Testing Abstract Classes: Strategies and Best Practices

Master Stubbing and Mocking with Mockito

Stubbing Abstract Methods

```
import static org.mockito.Mockito.*;

public abstract class BuildingDesign {
    public abstract String getMaterial();
}

BuildingDesign mockBuildingDesign = mock(BuildingDesign.class);
when(mockBuildingDesign.getMaterial()).thenReturn("Steel");
```

Using CALLS_REAL_METHODS

```
import static org.mockito.Mockito.*;

public abstract class BuildingDesign {
    public abstract String getMaterial();
    public String getDesignStyle() {
        return "Modern";
    }
}

BuildingDesign mockBuildingDesign = mock(BuildingDesign.class, CALLS_REAL_METHODS);
String designStyle = mockBuildingDesign.getDesignStyle(); // Returns "Modern"
```

Using thenCallRealMethod

```
import static org.mockito.Mockito.*;

public abstract class BuildingDesign {
    public abstract String getMaterial();
    public String getDesignStyle() {
        return "Modern";
    }
}

BuildingDesign mockBuildingDesign = mock(BuildingDesign.class);
when(mockBuildingDesign.getMaterial()).thenReturn("Steel");
when(mockBuildingDesign.getDesignStyle()).thenCallRealMethod();

String designStyle = mockBuildingDesign.getDesignStyle(); // Returns "Modern"
```

Create and Test Anonymous Subclasses

class Document

```java
// Abstract class that needs to be tested
public abstract class Document {
    public abstract void save();
    public abstract void print();
}
```

Testing the save Method

```java
class DocumentTest {

    @Test
    void testSaveMethod() {
        // Create an anonymous subclass of Document with a concrete implementation of the save method
        Document doc = new Document() {
            @Override
            public void save() {
                // Implementation for testing purposes
                System.out.println("Document saved!");
            }

            @Override
            public void print() {
                // The print method is not implemented because it's not the focus of this test
                throw new UnsupportedOperationException("Not implemented yet");
            }
        };

        assertDoesNotThrow(doc::save, "The save method should not throw an exception");
    }
}
```

Utilize JUnit 5 Features for Dynamic and Parameterized Testing

Parameterized Test

```
public abstract class AbstractOperationTest {

    public abstract int operation(int a);

    @ParameterizedTest
    @ValueSource(ints = {1, 2, 3, 5, 8, 13, 21})
    void testOperationWithMultipleValues(int input) {
        // Assume operation() should return a positive number for positive inputs
        int result = operation(input);

        assertTrue(result > 0, "The result should be positive for positive inputs");
    }
}
```

Dynamic Test

```java
public abstract class AbstractStockAnalyzerTest {

    // The createInstance method must be implemented by subclasses to provide a concrete instance of StockAnalyzer
    public abstract StockAnalyzer createInstance();

    @TestFactory
    Stream<DynamicTest> dynamicTestsForStockPriceAnalysis() {
        StockAnalyzer analyzer = createInstance();
        List<StockData> stockDataSamples = getStockDataSamples();

        return stockDataSamples.stream()
                .map(stockData -> DynamicTest.dynamicTest(
                        "Testing analysis for stock: " + stockData.getSymbol(),
                        () -> {
                            AnalysisResult result = analyzer.analyze(stockData);
                            assertNotNull(result, "Analysis result should not be null");
                            assertTrue(result.isHealthy(), "The stock should be marked as healthy based on the analysis criteria.");
                        }))
                .collect(Collectors.toList());
    }

    private List<StockData> getStockDataSamples() {
        // This method would return a list of StockData objects representing different stock scenarios
        // For instance, high volatility, steady growth, etc.
        // We will mock this for the example purpose
        return Arrays.asList(
            new StockData("AAPL", new BigDecimal("150.10"), ...),
            new StockData("GOOGL", new BigDecimal("2800.00"), ...),
            new StockData("AMZN", new BigDecimal("3100.50"), ...)
            // ... more data
        );
    }
}
// Assume StockData and AnalysisResult are classes that hold stock information and analysis outcome respectively.
```

Adopt the JUnit 5 Extension Model

Writing a Custom Extension

```java
// This is a custom extension that implements the BeforeEachCallback interface,
// which is used to perform actions before each test execution.
public class DatabaseConnectionExtension implements BeforeEachCallback, AfterEachCallback {

    // This method is called before each test execution.
    // It sets up a mock database connection that can be used by the test.
    @Override
    public void beforeEach(ExtensionContext context) throws Exception {
        // Initialize the mock connection
        MockDatabaseConnection mockConnection = MockDatabaseConnection.create();

        // Store the mock connection in the test context for retrieval in the test
        getStore(context).put("DB_CONNECTION", mockConnection);

        // You can output a log for debugging purposes
        System.out.println("Mock database connection created for test: " + context.getDisplayName());
    }

    // This method is called after each test execution.
    // It cleans up the mock database connection set up before the test.
    @Override
    public void afterEach(ExtensionContext context) throws Exception {
        // Retrieve and close the mock connection
        MockDatabaseConnection mockConnection = getStore(context).get("DB_CONNECTION", MockDatabaseConnection.class);
        mockConnection.close();

        // Log the closure of the connection
        System.out.println("Mock database connection closed for test: " + context.getDisplayName());
    }

    // Utility method to get the ExtensionContext Store
    private Store getStore(ExtensionContext context) {
        return context.getStore(Namespace.create(getClass(), context.getRequiredTestMethod()));
    }
}
```

Applying the Custom Extension

```java
// This test class demonstrates the use of the custom DatabaseConnectionExtension.
// The @ExtendWith annotation is used to tell JUnit to invoke the extension code at the appropriate points in the test lifecycle.
@ExtendWith(DatabaseConnectionExtension.class)
public class AbstractDatabaseManagerTest {

    private AbstractDatabaseManager manager;

    // This method is called before each test method.
    // It retrieves the mock database connection set up by the extension and uses it to create an instance of the manager.
    @BeforeEach
    public void setUp(ExtensionContext context) {
        // Retrieve the mock connection from the test context
        MockDatabaseConnection mockConnection = getStore(context).get("DB_CONNECTION", MockDatabaseConnection.class);

        // Create an instance of the manager with the mock connection
        manager = new AbstractDatabaseManager(mockConnection) {
            // Implementation of abstract methods, if necessary, for the test
        };
    }

    // Example of a test method that would use the mock database connection
    @Test
    public void testConnectionIsValid() {
        // The test can now interact with the 'manager' which is already configured
        // with a mock database connection thanks to our custom extension.
        assertTrue(manager.isConnectionValid());
    }

    // Utility method to get the ExtensionContext Store, similar to the one in the extension
    private Store getStore(ExtensionContext context) {
        return context.getStore(Namespace.create(getClass(), context.getRequiredTestMethod()));
    }
}
```

The @Testable Annotation

```java
import org.junit.jupiter.api.Testable;

// This abstract class represents a typical component in an application
// that handles database operations.
public abstract class AbstractDatabaseHandler {

    // An abstract method that needs to be implemented by subclasses
    // to perform an operation in the database.
    public abstract boolean performOperation(String data);

    // A concrete method that uses the abstract operation method.
    // It is annotated with @Testable to indicate that it's a point of interest for testing.
    @Testable
    public boolean handleData(String data) {
        // Pre-processing or validation logic can go here
        if (data == null || data.isEmpty()) {
            return false;
        }
        // Calling the abstract method which will be implemented by a subclass
        return performOperation(data);
    }
}
```