Software Testing - A3

Ingmar Falk

January 3, 2024

1 Introduction

In this report, I will analyze the paper *To Mock or Not To Mock? - An Empirical Study on Mocking Practices*, by the *Delft University of Technology*. To collect data for this paper, the researchers conducted a survey among developers from a handful of projects as well as created a tool to collect statistics about the usage of mocking in the tests of said projects. Among those projects are *SonarQube*, the *Spring Framework*, *VRaptor* and *Alura*. All of these projects are written in Java and use javas most popular mocking framework *Mockito*.

2 Methodology

2.1 Data Collection

The goal of this paper was to investigate how and why developers make use of mock objects when writing tests. There are 3 main questions, that were applied to all projects:

- What test dependencies do developers mock?
- Why do developers decide to (not) mock specific dependencies?
- What are the main challenges experienced with testing using mocks?

To answer these questions the team created a tool called *MockExtracor*. Using this tool, they extracted data about all of the classes, lines of code, tests with/without mocks and other information useful for later analysis.

Project	# of classes	LOC	# of test units	# of test units with mock	# of mocked dependencies	# of not mocked dependencies	Sample size of mocked (CL=95%)	Sample size of not mocked (CL=95%)
Sonarqube	5,771	701k	2,034	652	1,411	12,136	302	372
Spring Framework	6561	997k	2,020	299	670	21,098	244	377
VRaptor	551	45k	126	80	258	1,075	155	283
Alura	1,009	75k	239	91	229	1,436	143	302
Total	13.892	1.818k	4.419	1.122	2,568	35,745	844	1,334

Figure 1: Image taken from the paper, showing the data collected from the projects

2.2 Analysis

Due to the size of the projects and the amount of data collected, the team decided to only use a random sample of the original data. Defining an automated analysis process would have been extremely challenging, so the researchers went with a manual approach. Each of the researchers was responsible for two projects, which they then analyzed by hand and entered the data into a spreadsheet. One of the main objectives of this analysis was to assign broad categories to each analyzed unit.

2.3 Categorization

After assigning a category to each unit in the analysis stage, it was now time to merge the categories to reduce the number from 116 to a more manageable number. This was done by grouping similar categories together and then merging them into one. In total, the researchers ended up with 7 categories, which are: *Database* , *Web Service* , *External Dependencies* , *Domain Object* , *Java Libraries* , *Test Support*.

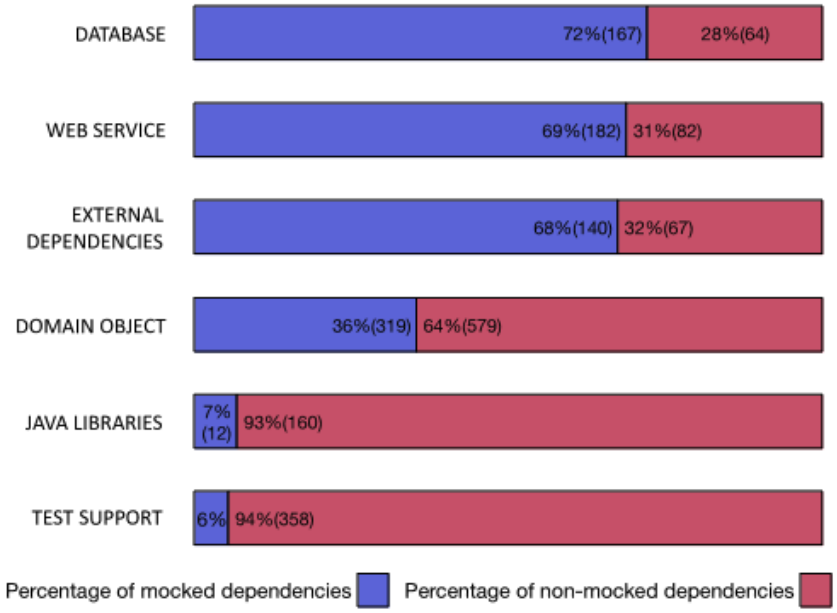

Figure 2: Image taken from the paper, showing the final categories

2.4 Interviews

To get a better understanding of the data, the researchers conducted interviews with developers from the projects and inquired about their testing practices and the usage of mocks. The interviews were semi-structured, meaning that the researchers had a set of questions prepared, but were also free to ask follow-up questions. The prepared questions were based on the data collected in the previous steps. The interviewers made sure to ask questions about the interviewee's reasoning for (not) mocking a certain category of dependencies, exceptions and the pros and cons regarding that category.

2.5 Survey

The survey was also based on all previous stages and aimed to expand upon the concepts that emerged from them. The questions were all based on the previously discovered data. It consisted of 4 main parts:

- Part 1. Participants experience in Software Development and Mocking
- Part 2. How often do participants make use of mocks in each found category
- Part 3. How often do participants make use of mocks in specific situations (complex, coupled, slow, etc.)
- Part 4. Challenges of mocking (Open-ended and Optional)

They received 105 responses. Experience:

- 21 % – 1-5 years
- 64 % – 6-15 years
- 15 % – 16+ years

With the languages used:

- 24 % – Java
- 18 % – C#
- 19 % – JavaScript
- Other

3 Threats to Validity

3.1 Construct Validity

The study acknowledges potential issues with the research instruments, particularly the use of MOCKEXTRACTOR for collecting dependencies through static code analysis. This tool may not capture dynamic behaviors, leading to some dependencies being incorrectly categorized as "non-mocked." The study mitigates this by using random samples in manual analysis and manually inspecting MOCKEXTRACTOR results in 100 test units, which showed no issues in those cases. Agreement among researchers analyzing the data was 89%.

3.2 Internal Validity

Concerns are raised about factors not considered that could affect the variables and relationships being investigated. The study interviews developers to understand why certain dependencies are mocked or not, but a single developer may not have complete knowledge of all implementation decisions. To address this, the study presents data from the first research question first and avoids asking questions about overall categories coined in RQ1. The potential influence of other factors, like current literature and other projects, is also acknowledged, and interviewees are reminded to focus on their specific project.

3.3 External Validity

There are concerns about the generalizability of the results due to a small sample size, consisting of four Java systems (one of them being closed source). To mitigate this, the opinions of 105 developers from various projects are collected. The study suggests that further research in different projects and programming languages should be conducted. Additionally, there may be self-selection bias in the survey respondents and the study lacks information about the nature of the the overall population that responded to the survey.

4 Results

4.1 RQ1: What test dependencies do developers mock?

The categories explained are as follows:

- Domain object: Classes that contain system rules, often dependent on other domain objects and not external resources.
- Database: Classes interacting with external databases, either as external libraries or as classes depending on such libraries.
- Native Java libraries: Java's built-in libraries, such as Java I/O and Java Util classes.
- Web Service: Classes performing HTTP actions, similar to databases in terms of external dependencies.
- External dependency: Libraries or classes using external libraries outside the project.
- Test support: Classes that support testing, like fake domain objects, test data builders, and web services for testing.
- Unresolved: Dependencies that couldn't be resolved, often from sub-modules with unavailable source code.

The percentage of mocked dependencies varies across projects, with databases and domain objects showing some differences. For instance, Sonarqube mocks around 47% of domain object dependencies, while other projects mock them around 30% of the time. For databases, SonarQube and Alura mock around 60% of the dependencies, while Spring mocks 94% of them. Web services and databases are the most frequently mocked dependencies, while domain objects exhibit a less clear trend with around 36% of them being mocked Test support and Java libraries are almost never mocked, which is expected since the former includes classes specifically created to support testing, and the latter comprises Java's core libraries that are typically reliable and don't require mocking

4.2 RQ2: Why do developers decide to (not) mock specific dependencies?

- Mocks are commonly used when the concrete implementation of a dependency is not simple. Dependencies that are highly coupled, complex to set up, contain complex code, perform slow tasks, or rely on external resources (e.g., databases, web services, or external libraries) are often candidates for mocking.
- Developers prefer not to use mocks when testing the integration with an external dependency itself, as they want to ensure that the integration works correctly. However, they do use mocks when unit testing a class that depends on a class dealing with external resources to isolate the unit being tested.
- Interfaces are often mocked instead of specific implementations to avoid relying on a particular implementation.
- Domain objects are usually not mocked, especially when they are simple and don't deal with external resources. However, complex domain objects or highly coupled ones may be mocked.
- Native Java objects and libraries, which are easy to instantiate with desired values, are typically not mocked. However, there may be exceptions, such as dealing with complex file operations.
- Database, web services, and external dependencies are considered good candidates for mocking due to their complexity and slow setup. Developers commonly mock these dependencies when testing.
- The decision to mock or not mock depends on whether the focus of testing is on the integration itself. When integration testing is required, mocks are less likely to be used, but for unit testing, mocks are preferred for certain types of dependencies.
- Survey respondents generally align with the interviewees' perspectives, indicating a preference for mocking web services, external dependencies, and databases when not testing integration

4.3 RQ3: What are the main challenges experienced with testing using mocks?

- Dealing with Coupling: Mocking practices can introduce coupling between tests and production code. Additionally, high coupling among production classes can make mocking challenging, especially when proper decoupling and dependency isolation haven't been implemented.

- Getting Started with Mocks: Mocks may be a new concept for some developers, requiring experienced developers to teach junior developers. Some developers may struggle to understand and use mock objects effectively.

- Mocking in Legacy Systems: Testing single units in legacy systems may require extensive mocking, sometimes even mocking the entire system. Tools like PowerMock may be necessary for classes not designed for testability. In some cases, mocking is the only way to perform unit testing in legacy systems with poor architecture.

- Non-Testable/Hard-to-Test Classes: Various technical details can hinder the use of mock objects, such as static methods in Java, file uploads in PHP, interfaces in dynamic languages, and specific language features like LINQ in C#.

- Relationship Between Mocks and Code Quality: Excessive use of mocks is seen as an indicator of poorly engineered code. Mocks can hide deeper problems in the system's design, such as highly coupled and complex classes. Well-structured production code should ideally require fewer mocks. Using too many mocks may lead to test readability and maintenance issues.

- Unstable Dependencies: Maintaining the behavior of mock objects compatible with the behavior of the original class can be challenging, especially when the production class frequently changes. Poorly designed or highly coupled production classes can make mock objects more unstable and prone to change.