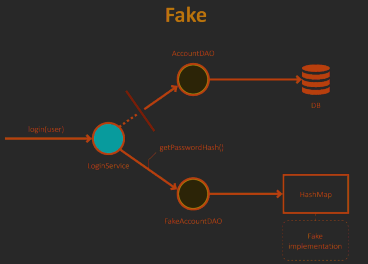
Mocking and Specification Testing

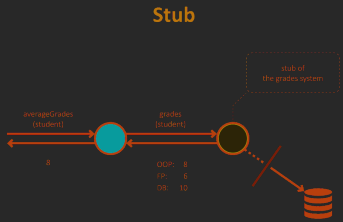
Isaac Griffith Idaho State University

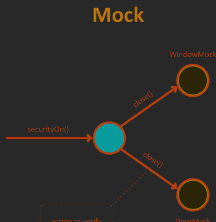
Test Doubles

Idaho State University | Computer Science

- Often our objects are hard to isolate because of interactions with other objects, databases, networks, etc.
- However, we have come up with several approaches to get around this







Mock Objects

Idaho State University | Computer Science

- **Mock Objects** are dummy implementations for an interface or class
- These allow us to define the output of certain method calls
- By recording the interaction with the system
- Allowing tests to validate the outcome

The Mocking Process

- Follows the AAA method:
  - **Arrange** - Setup the mock dependencies for the class under test
  - **Act** - Execute the code in the class under test
  - **Assert** - Validate if the code executed as expected

Outcomes

Idaho State University | Computer Science

After today's lecture you will be able to:

- Describe the basic concepts of Mock Objects
- Use the Mockito Project to provide Mocking for JUnit Jupiter
- Describe the basic concepts of specification testing
- Use the Spock framework to implement and execute specification tests
- Understand how Spock is useful for BDD
- Evaluate the effectiveness of your tests using the code coverage tool jacoco

Mock Objects

CS 2263

Mockito

- A popular open source framework for mocking Java objects
- Works in conjunction with JUnit
- Greatly simplifies the Mocking experience
- Although it does provide many great advantages it is not without its limitations…

Mockito and Gradle

We can add mockito to our gradle builds by including the following dependencies

```
dependencies {
    testImplementation 'org.mockito:mockito-inline:4.0.0'
    testImplementation 'org.mockito:mockito-junit-jupiter:4.0.0'
}
```

Mockito Mock Objects

- There are several ways to create mock objects with Mockito
- The two basic approaches are by either:
  - Using the `@Mock` annotation
  - Using the static `mock()` method
- **Note:** if you use the `@Mock` annotation, you will need to trigger the initialization of annotated fields
  - This can be done using `@ExtendWith(MockitoExtension.class)` annotation on the test class

Example

Let's start with the data model

```
public class Database {
  public boolean isAvailable() {
    return false;
  }
  public int getUniqueId() {
    return 42;
  }
}

public class Service {
  private Database database;

  public Service(Database database) {
    this.database = database;
  }

  public boolean query(String query) {
    return database.isAvailable();
  }

  @Override
  public String toString() {
    return "Using database with id: " +
      String.valueOf(database.getUniqueId());
  }
}
```

Add the test with Mocks

```
import static org.junit.jupiter.api.Assertions.assertNotNull;
import static org.junit.jupiter.api.Assertions.assertTrue;
import static org.mockito.Mockito.when;

import org.junit.jupiter.api.Test;
import org.junit.jupiter.api.extension.ExtendWith;
import org.mockito.Mock;
import org.mockito.junit.jupiter.MockitoExtension;

@ExtendWith(MockitoExtension.class)
class ServiceTest {

    @Mock
    Database databaseMock;

    @Test
    public void testQuery()  {
        assertNotNull(databaseMock);
        when(databaseMock.isAvailable()).thenReturn(true);
        Service t  = new Service(databaseMock);
        boolean check = t.query("* from t");
        assertTrue(check);
    }
}
```

Spec Testing

CS 2263

Why not just use JUnit or TestNG?

- JUnit and TestNG are good testing frameworks, but
  - often need an additional mocking or stubbing library
  - syntax generally limited to Java

- Spock has some built-in advantages:
  - mocking and stubbing part of core
  - succinct syntax for data driven testing
  - leverages Groovy syntax

Do we need another framework?

- Spock incorporates the best concepts from JUnit, RSpec, jMock, and Mockito
- The answer is obviously yes; otherwise there wouldn't be this lecture, right?
- Plus Spock makes testing fun!

How Spock Measures Up

- Unit Testing
  - JUnit
  - TestNG
- Mocking and Stubbing
  - EasyMock
  - jMock
  - Mockito
  - PowerMock
  - jMockit
- BDD
  - Cucumber
  - JBehave

The Spock Framework does all of this

- Additionally, Spock has a built in JUnit

Comparison to JUnit Concepts

JUnit	Spock
Test Class	Specification
Test	Feature
Test Method	Feature method
@Before	setup()
@After	cleanup()
Assertion	Condition
@TEST(expected…)	Exception condition

Simple Broken Tests

Idaho State University | Computer Science

JUnit

```
public class CalculatorTest {
  Calculator calculator;

  @Before
  public void setup() {
    calculator = new Calculator();
  }

  @Test
  public void testSimpleAddition() {
    assertEquals("2+2=4", 5,
        calculator.add(2, 2));
  }
}
```

Spock

```
class CalculatorSpec extends Specification {
  def calculator

  def setup() {
    calculator = new Calculator()
  }

  def "Test Simple Addition"() {
    expect:
      calculator.add(2, 2) == 5
  }
}
```

Simple Broken Test Results

Idaho State University | Computer Science

JUnit Failed Test Output

```
CalculatorTest > testSimpleAdditon FAILED
  java.lang.AssertionError: 2+2=4 expected:<5> but was:<4>
    at org.junit.Assert.fail(Assert.java:88)
    at org.junit.Assert.failNotEquals(Assert.java:834)
    at org.junit.Assert.assertEquals(Assert.java:645)
    at CalculatorTest.testSimpleAddition(CalculatorTest.java:15)
```

Spock Failed Test Output

```
CalculatorSpec > Test Simple Addition FAILED
    Condition not satisfied:

    calculator.add(2,2) == 5
    |          |        |
    |          4        false
    Calculator@11dd18e
        at CalculatorSpec.Test Simple Additon(CalculatorSpec.groovy:13)
```

Spruced up using Spock Parameterization

Idaho State University | Computer Science

```
class CalculatorSpec extends Specification {
  @Unroll
  def "#a + #b = #c"() {
    setup:
      def calculator = new Calculator()
    expect:
      calculator.add(a, b) == c
    where:
      a | b || c
      2 | 2 || 4
      2 | 2 || 5
  }
}
```

Parameterized Test Output

Idaho State University | Computer Science

Spock Passed Test Output:

```
CalculatorSpec > 2 + 2 = 4 PASSED
```

Spock Failed Test Output:

```
CalculatorSpec > 2 + 2 = 5 FAILED
    Condition not satisfied:

    calculator.add(a,b) == c
    |          |   | |  |  |
    |          4   2 2  |  5
    Calculator@11dd18e  false
        at CalculatorSpec.#a + #b = #c(CalculatorSpec.groovy:19)
```

Spock and Mocking


```
def "Mocked calculator"() {
  setup:
    def calculator = Mock(Calculator)
  when:
    calculator.add(2,2)
  then:
    1 * calculator.add(2,2)
}
```

Spock and Mocking

Idaho State University | Computer Science

```
def "Mocked calculator"() {
  setup:
    def calculator = Mock(Calculator)
  when:
    calculator.add(2,2)
  then:
    1 * calculator.add(2,2)
    0 * calculator.add(_,_)
}
```

Spock and Stubbing


```
def "Stubbed calculator"() {
  setup:
    def calcultor = Stub(Calculator) {
      add(2,2) >> 4
    }
  expect:
    calculator.add(a,b) == c
  where:
    a | b || c
    2 | 2 || 4
}
```

Spock and Stubbing


```
def "Stubbed calculator"() {
  setup:
    def calcultor = Stub(Calculator) {
      add(2,2) >> 4
      add(_,_) >> {x, y -> x + y}
    }
  expect:
    calculator.add(a,b) == c
  where:
    a | b || c
    2 | 2 || 4
    3 | 3 || 6
    4 | 3 || 7
}
```

Spock and BDD

Idaho State University | Computer Science

Classic Example of BDD:

Scenario: Multiple Givens
Given: one thing
And: another thing
And: yet another thing
When: I open my eyes
Then: I see something
But: I don't see something else

Valid Spock Code

```
def "Multiple Givens"() {
  given: "one thing"
  and: "another thing"
  and: "yet another thing"
  when: "I open my eyes"
  then: "I see something"
  and: "I don't see something else"
}
```

Spock plus BDD and Parameterization

Idaho State University | Computer Science

```
@Unroll
def "BDD: #a + #b = #c"() {
  given: "a new calculator"
    def calculator = new Calculator()
  and: "nothing is done to the calculator before addition"
  when: "adding two values together"
    def sum = calculator.add(a, b)
  then: "the result is the expected sum"
    c == sum
  where:
    a | b || c
    2 | 2 || 4
    3 | 2 || 5
}
```

Code Coverage

CS 2263

What is Code Coverage?

- **Code coverage** is a metric used to evaluate a test suite
- Essentially it evaluates how much of a program was executed by the test suite
- Thus, it returns a percentage value:
  - Across each method
  - Across each class
  - Across each package
  - Across the entire program
- Coverage also gives us the ability to determine when we can stop testing a system.
  - Our goal with testing is to simply cover the entire program
  - Thus, we need only write enough tests to achieve 100% coverage

Coverage with Jacoco

- Jacoco is a tool which works with your test environment to determine how much coverage your tests provide.
- It informs us the percentage of lines covered by the tests executed
  - Show this at the Package, class, and method level
- The ultimate goal is to achieve 100% code coverage, but often this is not possible.
  - Thus, a good stopping goal is 85% code coverage

Using Jacoco

Idaho State University | Computer Science

build.gradle

```
plugins {
  id 'jacoco'
}

jacoco {
  toolVersion = "0.8.6"
  reportsDirectory = file("$builddir/jacoco")
}

test {
  // report is generated after tests run
  finalizedBy jacocoTestReport
}

jacocoTestReport {
  // tests are required to run first
  dependsOn test
  reports {
    xml.enabled false
    csv.enabled false
    html.destination file("${buildDir}/jacocoHtml")
  }
}
```

- Jacoco works with the `test` phase of the build
- Adds the following tasks:
  - **jacocoTestCoverageVerification** - verifies code coverage metrics based on specified rules for the test task
  - **jacocoTestReport** - generates the code coverage report for the test task

Setting Coverage Goals

Idaho State University | Computer Science

build.gradle

```
jacocoTestCoverageVerification {
    violationRules {
        rule {
            limit {
                counter = 'LINE'
                value = 'COVEREDRATIO'
                minimum = 0.85
            }
        }
    }
}
```

- Rules can note the element (`CLASS`, `LINE`, `METHOD`, etc)
- Rules can also have an includes and excludes section both of which contain a list of classes

- This rule states that we have a minimum of 85% line coverage
- We could also replace "`LINE`" with:
  - `BRANCH` - number of execution branches
  - `CLASS` - number of classes
  - `INSTRUCTION` - number of code instructions
  - `METHOD` - number of methods
- We can replace the value "COVEREDRATIO" with:
  - `COVEREDCOUNT` - absolute number of covered items
  - `MISSEDCOUNT` - absolute number of items not covered
  - `MISSEDRATIO` - ratio of items not covered
  - `TOTALCOUNT` - total number of items

For Next Time

- Review the Moodle Mocking and Spec Testing Resources
- Review this lecture
- Come to Lecture!

Are there any questions?