Improving Code Coverage with Mockito and PowerMock

Arnob Mallick (am94376), Jason Beere (beerejm), & Kevin John Cherian Joseph (kc43529)

Abstract— JUnit is the most popular way of unit testing Java applications, but how well does JUnit perform on multithreaded, distributed applications? Could adding Mockito and PowerMock to JUnit tests improve the code coverage? Our paper shows code coverage dramatically improves adding Mockito to JUnit tests for testing multithreaded distributed applications and refactoring the application is better than adding PowerMock to these tests.

Index Terms— Distributed applications, Multiprocessing/multiprogramming/multitasking, Test coverage of code, Testing tools

1 Introduction

THE scope of this project is to compare statement coverage utilizing three testing tools: JUnit, Mockito, and PowerMock. Statement coverage will be compared between tests written using JUnit by itself against tests written using JUnit in conjunction with Mockito and PowerMock. JUnit will be used to cover as many meaningful statements as possible and statement coverage will be measured. Next, Mockito and PowerMock will be added to improve the coverage and statement coverage will be measured again. The goal is to maximize statement coverage. Branch coverage will be measured as an additional measurement but is not the main goal of the project.

2 Background

2.1 JUnit

JUnit is a Java library for unit testing Java code [1]. JUnit tests evaluate Java code by instantiating classes, executing methods, and validating return values or object states against expected results. All this is done without any modification or manipulation of the Java code being tested.

2.2 Mockito

Mockito is a lightweight Java library mocking framework which works with JUnit [2]. Mockito helps mock class dependencies and lets developers focus on the class under test. The mockings are done prior to test execution, and verifications are done after. This makes the tests clean and readable compared to other mocking frameworks which follow a pattern of expect-run-verify. Mockito provides a variety of features to mock objects like injected mocks, partial mocks using spies, iterative mocking, and mocking using argument matchers.

2.3 PowerMock

PowerMock is a library which extends Mockito and provides more complex tools for testing via a custom classloader and bytecode manipulation [3]. PowerMock allows developers to mock static methods, constructors, private methods, and other constructs not mockable in Mockito.

2.4 Test Application

The application chosen for testing, called Chord with Fusion (CwF), was written for a distributed systems class. CwF is a peer-to-peer shared storage application which stores the names and lyrics of songs built using JavaSE 8. Each peer is a Java process acting as a Chord node or a Fusion node. Chord is a distributed computation protocol where all the nodes in the distributed system are arranged in a ring with strategic links around the network to achieve O(log n) lookup time [4]. Fusion is a fault tolerance technique to reduce the number of backups necessary in a distributed system by fusing data using Reed Solomon erasure codes from several nodes into an aggregated backup [5]. Table 1 lists the class details of the test application.

TABLE 1
Application Class Details

Class Name	Number of Lines	Number of Branches	Key Components
ChordNode	215	124	java.net.* , java.io.*
ChordServer	48	18	java.util.Scanner multithreading
DataTuple	14	0	
FaultRecovery AgentThread	164	64	java.net.* , java.io.*
FingerTable	36	26	java.net.*
FixFingersThread	13	0	java.util.Random,
FusedBackUpTable	67	26	java.net.* , java.io.*
FusionServer	33	10	java.net.* , java.io.* multithreading
FusionServer Thread	91	38	java.net.* , java.io.*
HashUtils	14	2	java.security.*
LyricsTable	58	16	synchronizer
MessageConstants	50	14	static methods
MessageHandler	76	54	java.net.*
MessageRouter	17	2	java.net.* , java.io.*
PingPredecessor Thread	11	6	java.io.*
ServerDataTable	93	38	java.net.*, java.nio.*
StabilizeThread	25	16	java.net.*

The application nodes use multithreading with TCP Sockets for communication and use synchronization to persist data. The use of `java.io.*` and `java.net.*` packages implies CwF has interactions with operating system calls. During initial development of this application, unit testing was not taken into consideration, hence the code was not catered to be unit testable. The class FixFingersThread uses non-deterministic logic to lookup Chord nodes using java.util.Random. These characteristics make CwF an ideal candidate to learn about Mockito and PowerMock.

2.5 Code Coverage

Code coverage tools are used in white box testing to measure the number of lines executed in a program and return statistics about the coverage. Most tools only count lines executed and skip whitespace lines, comments, bracket only lines, and definition lines (like '`int x;`'). Some tools can distinguish between partial execution and complete execution of lines. The code coverage tools output statistics like number of lines executed, total number of lines, and the percent of lines executed. Many tools inform the developers of the lines which are executed (e.g. lines 1, 3, 4, & 7) and not executed (e.g. 2, 5, & 6). Developers can write additional tests to target lines missed to improve code coverage.

3 The Project

Testing for code coverage is a very open ended task. The scope of this project is constrained in a couple key dimensions. First, the qualities of the application code selected are intentionally not ideal for testing. CwF is multithreaded, includes network communication, and was written hastily for a class assignment without any consideration for testability. Second, modifying the code for improved testability is outside of the scope of the project (like 'legacy code'), but the code still needs to be unit tested. With these restrictions in place, the goal is to compare the effectiveness of JUnit alone versus JUnit with the assistance of a variety of other testing libraries. The results will show the value of these additional libraries when testing difficult code.

3.1 Project setup

We decided to have separate packages for JUnit, Mockito and PowerMock test cases, allowing us to measure coverage independently for each unit test framework. These packages were included in the existing CwF application project [6]. Table 2 contains details of the unit testing libraries and dependencies used for this project.

TABLE 2
Unit Test Libraries & Dependencies Used

Library	Version
junit	4.12
mockito-core	3.0.0
powermock-api-mockito2	2.0.7
powermock-api-support	2.0.7
powermock-core	2.0.7
powermock-module-junit4	2.0.0
powermock-module-junit4-common	2.0.0
powermock-reflect	2.0.7
system-rules	1.19.0
javaassist	3.24.1-GA
objenesis	3.0.1
byte-buddy	1.9.10
byte-buddy-agent	1.9.10

3.2 JUnit

We wrote unit tests in three stages: first JUnit tests, next added Mockito tests, and finally added PowerMock tests. All 17 Java classes were analyzed and we began to write JUnit tests to achieve the highest amount of statement coverage, starting with the six classes best suited for JUnit tests like DataTuple, LyricsTable, and MessageConstants. The methods in these classes typically have primitive types, common classes, or nothing in the parameter lists and have minimal external dependencies. Once the team covered as much of the six classes with JUnit tests, we added the other 11 Java classes. These classes were much harder to achieve significant statement coverage. Some classes could not achieve meaningful coverage with JUnit alone like FusionServerThread, FusionServer, and FaultRecoveryAgentThread. JUnit tests for these three classes were omitted rather than covering a few lines. JUnit tests were written for the rest of the eight classes to maximize statement coverage, as if Mockito and PowerMock weren't going to be used. Once this was completed, we added in Mockito.

3.3 Mockito

Mockito provided a variety of functions to stub objects. For our test application, we used various mock functions to mock TCP Sockets and I/O operations. Mockito helped us focus on the class being tested, while mocking other class dependencies not under test. Figure 1 is a sample test case class containing some of the key Mockito features which helped improve code coverage in CwF. Unit tests with Mockito are run using MockitoJUnitRunner. This class is responsible to scan a Mockito unit test class, create stubs for mocked objects, execute test cases, and finally verify behaviour of mocked objects. CwF persists details of various nodes in a P2P network. Mockito helped mock this data and kept our unit test clean and readable compared to JUnit test cases using the `when().thenReturn()` function which mocks method calls to class dependencies. Line #3 in Figure 1 provides a sample syntax for this functionality.

Using out of the box ArgumentMatchers like `anyInt()`, for e.g. Line#4 in Figure 3 helped create a more generalized mocking of method calls. CwF used a lot of `while(true)` loops for heartbeat checks among nodes, Mockito's iterative style stubbing helped respond with different values for each iteration or throw an exception (e.g. Line#5 in Figure 1).

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

@RunWith(MockitoJUnitRunner.class)
public class SampleTest(){

#1 @Mock private List<String> mockObj;

#2 @Test public void testCase1(){

#3   when(mockObj.get(0))
       .thenReturn("someName");
#4   when(mockObj.get(anyInt()))
       .thenReturn("someValue");
#5   when(mockObj.get(1))
       .thenReturn("someOne")
       .thenThrow(new Exception());
#6   List<String> mockObj2 =
       spy(new ArrayList());

#7   verify(mockObj, atMost(3))
       .get(anyInt());
#8   verifyZeroInteractions(mockObj2);

   }
}
```

Fig. 1. Sample Mockito Test Case

In some instances we had to use a mocked object but still call a real method implementation to get better coverage, Mockito's `spy()` function provided this capability. Verification of interactions with mocked methods helped assert the usage of a mocked object, like the number of times a method was called in a mocked object (e.g. in Line# 7, 8 in Figure 1).

3.4 Power Mock

A common design pattern used to simplify the task of injecting mock instances is to supply references to these mock objects in the constructor of the class being tested. CwF rarely uses this pattern. Rather instances of classes were almost always created inline with the logic (e.g. `new ArrayList<String>();`). PowerMock allows the test runner to detect when the 'new' keyword is used (a constructor is called) and inject a Mockito mock object in place of the constructor. Figure 2 demonstrates a sample case where two constructors are mocked.

```
public ChordNode(InetSocketAddress address) {
    m_address = address;
    m_stabilizer = new StabilizeThread(this);
    m_fixFingers = new FixFingersThread(this);
}

public class ChordNodePowerMockTest() {

  @Test public void testCase1() {

    // setup mocks
    StabilizeThread stMock =
      mock(StabilizeThread.class);
    PowerMockito
      .whenNew(StabilizeThread.class)
      .withAnyArguments()
      .thenReturn(stMock);

    FixFingersThread fftMock =
      mock(FixFingersThread.class);
    PowerMockito
      .whenNew(FixFingersThread.class)
      .withAnyArguments()
      .thenReturn(fftMock);

    // execute test

    // assertions
  }
}
```

Fig. 2. Sample PowerMock constructor mock injection

PowerMock also handled mocking Socket communications well. Typical socket-based Java code will instantiate Socket instances in-line, read or write from the Socket, and then close it. Or, in the case of long lasting Sockets, the instance will be created and a reference to the Socket will be shared amongst several parts of the code. For a distributed system making frequent function-like requests through network communication, PowerMock grants the ability to mock Socket communication the same way one can mock method calls. Figure 3 demonstrates how to mock the InputStream of a Socket.

```
Socket soMock = mock(Socket.class);
PowerMockito
  .whenNew(Socket.class)
  .withAnyArguments()
  .thenReturn(soMock);
when(soMock.getOutputStream())
  .thenReturn(new ByteArrayOutputStream());
InputStream isMock01 = new
  ByteArrayInputStream("expected".getBytes());
when(soMock.getInputStream())
  .thenReturn(isMock01);
```

Fig. 3. Sample PowerMock constructor mock injection

3.5 System Rules

We encountered several cases where a code path ends in a call to `System.exit()`. When `System.exit()` is encountered during a test, by default the entire JVM is shut down. This includes your test runner, which implies no test results. We had the option to either ignore all code paths containing `System.exit()` and forego statement coverage on lines covered in these paths, or find a workaround. The System Rules library was a perfect fit for our problem [7]. The library provides a convenient way to write a test expecting a call to `System.exit()`, detects the call, and terminates the test without shutting down the JVM. The library has many other functions supporting other aspects of the System class which we did not explore.

3.6 Code Coverage Tools

Many IDEs have code coverage tools as standard or available as a plugin. IntelliJ and Eclipse with EclEmma were used in this project. IntelliJ's was used to measure statement coverage and Eclipse with EclEmma was used to measure branch coverage. Statement coverage for tests

using PowerMock only worked in IntelliJ. Branch coverage for tests using PowerMock was not possible to measure with either tool. IntelliJ and EclEmma count branches differently so no results were compared across different code coverage tools.

4 Results & Analysis

JUnit with Mockito, PowerMock, and System Rules dramatically improved the statement coverage over JUnit alone. JUnit tests covered 53% of all statements (576 out of 1087). Adding Mockito, PowerMock, and System Rules increased coverage to 74% of all statements (808 out of 1087), an over 40% increase in coverage. The following chart shows a breakdown by Java class. The green bars show statement coverage with JUnit alone, the blue bars show how many additional statements are covered by adding the additional testing libraries. The red shows the number lines still uncovered.

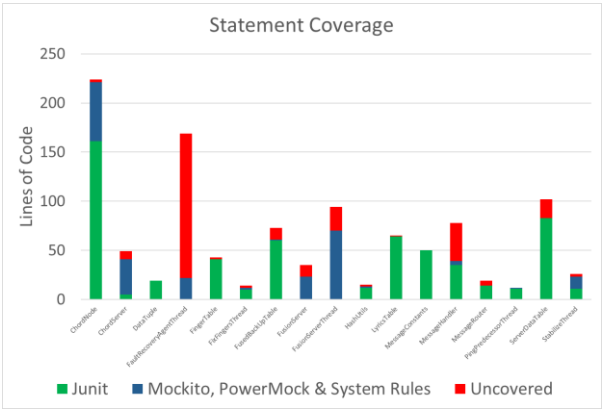

Fig. 4. Statement Coverage using JUnit, Mockito, PowerMock, & System Rules

Some classes were not able to be meaningfully covered, so JUnit tests were skipped leaving many lines uncovered. ChordServer, FaultRecoveryAgentThread, and FusionServerThread are some examples. Coverage of FaultRecoveryAgentThread, which extends the Thread class, was difficult to test because the class contains many long private methods that required deep stubbing of data. Mockito was able to fully cover the constructor and mostly cover the overridden run method. Mocking private methods is possible using PowerMock but is not easy. Overall Mockito and PowerMock really improved statement coverage.

Due to time constraints, branch coverage was not a focus when writing the tests for CwF, however we still measured branch coverage as a sanity check. PowerMock was incompatible with our coverage tools, so the results were measured only using JUnit, Mockito, and System Rules. Adding Mockito tests improved our branch coverage. JUnit alone had 47% branch coverage (215 out of 454 branches) while JUnit, Mockito and System Rules had 57% coverage (261 out of 454 branches). Figure 5 shows the branch coverage metrics by Java class. The green bar is branch coverage for JUnit tests, the blue is the additional branches covered when adding Mockito and System Rules to JUnit, and the red bars are the uncovered branches in the code. Additionally, if we found a way to measure branch coverage with PowerMock tests included, the results would have been much better.

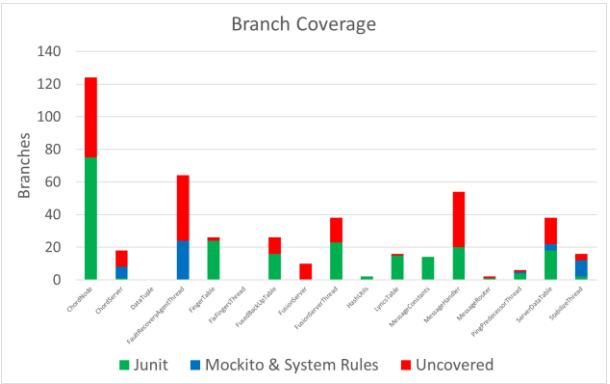

Fig. 5. Branch Coverage using JUnit, Mockito & System Rules

Some classes did not have any branches, like DataTuple and FixFingerThread. FusionServer, ChordNode, and ChordServer tests heavily relied on PowerMock. Both IntelliJ and Eclipse with EclEmma failed during the PowerMock tests, showing no coverage (neither statement nor branch). Regardless, we were able to demonstrate branch coverage can be improved using Mockito and System Rules.

5 Lessons Learned

Our team learned several valuable lessons about software development, testing, code coverage, multi-threaded applications, distributed applications, JUnit, Mockito, and PowerMock.

Mockito and PowerMock initially seemed like a silver bullet - simple tests with the promise of 100 percent statement coverage. PowerMock can access private classes and static classes and Mockito can target a single class for testing, even when the class requires other classes. Mockito and PowerMock are only tools in a developer's suitcase and do not solve all problems. Many of the problems PowerMock can solve are more easily solved by refactoring the code for testing. For instance, by wrapping `java.net` or `java.io` objects & 3rd-Party classes; or refactoring `while (true)` loops. PowerMock can mock private methods in public classes, however not easily. PowerMock cannot mock public methods in private inner classes nor package variables in external packages. The need to mock private methods to achieve statement coverage is generally indicative of poor design.

Design of classes and constructors is important when writing applications to make unit testing easier. `System.exit()` should be avoided as this shuts down the JVM and stops running any pending tests. The System Rules Library can be used to get around this problem, but a better way is to throw an exception instead of using a `System.exit()`. One should design classes and constructors

while keeping dependency injection for testing in mind. Dependency injection occurs when objects receive other objects, for instance the MessageRouter constructor needs a ChordNode object. A ChordNode constructor needs a ServerDataTable object and a InetSocketAddress object. Can MessageRouter be unit tested in this setup or is the base test an integration test? Does a failure of a MessageRouter test mean a problem with MessageRouter as long as the ChordNode and ServerDataTable tests pass? Dependency injection is often required in complex applications, but one should design classes in such a way that libraries like PowerMock are not needed.

Sockets can be tested using JUnit, however tricks need to be employed to get meaningful statement coverage. Adding PowerMock to mock sockets makes things much easier. Mock network communications satisfied a missing link necessary to effectively test a distributed system.

IntelliJ's & EclEmma's code coverage tools had problems with multi-threaded applications when statements took a long time to execute. We noticed this on code following a statement to access a SHA-1 (Secure Hash Algorithms-1) library. When we started the thread with `Thread.start()`, code coverage stopped during the attempt to get the SHA-1 instance. Code coverage measurement stopped after the thread was started. To get around this, we added print statements to the code and started the thread using `Thread.run()`. The thread kept running until the infinite loop was manually stopped. The code coverage tool did not return any measurements, but we were able to count which lines were executed.

IntelliJ and EclEmma measure branch coverage differently. An example is an 'if' statement with no 'else if' or 'else' branches. EclEmma counts two branches: `if (true)` as one branch and `if (false)` as a second branch. IntelliJ only counts one branch: `if (true)`. The implied `if (false)` path is not counted as a branch since the `if (false)` path does not contain any statements. EclEmma could not do statement coverage for PowerMock tests and neither coverage tool could do branch coverage for PowerMock tests. IntelliJ went into an infinite loop, while EclEmma measured the branch coverage as zero for these classes.

6 Conclusion

JUnit performed well considering the project being tested was a multithreaded distributed application - an application better tested during system integration. Utilizing JUnit with sockets can be done, but requires some work and tricks to get better code coverage. Mockito makes unit testing with sockets much easier. Test setup was much easier with Mockito than with JUnit alone - especially with classes with lots of dependency injection.

JUnit tests covered 53% statement coverage and 47% branch coverage. Mockito dramatically improved statement and branch coverage over JUnit. Mockito and PowerMock with JUnit covered 74% of the statements and 57% of the branches (PowerMock not included for branches). Adding PowerMock improved coverage, but took much longer to write tests. Using JUnit with Mockito seems like the right balance. Adding PowerMock helps overcome bad code design, but the time invested working with PowerMock is better spent refactoring the code to improve testability.

Code coverage tools measure coverage differently, sometimes don't measure all the code executed, and sometimes have incompatibilities with libraries. Code coverage tool statistics are usually treated as exact measurements, when really the statistics are approximations.

This exercise was a great learning experience. We were happy with what we could achieve with JUnit, Mockito, PowerMock and System Rules.

7 Future Work

Follow-on work can be done with this project. CwF can be refactored to improve unit testing with JUnit and Mockito. Best practices can be given about how to best refactor code for testing multithreaded distributed applications. Unit tests for other projects, including multithreaded and distributed projects, can be written using JUnit and Mockito. Java Pathfinder (JPF) or the ideas behind JPF can improve unit tests and code coverage for multithreaded distributed applications.

Additional work can be done on code coverage tools. We can develop code coverage tools that play nicely with multithreaded applications. EclEmma and IntelliJ's code coverage tools can be made to work with PowerMock (IntelliJ does not do branch coverage with PowerMock and EclEmma does neither statement nor branch coverage with PowerMock).

Mockito and PowerMock can be used for TDD. Work can be done to develop an application from scratch doing TDD using Mockito and PowerMock. The process can be documented and lessons learned can be explored.

References

[1] JUnit, "JUnit – About," 1 January 2020. [Online]. Available: https://junit.org/junit4/. [Accessed 7 May 2020].

[2] Szczepan Faber and friends, "Mockito framework site," [Online]. Available: https://site.mockito.org/. [Accessed 7 May 2020].

[3] PowerMock, "powermock/powermock: PowerMock is a Java framework that allows you to unit test code normally regarded as untestable.," 30 March 2020. [Online]. Available: https://github.com/powermock/powermock. [Accessed 7 May 2020].

[4] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek and H. Balakrishnan, "Chord: A Scalable Peer-to-peer Lookup Service for Internet Applications," August 2001. [Online]. Available: http://users.ece.utexas.edu/~garg/resources/chord_sigcomm.pdf. [Accessed 7 May 2020].

[5] B. Balasubramanian and V. K. Garg, "A Fusion-based Approach for Handling Multiple Faults in Distributed Systems," [Online]. Available: http://users.ece.utexas.edu/~garg/resources/FusionICDCS2011.pdf. [Accessed 7 May 2020].

[6] J. Beere, K. Cherian and A. Mallick, "kevincherianjoe/s20_software_testing_project: Spring 2020 Software Testing Final Project," 30 April 2020. [Online]. Available: https://github.com/kevincherianjoe/s20_software_testing_project. [Accessed 7 May 2020].

[7] M. Philipp and S. Birkner, "System Rules," [Online]. Available: https://stefanbirkner.github.io/system-rules/. [Accessed 7 May 2020].