Reflection Applied: Mocking

CPSC 501: Advanced Programming Techniques
Winter 2025

Jonathan Hudson, Ph.D
Assistant Professor (Teaching)
Department of Computer Science
University of Calgary

Wednesday, March 5, 2025

UNIVERSITY OF CALGARY

Advanced Testing

State vs Interaction Testing

- JUnit is designed around state testing
  - You run a function and ensure end state matches post-condition of function interface
  - You have limited awareness of anything that the function does outside of exposed behaviour
    - Values provided (an accessible information in them that could be modified)
    - Returned result

- Interaction testing
  - You verify if interactions between functions are as expected
  - Mockito (Or similar testing frameworks) let you do this
    - Mockito can also assist state testing by allowing intercession
      - letting you stub interactions so that a test doesn't rely on correctness of dependent classes
      - helps reduce tests that have grown into integration test back into unit tests

Stubs

- Test Stub (old method)
  - Hand-coded object for testing, have to sub in fully coded replacement
  - Rather time consuming
  - Becomes parallel development
  - Often requires rewriting existing code so that it is designed around interface that allows this replacement (rather than copy-paste code back and forth when testing

Stubs

- Maybe we want to test code that involves a lookup a book in a storage object by id
  - storage.getById(String)
  - We aren't testing the lookup, but only later properties post-retrieval
    - Storage could involve something complex like a red-black-tree which could have bugs!
  - With a stub we could code a pretend storage object good enough for our limited tests (maybe using simple if/then)
    - if id == lookup_id return new Book("title")
  - Our test could sub-in this universally non-functional stub in place of the other code, so that the test can rely on the correct book always being returned

Stubs

```
interface Library {
        public Book getById(String name);
}
public StubLibrary implements Library{
        public Book getById(String name){
                if name.equals(“Book1”)
                        return new Book(“title1”);
                if name.equals(“Book2”)
                        return new Book(“title2”);
                return null;
        }
}
```

```
public RealLibrary implements Library{
        public Book getById(String name){
                //relies on buggy RedBlackTree impl.
        }
}
```

Mocking

Test Doubles

- Mockito lets you mock (or make a double) for different things

- Mock Object
  - Created using Mockito's framework API
    - There are others like EasyMock, Jmocking, etc.
  - Has exact same interface as original object
    - We don't need to have made an interface
  - Allows us to define all functionality ourselves
    - We can do intercession
  - Allows us to monitor how the mock object is used
    - Framework latches on something like a profiler
    - Verify!

Usage Pattern

1. Setup mocks for all dependencies you want to replace
    1. This can be done setting up the test class
    2. Or can be done in the test itself
2. Set-up intercession
    1. when() -> do X
        1. thenReturn() is most common
3. Call tested logic
4. Verify results
    1. Regular unit testing of state
    2. Interaction testing using the profiled Mockito
        1. verify()

Dangers

- Mocking lets us bypass the interface limitations the object provided
- This creates tighter coupling of test code and actual code
- It can be easy to make brittle tests
  - Tests that are dependent on how the programmer chose to solve the problem
    - Ex. Maybe you verify how many times a sub-api is accessed when the outer function is called
    - Some solutions might do it X times and other programmers might to it Y times

    - Another Ex.
    - Testing the SQL string passed to the DB (there are many equivalent queries that get same results)
- Try to avoid getting to (white box test) implantation specific in the testing

Mockito

- Open Source
- Most common mocking framework
- https://site.mockito.org/
- Can add it through Maven

- 5.14.2 is most recent
- MIT license

Benefits

- No custom stubs
- Simple post-verification setup (others like EasyMock require you to write expected behaviour first -> a more academic approach)
- Refactoring-safe
  - Design means most refactoring triggers will also trigger refactoring in tests
- Supports returns and exceptions
- Flexible parameters setting
- Single jar
- Easy to learn
- Most commonly found Q/A on internet for mocking in Java

Intro Examples

First Examples

- Imported

  import static org.mockito.Mockito.*;

- Mock

  List mockedList = mock(List.class);

- Intercession

  when(mockedList.get(0)).thenReturn(“first”);
  when(mockedList.get(or(eq(1),eq(2))).thenReturn(“more”);
  when(mockedList.get(gt(2)).thenThrow(new RuntimeException());

- Others
  - anyInt, anyString
  - Custom parameter matchers (Hamcrest) https://hamcrest.org/

First Examples - Verify

- Verify interaction

```
verify(mockedList).clear();
verify(mockedList).add(“one”);
verify(mockedList).add(anyString());
verify(mockedList, atLeastOnce()).add(“one”);
verify(mockedList, times(3)).add(anyString());
```

- Others
  - never, atMost
  - verifyZeroInteractions

More Examples

Basics

First test is regular JUnit tests

Second test using a mock Calculator

add() is never actually used

```
public class Calculator {
    public int add(int a, int b) {
        return a + b;
    }
}
```

```
@Test
  public void testAdd1() {
     Calculator calculator = new Calculator();
     int result = calculator.add(2, 3);
     assertEquals(5, result, "2 + 3 should equal 5");
  }

  @Test
  public void testAdd2() {
     Calculator calculator = mock(Calculator.class);
     when(calculator.add(2,3)).thenReturn(5);
     int result = calculator.add(2,3);
     assertEquals(5, result, "2 + 3 should equal 5");
  }
```

Escape Dependency Chain

We have a **UserService** that relies on **UserRepository**

Any time we make a test for isUserActive we need userRepository.findById() to be correct as well!

But we haven't even completed it yet

```
public class User {
    private int id;
    private String name;
    private boolean active;
    public User(int id, String name, boolean active) {
        this.id = id;
        this.name = name;
        this.active = active;
    }
    public boolean isActive() {
        return active;
    }
}
```

```
public class UserService {
    private UserRepository userRepository;

    public UserService(UserRepository userRepository) {
        this.userRepository = userRepository;
    }

    public boolean isUserActive(int userId) {
        User user = userRepository.findById(userId);
        return user != null && user.isActive();
    }
}
```

```
public class UserRepository {
    public User findById(int userId) {
        return null;
    }
}
```

Escape Dependency Chain

We have a **UserService** that relies on **UserRepository**

Any time we make a test for isUserActive we need userRepository.findById() to be correct as well!

But we haven’t even completed it yet

```
@Test
  public void testIsUserActive() {
     // Create a mock UserRepository
     UserRepository userRepository = mock(UserRepository.class);

     // Define the behavior of the mock UserRepository
     User activeUser = new User(1, "John Doe", true);
     when(userRepository.findById(1)).thenReturn(activeUser);

     // Instantiate UserService with the mock UserRepository
     UserService userService = new UserService(userRepository);

     // Test the isUserActive method
     assertTrue(userService.isUserActive(1),
                          "User with ID 1 should be active");

     // Verify the mock UserRepository's findById method was called
    //with the correct argument
     verify(userRepository, times(1)).findById(1);
  }
```

Escaping service requirements (online service)

```
public class Network {

   private URL url;

   public Network(URL url) {
      this.url = url;
   }

   public boolean isUrlAvailable() throws IOException {
      return getResponseCode() ==
            HttpURLConnection.HTTP_OK;
   }

   private int getResponseCode() throws IOException {
      HttpURLConnection con = (HttpURLConnection)
                        this.url.openConnection();
      return con.getResponseCode();
   }
```

```
   public String getData() throws IOException {
      if(isUrlAvailable()){
         BufferedReader in = new BufferedReader(
               new InputStreamReader(url.openStream()));
         String inputLine;
         StringJoiner sj = new StringJoiner(",");
         while ((inputLine = in.readLine()) != null)
            sj.add(inputLine);
         in.close();
         return sj.toString();
      }
      return null;
   }
}
```

It can be hard to test code that requires and active internet connection

Also tests that look like DDoS to a service or botting will often end up blocked

Escaping service requirements (online service)

- We can actually mock the original URL (this allows us to access the openConnection) to have it return a mocked HttpURLConnection that will give the response we want
- In this case failure to find the URL (this allows us to see if isUrlAvailable would give expected response in this scenario (state check after intercession)

```
@Test
  public void givenMockedUrl_whenRequestSent_thenIsUrlAvailableFalse() throws Exception {
      HttpURLConnection mockHttpURLConnection = mock(HttpURLConnection.class);
      when(mockHttpURLConnection.getResponseCode()).thenReturn(HttpURLConnection.HTTP_NOT_FOUND);

      URL mockURL = mock(URL.class);
      when(mockURL.openConnection()).thenReturn(mockHttpURLConnection);

      Network network = new Network(mockURL);
      assertFalse(network.isUrlAvailable(), "Url should not be available: ");
  }
```

Escaping service requirements (online service)

- Here I use URL mock to make the HTML response from connecting to a website

```
@Test
  public void givenMockedUrl_whenRequestSent_thenGetDefaultData() throws Exception {
     HttpURLConnection mockHttpURLConnection = mock(HttpURLConnection.class);
     when(mockHttpURLConnection.getResponseCode()).thenReturn(HttpURLConnection.HTTP_OK);
     URL mockURL = mock(URL.class);
     when(mockURL.openConnection()).thenReturn(mockHttpURLConnection);
     String fakeData = "<html>" +
           "<body>Hello</body>" +
           "</html>";
     InputStream is = new ByteArrayInputStream(fakeData.getBytes(StandardCharsets.UTF_8));
     when(mockURL.openStream()).thenReturn(is);

     Network network = new Network(mockURL);
     assertEquals(fakeData, network.getData());
  }
```

Escaping service requirements (database)

- Here I don’t want my unit test to rely on the actual database (or modify it)
- So I’ll want to modify my tests to bypass need for getDBConnection() to have been executed

```
public class Database {
    private Connection dbConnection;
    public void getDBConnection() throws SQLException {
        dbConnection = DriverManager.getConnection("jdbc:sqlite:sqlite.db");
    }
    public List<String> getNames(String query) throws SQLException {
        List<String> names = new ArrayList<>();
        ResultSet resultSet = dbConnection.createStatement().executeQuery(query);
        do{
            names.add(resultSet.getString("first_name")+" "+resultSet.getString("last_name"));
        }while(resultSet.next());
        return names;
    }
}
```

Escaping service requirements (database)

- We can mock a private variable by using @InjectMocks for the Class, and then @Mock for the field to mock inside
- I'll need to trigger this for each test

```
@InjectMocks
private Database database;
@Mock
private Connection connection;

@BeforeEach
public void setUp() {
   MockitoAnnotations.initMocks(this);
}
@Test
public void testGetNames() throws Exception {
   ResultSet rs = Mockito.mock(ResultSet.class);
   Mockito.when(rs.getString("first_name")).thenReturn("Jonathan");
   Mockito.when(rs.getString("last_name")).thenReturn("Hudson");
Mockito.when(connection.createStatement()).thenReturn(Mockito.mock
(Statement.class));

Mockito.when(connection.createStatement().executeQuery(Mockito.any
())).thenReturn(rs);

      List<String> names = database.getNames("SELECT first_name,
last_name FROM person WHERE height > 1.82;");
      assertEquals(1, names.size());
      assertEquals("Jonathan Hudson", names.getFirst());
      Mockito.verify(connection.createStatement(), Mockito.times(1));
   }
```

Escaping service requirements (database)

- Then I can bypass the need for DBConnection to have been setup
- getNames will now trigger the intercession on executeQuery to return my mock ResultSet *(I need a mock Statement returned by the connection I've mocked as well in the injection into Database class private variable, otherwise createStatement is null)*

```
@Test
  public void testGetNames() throws Exception {
    ResultSet rs = Mockito.mock(ResultSet.class);
    Mockito.when(rs.getString("first_name")).thenReturn("Jonathan");
    Mockito.when(rs.getString("last_name")).thenReturn("Hudson");

    Mockito.when(connection.createStatement()).thenReturn(Mockito.mock(Statement.class));
    Mockito.when(connection.createStatement().executeQuery(Mockito.any())).thenReturn(rs);

    List<String> names = database.getNames("SELECT first_name, last_name FROM person WHERE height > 1.82;");
    assertEquals(1, names.size());
    assertEquals("Jonathan Hudson", names.getFirst());
    Mockito.verify(connection.createStatement(), Mockito.times(1));
  }
```

Onward to … aspects.

Jonathan Hudson
jwhudson@ucalgary.ca
https://pages.cpsc.ucalgary.ca/~jwhudson/

UNIVERSITY OF CALGARY