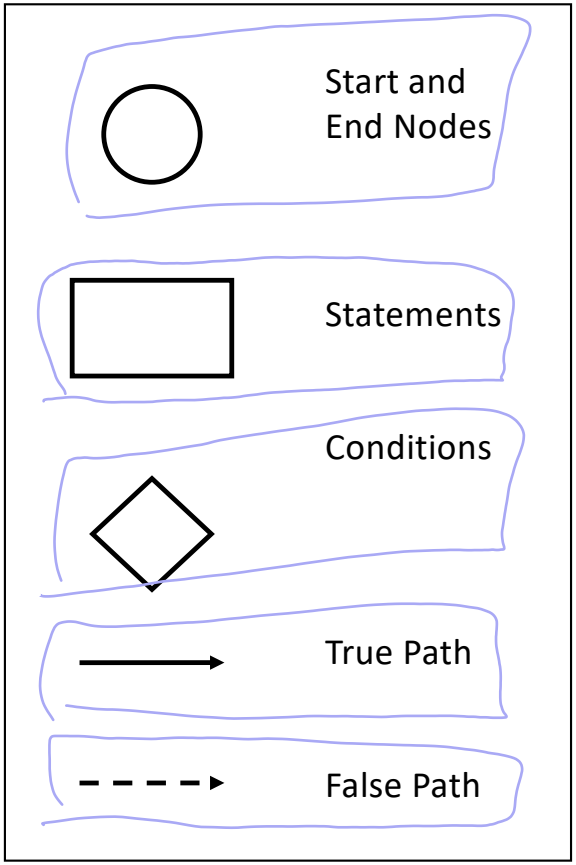
The Control Flow Graph (CFG)

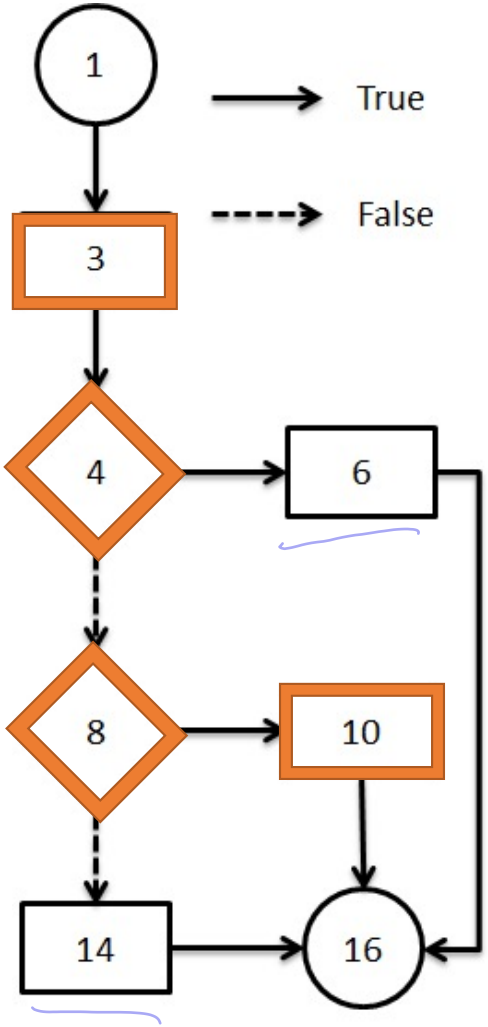
https://github.com/prmr/SoftwareDesign/blob/master/modules/Module-04.md

Example:



The Control Flow Graph (CFG)





Statement Coverage

- Input: (1,2,1)

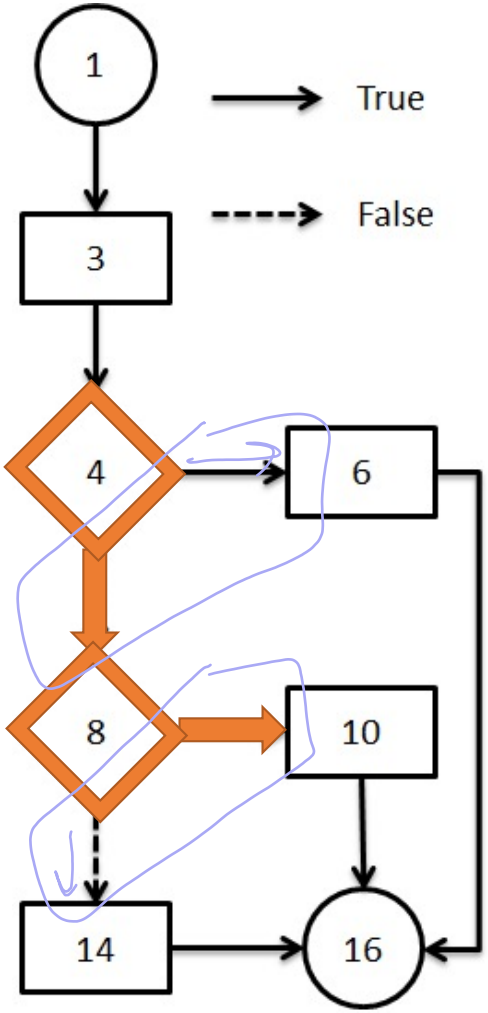
= (Number of statements executed/ total number of statements)

= 4/6 = 67%

(start and end nodes are ignored)

percent of lines covered
doesn't test lines 6, 14

```
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

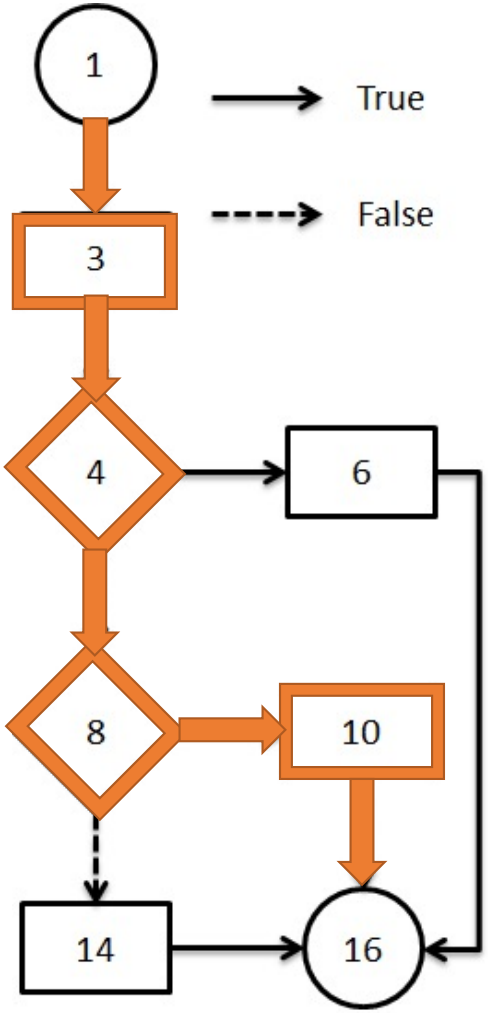
- Input: (1,2,1)

= (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)

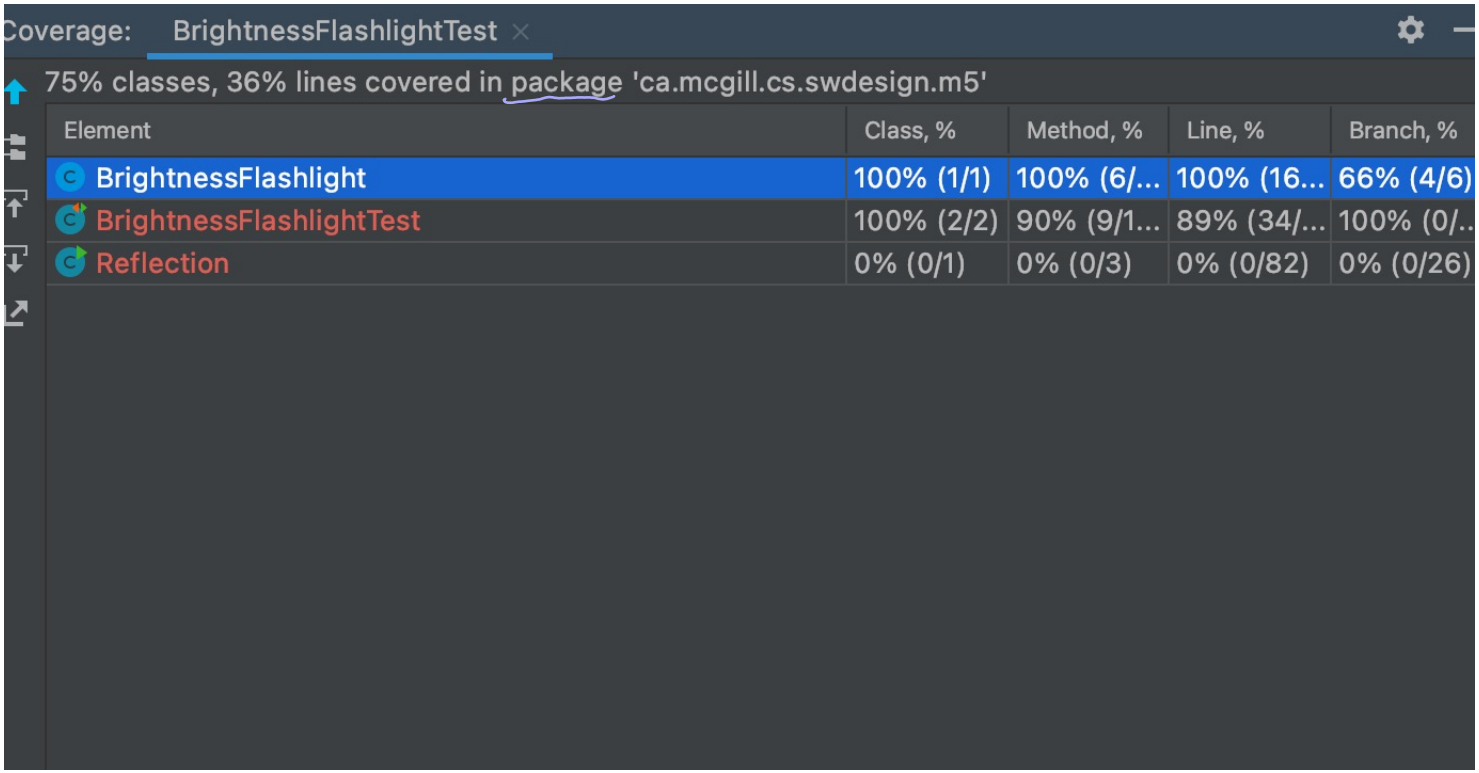
= (Number of paths executed / total number of paths)

= 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 }
```

Using test coverage tools



Summary

- Concepts and Principles:

Unit testing, regression testing, test suites, test coverage;

- Programming Mechanisms:

Unit testing frameworks, JUnit, metaprogramming, annotations;

- Design Techniques:

Test suite design and organization