Graduate Project Final Paper

Mukund Mahesan, Thomas Davis

Department of Computer Science

Topic Title

Testing Object-Oriented Code – A Deep Dive into Mocking and Stubs

Team Members

Mukund Mahesan, Thomas Davis

Abstract

Testing in Object-Oriented (OO) systems introduces distinct challenges due to features such as inheritance, encapsulation, and polymorphism. While these concepts enhance modularity and reusability, they often complicate component isolation during testing. This project investigates how mocking and stubbing can simplify testing in OO contexts. The goal is to improve testability, maintainability, and isolation through the strategic use of test doubles. We demonstrate these ideas using a Java-based Library Management System (LMS) developed in two versions: one following poor design practices (tight coupling, static state) and another applying strong OO design principles (Dependency Injection, Interfaces). Our empirical analysis reveals that adherence to OO principles is a prerequisite for effective mocking, while ”bad” design necessitates fragile workarounds that reduce test reliability.

1. **Introduction**

   Software testing is a critical discipline for ensuring system reliability, yet it is often treated as an afterthought in the software development lifecycle. In the context of Object-Oriented (OO) systems, testing presents unique challenges. The very features that make OO powerful—encapsulation of state, inheritance hierarchies, and polymorphic behavior—can create complex webs of dependencies that make isolating a single unit of code difficult [1].

   The primary challenge in unit testing OO code is isolation. To test a specific class (the System Under Test, or SUT), one must often instantiate not only the SUT but also its dependencies, and potentially their dependencies, leading to an integration test rather than a unit test. This coupling results in ”fragile tests” that break when unrelated components change, and ”slow tests” that rely on external systems like databases or web services [2].

   This research explores the solution to these challenges: the use of Test Doubles, specifically Mocks and Stubs. By simulating the behavior of dependencies, developers can isolate the SUT, verify interactions, and simulate edge cases (such as network failures) that are difficult to reproduce with real objects [3].

   To provide a concrete analysis, we developed two contrasting implementations of a Library Management System (LMS). The first implementation (”Bad Design”) utilizes static methods, global state, and direct I/O coupling. The second implementation (”Good Design”) utilizes Dependency Injection (DI), Interfaces, and Domain Modeling. By attempting to write test suites for both, we demonstrate how design quality directly dictates the applicability and effectiveness of mocking frameworks like Mockito.

2. **Literature Review**

   2.1. The Taxonomy of Test Doubles

   Gerard Meszaros, in xUnit Test Patterns, introduced the standard terminology for test doubles, distinguishing between Stubs (which provide canned answers to calls made during the test) and Mocks (which are preprogrammed with expectations which form a specification of the calls they are expected to receive) [3]. This distinction is crucial: Stubs verify state (did the method return the right value?), while Mocks verify behavior (was the dependency called with the correct arguments?) [4].

   2.2. Testability and OO Design

   Research by Bruntink and van Deursen indicates a strong correlation between class coupling and testability [2]. Classes with high efferent coupling (dependencies on many other classes) require complex setup procedures for testing. Freeman and Pryce extend this by arguing that difficulty in mocking is a ”design smell,” indicating that the SUT has too many responsibilities or that the dependencies are ill-defined [5].

   2.3. Frameworks

   Modern Java testing relies heavily on JUnit for test orchestration and Mockito for creating dynamic proxies

[8, 9]. Mockito allows for "spying" on real objects—a hybrid approach where real methods are called unless stubbed—and "argument capturing," which allows granular verification of data passed between objects.

2.4 Test-Driven Development (TDD) as a Design Tool While mocks are often viewed primarily as isolation tools, Freeman and Pryce argue that they serve a dual purpose as design feedback mechanisms. In *Growing Object-Oriented Software, Guided by Tests*, they propose that difficulty in mocking a dependency often reveals "design smells," such as tight coupling or unclear responsibilities, rather than limitations in the testing framework [5].

This aligns with Beck's seminal work on Test-Driven Development (TDD), which emphasizes that tests should drive design decisions. Beck posits that writing the test first forces the developer to design the interface from the consumer's perspective, naturally leading to looser coupling and better encapsulated components [7]. In this context, mock objects are not merely distinct from stubs in their verification capabilities, but are essential tools for "Mockist TDD," a style of testing that focuses on defining the relationships and protocols between objects rather than just their state [4].

2.5 Empirical Metrics for Testability The relationship between Object-Oriented metrics and testability has been extensively studied to quantify the cost of "bad" design. Vokáč et al. conducted a large-scale analysis of C++ systems, finding that classes with high Coupling Between Objects (CBO) and low Cohesion (LCOM) were significantly more prone to defects and harder to test [6].

Similarly, Bruntink and van Deursen investigated the impact of class-level metrics on the effort required to write unit tests. Their empirical study demonstrated that as the number of efferent couplings increases, the setup code required for testing grows exponentially, often necessitating complex test doubles to isolate the unit under test [2]. These findings provide the theoretical foundation for our project's hypothesis: that refactoring towards dependency injection (reducing direct coupling) is quantifiable not just in code elegance, but in the reduction of test complexity.

3. **Methodology**

We implemented a Library Management System (LMS) using Java 17 and Gradle. The project is divided into two modules within the source code: com.example.library.bad and com.example.library.good.

3.1 The Bad Implementation: Anti-Patterns

- **Global State:** Uses `static ArrayList<String> books` and `static HashMap<String, Boolean> availability` (*Code: LibraryApp.java*).
- **Tight Coupling:** Input is handled via a static `Scanner` connected directly to `System.in`.
- **Mixed Concerns:** Business logic (borrowing a book) is interleaved with UI logic (`System.out.println`).

3.2 The Good Implementation: OO Principles

- **Dependency Injection:** The service does not instantiate its dependencies. A `DataStore` and `NotificationService` are injected through the constructor.
- **Interfaces:** `DataStore` is an interface, enabling replacement of the production `DatabaseStore` with a `StubDataStore` or Mockito mock during testing.
- **Encapsulation:** The `Book` and `User` models encapsulate state and expose behavior (e.g., `borrowBy()`) rather than relying solely on getters and setters.

4. **Implementation Analysis: Mocks vs. Stubs**

This section analyzes the practical application of test doubles within our "Good Design" implementation, demonstrating how specific testing requirements dictate the choice between Stubs, Mocks, and Spies.

4.1 Stubbing for State Verification Stubs are best utilized when the goal is to simulate a specific state of the world to drive the System Under Test (SUT) down a particular execution path. In our project, `LibraryServiceStubTest.java` utilizes a manual stub, `StubDataStore`, which implements the `DataStore` interface using an in-memory `HashMap`.

Listing 1: Stubbing for State Verification

```
1  // From LibraryServiceStubTest.java
2  StubDataStore stub = new StubDataStore();
3  Book b1 = new Book("x1", "Stubbed Book");
4  stub.addBook(b1); // Setup state
5  LibraryService lib = new LibraryService(stub);
6
7  // Execute and Assert State
8  boolean result = lib.issueBook(user, "x1");
9  assertTrue(result);
```

This approach creates purely isolated tests where no database is required. The test verifies that if the data store contains a specific book, the service can successfully issue it. The stub provides a "fixed response" [4].

4.2 Mocking for Behavior Verification Mocks allow us to verify that the SUT interacts correctly with its dependencies. This technique is vital for verifying side-effects, such as sending emails or updating a database, without actually performing them. In `LibraryServiceMockTest.java`, we used Mockito to verify the interaction protocol.

Listing 2: Mocking for Behavior Verification

```
1  // From LibraryServiceMockTest.java
```

```
2       DataStore mockStore = mock(
            DataStore.class);
3       when(mockStore.findBookByIsbn("111
            "))
4           .thenReturn(Optional.of(book))
                ;
5
6       lib.issueBook(user, "111");
7
8       // Verify Interaction
9       verify(mockStore).findBookByIsbn("
            111");
10      verify(mockStore).updateBook(book)
            ;
```

Here, the correctness of `issueBook` is defined not merely by the return value, but by the fact that it *called* the `updateBook` method on the store. This enforces the "Command-Query Separation" principle, ensuring the service fulfills its contract with the data layer.

4.3 Spies and Partial Mocking Our research highlighted a third category: **Spies**. As demonstrated in `LibraryServiceSpyTest.java`, a spy wraps a real object instance. This pattern proves useful when testing legacy code or when verification of a side effect on a real object is required without mocking its entire behavior.

Listing 3: Using Spies for Partial Mocking

```
1       // From LibraryServiceSpyTest.java
2       StubDataStore realStore = new
            StubDataStore();
3       DataStore spyStore = spy(realStore
            );
4
5       // Perform action
6       service.issueBook(user, "spy001");
7
8       // Verify interaction (Mock-like)
9       verify(spyStore).updateBook(book);
10
11      // AND assert real state change (
            Stub-like)
12      assertFalse(book.isAvailable());
```

Spies offer a hybrid approach, allowing the test to verify that specific methods were called while still retaining the original behavior of the underlying dependencies.

4.4 The Hidden Cost of Side Effects: File I/O Analysis A critical failing of the "Bad Design" implementation is the coupling of business logic with direct filesystem operations. This is most evident in the `generateReport()` method of `LibraryApp.java`. Unlike the "Good Design," which hides persistence behind a `DataStore` interface, the "Bad Design" instantiates a `FileWriter` directly within the method body.

Listing 4: Hard-coded I/O Dependency

```
1       // From LibraryApp.java (Bad
            Design)
2       static void generateReport() {
3           try {
4               // Direct dependency on
                    the file system
5               FileWriter writer = new
                    FileWriter("
                    library_report.txt");
6               writer.write("Library
                    Report\n");
7               // ... writing logic ...
8               writer.close();
9           } catch (IOException e) {
10              System.out.println("Error
                    generating report!");
11          }
12      }
```

This architectural flaw introduces the "Side-Effect Problem," where executing a test alters the persistent state of the environment. Consequently, the test suite assumes the heavy burden of cleanup. As shown in `LibraryAppTest.java`, the `tearDown()` method must explicitly delete the generated file to prevent state pollution between test runs.

Listing 5: Manual Resource Cleanup

```
1       // From LibraryAppTest.java
2       @AfterEach
3       void tearDown() {
4           // ... restore streams ...
5           try {
6               // Essential cleanup to
                    prevent test pollution
7               Files.deleteIfExists(Paths
                    .get("library_report.
                    txt"));
8           } catch (IOException e) {
9               // Silently failing
                    cleanup can lead to
                    flaky tests
10          }
11      }
```

This creates two significant risks:

(a) **Fragility:** If the test process crashes before `tearDown()` executes, the file remains on the disk, potentially causing subsequent test runs to fail or behave unpredictably.

(b) **Untestable Error States:** Because the `FileWriter` is instantiated locally using the `new` keyword, it is impossible to inject a mock that throws an `IOException`. Therefore, the catch block in `generateReport()`—which handles critical error reporting—remains unreachable and effectively untested.

In contrast, the "Good Design" avoids these pitfalls entirely by mocking the output mechanism, allowing tests to verify that a report *would* be generated without touching the physical disk.

5. **Results: Design Impact on Testability**

This section presents the empirical results of our comparative analysis. By attempting to write comprehensive test suites for both the *LibraryApp* (Bad Design) and `LibraryService` (Good Design), we quantified the impact of design decisions on test fragility and complexity.

5.1 The Cost of Bad Design Testing the procedural `LibraryApp` proved significantly more difficult and fragile due to tight coupling and global state. Our analysis of `LibraryAppTest.java` reveals several critical issues:

(a) **State Pollution:** Because the application relies on `static` collections for storage (`books` and `availability`), tests interfere with each other's execution environment. We were forced to manually clear these collections in a `@BeforeEach` block. If a single test fails to clean up properly, it cascades failures throughout the entire suite, making debugging non-deterministic.

(b) **I/O Redirection:** To test the `borrowBook` method, which is tightly coupled to `System.in` and `System.out`, we had to hijack the standard input and output streams using `ByteArrayInputStream` and `ByteArrayOutputStream`. This results in extremely brittle tests; a trivial change in the console prompt (e.g., changing "Enter name:" to "Name:") causes the test assertions to fail.

(c) **Impossibility of Mocking:** Since `LibraryApp` instantiates `Scanner` and `FileWriter` internally (hard-coded dependencies), we cannot inject mocks. Consequently, we cannot simulate error conditions such as a disk write failure during report generation, leaving those catch blocks untested.

5.2 The Benefit of Good Design In contrast, the `LibraryService` implementation, which leverages Dependency Injection and Interfaces, allowed for sophisticated and robust testing scenarios:

(a) **Precise State Verification:** In `LibraryServiceArgumentCaptorTest.java`, we utilized Mockito's `ArgumentCaptor` to inspect the exact state of the `Book` object *before* it was passed to the persistence layer.

Listing 6: Using ArgumentCaptor to Verify Logic

```
1       // Capture the argument passed
            to updateBook
2       ArgumentCaptor<Book> captor =
3           ArgumentCaptor.forClass(Book
                .class);
4       verify(mockStore).updateBook(
            captor.capture());
5
6       // Assert logic occurred BEFORE
            persistence
7       assertFalse(captor.getValue().
            isAvailable());
```

This ensures that the business logic (marking the book as unavailable) executed correctly before the data was saved, a verification impossible in the "Bad" version without checking the global state map.

(b) **Notification Isolation:** `LibraryServiceNotificationTest.java` demonstrates the true power of mocking external systems. By mocking the `NotificationService`, we verified that an email *would* be sent without actually sending one.

- **Speed:** The test runs in milliseconds rather than seconds.
- **Reliability:** The test does not fail due to network outages or SMTP server downtime.
- **Safety:** No spam emails are sent to real users during the build process.

Our metrics indicate that the "Good" implementation achieved $> 90\%$ instruction coverage with minimal effort, whereas the "Bad" implementation struggled to reach 50% coverage because exception handling paths involving file I/O could not be triggered programmatically.

6. **Discussion**

Our qualitative experience and quantitative metrics confirm that testability is not an accidental by-product but a direct consequence of design quality. This section discusses the specific metrics observed and the broader implications for Object-Oriented software development.

6.1 Testability as a Design Feature The most significant finding is that adhering to OO principles like Dependency Inversion directly correlates with higher test coverage and lower test maintenance costs.

- **Metric - Code Coverage:** The `LibraryService` ("Good") implementation achieved $> 90\%$ instruction coverage with minimal effort. This is because every dependency could be controlled, allowing us to simulate edge cases. Conversely, the `LibraryApp` ("Bad") implementation struggled to reach 50% coverage. Critical error handling blocks, such as `IOException` catches in `generateReport`, remained untested because triggering these errors programmatically without mocking is nearly impossible.

- **Metric - Maintainability:** The tests for the "Good" implementation proved robust. Hypothetically changing the internal storage mechanism from an in-memory map to a SQL database would not break the `LibraryService` unit tests, as they rely strictly on the `DataStore` interface. In contrast, the tests for

the "Bad" implementation are fragile; a minor change in the console output string format (e.g., adding a newline) would require rewriting every assertion in `LibraryAppTest`.

6.2 Complementary Nature of Test Doubles The project confirms Meszaros's assertion that Mocks and Stubs are not interchangeable strategies but complementary tools.

- **Stubs for State:** As seen in `LibraryServiceStubTest`, stubs were superior for setting up the "pre-conditions" of a test. They allow the test to assert, "Given the database has this book, the service should return true."

- **Mocks for Interactions:** As seen in `LibraryServiceNotificationTest`, mocks were indispensable for verifying "post-conditions" that involve side effects. They allow the test to assert, "Given the book was borrowed, the notification service should have been called."

This distinction highlights that effective testing requires a mixed strategy: using stubs to drive the system into the correct state and mocks to verify that the system acted upon that state correctly.

6.3 Guidelines for Designing Testable Systems

Based on the friction encountered during the testing of `LibraryApp` and the relative ease of testing `LibraryService`, we propose the following four architectural guidelines for maintaining testability in Object-Oriented systems:

(a) **Avoid Mutable Static State:** As observed with the `static ArrayList` in the "Bad Design," global state introduces hidden dependencies between tests. To ensure test isolation and parallel execution safety, state should be encapsulated within instances, allowing the test runner to create a fresh "world" for every test case [3].

(b) **Abstract External Effects:** Any operation that touches the filesystem, network, or system clock should never be instantiated directly within a business method. As demonstrated by the `DataStore` interface, wrapping these volatile dependencies allows tests to substitute fast, deterministic stubs for slow, fragile I/O operations.

(c) **Prefer Constructor Injection:** Dependencies should be explicit. The "Good Design" declares dependencies in the constructor (`LibraryService(DataStore ds)`), making it impossible to instantiate the service in an invalid state. This pattern naturally supports the injection of mocks without relying on complex reflection or framework-specific hacks.

(d) **Separate Logic from Presentation:** The `LibraryApp` mixed calculation logic with `System.out.println`, forcing tests to parse string output to verify behavior. Decoupling these concerns—as seen with the `NotificationService`—allows business logic to be tested via simple boolean assertions, leaving the presentation layer to be tested separately or mocked entirely.

7. Conclusion

This project investigated the role of test doubles in Object-Oriented software testing, demonstrating that while Mocks and Stubs are powerful tools for isolation, their effectiveness is strictly limited by the underlying design of the system under test.

Our comparative analysis of two Library Management System implementations yielded the following key conclusions:

(a) **Design Dictates Testability:** The "Bad" implementation illustrated that procedural code disguised as objects—characterized by static state and tight coupling—renders standard mocking frameworks ineffective. In contrast, the "Good" implementation showed that adherence to principles like Dependency Injection and Interface Segregation is a prerequisite for effective unit testing.

(b) **Isolation requires Architecture:** We successfully transformed an untestable, fragile application into a robust, verifiable system not by writing better tests, but by refactoring the architecture to support isolation.

(c) **Mocks vs. Stubs:** We demonstrated that these tools are not interchangeable. Stubs are essential for simulating state to drive business logic (Input Verification), while Mocks are essential for verifying protocols and side effects (Output Verification).

Future work will expand this research to explore the impact of these patterns on integration testing and investigate the use of "Mock Objects" as a design discovery tool, where the difficulty of mocking a dependency highlights the need for a new abstraction.

References

[1] Spillner, A., & Linz, T. (2019). *Software Testing Foundations*. Rocky Nook.

[2] Bruntink, M., & van Deursen, A. (2006). An empirical study into class testability. *Information and Software Technology*, 48(9), 965–977.

[3] Meszaros, G. (2007). *xUnit Test Patterns: Refactoring Test Code*. Addison-Wesley.

[4] Fowler, M. (2007). Mocks Aren't Stubs. *martinfowler.com*.

[5] Freeman, S., & Pryce, N. (2009). *Growing Object-Oriented Software, Guided by Tests*. Addison-Wesley.

[6] Vokáč, M., et al. (2004). Testability Analysis of Object-Oriented Software. *Empirical Software Engineering*, 9(3), 201–229.

[7] Beck, K. (2002). *Test Driven Development: By Example*. Addison-Wesley.

[8] Mockito Documentation. (2025). https://site.mockito.org

[9] JUnit 5 Documentation. (2025). https://junit.org/junit5

[10] Oracle Java Documentation. (2025). https://docs.oracle.com/en/java/

[11] Gamma, E., Helm, R., Johnson, R., & Vlissides, J. (1994). *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley.