# A Quantitative and Qualitative Empirical Evaluation of a Test Refactoring Tool

by

Aliasghar Iman

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Applied Science
in
Electrical and Computer Engineering

Waterloo, Ontario, Canada, 2021

## Abstract

Reducing the gap between what practitioners want vs. what researchers assume they want is one of the vital challenges in software projects. When it comes to software tools, many people develop tools, but only some tools end up being useful to developers. Since this problem is more attached to short-term industry imperatives, several new software development practices and methodologies have been established to get frequent feedback from potential clients and adjust the project based on their feedback, to address this issue. We thought that agile-style techniques as in industry could be transplanted to evaluate the usefulness of tools in software engineering and programming languages research systems.

JTestParametrizer is an existing refactoring tool that aims to automatically refactor the method-scope renamed clones in test suites. This research aimed to use different practices and methodologies formulated on the aforementioned concept to evaluate, modify, and extend the JTestParametrizer tool. First, we ran the tool on 18 benchmarks that we picked for our benchmark suite. Then by studying the feedback that we got from quantitative results, we detected and fixed some conceptual and non-conceptual bugs in the tool. Next, we developed questionnaires and used manual assessments and pull requests submitted to developers to solicit feedback on the quality of the tool. Then after studying this feedback, we modified the tool and added new configurations to adjust the tool to the feedback.

Furthermore, we used a technique similar to the Minimum Viable Product technique in industry to collect feedback on potential features for JTestParametrizer before actually implementing them. We did this by manually applying the effect of different potential features that the tool could have on cases used for pull requests. By studying feedback from these manually modified pull requests, we determined the factors that the practitioners care about the most in the context of refactoring unit tests, allowing us to formulate and support hypotheses about suitable features to implement in JTestParametrizer next.

## Acknowledgements

I would like to thank my supervisor, Professor Patrick Lam, for his priceless guidance and assistance throughout my program. It was a pleasure to learn from him. His patience and constant support were crucial factors that helped me advance my work during an unprecedented time of a pandemic. I like to thank Professor Derek Rayside and Professor Mahesh Tripunitara for reading my thesis and providing me with invaluable insights. Finally, I want to thank my family for their unconditional love and support.

## Dedication

I dedicate this thesis to my family.

# Table of Contents

# List of Figures

# List of Tables

# Chapter 1

# Introduction

Researchers develop many tools, but only some of those tools end up being helpful to developers. This could be due to many reasons, but one of the important ones is that the tool does not solve a problem developers have.

Particularly for refactoring tools, Eilertsen et al. [17] conducted a study to determine how to enhance the usability of these tools. The problem was that even though developers have access to various refactoring tools, most developers avoid using most of these tools. The identified reasons for this disuse include a lack of awareness by developers, a lack of predictability of the tools, and a lack of need for the tools. Eilertsen et al. studied the experiences of 17 developers working on three change tasks designed to be amendable to the use of refactoring tools. Based on this lab study, they provided some insights into what developers need from a refactoring tool.

Furthermore, Sadowski et al. [25] expressed lessons from building static analysis tools at Google and mentioned reasons engineers do not always use these tools. Some of these reasons include the tool not being trustworthy, not being integrated, and issues identified by the tool being too expensive to fix. Sadowski et al. also provided some critical insights, including that “static analysis authors should focus on the developer and listen to their feedback”, “careful developer workflow integration is key for static analysis tool adoption”, and “static analysis tools can scale by crowdsourcing analysis development”.

Determining what practitioners want vs. what researchers assume they want is a crucial challenge of Programming Languages / Software Engineering research. This problem can happen in both research and industry, but typically, the consequences are more apparent in the industry since this problem is more attached to short-term industry imperatives, whereas in research, how authors represent their work and the factors that they learn can have higher importance. Lack of evaluation and feedback is a factor that leads to this issue. If we seek feedback from practitioners on the quality of our work throughout our research and adjust our work based on that, there will be a lower chance of our final product being something that no one would want to use.

Over the years, there have been plenty of new software development practices and methodologies to mitigate the "something developers assumed that the customer wants vs. what customers wanted" issue. Many newer software development practices (such as Agile) are based on getting frequent feedback from potential customers, responding to changes over following a plan, and adjusting the project based on their feedback. Even though the better response to this problem in the industry compared to research could be because this problem is more tied to short-term industry imperatives, we could still benefit from using agile-style techniques to evaluate the usefulness of tools in software engineering and programming languages research systems.

This work aimed to evaluate, modify, and extend an existing automatic test refactoring tool called JTestParametrizer based on the mentioned concepts. JTestParametrizer [29] is a refactoring tool that aims to refactor the method-scope renamed clones in test suites automatically. The idea behind this tool is that when developers write unit tests, in order to increase the coverage criteria, they usually tend to create multiple test methods that are very similar and only differ in tiny details. This will lead to having duplicated code in tests. Now, if we can automatically deduplicate the tests, we are decreasing the code repetition in the test, which, in theory, will lead to higher maintenance and higher overall quality of the test code.

However, the major problem with the JTestParametrizer tool was that even though, in theory, it should have increased the overall quality of the test code, there was no proof or empirical evidence to show JTestParametrizer's effectiveness. The main goal of this research was to provide that empirical evidence that showed the JTestParametrizer's effectiveness, and in doing so, modify and extend the tool based on the feedback that we got to enhance the overall quality of the tool.

We ran the tool on 18 open-source benchmark projects, and based on the quantitative feedback, we fixed some conceptual and non-conceptual bugs in the tool. Then we tried different processes such as questionnaires and manual assessment of the refactoring cases to provide quantitative feedback on the tool. After analyzing the feedback, we figured that one of the three parametrization techniques in JTestParametrizer decreases the overall quality of the refactored test methods. We added a configuration to the tool to discard that specific parametrization technique, and then we ran the tool with the new configuration on all the 18 benchmarks again and studied the results. Based on this new feedback, we determined that even after using the new configuration, there is no significant enhancement in the overall quality of the refactored tests.

Furthermore, we submitted vetted proposed refactorings upstream using Pull Requests to get the developer's feedback on the quality of the tool. First, we submitted one Pull Request only containing the refactoring cases created by running the tool on Jimfs benchmark. This confirmed our previous assessment that the current version of the JTestParametrizer tool was not significantly enhancing the overall quality of the refactored tests.

Then, we used a technique similar to the Minimum Viable Product technique to determine which potential feature of the JTestParametrizer tool will have the best feedback before implementing that feature. We did this by manually implementing the effect of each potential feature to the refactored test methods that we selected for a Pull Request. After investigating the feedback for these Pull Requests, we determined the best potential feature that will provide the highest enhancement to the quality of the JTestParametrizer tool. We also found some new configurations for the tool that might positively affect the overall quality.

Throughout this work, we also learned some heuristics for determining which benchmarks would suit which processes the best. We recognized the importance of "maintenance mode", "estimated response time", and "familiarity with the benchmark's domain" factors when choosing a benchmark to get feedback using the Pull Requests process. We compared three processes (Questionnaire, Manual Quality Evaluation, and Submitting Pull Requests) for getting qualitative feedback. We discussed the priority of qualitative feedback vs. quantitative feedback. Furthermore, we learned some critical factors developers care about when refactoring unit tests, such as the higher importance of understandability and readability in tests vs. production code and the lower importance of code repetition in tests vs. production code.

# Chapter 2

# Reproducible Research

Researchers have advocated for transparency and reproducibility as essential tools used in modern academic literature for establishing the validity and reliability of research efforts. The challenges in performing reproducible research range from technical issues concerning input data, under-specification in methodology or metrics, obfuscated or unavailable codebases, and selective or exaggerated reporting [3] to the general culture of the computer science community, which typically has a lower than ideal emphasis on reproducibility, both in the review process and during the development of scientific publications [4].

Research in Programming Languages has been notoriously susceptible to such challenges as reproducing the results of one experiment is also heavily affected by the environment and software specifications of the set-up used by the authors. This is closely related to the age-old problem of cross-platform software development and deployment, which has shaped many trends in the industry. For instance, the Write Once, Run Anywhere principle in Java and the recent explosion of scripting languages in popularity have all been more or less related to the need for transferability of software projects from one platform to the other.

This problem is exacerbated in complex systems where diverse artifacts are brought into play to carry out an objective. Slight differences in OS kernels, distributions, package versions, configurations, and file systems can hinder the successful redeployment of a software set-up.

Guo et al. [19] have approached this problem by introducing a packaging framework called CDE which recreates the file system tree of the source environment accurately and ships the code along with its input data to an arbitrary environment by creating a fake root containing the mentioned artifacts in the destination machine. While the authors' idea of shipping the entire environment as an apparatus for reproducible software runs is interesting, their focus is entirely on the file system. In contrast, many different factors such as the kernel, pre-loaded system object files, host specs, networks, shared resources can play a role in the successful reproduction of the software execution.

This strategy has evolved into modern containerization technologies such as Docker and LXC, which make use of various features in the Linux kernel (e.g., namespaces, Cgroups, Netfilter, AppArmor, Netlink) to replicate the desired environment as accurately as possible in isolation from other containers and the host operating system. Thus, containers provide a comprehensive framework for creating the desired environment in machines with different OS distributions and installed packages.

This has led many researchers to advocate for containers as a solution for reproducible research. For instance, Boettiger et al. [12] have proposed Docker as such a solution while defining best practices in doing so, e.g., using the containers during development, test cases, and checks, using Dockerfiles instead of manual instructions. Virtualization technologies, in general, can play an essential role in future research by providing a snapshot of the testing environment and the evaluation results, making a robust case for an experiment.

Without such arrangements, reproducing studies on programming languages and software evaluations can often be challenging. For instance, in a comprehensive technical report, Berger et al. [7] follow the footsteps of the authors in [24] along with their GitHub repository, documenting the results and cross-checking the claims. In this effort, reproducing some of the case studies has proven technically complex, with many steps in between. While most claims in [24] have been convincingly proven, some results seem to be contradicting the authors' claims.

Having a reliable method and baseline for comparing results in such measurements is of the highest importance for the future of this research area. With a combination of the mentioned methods, we expect the openness, transparency, and reliability of such efforts to be enhanced. Contradictory measurements can thus be investigated much more effectively as the testing environment along with its input data would be effectively specified and reproduced easily by consequent research projects building upon the same methods.

## 2.1 Reproducibility Challenges Related to This Work

The first step to empirically evaluate the JTestParametrizer tool was to run the tool on the selected benchmarks (see Chapter 4). I included all five benchmarks Zhao used to evaluate the JTestParametrizer tool [29] in the benchmark suite for this work.

I used the exact version of those benchmarks that Zhao used since the git commit for those versions was documented. I also used the exact version of the JTestParametrizet tool. However, I could not reproduce the same experiment as Zhao's since the JTestParametrizer tool takes an XLS file, including all the potential nominees for refactoring, as an input (see Section 4.4), and I did not have all the information to create the exact XLS files Zhao used. To create these XLS files, we both used Deckard's clone detection ability to create some cluster files and then ran a python script on the cluster files to generate the XLS files. However, Deckard's clone detection ability takes three arguments to operate, and I could not deduce which three numbers Zhao used for his work. Hence, I could not reproduce his XLS files. Consequently, I got different quantitative findings for those five benchmarks as I encountered several compile errors when building the refactored benchmarks (after running the JTestParametrizer tool), while Zhao stated that he did not encounter any compile errors on those five benchmarks.

Another crucial challenge for the reproducibility of this work is that JTestParametrizer is an Eclipse-based tool. To run the tool on a benchmark, we need two Eclipse workspaces, one for the tool and one for the benchmark. Now there are two essential things that we need to consider for reproducing this work. First, we need to have the Eclipse run configurations for each benchmark, including the main arguments, VM arguments, working directory, and program arguments. Second, we need to have all the information about the workspaces, including the versions and all the plugins they are using; otherwise, we will not recreate the exact results.

To address all the mentioned issues for reproducibility of this work, I documented all the required information in a Figshare article (`https://figshare.com/articles/software/A_Quantitative_and_Qualitative_Empirical_Evaluation_of_a_Test_Refactoring_Tool_-_Information_for_Reproducing_Results/16755622`).

This Figshare includes three base directories. The "refactoring_nominees_XLS_files" directory includes the 18 refactoring nominees XLS files that I used for 18 benchmarks. It is possible to create these XLS files by using Decard's clone detection ability and the information documented in Section 4.4. However, I thought it might be better to document the final XLS files as well. The "eclipse_workspaces" directory includes all the information about the two workspaces I used for this work, including all the versions and plugins I used in each workspace. Finally, the "eclipse_run_configurations" directory includes all the arguments needed to create a run configuration for each of the 18 benchmarks.

# Chapter 3

# Evaluating PL/SE Research

Often in the programming languages research community, the worth of an idea is evaluated empirically. Research developments depend on empirical evidence to prove their effectiveness. In August of 2017, SIGPLAN [11] formed an ad hoc committee on Programming Languages Research Empirical Evaluation to find out best practices for putting together better empirical evaluation in PL research that would lead to more reliable conclusions. They examined the literature to distinguish common forms of empirical evaluation used in PL research. In doing so, they identified some common inadequacies even in recent papers in highly regarded venues. Based on this, they came up with a one-page Empirical Evaluation Checklist that includes the best practices and guidelines for empirical evaluation in programming languages research.

One of the common inadequacies that they found states that, unfortunately: "(Programming Languages) papers we looked at often subset a benchmark, or cherry-picked particular programs.". This threatens the validity of the claims. We included the documented process of choosing the benchmarks and benchmark requirement list to avoid this issue in our work.

"An unsound claim can misdirect a field, encouraging the pursuit of unworthy ideas and the abandonment of promising ideas.". S. M. Blackburn et al. [9] believed that having a methodical approach to exploring, exposing, and addressing the root of unsound claims and poor exposition would help to solve this issue. They proposed a framework that identifies two categories of sins: Three sins of reasoning that lead to unsound claims and two exposition sins that lead to poorly specified claims and evaluations. Their framework provides practitioners with methodological techniques for evaluating the integrity of their work or the work of others.

Krishnamurthi et al. [21] advocated the importance of Artifact Evaluation Committees in programming languages research. Software artifacts play a central role in the programming languages field, yet we rarely provide a software artifact for evaluation when we publish research. Krishnamurthi et al. stated that “If a paper makes, or implies, claims that require software, those claims must be backed up.”. They also discussed the artifact evaluation process, the mechanics of artifact evaluation, who should evaluate artifacts, whether the artifacts should be published, and the benefits of artifact evaluation, such as that “Artifact evaluation encourages authors to produce reasonable artifacts, which are the cornerstone of future research.”.

## 3.1 Concrete Examples of Evaluations in PL Research

Linking user bug reports and code changes for fixing those bugs are missing for several software projects since the bug tracking and version control systems are often maintained independently. There have been some solutions, such as ReLink [27], proposed for this problem. However, Bissyandé et al. [8] believed that the presentation of the effectiveness of ReLink is subject to several issues, including a reliability issue with their ground truth datasets in addition to the extent of their measurement. They proposed a benchmark for evaluating this bug-linking solution, and they designed some research questions for quantitative and qualitative assessment of the effectiveness of this tool. Furthermore, they applied their benchmark to ReLink to determine the strengths and limitations of this tool.

The $i^*$ modeling framework is a modeling language used in the early system modeling phase to help understand the problem domain. This framework has been widely used in research and some industrial projects. However, Estrada et al. [18] concluded that no empirical evaluation existed to identify the strengths and weaknesses of this framework. They presented an empirical evaluation of the $i^*$ framework using industrial case studies. They conducted their work in collaboration with an industrial partner who was using object-oriented and model-driven approaches for their software development. Estrada et al. report lessons learned from this experience showing the strengths and weaknesses of this framework and, they believe that this evaluation could play a crucial role in guiding extensions of the $i^*$ framework.

Questionnaires are one method for soliciting feedback. However, a questionnaire does not guarantee quality results because it is difficult to find the right engaged target audience for a technical software tool questionnaire. Furthermore, feedback from the questionnaires tends to have a lower level of detail. Laugwitz et al. [22] designed a user experience questionnaire to get feedback on six factors: Attractiveness, Perspicuity, Efficiency, Depend-ability, Stimulation, and Novelty. Their results indicated a satisfying level of reliability and construct validity.

Another method for getting feedback on the quality of a software tool and qualitative evaluation is manual self-assessment. The feedback received from this assessment can be as detailed as required. This method does not need external help, but it could be vulnerable to potential unconscious bias and wrong assumptions about what factors would be the most crucial for the overall quality of the tool.

One of the best techniques to obtain valuable feedback for a specific software tool would be to contact potential customers directly about the quality of the changes that our tool has made to their codebase. Submitting pull requests is a method that makes this technique possible. This allows us to determine which factors are the most crucial for the quality of our tool from the potential customers' point of view. However, using this method to get feedback requires spending more time.

When using the methods that require external help for getting feedback on the overall quality of changes (such as using questionnaires or submitting pull requests), an essential consideration is that ethical standards should be held whenever a method seeks external help. Violations of those ethical standards can cause irremediable consequences.

An example of these consequences occurred when the University of Minnesota got a university-wide ban by the Linux kernel. One of their systems-security researchers submitted pull requests to the Linux kernel for a hidden purpose that they did not state, which the Linux Foundation deemed very unethical. Developers were offended that the university had purposely wasted the reviewers' time. This resulted in a university-wide ban following an email from Linux Foundation fellow Greg Kroah-Hartman which stated: "I suggest you find a different community to do experiments on," and "You are not welcome here." [13]

# Chapter 4

# Selecting Benchmarks

The choice of benchmarks directly affects the result of the research. Choosing the proper benchmark collection is one of the most challenging and essential parts of any software engineering/programming languages research, and yet it is frequently underestimated.

I started with some considerations about my benchmark collection and learned more along the way. This chapter will discuss considerations for the benchmark collection, the forced constraints I dealt with, the constraint that I added, the process of choosing the benchmarks, preparing the needed data for a specific benchmark, each of the benchmarks, different usages of benchmarks, and what I have learned.

## 4.1 Existing Benchmark Collections

Industrial consortia and researchers have created collections of benchmarks for various purposes. The earliest Java benchmark collection was SPEC JVM 98 [1, 2], designed to measure the performance of both Java virtual machine client platforms and hardware systems. Then there is the DaCapo benchmark collection [10], consisting of open-source, real-world Java applications. They introduced new value, time-series, and statistical metrics for static and dynamic properties such as code complexity, code size, heap composition, and pointer mutations. Nevertheless, both these benchmark collections are mainly trying to evaluate the performance and memory management of a virtual machine.

Furthermore, there exists the Qualitas Corpus [26]. Per the authors, “Qualitas Corpus is a curated collection of software systems intended for analyzing empirical studies of code artifacts with the primary goal of providing a resource that supports reproducible studies of software.”. However, Qualitas’s open-source Java software systems are not necessarily executable. Finally, XCorpus [14] is “a set of 76 executable, real-world Java programs, including a subset of Qualitas Corpus, XCorpus uses a harness that is a combination of built-in and generated test cases, resulting in a branch coverage that is significantly better than what is available from DaCapo.”.

## 4.2 Constraints

The JTestParametrizer tool forced some conditions on the candidate benchmarks, and I added some more conditions to either simplify the process of running the benchmarks or increase the potential quality of the feedback. Due to these constraints and because I knew that the feedback I was looking for was mainly on the quality of the refactorings and not quantitative metrics such as performance, I decided to create a new specific set of benchmarks for this work.

Here, I will discuss the main constraints on benchmarks for this part. The JTestParametrizer tool required:

1. That the benchmarks be Java projects because the JTestParametrizer tool is designed for the Java programming language.
2. That the benchmarks run and pass all the test runs successfully on my machines (I used a macOS machine to run the JTestParametrizer tool and an Ubuntu machine for running Deckard [20]).
3. That the benchmarks run successfully on Eclipse because the JTestParametrizer is an Eclipse-based tool.

Furthermore, here are three constraints that I added to simplify the process of running the benchmarks or increase the potential value of the feedback on the quality of refactorings:

1. I added a constraint to use a setup based on the Maven project management tool. Having this constraint enables having the same build setup process for all of the benchmarks, which will lead to simplicity and consistency of the process. However, it limits the benchmarks to Maven projects.
2. I added another constraint to restrict the minimum number of stars and forks on GitHub for candidate benchmarks to have more relevant and widely used benchmarks.

Stars and forks are indicators of how many people decide to use or work on a specific project, and even though they are not the perfect metrics, to some degree, they show the reliability and the quality of that project and its developers. Therefore, this constraint can enhance the potential quality of feedback from the developers. Nevertheless, it limits the benchmarks to project with a certain minimum number of stars and forks.

3. I added another constraint to discard the benchmarks that took more than one hour to run all the test runs on the macOS machine. Each benchmark will be run repeatedly, and using the projects that take longer to build has no particular advantage to make up for the unnecessary time it takes. Therefore, this constraint will help to simplify the process of running benchmarks by saving time. However, it adds a time constraint on the possible candidate benchmarks.

## 4.3 Process of Choosing the Benchmarks

To find candidate benchmarks that satisfy the stated constraints, I started with the benchmarks that Jun Zhao used in his thesis [29]. And then, I went through over a hundred open source projects to find the candidates that fit our requirements.

I used the suggested Java projects on GitHub lists by IssueHunt, awesomeopensource, and Henn Idan to find potential open-source Java project candidates, although I discarded the Github projects where their number of stars plus their number of forks on GitHub was less than 800.

I also considered all the open-source Java projects from the Google, Spotify, Apache, Airbnb, and Netflix companies on GitHub that had the minimum number of stars + forks as potential benchmark candidates.

### 4.3.1 Benchmark Requirements Checklist

Checklists are a helpful way for quickly evaluating things. They help to be more organized and to not skip any vital step in the process. I used the following checklist to determine if a candidate benchmark has the minimum requirements needed to be in the benchmark collection for this work.

1. Ensure that the candidate is an open-source Java project built by Maven.

2. Ensure that the candidate contains tests.

3. Ensure that I can build the candidate on my machines (macOS and Ubuntu machines) and that all the test runs pass successfully; if this is not the case, spend up to one hour fixing the issues. If I cannot fix it, discard this candidate.

4. Ensure that I can build the candidate on Eclipse; if this is not the case, spend one hour fixing the issues. If I cannot fix it, discard this candidate.

5. Create the cluster files for that project using Deckard, and then use the cluster files to create the XLS file of the potential nominees for refactoring. Now, discard this candidate if the created XLS file is empty, meaning that there are no nominees for refactoring.

If a project satisfied all these conditions, then I could consider it as a potential benchmark. However, these were the minimum requirements, and some projects had these requirements but still failed to make the benchmark collection due to having a low number of refactoring nominees or not having any refactored cases after running the JTestPrametrizer tool.

## 4.4 Deckard and Potential Nominees for Refactoring

One of the primary inputs of the JTestParametrizer tool for every benchmark is an XLS file that includes basic information (folder, class, package, method, start line, end line, and clone group size) about the potential nominees for refactoring in the benchmark. I created this XLS file by running a process on the cluster files that I receive from running Deckard's clone detection tool on that benchmark.

To use Deckard's clone detection ability, we need to set up specific parameters such as MIN_TOKENS ("minimum number of tokens required for clones"), STRIDE ("size of the sliding window"), and SIMILARITY described in Deckard: scalable and accurate tree-based detection of code clones [20].

Jun Zhao did not document the parameters he used in his research. I deduce that he used 0 for STRIDE (equivalent to no merging of vectors), but I could not deduce what he used for MIN_TOKENS and SIMILARITY. Increasing the MIN_TOKENS will decrease the number of potential nominees for refactoring since the nominees have to have at least that many tokens to be eligible. On the other hand, if the MIN_TOKENS is too low, we will have nominees that can be refactored, but the refactoring would not be worth doing since the nominees were very trivial. Furthermore, SIMILARITY should be a number between 0.9 and 1. If it is too high, we will miss several nominees (false negatives), and if it is too low, we will have poor nominees (false positives).

After considering eight different sets of values, MIN_TOKENS=50–100, STRIDE=2–0, and SIMILARITY=0.95–0.90, for these three parameters for the three benchmarks Gson, Joda-time, and Jfreechart, and evaluating the differences between the final XLS files created by each set of variables, I decided to use 50 for the MIN_TOKENS, 0 for STRIDE, and 0.95 for SIMILARITY for every benchmark to keep the results consistent. These chosen parameters produced good results for the three benchmarks that I was using for evaluation.

With 50 for MIN_TOKENS, we will not have many false positives (nominees that can be refactored, but it would be better if not, because they are already trivial), and 50 is not too high to miss good nominees because of the eligibility constraint. Furthermore, Jiang et al. [20] state that for clone detection, SIMILARITY could be a number between 0.9 and 1, and with 0.95, it is not too high to miss on several nominees, and it is not too low to have poor nominees. Also, for the three benchmarks discussed above, changing the Similarity from 0.9 to 0.95 did not considerably impact the final list of nominees.

## 4.5 Our Benchmark Suite

In this section, I will go over the projects that I used as benchmarks for this work. First, I will go over the numeric facts of each benchmark in table 4.1, then I will explain each of the benchmarks briefly.

### 4.5.1 Numeric Facts

Table 4.1 lists the numbers related to the essential aspects of the GitHub repository, including the number of stars, number of forks, number of GitHub issues, number of contributors, number of closed pull requests, and number of open pull requests, on GitHub, for each of the projects used as benchmarks. As for the number of lines of java code, I used SLOC-Count[1] to measure it for the latest version of each benchmark.

There are 14 repositories in this table but 18 benchmarks in total. This is because two of the benchmarks (Netty/Codec-http and Netty/Buffer) are subprojects of the same GitHub

[1]Source Lines of Code Count, https://dwheeler.com/sloccount/

repository, Netty. Also, I used two different versions of each of the three repositories Gson, Bootique, and Joda-time, as separate benchmarks.

| Repository | Stars | Forks | Lines of Java | GitHub Issues | Contributors | Closed PRs | Open PRs |
|---|---|---|---|---|---|---|---|
| Gson | 20k | 3.9k | 25.6k | 503 | 114 | 358 | 151 |
| Jimfs | 2k | 245 | 17.4k | 26 | 23 | 96 | 4 |
| Bootique | 1.3k | 286 | 18.5k | 31 | 17 | 84 | 4 |
| Joda-time | 4.7k | 888 | 86.5k | 23 | 77 | 160 | 3 |
| Commons-lang | 2.2k | 1.3k | 85.1k | - | 161 | 682 | 107 |
| Commons-io | 800 | 519 | 41.5k | - | 76 | 232 | 29 |
| Commons-collections | 506 | 339 | 67.6k | - | 57 | 215 | 28 |
| Jfreechart | 732 | 296 | 133.5k | 65 | 22 | 75 | 65 |
| Netty | 27.4k | 13.5k | 312.1k | 451 | 531 | 5,963 | 43 |
| Checkstyle | 6.2k | 8k | 286.5k | 642 | 290 | 6,530 | 41 |
| Git-commit-id-maven-plugin | 1.3k | 253 | 3.7k | 23 | 73 | 241 | 3 |
| Docker-maven-plugin | 2.6k | 556 | 2.5k | 10 | 38 | 160 | 11 |
| Maven | 2.7k | 2k | 91.9k | - | 143 | 428 | 55 |
| Mybatis-3 | 16.1k | 10.9k | 60.8k | 123 | 182 | 1,156 | 55 |

Table 4.1: Numeric Characteristics of Benchmarks

Table 4.2 lists the necessary information that we retrieved for each benchmark before running the JTestParametrizer tool on that benchmark. This information includes the version of that benchmark, the number of Java code lines for that specific version using SLOCCount, the number of tests run, failures, errors, and skipped from building that benchmark and running all the tests on my macOS machine, and finally, the number of refactoring nominees that I got from running a process on the cluster files that I got from running the Deckard clone detection tool on that version of the benchmark.

### 4.5.2 Gson

Gson is an open-source Maven project described as "a Java library that can be used to convert Java Objects into their JSON representation. It can also be used to convert a JSON string to an equivalent Java object. Gson can work with arbitrary Java objects including pre-existing objects that you do not have source-code of."

I used two versions of the Gson project, the first one being gson-parent-2.8.5 (21 May 2018) which was the version that Jun Zhao used, and the second one f319c1b8e5 (27 May 2021), which was the version I used to create the pull request. Each run of all tests took under a minute. Furthermore, Gson ran successfully on Eclipse. After running Deckard's clone detection tool with the stated parameters and analyzing the cluster files, I got 39

| Repository | Version | Lines of Java | Tests run | Failures | Errors | Skipped | Nominees |
|---|---|---|---|---|---|---|---|
| Gson | f649e05 | 25193 | 1050 | 0 | 0 | 1 | 39 |
| Gson | f319c1b | 25269 | 1063 | 0 | 0 | 1 | 42 |
| Jimfs | 3c9d8ba | 17472 | 5834 | 0 | 0 | 0 | 45 |
| Bootique | d0648eb | 18589 | 231 | 0 | 0 | 0 | 22 |
| Bootique | 9939bc6 | 18591 | 228 | 0 | 0 | 0 | 23 |
| Joda-time | 0ae5311 | 86138 | 4222 | 0 | 0 | 0 | 261 |
| Joda-time | 27edfff | 86536 | 4238 | 0 | 0 | 0 | 260 |
| Commons-lang | 425d808 | 77224 | 4068 | 0 | 0 | 5 | 154 |
| Commons-io | e4ff4a5 | 40336 | 1852 | 0 | 0 | 6 | 32 |
| Commons-collections | 7d8b979 | 67647 | 16923 | 0 | 0 | 4 | 47 |
| Jfreechart | d03e68a | 132452 | 2176 | 0 | 0 | 0 | 124 |
| Netty/Codec-http | e69107c | 41014 | 858 | 0 | 0 | 0 | 47 |
| Netty/Buffer | e69107c | 33564 | 10458 | 0 | 0 | 1198 | 20 |
| Checkstyle | 6cbc1dc | 255741 | 3528 | 0 | 0 | 0 | 130 |
| Git-commit-id-maven-plugin | 4a1ac8f | 7238 | 214 | 0 | 0 | 1 | 4 |
| Docker-maven-plugin | 84020ac | 2434 | 59 | 0 | 0 | 0 | 7 |
| Maven/Maven-core | 3fabb63 | 38653 | 388 | 0 | 0 | 4 | 26 |
| Mybatis-3 | 1d82865 | 60825 | 1675 | 0 | 0 | 14 | 26 |

Table 4.2: Benchmarks' Refactoring Nominees

test method pairs as refactoring nominees for the first version, and for the second version, 42 test method pairs as refactoring nominees.

### 4.5.3 Jimfs

Jimfs is an open-source Maven project described as "an in-memory file system for Java 7 and above, implementing the java.nio.file abstract file system APIs."

I used the 3c9d8babec version (1 June 2021), which was also the version that I used to create the pull request for this benchmark. Each run of all tests took about one and a half minutes.

I fixed the problems with Eclipse by changing the setting for Maven → Errors/Warnings in Eclipse. I set the "groupId" duplicate of parent groupId to Warning, "version" duplicate of parent version to Warning, Out-of-date project configuration to Error, Plugin execution not covered by lifecycle configuration to warning, and finally, Overriding manged version to Warning.

The number of test refactoring nominees I got for this version using Deckard's clone detection tool with specified parameters was 45.

### 4.5.4 Bootique

Bootique is an open-source Maven project described as “a minimally opinionated java launcher and integration technology which is intended for building container-less runnable Java applications.”

I used two versions of the Bootique project, the first one being the d0648eb612 (19 Feb 2021) and the second one 9939bc640c (11 Jun 2021), which was the version I used to create the pull request. Each run of all tests took under a minute. Furthermore, it ran successfully on Eclipse. After running Deckard’s clone detection tool with the stated parameters and analyzing the cluster files, I got 22 test method pairs as refactoring nominees for the first version, and for the second version, 23 test method pairs as refactoring nominees.

### 4.5.5 Joda-time

Joda-time is an open-source Maven project expressed as “a quality replacement for the Java date and time classes. The design allows for multiple calendar systems while still providing a simple API.”

I used two versions of the Joda-time project: the 0ae5311895 (30 May 2018), which was the version that Jun Zhao used, and the second one, 27edfffa58 (20 Apr 2021), which was the version I used to create the pull request. For both versions, I changed the Maven compiler from 1.5 to 1.8 in the pom.xml file to fix the build errors. Each run of all tests took under a minute. Furthermore, it ran successfully on Eclipse. After running Deckard’s clone detection tool with the stated parameters and analyzing the cluster files, I got 261 test method pairs as refactoring nominees for the first version, and for the second version, 260 test method pairs as refactoring nominees.

### 4.5.6 Commons-lang

Commons-lang is an open-source Maven project expressed as “a package of Java utility classes for the classes that are in java.lang’s hierarchy, or are considered to be so standard as to justify existence in java.lang.”

I used the 425d8085cf version (4 Nov 2017) with some custom changes, the same version that Jun Zhao used. Each run of all tests took just over one minute. This project ran successfully on Eclipse. The number of test refactoring nominees I got for this version using Deckard’s clone detection tool with specified parameters was 154.

### 4.5.7 Commons-io

Commons-io is an open-source Maven project described as “a library containing utility classes, stream implementations, file filters, file comparators, endian transformation classes, and much more.”

I used the e4ff4a589b version (31 May 2021). Each run of all tests took about seven minutes. There were some issues when running the project on Eclipse, but those issues were fixed by changing the setting for Maven → Errors/Warnings (in Eclipse). The number of test refactoring nominees I got for this version using Deckard’s clone detection tool with specified parameters was 32.

### 4.5.8 Commons-collections

Commons-collections is an open-source Maven project described as “a package containing types that extend and augment the Java Collections Framework.”

I used the 7d8b979612 version (28 May 2021). Each run of all tests took about one minute. There were some issues when running the project on Eclipse, but those issues were fixed by changing the setting for Maven → Errors/Warnings (in Eclipse). The number of test refactoring nominees I got for this version using Deckard’s clone detection tool with specified parameters was 47.

### 4.5.9 Jfreechart

Jfreechart is an open-source Maven project described as “a comprehensive free chart library for the Java(tm) platform that can be used on the client-side (JavaFX and Swing) or the server-side (with export to multiple formats including SVG, PNG, and PDF).”

I used the d03e68acaf version (7 Feb 2019). I changed the project.source.level and project.target.level from 1.6 to 1.8 in the pom.xml file to fix the build errors. Each run of all tests took less than a minute. The project ran successfully on Eclipse. The number of test refactoring nominees I got for this version using Deckard’s clone detection tool with specified parameters was 124.

### 4.5.10 Netty/Codec-http

Codec-http is a subproject of the Netty project. Netty is an open-source Maven project described as “an asynchronous event-driven network application framework for rapid development of maintainable high-performance protocol servers & clients."

I used the e69107ceaf version (8 Jun 2021). Each run of all tests took just under a minute. There were some issues when running the project on Eclipse, but those issues were fixed by changing the setting for Maven → Errors/Warnings (in Eclipse). The number of test refactoring nominees I got for this version using Deckard's clone detection tool with specified parameters was 47.

### 4.5.11 Netty/Buffer

Buffer is another subproject of the Netty project. I used the same e69107ceaf version (8 Jun 2021) for Buffer too. Each run of all tests took less than two minutes. There were some issues when running the project on Eclipse, but those issues were fixed by changing the setting for Maven → Errors/Warnings (in Eclipse). The number of test refactoring nominees I got for this version using Deckard's clone detection tool with specified parameters was 20.

### 4.5.12 Checkstyle

Checkstyle is an open-source Maven project described as "a tool for checking Java source code for adherence to a Code Standard or set of validation rules (best practices)."

I used the 6cbc1dc3c0 version (31 May 2021). Each run of all tests took about four minutes. There were some issues when running the project on Eclipse, but those issues were fixed by changing the setting for Maven → Errors/Warnings (in Eclipse). The number of test refactoring nominees I got for this version using Deckard's clone detection tool with specified parameters was 130.

### 4.5.13 Git-commit-id-maven-plugin

Git-commit-id-maven-plugin is an open-source Maven project described as "a Maven plugin which includes build-time git repository information into a POJO / *.properties). Make the apps tell which version exactly they were built from!"

I used the 4a1ac8fbad version (23 Apr 2021). Each run of all tests took less than four minutes. There were some issues when running the project on Eclipse, but those issues were fixed by changing the setting for Maven → Errors/Warnings (in Eclipse). The number of test refactoring nominees I got for this version using Deckard's clone detection tool with specified parameters was 4.

### 4.5.14 Docker-maven-plugin

Docker-maven-plugin is an open-source Maven project described as "A Maven plugin for building and pushing Docker images."

I used the 84020acb55 version (14 Jan 2020). Each run of all tests took less than a minute. The project ran successfully on Eclipse. The number of test refactoring nominees I got for this version using Deckard's clone detection tool with specified parameters was 7.

### 4.5.15 Maven/Maven-core

Maven-core is a subproject of the Maven project. Maven is an open-source project described as "software project management and comprehension tool. Based on the concept of a project object model (POM), Maven can manage a project's build, reporting, and documentation from a central piece of information."

I used the 3fabb639a3 version (15 May 2021). Each run of all tests took about three minutes. There were some issues when running the project on Eclipse, but those issues were fixed by changing the setting for Maven $\rightarrow$ Errors/Warnings (in Eclipse). The number of test refactoring nominees I got for this version using Deckard's clone detection tool with specified parameters was 26.

### 4.5.16 Mybatis-3

Mybatis-3 is an open-source Maven project described as "The MyBatis SQL mapper framework makes it easier to use a relational database with object-oriented applications."

I used the 1d82865816 version (11 Apr 2021). Each run of all tests took slightly less than four minutes. The project ran successfully on Eclipse. The number of test refactoring nominees I got for this version using Deckard's clone detection tool with specified parameters was 26.

## 4.6 Benchmarks for Different Types of Feedback

As we previously discussed in chapter 3, we were aiming for different types of feedback.

A refactoring case might be correct, meaning that it does not cause any errors or failures, and the behavior of the refactored test case stays the same, but that refactoring case might still be poor due to decreasing the maintainability or readability of the code. In that sense, verifying the correctness of refactorings and validating their quality are two different topics.

I used all 18 benchmarks to get numerical metrics and statistics. Besides, I used Gson, Joda-time, Jfreechart, Commons-lang, Bootique, and Jimfs benchmarks to verify the JTest-Parametrizer tool's correctness and to find the errors and bugs of the JTestParametrizer tool. If an error occurred when running the JTestParametrizer tool on a benchmark, or if after running the JTestParametrizer on a benchmark that benchmark had some compile errors, or if after running the JTestParametrizer on a benchmark, some of the tests run of that benchmark resulted in failures or errors, then I documented those incidents, and in most cases, tried to find the source of that problem and fix the JTestParametrizer tool.

I used Gson and Joda-time to develop practical examples for the questionnaire to validate the quality of the refactoring instead of the correctness (all the examples were correct), which we ended up not pursuing due to the reasons explained in chapter 7.

We manually validated refactoring quality for the benchmarks Jimfs, Gson, Joda-time, and Bootique. We also sought developer feedback about test refactorings of these benchmarks using pull requests.

## 4.7 Factors Learned Throughout

Besides the three necessary constraints and the three unnecessary ones discussed earlier, we learned some factors that led us to create new optional constraints throughout the work. These newly created constraints were not necessarily for all benchmarks and were based on the type of feedback we wanted for that benchmark.

We did not use all the benchmarks the same way. There were some benchmarks such as Checkstyle that we only used for getting numerical metrics and quantitative evaluation. Whereas, there were benchmarks like Gson that we used not only for that purpose but also for coming up with practical examples for the questionnaire, validating the quality of the refactorings by ourselves, and validating the quality of the refactorings by their developers.

Here we discuss three of these new constraints that are specialized for certain types of feedback.

### 4.7.1 Maintenance Mode

For the benchmarks used for validating the quality of the refactorings by their developers, it would be best if there are some constraints to check that that specific benchmark is not in maintenance mode. For instance, after submitting the pull request for the Gson benchmark, the first part of the feedback that we got was this: "This project is in maintenance mode, and we are generally going to be reluctant to accept PRs that are essentially cosmetic, especially if it is not trivially obvious that they do not change anything.". Also, in the case of Joda-time Benchmark, after going over all the steps and just before submitting the pull request, we realized that the project is in maintenance mode.

How to figure out if a benchmark is in maintenance mode? There are multiple ways. After learning about the importance of this factor, we did the following steps to determine whether a project is in maintenance mode for the following candidate benchmark for sending a pull request:

1. Examine the last few commits to the codebase and checking if they are adding new features.
2. Check out the time gap between the last few commits to see how frequently they are committing.
3. Investigate if they mention that they are in maintenance mode in a message somewhere, such as the pull request message.

For instance, for the Joda-time case, the last commit was about four months ago, the last few commits were all about the version release, and the pull request message pointed out that they are in maintenance mode.

### 4.7.2 Estimated Response Time

Another good constraint for the benchmarks used to validate the quality of the refactorings by their developers would be to prioritize the projects that take much less time to provide feedback. For instance, the Gson project provided feedback to our pull request in less than an hour, whereas the Bootique project has not reacted to our pull request after a month.

How to check this? Checking the list of open/closed pull requests for that project and looking at the last few closed pull request and some of the open ones will provide a reasonable estimate on how long it takes for them to come up with feedback. Furthermore, the number of contributors can be an indicator too.

### 4.7.3 Familiarity with Benchmark's Domain

One factor to have in mind when choosing benchmarks used to validate the quality of the refactorings is that when we evaluate the quality of changes, we have to understand the domain of that project and understand the semantics behind the changes. So choosing the projects that we understand better would be very helpful. Especially when we want to evaluate the quality of each refactoring case by giving them a rating, being familiar with the context of refactoring enhances our judgment.

Nevertheless, even though a benchmark might not be ideal for getting developer feedback, it does not necessarily mean that it will not be great for retrieving quantitive feedback or feedback on the quality of the refactoring cases using a questioner or self-assessment. For instance, although I could not get the developer feedback that I was hoping for Joda-time (due to it being in maintenance mode), I used Joda-time for self-assessment of the quality of refactorings.

# Chapter 5

# JTestParametrizer Tool

JTestParametrizer is a refactoring tool that Zhao presented in his thesis [29]. This tool aims to refactor the method-scope renamed clones in test suites automatically. It uses three parametrization techniques “Type Parametrization”, “Data Parametrization”, and “Behavior Parametrization” to refactor clone pairs with type, data, and behavior differences. This tool operates at the Abstract Syntax Tree level. It extracts a parametrized template method and instantiates it with suitable parameter values.

Type differences in a clone pair refer to entries with different type identifiers. The tool uses the common superclass of the different types as the bound of the extracted generic type for the “Type Parametrization” technique. Furthermore, data differences in a clone pair refer to differences in literal values and variables. For the “Data Parametrization” technique, the tool replaces the differing values with a parameter and passes the particular values as arguments to the extracted template method. Finally, behavioral differences refer to method invocation calls with the same length argument lists but different signatures in the clone pair. The “Behavior Parametrization” technique used in the tool is more complex than the other two.

The JTestParametrizer tool works as an Eclipse application, with the tool being on one Eclipse workspace and the benchmark project being on another Eclipse workspace. One of the most crucial inputs for this tool is an XLS file for each benchmark containing information about the refactoring nominees in the project we are using as the benchmark. As I explained in Section 4.4, we used Deckard’s clone detection ability to create cluster files for the project and then ran a process to retrieve the desired information from the cluster files into an XLS file. I included all the refactoring nominees XLS files I created and used in the Figshare article mentioned in Chapter 2.

Zhao used five open-source Java benchmark projects (Jfreechart, Gson, Commons-lang, Commons-io, and Joda-time) for the empirical study. He claimed that all of the refactored tests methods were compilable. I included these five benchmarks as part of the 18 benchmarks I used in this work's benchmark suite. Due to a lack of documentation of the Deckard parameters, I could not reproduce Zhaos's exact refactoring nominees' XLS files for these five benchmarks. Consequently, using the XLS files that I created with parameters explained in Section 4.4, I got inconsistent results (represented in Section 6.1) as 13 refactoring cases were causing compile errors in these five benchmarks.

## 5.1 Parameterization Techniques

As aforementioned, the JTestParameterizer tool uses three parameterization techniques to refactor the test methods. Here, I will provide three simplified examples for each of these three techniques. Each of these three examples only uses one of the three techniques.

### 5.1.1 Data Parameterization Simplified Example

In the example shown in Figure 5.1, the two test methods only differ in an integer value as one of them invokes f(5) while the other one invokes f(7). The rest of the two test methods are the same, meaning that there is code duplication.

JTestParameterizer extracts the body of these two test methods into a template method (helper function) but parametrizes the value that was different between the two into an integer argument. Now, both test methods can invoke the same created template method with different arguments.

### 5.1.2 Type Parameterization Simplified Example

The example in Figure 5.2 presents two test methods with the same body, except for a difference in the type of variable x. In the first test method, the type of x is A1, while in the second test method, the type of x is A2. Furthermore, A1 and A2 are both subclasses of type A. The rest of the two test methods are the same, and regardless of the type of x, they both follow the same logic.

JTestParameterizer extracts the body of these two test methods into a template method but also declares a generic type TA constrained to extend the type A to represent types A1

```java
// BEFORE:
public void firstTestMethod() {
   ... // some test code here
   x = y.f(5);
   ...
}

public void secondTestMethod() {
   ... // same test code appears here again (code duplication)
   x = y.f(7);
   ...
}

// AFTER:
public templateMethod(Integer int1) {
   ... // same test code appears here
   x = y.f(int1);
   ...
}

public void firstTestMethod() {
   templateMethod(5);
}

public void secondTestMethod() {
   templateMethod(7);
}

```

Figure 5.1: Data Parameterization Simplified Example

```java
// BEFORE:
public void firstTestMethod() {
   ... // some test code here
   A1 x = new A1();
   ...
}

public void secondTestMethod() {
   ... // same test code appears here again (code duplication)
   A2 x = new A2();
   ...
}

// AFTER:
public <TA extends A> void templateMethod(Class<TA> clazzTA) {
   ... // same test code appears here
   TA x = clazzTA.newInstance();
   ...
}

public void firstTestMethod() {
   templateMethod(A1.class);
}

public void secondTestMethod() {
   templateMethod(A2.class);
}

```

Figure 5.2: Type Parameterization Simplified Example

and A2. The created template method uses this generic type as its argument and uses the "newInstance" method to invoke a new instance of that class in its body. Consequently, both the first and second test methods can invoke the same template method with different arguments A1.class and A2.class.

### 5.1.3 Behavior Parameterization Simplified Example

The example in Figure 5.3 presents two test methods with the same body except for a difference in a method invocation which consequently could differentiate the behavior of the two test methods. In the first test method, function f is invoked on variable y, while in the second test method, function g is invoked on variable y. The rest of the two test methods are the same, but the behavior of the two test methods could be different.

To refactor these two test methods, the JTestParameterizer creates an interface with a method to encapsulate the behavioral difference between the two test methods. Then it creates two new adaptor implementation classes that both implement that created interface. The first adaptor implementation class implements the interface's method with the behavior of the first test method. In contrast, the second adaptor implementation class implements the interface's method with the behavior of the second test method. Next, the JTestParameterizer tool creates a template method with the extracted bodies of the two test methods. However, the template method gets an instance of the created interface as an argument and then invokes the interface's method in its body. Consequently, the two test methods can invoke the created template method now, but the first test method will pass the first adaptor implementation class as the argument while the second test method passes the second adaptor implementation class as the argument to the template method.

## 5.2 JUnit 5's Parameterized Tests

JUnit 5 [6] provides a Parameterized Tests facility that makes it possible to run a test multiple times with different arguments. This feature is available using annotations. Instead of putting the regular "@Test" annotation, using "@ParameterizedTest" will enable this feature, and then we need to declare at least one source of values for the arguments in each invocation of that parameterized test.

Using this feature will help prevent the problem of having many test methods that are very similar and only differ in some data values, which will help to reduce the duplicated code in test methods and reduce the number of test methods. However, this feature works as a prevention mechanism and requires the developer to know which test methods will be similar and only differ in data values. In comparison, the JTestParameterizer tool works as a refactoring mechanism that scans existing test methods and deduplicates them by creating new parametrized template methods. Furthermore, the JTestParameterizer tool also refactors the test methods with type differences and behavior differences. This is due to three parameterization techniques used in the JTestParameterizer tool (Data Parametrization, Type Parametrization, and Behavior Parameterization).

```java
// BEFORE:
public void firstTestMethod() {
   ... // some test code here
   x = y.f();
   ...
}

public void secondTestMethod() {
   ... // same test code appears here again (code duplication)
   x = y.g();
   ...
}

// AFTER:
interface behaviorAdaptor {
   int method(Y y);
}

class firstTestMethodAdaptorImpl implements behaviorAdaptor {
   public int method(Y y) {
      return y.f();
   }
}
class secondTestMethodAdaptorImpl implements behaviorAdaptor {
   public int method(Y y) {
      return y.g();
   }
}

public templateMethod(behaviorAdaptor adaptor) {
   ... // same test code appears here
   x = adaptor.method(y);
   ...
}

public void firstTestMethod() {
   templateMethod(new firstTestMethodAdaptorImpl());
}

public void secondTestMethod() {
   templateMethod(new secondTestMethodAdaptorImpl());
}
```

Figure 5.3: Behavior Parameterization Simplified Example

# Chapter 6

# JTestParametrizer Quantitative Results and Discussion

This chapter explains the quantitative results obtained from running the JTestParametrizer tool on the benchmarks. Furthermore, it shows how we examined the quantitative results and used the retrieved information to determine and fix some of the existing bugs in the JTestParametrizer tool.

Running JTestParametrizer on the benchmarks enabled us to figure out the stats for the applicability of the JTestParametrizer. Furthermore, when I encountered run-time errors when running the tool on a benchmark, compile errors in the benchmark after running the tool, or errors or failures in the test runs, I examined those problems that led to fixing bugs in the JTestParametrizer tool. This feedback helped with the verification of the correctness of the tool in that sense.

Suppose I do not encounter any of the stated problems (run-time errors when running the tool on a benchmark or compile errors in the benchmark after running the tool, or errors or failures in the test runs) when working on a benchmark. In that case, it does not necessarily mean that there is no problem with the correctness of the JTestParametrizer tool. However, if I do encounter a run-time error when running the tool on a benchmark or a compile error when building the refactored benchmark, then it does mean that there is a bug in the tool.

This feedback was exceedingly feasible to collect because I could do so without any external help. Moreover, since I did not need external help, I had a more reliable estimate of how much time this feedback would take. Also, spending more time on this process equals faster feedback, which does not necessarily hold for other processes for getting feedback. Furthermore, using this process does not require being familiar with the domain of that specific benchmark.

However, this feedback will not help with evaluating the quality of refactoring cases at all. That is, using this feedback, I can detect things that are definitely wrong and count how often the tool works, but there is a lot that I am not catching, especially domain-specific things.

## 6.1 Quantitative Results

Table 6.1 presents the quantitative results of running the JTestParametrizer tool on the benchmarks. The first two columns (Repository, version) identify the benchmark. The third column (Nominees) represents the number of refactoring nominees, also seen in Table 4.2. The fourth column (Run-time errors) denotes the number of refactoring nominees that caused a run-time error when running the JTestParametrizer tool on that benchmark. The fifth column (Compile-time errors) represents the number of refactoring nominees that caused a compile error when I was trying to build the benchmark after running the JTestParametrizer tool. The sixth column (Refactored) represents the number of refactored nominees; I got this after excluding those in the fourth and fifth columns categories. Columns Tests run, Failures, Errors, and Skipped represent the result of running the test cases of the refactored benchmark.

| Repository | Version | Nominees | Run-time errors | Compile-time errors | Refactored | Tests run | Failures | Errors | Skipped |
|---|---|---|---|---|---|---|---|---|---|
| Gson | f649e05 | 39 | 0 | 0 | 17 | 1050 | 0 | 0 | 1 |
| Gson | f319c1b | 42 | 0 | 0 | 18 | 1063 | 0 | 0 | 1 |
| Jimfs | 3c9d8ba | 45 | 0 | 1 | 5 | 5834 | 0 | 0 | 0 |
| Bootique | d0648eb | 22 | 0 | 2 | 10 | 231 | 0 | 1 | 0 |
| Bootique | 9939bc6 | 23 | 0 | 2 | 11 | 228 | 0 | 1 | 0 |
| Joda-time | 0ae5311 | 261 | 1 | 1 | 112 | 4224 | 5 | 6 | 0 |
| Joda-time | 27edfff | 260 | 1 | 1 | 112 | 4240 | 5 | 6 | 0 |
| Commons-lang | 425d808 | 154 | 1 | 8 | 62 | 4068 | 2 | 4 | 5 |
| Commons-io | e4ff4a5 | 32 | 0 | 3 | 9 | 1852 | 0 | 0 | 6 |
| Commons-collections | 7d8b979 | 47 | 1 | 1 | 4 | 16923 | 0 | 0 | 4 |
| Jfreechart | d03e68a | 124 | 0 | 1 | 65 | 2176 | 2 | 3 | 0 |
| Netty/Codec-http | e69107c | 47 | 0 | 3 | 12 | - | - | - | - |
| Netty/Buffer | e69107c | 20 | 0 | 3 | 2 | - | - | - | - |
| Checkstyle | 6cbc1dc | 130 | 0 | 1 | 25 | 3528 | 0 | 2 | 0 |
| Git-commit-id-maven-plugin | 4a1ac8f | 4 | 0 | 0 | 2 | 214 | 0 | 0 | 1 |
| Docker-maven-plugin | 84020ac | 7 | 0 | 0 | 2 | 59 | 0 | 0 | 0 |
| Maven/Maven-core | 3fabb63 | 26 | 0 | 0 | 12 | 388 | 0 | 0 | 4 |
| Mybatis-3 | 1d82865 | 26 | 0 | 0 | 0 | 1675 | 0 | 0 | 14 |

Table 6.1: The JTestParametrizer Tool Quantitative Results

To determine the fourth column (Run-time errors), which represents the number of refactoring nominees that caused a run-time error when I ran the JTestParametrizer tool on that benchmark, I first ran the JTestParametrizer on the benchmark. If there were no run-time errors, the number would be zero, but if there were a run-time error, I would use a combination of log files and debugging using breakpoints to find out which nominee was causing that compile error. Then I would record this problem, discard that refactoring nominee and repeat this process until there are no more run-time errors.

To determine the fifth column (Compile-time errors), which represents the number of refactoring nominees that caused a compile error when I was trying to build the benchmark after running the JTestParametrizer tool, I tried to build the refactored benchmark. If I did not get any compile errors, then the number would be zero, but if I did get compile errors, then I checked them one by one, figured out which refactored test cases are causing these compile errors using git diff, and then determined which nominees are responsible for these refactored test cases using the refactoring nominees XLS file. Then I recorded these problems and discarded these refactoring nominees that led to compile errors.

To determine the numbers in the sixth column (Refactored), after following the explained process for the fourth and fifth columns and discarding those refactoring nominees, I ran the JTestParametrizer tool on the benchmark one more time, knowing that I would not get any run-time errors and that the refactored benchmark would not have any compile errors. Then I examined the log file and determined how many refactoring cases were in that last execution of the JTestParametrizer tool on that benchmark.

To determine the numbers in columns seven, eight, nine, and ten, I used the "mvn clean test" (or "mvn clean test -Drat.skip=true" if the benchmark is using Apache RAT[1] in Maven) command on the refactored benchmark that I got from the process explained for the sixth column, and then I reported the number of tests run, failures, errors, and skipped. I could not run the "mvn clean test" command on the refactored Netty successfully because a copyright header would cause errors when running that command for modified Netty.

## 6.2 Potential Errors

As discussed earlier, any occurrence of run-time errors when running the JTestParametrizer tool on a benchmark or any occurrence of compile errors when building a refactored benchmark represents a potential problem in the JTestParametrizer tool. Furthermore, any fail-

[1]Apache RAT (Release Audit Tool), `http://creadur.apache.org/rat/`

ures or errors resulting from running tests for a refactored benchmark can also hint at a potential problem in the JTestParametrizer tool.

As seen in Table 6.1, there were multiple cases of those occurrences in columns Run, Compile, Failures, and Errors. I began with identifying which nominees were causing each of those occurrences and recorded those nominees to go over them one by one.

I explained how I recorded the nominees that were causing the run-time or compile-time errors in Section 6.1. As for the test errors and failures, I followed a similar process. I determined which test cases were failing or causing errors. Then using the refactoring nominees XLS file for that benchmark (which contains the name of all the test methods involved in a refactoring case), I identified the refactoring nominees that were causing those problems.

## 6.3 Debugging Procedure

Now that I had recorded all the refactoring nominees leading to potential errors, I used the following procedure to go over them, one at a time.

1. I isolated that specific refactoring nominee by running JTestParametrizer with only the nominee of interest. To do so, I retracted all changes to the benchmark and then ran the tool on the benchmark one more time, but I used a different XLS file this time around and only put that one specific nominee into the XLS file. This way, the only refactoring was that one case.

2. I compared the refactored test and the original test manually and tried figuring out what the test case was doing, what caused the problem in the refactored version, and why the refactored version is wrong.

3. I used breakpoints and the new XLS file mentioned in the first step to go over this refactoring case until I figured out what part of the JTestParametrizer code is responsible for the problem in the refactored test.

4. I tried understanding why that part of the JTestParametrizer code is wrong and how I can modify it to fix that problem. Then I came up with a solution and modified the JTestParametrizer tool.

5. I checked whether the modified tool fixed that specific problem. If not, I would go back to step four and try coming up with a new solution.

6. I ran the modified JTestParametrizer tool on the five benchmarks Gson, Joda-time, Bootique, Commons-lang, and Jfreechart to ensure that the modified version is not causing any new problems. If it did, I would try to understand the new problem and why it is happening, and how to change the solution in step four to avoid the new problem.

7. Finally, if the modified tool fixed the problem with this specific refactoring nominee, and did not cause any new problems in those five benchmarks, then I used the modified version as the main version from that point forward.

This process was time-consuming. Depending on the complexity of the problem in the JTestParametrizer tool, fixing each of these errors took from a few days to a few weeks. I only used five benchmarks (instead of using them all) to check whether the modified tool caused any problems, which saved some time but was a time-consuming process nonetheless.

Every fix enhanced our confidence about the correctness of the code a bit more. Some modifications only affected one of those refactoring nominees, whereas some were fixing multiple issues because the same problem was repeated in multiple places, and modifying the tool to fix one occurrence of that problem, also fixed the other ones.

Next, I discuss an example of these errors and corresponding modifications to the JTestParametrizer tool for fixing it.

### 6.3.1 Protected Method Access Issue Example

Here, I will go over one of the refactoring cases causing a compilation error in the refactored Bootique benchmark and how I followed the debugging procedure explained in Section 6.3 to fix that issue.

1. The refactoring nominee that was causing this problem included two test methods in the Bootique benchmark: OffsetDateTimeDeserializerIT.testDeserialization03 and ZonedDateTimeDeserializerIT.testDeserialization03. I created a new XLS file with only this refactoring nominee inside that file so that after running the JTestParametrizer tool on Bootique benchmark using this new XLS file, the only refactoring case would be the one that I want to examine.

2. Figure 6.1 shows the original test methods before the refactoring, and Figure 6.2 shows the methods after refactoring. Since the two participating methods in this

```java
//  OffsetDateTimeDeserializerIT.java

public void testDeserialization03() throws IOException {
  Bean1 bean1 = deserialize(Bean1.class, "a: \"x\"\n" +
    "c:\n" +
    "  offsetDateTime: 2017-09-02T10:15:30+01:00");
  assertEquals(OffsetDateTime.of(2017, 9, 2, 10, 15, 30, 0,
    ZoneOffset.ofHours(1)), bean1.c.offsetDateTime);
}

//  ZonedDateTimeDeserializerIT.java

public void testDeserialization03() throws IOException {
  Bean1 bean1 = deserialize(Bean1.class, "a: \"x\"\n" +
    "c:\n" +
    "  zonedDateTime: 2017-09-02T10:15:30+01:00");
  assertEquals(ZonedDateTime.of(2017, 9, 2, 10, 15, 30, 0,
    ZoneOffset.ofHours(1)), bean1.c.zonedDateTime);
}

```

Figure 6.1: Compilation error example: before refactoring

refactoring were in two different classes/files, the JTestParametrizer tool created a template class/file, shown in Figure 6.3.

The invocation of the "deserialize" method in line seven of Figure 6.3 is causing the compile error. The problem is that the "deserialize" method is undefined for the type "DateTimeDeserializerITTestDeserialization03Template".

Both "OffsetDateTimeDeserializerIT" and "ZonedDateTimeDeserializerIT" classes extend the "DeserializerTestBase" class, and the "deserialize" method is a protected method of the "DeserializerTestBase" class. With this in mind, all the methods inside the "OffsetDateTimeDeserializerIT" and "ZonedDateTimeDeserializerIT" classes have access to the "deserialize" method through inheritence, but the newly created "dateTimeDeserializerITTestDeserialization03Template" method will not have access to it, since "DateTimeDeserializerITTestDeserialization03Template" class is not extending the "DeserializerTestBase" class.

3. The problem is that this issue was not considered in the JTestParametrizer code. This issue could have been addressed in "CloneRefactor.java" by checking if both methods in the clone both have at least one invocation of an inherited protected

```java
// OffsetDateTimeDeserializerIT.java

public void testDeserialization03() throws Exception {
  DateTimeDeserializerITTestDeserialization03Template.
    dateTimeDeserializerITTestDeserialization03Template(
    new OffsetDateTimeDeserializerITTestDeserialization03AdapterImpl(),
    " offsetDateTime: 2017-09-02T10:15:30+01:00",
    bean1.c.offsetDateTime);
}

class OffsetDateTimeDeserializerITTestDeserialization03AdapterImpl
    implements DateTimeDeserializerITTestDeserialization03Adapter {
  public Object of(int i1, int i2, int i3, int i4, int i5, int i6,
      int i7, ZoneOffset zoneOffset1) {
    return OffsetDateTime.of(i1, i2, i3, i4, i5, i6, i7, zoneOffset1);
  }
}

// ZonedDateTimeDeserializerIT.java

public void testDeserialization03() throws Exception {
  DateTimeDeserializerITTestDeserialization03Template.
    dateTimeDeserializerITTestDeserialization03Template(
    new ZonedDateTimeDeserializerITTestDeserialization03AdapterImpl(),
    " zonedDateTime: 2017-09-02T10:15:30+01:00",
    bean1.c.zonedDateTime);
}

class ZonedDateTimeDeserializerITTestDeserialization03AdapterImpl
    implements DateTimeDeserializerITTestDeserialization03Adapter {
  public Object of(int i1, int i2, int i3, int i4, int i5, int i6,
      int i7, ZoneOffset zoneOffset1) {
    return ZonedDateTime.of(i1, i2, i3, i4, i5, i6, i7, zoneOffset1);
  }
}

```

Figure 6.2: Compilation error example: refactored test methods

```java
//  DateTimeDeserializerITTestDeserialization03Template.java

public class DateTimeDeserializerITTestDeserialization03Template {
  public static void dateTimeDeserializerITTestDeserialization03Template(
      DateTimeDeserializerITTestDeserialization03Adapter adapter,
      String string1,  Object object1) throws Exception {
    Bean1 bean1=deserialize(Bean1.class,"a: \"x\"\n" + "c:\n" + string1);
    assertEquals(adapter.of(2017,9,2,10,15,30,0,
      ZoneOffset.ofHours(1)),object1);
  }
}

interface DateTimeDeserializerITTestDeserialization03Adapter {
  Object of(int i1, int i2, int i3, int i4, int i5, int i6,
    int i7, ZoneOffset zoneOffset1);
}

```

Figure 6.3: Compilation error example: Template file

method and making a decision based on that, or by adding some conditions in "visit(MethodInvocation node)" method of the "RFVisitor" class.

4. First, I created a new ASTVisitor, shown in Figure 6.4, that would record all the method invocations and check if any of them bind to an inherited protected call. Then I used this visitor in the "CloneRefactor.refactor" method to discard the clones for which all their methods have at least one invocation that binds to a protected method. Figure 6.5 shows this condition.

5. Running this modified version of the JTestParametrizer tool on Bootique benchmark fixed the problem I aimed to fix.

6. Initially, I came up with another solution in step four which passed step five but failed step six since the new condition I added in "CloneRefactor.refactor" was not strong enough, and it discarded a refactoring nominee in Joda-time that had no problems. Technically, discarding a refactoring nominee that is not causing any issues is not an error and will not threaten the correctness of the tool, but it will decrease the number of cases that the tool can refactor.

   So I went back to step four and tried coming up with a new solution that did not have this issue. The new solution, stated in step four, did not affect the results of running the tool on Gson, Joda-time, Commons-lang, and Jfreechart.

```java
public class CheckProtectedCallsVisitor extends ASTVisitor {
  private Set<MethodInvocation> methodInvocations;
  public CheckProtectedCallsVisitor() {
    methodInvocations = new HashSet<MethodInvocation>();
  }

  @Override
  public boolean visit(MethodInvocation methodInvocation) {
    methodInvocations.add(methodInvocation);
    return true;
  }

  public boolean hasProtectedCall() {
    for(MethodInvocation methodInvocation : methodInvocations) {
      if((methodInvocation.getExpression() == null) &&
         ((methodInvocation.resolveMethodBinding().getModifiers() &
           Modifier.PROTECTED) == Modifier.PROTECTED)) {
        return true;
      }
    }
    return false;
  }
}

```

Figure 6.4: JTestParametrizer tool: new visitor

```java
  CheckProtectedCallsVisitor checkVisitor1 =
    new CheckProtectedCallsVisitor();
  CheckProtectedCallsVisitor checkVisitor2 =
    new CheckProtectedCallsVisitor();
  method1.getBody().accept(checkVisitor1);
  method2.getBody().accept(checkVisitor2);
  if(checkVisitor1.hasProtectedCall() &&
     checkVisitor2.hasProtectedCall()) {
    log.info("marked non-refactorable: both methods have calls to
              a protected inherited method in their bodies!");
    countSkip++;
    return;
  }

```

Figure 6.5: JTestParametrizer tool: new condition in CloneRefactor

7. I used this new modified version of the JTestParametrizer tool as the new base and updated the quantitative results using this new version of the tool.

   The numbers in Table 6.1 are based on the latest version of the JTestParametrizer that passed these seven steps.

## 6.4 Undetectable Errors

Even if I do not encounter any run-time errors when running the tool on a benchmark or any compiler errors when building that refactored benchmark, or any failure or errors when running the tests of that refactored benchmark, it does not necessarily mean that running the JParametrizer tool did not cause any problems on that benchmark.

Due to existing bugs in the JTestParametrizer tool, running the JTestParametrizer tool on a benchmark might change the behavior of a test method, and even though that test method might still pass, it will not cover what it was supposed to cover. This problem can not be detected by examining the quantitative results, and the only way to detect this problem is by going through every refactored test manually and check the correctness of the changes.

### 6.4.1 Undetectable Issue Example

Here, I will provide an example of this problem in the Jimfs benchmark that I noticed while doing a manual inspection on the refactoring cases of that benchmark. When trying to refactor the “testNormalizeNfc_pattern” and “testNormalizeNfd_pattern” methods in “PathNormalizationTest” class, displayed in Figure 6.6, the tool somehow ignored the difference between “NFC” and “NFD”, and assumed that both of them are “NFC”, and came up with the the refactored tests shown in Figure 6.7.

Now looking at Figure 6.7, we can see that both refactored test methods cover the scenario that the “testNormalizeNfc_pattern” was covering before. There will not be any errors, and both refactored tests will pass successfully, but this is an issue nonetheless since we lost the “testNormalizeNfd_pattern” test after the refactoring.

I followed the debugging procedure explained in Section 6.3 and found out that this problem was caused by a simple mistake in the “RFVisitor.visit(SimpleName node)” method. I continued the procedure and fixed that mistake by adding a new line in that

```java
//  PathNormalizationTest.java

public void testNormalizeNfc_pattern() {
  normalizations = ImmutableSet.of(NFC);
  assertNormalizedPatternMatches("foo", "foo");
  assertNormalizedPatternDoesNotMatch("foo", "FOO");
  assertNormalizedPatternDoesNotMatch("FOO", "foo");
  assertNormalizedPatternMatches("Am\u00e9lie", "Ame\u0301lie");
  assertNormalizedPatternDoesNotMatch("Am\u00e9lie", "AME\u0301LIE");
}

public void testNormalizeNfd_pattern() {
  normalizations = ImmutableSet.of(NFD);
  assertNormalizedPatternMatches("foo", "foo");
  assertNormalizedPatternDoesNotMatch("foo", "FOO");
  assertNormalizedPatternDoesNotMatch("FOO", "foo");
  assertNormalizedPatternMatches("Am\u00e9lie", "Ame\u0301lie");
  assertNormalizedPatternDoesNotMatch("Am\u00e9lie", "AME\u0301LIE");
}
```

Figure 6.6: Undetectable error example: before refactoring

```java
//  PathNormalizationTest.java

public void testNormalizeNfc_pattern() {
  this.pathNormalizationTestTestNormalizeTemplate();
}

public void testNormalizeNfd_pattern() {
  this.pathNormalizationTestTestNormalizeTemplate();
}

public void pathNormalizationTestTestNormalizeTemplate() {
  normalizations = ImmutableSet.of(NFC);
  assertNormalizedPatternMatches("foo", "foo");
  assertNormalizedPatternDoesNotMatch("foo", "FOO");
  assertNormalizedPatternDoesNotMatch("FOO", "foo");
  assertNormalizedPatternMatches("Am\u00e9lie", "Ame\u0301lie");
  assertNormalizedPatternDoesNotMatch("Am\u00e9lie", "AME\u0301LIE");
}
```

Figure 6.7: Undetectable error example: refactored test methods

```java
//  PathNormalizationTest.java

public void testNormalizeNfc_pattern() {
  this.pathNormalizationTestTestNormalizeTemplate(NFC);
}

public void testNormalizeNfd_pattern() {
  this.pathNormalizationTestTestNormalizeTemplate(NFD);
}

public void pathNormalizationTestTestNormalizeTemplate(
    PathNormalization pathNormalization1) {
  normalizations = ImmutableSet.of(pathNormalization1);
  assertNormalizedPatternMatches("foo", "foo");
  assertNormalizedPatternDoesNotMatch("foo", "FOO");
  assertNormalizedPatternDoesNotMatch("FOO", "foo");
  assertNormalizedPatternMatches("Am\u00e9lie", "Ame\u0301lie");
  assertNormalizedPatternDoesNotMatch("Am\u00e9lie", "AME\u0301LIE");
}
```

Figure 6.8: Undetectable error example: fixed refactored test methods

method. This was just a simple bug in the JTestParametrizer tool, and I did not make any conceptual changes to fix this issue.

The correct version of this refactoring case that I got after running the modified version of the JTestParametrizer tool on the benchmark is shown in Figure 6.8.

# Chapter 7

# JTestParametrizer Qualitative Results and Discussion

When working on a relatively large project, whether in PL or SE research, there will be a point where we need feedback to determine whether what we have is the right thing and to decide how to continue, otherwise we will waste a lot of time and energy.

In this chapter, I will explain the processes that I used for getting feedback on the quality of refactorings, how I used this feedback to modify the JTestParametrizer tool, how I gathered feedback on what would be the best way to extend the JTestParametrizer tool project, and finally, what I learned from this experience.

Up until this point, we only worked on the correctness of the JTestParametrizer tool. We tried fixing the existing errors and making sure that running the tool on a benchmark will not change the behavior of the test cases. However, the main idea behind the JTestParametrizer tool was to enhance the quality of test cases.

There were many potential errors in the JTestParametrizer tool that we have not addressed yet (represented in Table 6.1), but we decided that we should switch to validation of the quality of refactoring first, as it might affect the errors that we are getting.

It would be better to modify the tool to get something that is desirable and accepted by potential users and then work on the correctness of that version instead of fixing all of the errors in the version of the tool that we have now and then trying to change the tool to something desirable by the potential users of the tool. This will save time because there might be mistakes in the tool that will be eliminated by modifying the tool first, and we will not need to address those mistakes anymore.

## 7.1 The Questionnaire

Using a questionnaire is one way of getting feedback. It is inexpensive, easy to analyze, and scalable. However, a questionnaire does not come with any guarantee of quality results because it is very hard to find the right target audience for a technical questionnaire. Furthermore, even if we do so, there is no guarantee of the target audience being moderately engaged and putting any time or effort into the answers.

We decided to create a questionnaire to get feedback on the quality of the refactored test cases. Our questionnaire included:

1. seven demographic questions to determine the familiarity of that person with SE/PL research in general;
2. seven demographic questions to determine the familiarity of that person with our research, unit tests, and refactoring; and,
3. twenty-four questions on three refactoring cases that I picked from the refactored Gson benchmark compared to the original version

These twenty-four questions included comparisons based on maintainability, conciseness, readability, understandability, extensibility, and repetition criteria, and overall quality of the refactorings. I hand-picked those three refactoring cases since they represented three different refactoring groups in the Gson benchmark.

However, before submitting the questionnaire for the ethics clearance, we decided that it would be better if we use our time to seek feedback on the quality of the refactorings in a different way. This decision was mainly due to not having a great target audience for our questionnaire, potentially dealing with low-quality feedback that might not help our assessment, and knowing that we can get higher-quality feedback using other methods such as self-assessment and pull requests.

## 7.2 Manual Quality Evaluation

Manual quality evaluation is another way to get feedback on the quality of refactorings. A key advantage of this method is that I do not need assistance from others, and I do not need to wait for an unknown amount of time to get this feedback. Furthermore, I can force myself to study the domain of benchmarks before assessing, which I could not force using a questionnaire. This will guarantee a lower bound on the quality of the feedback that I get using this method. Also, by using this method, I will be in charge of the level of detail that I want in the feedback.

However, manual quality evaluation is a self-assessment, and like any other type of self-assessments, it could be vulnerable to unconscious bias. I asked for my supervisor's help in every assessment to decrease the unconscious bias that I might have had.

### 7.2.1 The Process of Manual Quality Evaluation

We used Gson (f319c1b version), Jimfs, and Bootique (9939bc6 version) for this work. First, I ran the JTestParametrizer on these three benchmarks, and then I discarded all the refactoring cases causing errors and failures on the test runs, which was only one case in the Bootique benchmark. Doing this resulted in 18 refactoring cases for Gson, five refactorings cased for Jimfs, and ten refactoring cases for Bootique.

Then we went over each of these 33 refactoring cases one by one, discussed the quality of that case for a few minutes, and then based on the effect of the changes on maintainability, conciseness, readability, understandability, extensibility, and repetition criteria, we allocated a rating between 0 and 10 to that refactoring case.

In our rating system, five represented a refactoring that did not change the quality of the test case and that the refactored version had the same overall quality as before. Anything above five meant that the refactoring was improving the overall quality of the test method. And anything below five meant that the refactoring was decreasing the overall quality of the code.

### 7.2.2 Rating Example

Here, I will go over one of the refactoring cases and explain how I evaluated this case. This refactoring case belongs to the Gson benchmark and is between "testExceptionWithoutCause" and "testErrorWithoutCause" methods which are both in the "ThrowableFunctionalTest" class.

Looking at Figure 7.1, we can see that, before running the tool, the only difference between the two test methods is in a class type ("RuntimeException.class" vs. "OutOfMemoryError.class"), and the rest is the same. Now, in Figure 7.2, JTestParametrizer tool creates a parametrized method (throwableFunctionalTestTestWithoutCauseTemplate) that will allows us to add similar test cases with different class types.

```java
//  ThrowableFunctionalTest.java

public void testExceptionWithoutCause() {
  RuntimeException e = new RuntimeException("hello");
  String json = gson.toJson(e);
  assertTrue(json.contains("hello"));

  e = gson.
    fromJson("{'detailMessage':'hello'}", RuntimeException.class);
  assertEquals("hello", e.getMessage());
}

public void testErrorWithoutCause() {
  OutOfMemoryError e = new OutOfMemoryError("hello");
  String json = gson.toJson(e);
  assertTrue(json.contains("hello"));

  e = gson.
    fromJson("{'detailMessage':'hello'}", OutOfMemoryError.class);
  assertEquals("hello", e.getMessage());
}
```

Figure 7.1: Rating example: before refactoring

The refactored code is slightly more maintainable because the shared part of the two test methods is in one place, and when we decide to change something about the scenario of the test, we do not need to worry about keeping the shared part consistent. The name of the template method is not ideal, but changing that name to a more meaningful and shorter name such as “testThrowableWithoutCauseTemplate” will make the refactored case more concise than the original version. The readability and understandability of code are slightly worse due to the increase in the complexity of the code. However, the extensibility of the code is higher since now we can easily add a similar test for a new class type. Furthermore, the refactored version has a lower repetition in the code.

Based on the criteria I mentioned, I allocated a 5 rating for the out-of-the-box version of this refactoring case, but a potential 6. That is, if we change the template method’s name to something more meaningful, the overall quality of the refactored code will be slightly higher than the overall quality of the original version.

```java
//  ThrowableFunctionalTest.java

public void testExceptionWithoutCause() throws Exception {
  this.throwableFunctionalTestTestWithoutCauseTemplate(
    RuntimeException.class);
}

public void testErrorWithoutCause() throws Exception {
  this.throwableFunctionalTestTestWithoutCauseTemplate(
    OutOfMemoryError.class);
}

public <TThrowable extends Throwable> void
    throwableFunctionalTestTestWithoutCauseTemplate(
    Class<TThrowable> clazzTThrowable) throws Exception {

  TThrowable e = clazzTThrowable.getDeclaredConstructor(String.class).
    newInstance("hello");
  String json = gson.toJson(e);
  assertTrue(json.contains("hello"));
  e = (TThrowable) gson.
    fromJson("{'detailMessage':'hello'}", clazzTThrowable);
  assertEquals("hello", e.getMessage());
}
```

Figure 7.2: Rating example: after refactoring

```java
//  Gson benchmark  -  JavaUtilConcurrentAtomicTest.java

public void testAtomicInteger() throws Exception {
  AtomicInteger target = gson.fromJson("10", AtomicInteger.class);
  assertEquals(10, target.get());
  String json = gson.toJson(target);
  assertEquals("10", json);
}

public void testAtomicLong() throws Exception {
  AtomicLong target = gson.fromJson("10", AtomicLong.class);
  assertEquals(10, target.get());
  String json = gson.toJson(target);
  assertEquals("10", json);
}
```

Figure 7.3: Behavioral example: before refactoring

### 7.2.3 Behavior Parameterization Evaluation

One of the parametrization techniques that Jun Zhao used in JTestParametrizer is Behavior Parameterization. In his thesis, Jun Zhao described Behavior parameterization as "parameterizing methods with behavioral differences, that is, method invocation calls having different signatures. These method calls must still have the same length argument lists but may have different method names or perhaps different receiver objects.".

Based on his thesis and JTestParameterizer code, he creates a common interface to collect all unified behavioral methods with compatible signatures. He adds an argument, "adapter", to the parameterized template method with the common interface type. Finally, for each pair of different method invocations in the clone pair, he extracts an interface method declaration into the common interface.

For example, in Figure 7.3, since the type of "target" is different in the two test methods, the "target.get()" method will invoke different methods. Hence this refactoring case has a behavioral difference.

Now, as shown in Figure 7.4, to refactor this clone pair, the tool creates an interface and two new classes, plus the template method. We allocated a 0 rating to this refactoring case as it clearly decreases the overall quality of the code due to decreasing the conciseness, lowering readability, lowering understandability, massively increasing the complexity, and potentially lowering the maintainability of the code.

```java
// Gson benchmark - JavaUtilConcurrentAtomicTest.java

public void testAtomicInteger() throws Exception {
  this.javaUtilConcurrentAtomicTestTestAtomicTemplate(
    new JavaUtilConcurrentAtomicTestTestAtomicIntegerAdapterImpl(),
    AtomicInteger.class);
}

public void testAtomicLong() throws Exception {
  this.javaUtilConcurrentAtomicTestTestAtomicTemplate(
    new JavaUtilConcurrentAtomicTestTestAtomicLongAdapterImpl(),
    AtomicLong.class);
}

public <TAtomic extends Number> void
    javaUtilConcurrentAtomicTestTestAtomicTemplate(
      JavaUtilConcurrentAtomicTestTestAtomicAdapter<TAtomic> adapter,
      Class<TAtomic> clazzTAtomic) throws Exception {

  TAtomic target = (TAtomic) gson.fromJson("10", clazzTAtomic);
  assertEquals(10, adapter.get(target));
  String json = gson.toJson(target);
  assertEquals("10", json);
}

interface JavaUtilConcurrentAtomicTestTestAtomicAdapter<TAtomic> {
  long get(TAtomic tAtomic1);
}

class JavaUtilConcurrentAtomicTestTestAtomicIntegerAdapterImpl
  implements
    JavaUtilConcurrentAtomicTestTestAtomicAdapter<AtomicInteger> {
  public long get(AtomicInteger target) {
    return target.get();
  }
}

class JavaUtilConcurrentAtomicTestTestAtomicLongAdapterImpl
  implements
    JavaUtilConcurrentAtomicTestTestAtomicAdapter<AtomicLong> {
  public long get(AtomicLong target) {
    return target.get();
  }
}
```

Figure 7.4: Behavioral example: after refactoring

After manual evaluation of 33 refactoring cases in Gson, Jimfs, and Bootique, we realized that the 8 cases with behavioral parametrizations had the lowest ratings out of all the cases, with a considerable gap, which was due to the lack of conciseness, very high complexity, lower readability, lower understandability, and arguably lower maintainability that the behavioral parametrization produces, which indicated that behavioral parametrization decreases the overall quality of the code.

We decided to modify the JTestParametrizer tool and add a new configuration that skips the cases that need the behavioral parametrization technique.

I implemented this option by adding a new class called "SimilarityAspects" in the "template" package that keeps track of the aspects of differentiation between the methods in a refactoring case (such as behavioral difference). Then I used an instance of this new class as a public variable in the "RFTemplate" class. Next, I set the behavioral differences to true in the "RFVisitor.visit(MethodInvocation node)" method in a condition that determines if behavioral parametrization is needed. Finally, I added a condition to skip the clones with behavioral differences set to true in the "CloneRefactor.refactor" method.

### 7.2.4 Discarding Behavioral Effect on Quantitative Results

Table 7.1 lists the quantitative results of running the JTestParametrizer tool, discarding behavioral clones on the benchmarks. The process that I used to get these results is the same process that I used in Section 6.1, with the only difference being the addition of the "-skipBehaviourals" option at the end of the arguments that I passed to the JTest-Parametrizer tool.

| Repository | Version | Nominees | Run-time errors | Compile-time errors | Refactored | Tests run | Failures | Errors | Skipped |
|---|---|---|---|---|---|---|---|---|---|
| Gson | f649e05 | 39 | 0 | 0 | 12 | 1050 | 0 | 0 | 1 |
| Gson | f319c1b | 42 | 0 | 0 | 13 | 1063 | 0 | 0 | 1 |
| Jimfs | 3c9d8ba | 45 | 0 | 0 | 4 | 5834 | 0 | 0 | 0 |
| Bootique | d0648eb | 22 | 0 | 0 | 8 | 231 | 0 | 1 | 0 |
| Bootique | 9939bc6 | 23 | 0 | 0 | 9 | 228 | 0 | 1 | 0 |
| Joda-time | 0ae5311 | 261 | 1 | 0 | 39 | 4224 | 3 | 4 | 0 |
| Joda-time | 27edfff | 260 | 1 | 0 | 39 | 4240 | 3 | 4 | 0 |
| Commons-lang | 425d808 | 154 | 1 | 8 | 13 | 4068 | 0 | 0 | 5 |
| Commons-io | e4ff4a5 | 32 | 0 | 1 | 4 | 1852 | 0 | 0 | 6 |
| Commons-collections | 7d8b979 | 47 | 1 | 1 | 2 | 16923 | 1 | 0 | 4 |
| Jfreechart | d03e68a | 124 | 0 | 1 | 13 | 2176 | 0 | 0 | 0 |
| Netty/Codec-http | e69107c | 47 | 0 | 3 | 8 | - | - | - | - |
| Netty/Buffer | e69107c | 20 | 0 | 2 | 1 | - | - | - | - |
| Checkstyle | 6cbc1dc | 130 | 0 | 1 | 16 | 3528 | 0 | 1 | 0 |
| Git-commit-id-maven-plugin | 4a1ac8f | 4 | 0 | 0 | 1 | 214 | 0 | 0 | 1 |
| Docker-maven-plugin | 84020ac | 7 | 0 | 0 | 1 | 59 | 0 | 0 | 0 |
| Maven/Maven-core | 3fabb63 | 26 | 0 | 0 | 11 | 388 | 0 | 0 | 4 |
| Mybatis-3 | 1d82865 | 26 | 0 | 0 | 0 | 1675 | 0 | 0 | 14 |

Table 7.1: Quantitative Results With Skip Behavioral Configuration

Furthermore, Table 7.2 shows the comparison between the test results of the tool with (Table 7.1) and without (Table 6.1) the discard behavioral configuration.

The number of tests run and skipped did not change for any of the benchmarks after adding the skip behavioral option, so I did not include those columns in this table. The four numerical columns on the left show the results, including behavioral refactorings, and the four columns on the right show the results without behavioral refactoring.

By examining Table 7.2, we can see that running the tool with the new configuration causes a lot fewer problems, failures, and errors. This might be due to discarding some refactoring nominees that might trigger a fault in the JTestParametrizer tool that other nominees do not. However, it is also highly likely that some of these errors were due to mistakes in the behavior parametrization code, and now that we decided to skip those cases, we do not need to deal with these mistakes anymore.

Furthermore, based on Table 7.2, before discarding the behavioral cases, we had 480 refactoring cases in total, which decreased to 194 after. However, this decrease was not uniform for all benchmarks since some of the benchmarks like Joda-time lost more than half (65% for Joda-time) of their refactoring cases, whereas some like Gson lost fewer than 30% of their cases. Also, 27 cases (not included in the 480) were causing compile errors in the benchmarks after running the tool without the discard behavioral configuration. After running the tool with the discard behavioral configuration, this number decreased to 17 cases (not included in 194). This means that 10 of the compile errors in the benchmarks after running the tool were caused by refactoring nominees with behavioral differences, but this does not necessarily mean that all ten mistakes leading to these compile errors were in the behavior parametrization code.

| Benchmark | | Without new configuration | | | | | With new configuration | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| Repository | Version | Run-time errors | Compile-time errors | Refactored | Failures | Errors | Run-time errors | Compile-time errors | Refactored | Failures | Errors |
| Gson | f649e05 | 0 | 0 | 17 | 0 | 0 | 0 | 0 | 12 | 0 | 0 |
| Gson | f319c1b | 0 | 0 | 18 | 0 | 0 | 0 | 0 | 13 | 0 | 0 |
| Jimfs | 3c9d8ba | 0 | 1 | 5 | 0 | 0 | 0 | 0 | 4 | 0 | 0 |
| Bootique | d0648eb | 0 | 2 | 10 | 0 | 1 | 0 | 0 | 8 | 0 | 1 |
| Bootique | 9939bc6 | 0 | 2 | 11 | 0 | 1 | 0 | 0 | 9 | 0 | 1 |
| Joda-time | 0ae5311 | 1 | 1 | 112 | 5 | 6 | 1 | 0 | 39 | 3 | 4 |
| Joda-time | 27edfff | 1 | 1 | 112 | 5 | 6 | 1 | 0 | 39 | 3 | 4 |
| Commons-lang | 425d808 | 1 | 8 | 62 | 2 | 4 | 1 | 8 | 13 | 0 | 0 |
| Commons-io | e4ff4a5 | 0 | 3 | 9 | 0 | 0 | 0 | 1 | 4 | 0 | 0 |
| Commons-collections | 7d8b979 | 1 | 1 | 4 | 0 | 0 | 1 | 1 | 2 | 1 | 0 |
| Jfreechart | d03e68a | 0 | 1 | 65 | 2 | 3 | 0 | 1 | 13 | 0 | 0 |
| Netty/Codec-http | e69107c | 0 | 3 | 12 | - | - | 0 | 3 | 8 | - | - |
| Netty/Buffer | e69107c | 0 | 3 | 2 | - | - | 0 | 2 | 1 | - | - |
| Checkstyle | 6cbc1dc | 0 | 1 | 25 | 0 | 2 | 0 | 1 | 16 | 0 | 1 |
| Git-commit-id-maven-plugin | 4a1ac8f | 0 | 0 | 2 | 0 | 0 | 0 | 0 | 1 | 0 | 0 |
| Docker-maven-plugin | 84020ac | 0 | 0 | 2 | 0 | 0 | 0 | 0 | 1 | 0 | 0 |
| Maven/Maven-core | 3fabb63 | 0 | 0 | 12 | 0 | 0 | 0 | 0 | 11 | 0 | 0 |
| Mybatis-3 | 1d82865 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

Table 7.2: With and Without Skip Behavioral Configuration Comparison

### 7.2.5 Manual Quality Evaluation Discussion

To reiterate, we went over all 33 refactoring cases in Gson, Jimfs, and Bootique and rated each one on a scale of 0 to 10 based on their overall quality. Then we examined the ratings and realized that the eight behavioral refactoring cases had the lowest ratings. After examining the behavioral parametrization process, we determined that it has a high chance of decreasing the overall quality of the code, which led to our decision to add a configuration to the JTestParametrizer tool that discards the behavioral refactoring nominees.

However, even after removing those eight cases, the average rating of the other 25 cases was slightly less than five. This means that even if we discard the behavioral cases, a lot of the refactoring cases will decrease the quality of the code instead of increasing it. Furthermore, this was all based on the ratings that we came up with, and there was a possibility that the developers of the projects will be more reluctant to accept some of the cases that we rated higher than five.

## 7.3 Selecting Pull Requests

Creating and submitting pull requests is yet another method of getting feedback on the quality of the refactoring cases or any proposed program transformation in general.

A significant advantage of this method is getting feedback from potential users of the JTestParametrizer tool, which increases its value since nobody's opinion about the changes in a code is as important as the opinion of developers of that code. Unlike manual quality evaluation, there will not be any unconscious bias helping the cases. If anything, there might be an unconscious bias against the refactoring cases as developers might be reluctant to change their code unless the quality of the code after changes is significantly higher.

However, this method of getting feedback has some disadvantages too. Selecting the right set of changes for an acceptable pull request is a time-consuming process. Some considerations have to be acknowledged, and ignoring them will lead to consequences. Pull requests are external feedback, and it will take an unknown amount of time to receive them. To select the right set of changes for a pull request, we need to have complete knowledge and understanding of the domain of the changes. Even though the feedback we are getting using this method is precious, we have no control over the level of detail of the feedback as it could be just a accept/reject or a more detailed message.

After the discussion in Section 7.2.5, we decided that the best way to determine whether developers would accept the refactoring cases produced by the JTestParametrizer tool is to seek the developers' feedback by opening pull requests. Furthermore, by manually modifying the cases for a pull request and studying the feedback we get for them, we might receive feedback on what JTestParametrizer should do to be more beneficial for the developers.

### 7.3.1 Important Considerations

When we submit a pull request, a developer will spend their time reviewing our pull request. We must acknowledge some considerations when selecting changes for a pull request to avoid wasting the reviewers' time. Here are some of those considerations in the context of our work:

1. We should ensure that the changes that we submit do not cause any errors or failures. This includes both semantic and syntactic errors and failures. Also, we need to manually ensure that the selected modification to the test methods does not change their behavior.
2. We should only select the modifications that, in our opinion, increase the quality of the code. Submitting pull requests is a method for helping the developers fix bugs in their code or increase the quality of their code, and if we send modifications that, even based on our own opinion, decrease the quality of the code only to see what their feedback will be, then that would be unethical and unacceptable.
3. We should avoid selecting many changes for one pull request, and instead, we should select cases that are representative of a group of cases. This is due to two reasons. First, selecting many changes lowers the chance of a pull request getting accepted. Second, if many of the changes in a pull request are very similar, we are potentially wasting the reviewers' time.

The considerations in items 2 and 3, imply manual quality evaluation. In other words, to meet the quality considerations, we have to manually evaluate the refactoring cases first. This is a disadvantage of the pull request method for getting feedback, but it is necessary, and avoiding it and just selecting cases for a pull request without these considerations is unethical and can potentially have severe consequences.

### 7.3.2 Process of Selecting Pull Requests

To create a pull request for a benchmark, I followed this procedure:

1. I manually evaluated all the refactoring cases that I got for that benchmark after running the JTestParametrizer tool.

2. I discarded all the cases that were causing any syntactic or semantic errors or failures.

3. I recorded and then discarded all the refactoring cases that I rated less than 5 in the manual quality evaluation process, which were the refactoring cases that decreased the code's overall quality.

4. I went over the remaining cases, and if two or more refactoring cases were very similar, I only selected the case with the highest rating and eliminated the other ones.

5. Finally, I went over the remaining cases one more time and recorded and then fixed existing minor issues. For instance, if JTestParametrizer proposed a method name that was uninformative or unhelpful, then I would record the problem and then try to pick a more meaningful name for that template method.

### 7.3.3 Minor Manual Modifications

The modifications specified in step 5 were truly minor since we wanted to get feedback on the quality of the refactoring cases that the current version of the JTestParametrizer tool produced and not on what it could produce. The reasoning behind these changes was to provide a pull request with slightly better quality since, as we mentioned before, submitting the raw version of that refactoring case with minor issues would be unethical, and it would lower the chance of acceptance.

To provide an example of these minor modifications, I used the same refactoring case that I used in Figure 7.1 and Figure 7.2. As I explained in Section 7.2.2, Figure 7.1 shows the test methods before running the JTestParametrizer tool, and Figure 7.2 shows the test methods and the newly created template method after running the JTestParametrizer tool. In Figure 7.5, we can see the only change I made in the proposed refactoring was to modify the name of the template method from "throwableFunctionalTestTestWithoutCauseTemplate" to "testThrowableWithoutCauseTemplate" which is a better-suited name.

```java
//  ThrowableFunctionalTest.java

public void testExceptionWithoutCause() throws Exception {
  this.testThrowableWithoutCauseTemplate(RuntimeException.class);
}

public void testErrorWithoutCause() throws Exception {
  this.testThrowableWithoutCauseTemplate(OutOfMemoryError.class);
}

public <TThrowable extends Throwable> void
    testThrowableWithoutCauseTemplate(
    Class<TThrowable> clazzTThrowable) throws Exception {
  TThrowable e = clazzTThrowable.getDeclaredConstructor(String.class).
    newInstance("hello");
  String json = gson.toJson(e);
  assertTrue(json.contains("hello"));
  e = (TThrowable) gson.
    fromJson("{'detailMessage':'hello'}", clazzTThrowable);
  assertEquals("hello", e.getMessage());
}
```

Figure 7.5: Rating example: after minor manual modifications

### 7.3.4 Representative Cases

In step 4, I explained that for the refactoring cases that were very similar, I only selected the one with the highest rating as the representative and discarded the rest. Here, I will go over the factors that made the test refactorings similar.

1. Jun Zhao explained in his thesis that the JTestParametrizer tool uses three different parametrization techniques: "Type Parameterization", "Data Parameterization", and "Behavior Parameterization". We saw an example of "Behavior Parameterization" in Figure 7.4, an example of "Data Parameterization" in Figure 6.8, and an example of Type Parameterization in Figure 7.2.

   Since we decided to discard the refactoring nominees with behavioral differences, we will have four options for each refactoring case as it might or might not have either or both type and data differences. If two of the refactoring cases fall into the same category regarding having or not having type and data differences, then they will be considered somewhat similar.

2. The test methods involved in a refactoring case can all be in the same class or different classes. If the test methods are not all in the same class, then the JTestParametrizer tool creates a new template class and creates the template method as a public static method in this new template class. The refactoring case used in Figure 6.3 would be an example of this situation.

   However, when all the test methods are in the same class, the JTestParametrizer tool creates the template method as a standard public non-static method in the same class. Every refactoring case example so far, except for the one used in Figure 6.3, is an example of this situation.

   Now, if two refactoring cases both fall into the same category regarding all the involved test methods being or not being in the same class, then we would consider those two refactoring cases somewhat similar.

If two refactoring cases are somewhat similar based on both of these categories, we would consider those refactoring cases similar. That is when one could potentially represent both in the pull request.

### 7.3.5 Discussion

To reiterate, in this section, I explained the importance of the feedback that we get from pull requests. I mentioned the three crucial considerations when selecting a pull request:

ensuring correctness, ensuring quality, and avoiding repetitive cases. Finally, I went over the process I used to select a pull request and explained the five steps: manual evaluation, discarding cases with errors and failures, discarding low-rated cases, selecting one representative for similar cases, and minor manual modification.

Looking forward, when we are trying to get feedback on what JTestParametrizer could do instead of what it is doing now, there will be a sixth step in the process of selecting the pull requests. In that sixth step, we will manually implement the feature that we have in mind on the output of step five, which will be a significant manual modification, unlike step five. Then based on the feedback that we get for that refactoring, we will decide if we want to pursue implementing that feature in the JTestParametrizer tool or not. So the sixth step will be different for every feature since it is directly related to the feature.

## 7.4 Submitting Pull Requests

In the previous section, I discussed the selection of pull requests. Now, if we submit the same refactoring cases that we got from the process in Section 7.3.2, though the feedback will be valuable, it will only help us to evaluate the quality of the current version of the JTestParametrizer tool.

However, submitting pull requests is a powerful method for getting feedback and can help to a much higher extent. By manually modifying the output of the process in Section 7.3.2, we can implement the effects of a feature that is not currently in the JTestParametrizer tool. Then based on the feedback that we get for that manually modified pull request, we will determine whether we should pursue that feature in the tool or not. There is precedent for that in the startup world of a Minimum Viable Product. Different world, but similar concept.

This can save much time since the other option is implementing the feature in the tool and evaluating it afterward. Furthermore, the feedback we get from a manually modified pull request will guide us in creating the subsequent manually modified pull request.

This leads to creating multiple pull requests for different features. However, we should not use the pull request requests as a black box or an oracle without any considerations. We must always state our clear intention for each pull request in its description and ensure that, based on our opinion, each pull request is improving the code's overall quality. This is a crucial consideration since a developer will review each pull request and put their time and effort into examining it.

This section will explain each pull request's idea, how I implemented it, the feedback I got for that pull request, how I interpreted the feedback, and how I used what I learned to create the subsequent pull request.

### 7.4.1 Jimfs Pull Request

For the Jimfs pull request, the idea was to use the raw output of the pull request selection process in Section 7.3.2. Furthermore, since the template method names that we got for the remaining cases after step 4 were acceptable, we did not do any minor modifications either, meaning that step 5 was unnecessary for this pull request.

After finishing the step 4, there remained four refactoring cases modifying three files. We used all of these cases for this pull request. The goal of this pull request was to get feedback on the quality of the refactoring cases that the current version of the JTest-Parametrizer tool produces.

We submitted the Jimfs pull request and the feedback that we got for it stated:

> "Thanks for the PR. I'm not going to merge it at the moment because it's not clear to me that this is an improvement... reducing duplication is generally good, but not as much so in tests where some amount of duplication can make the expected behavior more clear in each test case. But I'll leave it open for the moment because it's possible this could benefit from reducing duplication to some extent."

This was valuable feedback regarding the details of the reasoning behind the final decision. It was not clear to the reviewer that the changes improved the overall quality of the code, which was not unexpected since we rated the refactoring cases used for this PR around 6 in the manual evaluation process. This meant that we felt that the PR somewhat improved the quality of the code. The reviewer stated that they appreciated the reduction of duplication, but its effect was not enough to compensate for the decrease in the understandability of the expected behavior of the tests. Nevertheless, they kept the PR open as it might potentially be beneficial in reducing the duplication to some extent.

The feedback was clear. The reduction of duplication, improvement in extensibility, or other advantages of our modifications, should clearly compensate for the disadvantages, such as the decrease in understandability of the expected behavior of the tests, for it to be a good set of modifications. This could be done either by strengthening the advantages (e.g., more duplication reduction) or reducing the disadvantages (e.g., easier to understand).

### 7.4.2 First Gson Pull Request

After the feedback for Jimfs PR, we decided to test two ideas that increase the advantages of our modifications using manual modification in a new PR.

The first idea was to improve the extensibility of the refactoring cases using further parametrizations. This includes parameterizing the data values that are the same in all test methods involved in a refactoring case, but if they were not the same, the refactoring was still possible with parametrization.

The example that we used for this idea is based on the same refactoring nominee that we used in Figure 7.1, Figure 7.2, and Figure 7.5. So after step 5 in the pull request selection process, we noticed that if the values "hello", and "'detailMessage':'hello'", were different between the two test methods involved, this refactoring case would have still worked by parametrizing two new variables. Figure 7.6 shows the final version that we used for this pull request after manually parametrizing two new variables in Figure 7.5 which would achieve the effect of this idea on the code. We also tried to make the exception handling more straightforward in this final version to maintain understandability.

The second idea was to improve the advantage of reduction of duplication by manually involving three test methods in a refactoring case instead of two. With this idea, we can reduce more existing duplications in the code, and the template method created will potentially be more generalized since it has to be parametrized for all the differences between the shared part of the three test methods. To implement this idea, we manually examined the refactoring cases for the Gson benchmark and checked whether another test method had similar parts to the template methods created for the refactoring cases.

An earlier student had worked on refactoring triplets but found that it was quite hard. However, it was was worthwhile to try to do it manually to see whether it was worthwhile.

We figured that we could use the refactoring example for "CustomSerializerTest" and "testSubClassSerializerInvokedForBaseClassFieldsHoldingSubClassInstances", and manually change it so we can use the modified template method for reducing the duplication in the "testBaseClassSerializerInvokedForBaseClassFieldsHoldingSubClassInstances" method as well. Figure 7.7 shows the raw output of the JTestParametrizer tool for that refactoring case, and Figure 7.8 shows the final version that we used for this PR after applying the manual modifications.

We used these two refactoring cases in this pull request. This was the second pull request that we submitted, and unlike the first one, this one was to get feedback on a version of the JTestParametrizer tool that we have not implemented yet. Here is the feedback that we got:

```java
//  ThrowableFunctionalTest.java

public void testExceptionWithoutCause() {
  this.testThrowableWithoutCauseTemplate(
    RuntimeException.class, "hello", "{'detailMessage':'hello'}");
}

public void testErrorWithoutCause() {
  this.testThrowableWithoutCauseTemplate(
    OutOfMemoryError.class, "hello", "{'detailMessage':'hello'}");
}

public <TThrowable extends Throwable> void
    testThrowableWithoutCauseTemplate(
    Class<TThrowable> clazzTThrowable, String msg, String jsonString) {
  try {
    TThrowable e = clazzTThrowable.
      getDeclaredConstructor(String.class).newInstance(msg);
    String json = gson.toJson(e);
    assertTrue(json.contains(msg));
    e = (TThrowable) gson.fromJson(jsonString, clazzTThrowable);
    assertEquals(msg, e.getMessage());
  } catch (Exception e) {
    e.printStackTrace();
    throw new RuntimeException();
  }
}
```

Figure 7.6: First idea example: after manual modification

```java
//  CustomSerializerTest.java

public void testBaseClassSerializerInvokedForBaseClassFields()
    throws Exception {
  this.customSerializerTestTestClassSerializerInvokedForBaseClass-
    -FieldsSubClassTemplate(TestTypes.Base.class, BaseSerializer.NAME);
}

public void testSubClassSerializerInvokedForBaseClassFieldsHolding-
    -SubClassInstances() throws Exception {
  this.customSerializerTestTestClassSerializerInvokedForBaseClass-
    -FieldsSubClassTemplate(TestTypes.Sub.class, SubSerializer.NAME);
}

public void testBaseClassSerializerInvokedForBaseClassFieldsHolding-
    -SubClassInstances() {
  Gson gson = new GsonBuilder()
    .registerTypeAdapter(Base.class, new BaseSerializer()).create();
  ClassWithBaseField target = new ClassWithBaseField(new Sub());
  JsonObject json = (JsonObject) gson.toJsonTree(target);
  JsonObject base = json.get("base").getAsJsonObject();
  assertEquals(BaseSerializer.NAME,
    base.get(Base.SERIALIZER_KEY).getAsString());
}

public <TBase> void customSerializerTestTestClassSerializerInvokedFor-
    -BaseClassFieldsSubClassTemplate(Class<TBase> clazzTBase,
    String string1) throws Exception {
  Gson gson = new GsonBuilder()
    .registerTypeAdapter(Base.class, new BaseSerializer())
    .registerTypeAdapter(Sub.class, new SubSerializer()).create();
  ClassWithBaseField target =
    new ClassWithBaseField((TestTypes.Base) clazzTBase.newInstance());
  JsonObject json = (JsonObject) gson.toJsonTree(target);
  JsonObject base = json.get("base").getAsJsonObject();
  assertEquals(string1, base.get(Base.SERIALIZER_KEY).getAsString());
}
```

Figure 7.7: Second idea example: before manual modification

```java
//  CustomSerializerTest.java

public void testBaseClassSerializerInvokedForBaseClassFields() {
  Gson gson = new GsonBuilder()
    .registerTypeAdapter(Base.class, new BaseSerializer())
    .registerTypeAdapter(Sub.class, new SubSerializer()).create();
  this.testSerializerInvokedForBaseClassFieldsTemplate(gson,
    TestTypes.Base.class, BaseSerializer.NAME);
}

public void testSubClassSerializerInvokedForBaseClassFieldsHolding-
    -SubClassInstances() {
  Gson gson = new GsonBuilder()
    .registerTypeAdapter(Base.class, new BaseSerializer())
    .registerTypeAdapter(Sub.class, new SubSerializer()).create();
  this.testSerializerInvokedForBaseClassFieldsTemplate(gson,
    TestTypes.Sub.class, SubSerializer.NAME);
}

public void testBaseClassSerializerInvokedForBaseClassFieldsHolding-
    -SubClassInstances() {
  Gson gson = new GsonBuilder()
    .registerTypeAdapter(Base.class, new BaseSerializer()).create();
  this.testSerializerInvokedForBaseClassFieldsTemplate(gson,
    TestTypes.Sub.class, BaseSerializer.NAME);
}

public <TBase> void testSerializerInvokedForBaseClassFieldsTemplate(
    Gson gson, Class<TBase> clazzTBase, String string1) {
  try {
    ClassWithBaseField target = new
      ClassWithBaseField((TestTypes.Base) clazzTBase.newInstance());
    JsonObject json = (JsonObject) gson.toJsonTree(target);
    JsonObject base = json.get("base").getAsJsonObject();
    assertEquals(string1, base.get(Base.SERIALIZER_KEY).getAsString());
  } catch (Exception e) {
    e.printStackTrace();
    throw new RuntimeException();
  }
}
```

Figure 7.8: Second idea example: after manual modification — We manually modified the template method and then updated the first two refactored test methods accordingly and then manually refactored the third test method using the modified template method.

"Thanks for your contribution!

A few remarks.

1. This project is in maintenance mode, and we're generally going to be reluctant to accept PRs that are essentially cosmetic, especially if it is not trivially obvious that they don't change anything.
2. Anyway, I'm not really convinced that the change would be an improvement. The new helper methods seem more complicated than I would expect. I don't see why they catch and rethrow exceptions, for example. They also have type parameters that don't seem useful.
3. As a more general remark, refactoring test methods to remove duplication isn't as obviously beneficial as with production code. It's an advantage to be able to understand a test method in isolation, without having to look elsewhere in the test class.

So I think we're not going to accept this one. Thanks for thinking of us, though, and best wishes for your research!"

Once again, the reviewer is not convinced that the change would be an improvement. They mention that new helper methods seem more complicated than expected; there is confusion on the exception handling in the helper method, and they also have type parameters that do not seem useful.

We learned from this feedback to consider discarding the refactoring cases where the template method will need exception handling in its implementation. Also, we might need to re-evaluate the "type parametrization" and discuss when it could be less beneficial.

Furthermore, the reviewer mentions that the refactoring test methods to remove duplication is not as obviously beneficial as with production code since it is an advantage to understand a test method in isolation without looking elsewhere in the test class. This was the second time that we were getting this feedback, but it is more apparent this time. Developers will not accept the refactorings that we rated 6 or 7 because they are way more reluctant to use a helper function in test methods than using a helper function in the production code. This means that unless the refactoring case improves the code's quality by a lot, it will not be accepted.

After discussing the possible ideas to improve the quality of the code by a lot, we concluded that it is unlikely to do so when there are only two test methods involved in a refactoring case and that only cases that have a chance of significant improvements are the ones that involve many test methods. This decreases the potential applicability of JTestParametrizer's refactorings because situations where many tests share a similar body are hard to refactor automatically.

However, those situations are the ones where using a template method can highly improve the overall quality of the code due to reduction of duplication and even improve the code's readability and extensibility.

### 7.4.3 Second Gson Pull Request

This pull request was also created based on the feedback we got from the Jimfs pull request, and it is not affected by the feedback that we got from the first Gson pull request: I created both of these pull requests simultaneously but for testing different ideas.

The idea behind this pull request was to better leverage the extensibility that our refactoring cases add to the code. This way, we might be able to present the advantages of our refactoring cases better. After refactoring a method pair using the JTestParametrizer tool, we manually create new test cases that use the template method created for that refactoring case. By doing so, not only do we represent the potential increase in extensibility, but we can also increase the coverage of the tests. This will be easier for refactoring cases that the arguments of the created template method are of a primitive type like boolean.

We leveraged the refactoring nominee between "testReadArray" and "testBooleans" test methods in "JsonReaderTest.java" to implement this idea. After running the JTestParametrizer tool, I manually added two new test methods "testBooleansFalseFalse" and "testBooleansFalseTrue" using the template method that the JTestParametrizer tool created. Figure 7.9 shows the code before running the JTestParametrizer tool. Figure 7.10 shows the changes after running the JTestParametrizer tool, and Figure 7.11 shows the final manually modified version we used in this pull request.

I created this pull request at the same time as I created the first Gson pull request, and we intended to submit both of them at around the same time. However, when we got the feedback for the first Gson pull request, we decided not to submit this second Gson pull request because the feedback clearly stated that the Gson project was in maintenance mode, and even regardless, based on remarks number 2 and 3 that we got for the first Gson pull request, they would not accept these changes.

Therefore, unfortunately, we were not able to get any feedback for this pull request. Nevertheless, the idea behind this pull request can still be considered as a potential feature for the JTestParametrizer tool.

```java
//  JsonReaderTest.java

public void testReadArray() throws IOException {
  JsonReader reader = new JsonReader(reader("[true, true]"));
  reader.beginArray();
  assertEquals(true, reader.nextBoolean());
  assertEquals(true, reader.nextBoolean());
  reader.endArray();
  assertEquals(JsonToken.END_DOCUMENT, reader.peek());
}

public void testBooleans() throws IOException {
  JsonReader reader = new JsonReader(reader("[true,false]"));
  reader.beginArray();
  assertEquals(true, reader.nextBoolean());
  assertEquals(false, reader.nextBoolean());
  reader.endArray();
  assertEquals(JsonToken.END_DOCUMENT, reader.peek());
}
```

Figure 7.9: Third idea example: before running the JTestParametrizer tool

```java
//  JsonReaderTest.java

public void testReadArray() throws Exception {
  this.jsonReaderTestTestTemplate("[true, true]", true);
}

public void testBooleans() throws Exception {
  this.jsonReaderTestTestTemplate("[true,false]", false);
}

public void jsonReaderTestTestTemplate(String string1, boolean b1)
    throws Exception {
  JsonReader reader = new JsonReader(reader(string1));
  reader.beginArray();
  assertEquals(true, reader.nextBoolean());
  assertEquals(b1, reader.nextBoolean());
  reader.endArray();
  assertEquals(JsonToken.END_DOCUMENT, reader.peek());
}
```

Figure 7.10: Third idea example: before manual modification

```java
//  JsonReaderTest.java

public void testReadArray() throws IOException {
  this.testBooleansTemplate("[true, true]", true, true);
}

public void testBooleans() throws IOException {
  this.testBooleansTemplate("[true,false]", true, false);
}

public void testBooleansFalseFalse() throws IOException {
  this.testBooleansTemplate("[false,false]", false, false);
}

public void testBooleansFalseTrue() throws IOException {
  this.testBooleansTemplate("[false,true]", false, true);
}

public void testBooleansTemplate(
    String string1, boolean b1, boolean b2) throws IOException {
  JsonReader reader = new JsonReader(reader(string1));
  reader.beginArray();
  assertEquals(b1, reader.nextBoolean());
  assertEquals(b2, reader.nextBoolean());
  reader.endArray();
  assertEquals(JsonToken.END_DOCUMENT, reader.peek());
}
```

Figure 7.11: Third idea example: after manual modification — We manually further parameterized the template method adding another boolean argument. Then we updated the first two refactored test methods accordingly. Finally, we created the third and fourth test methods that both utilize the updated template method.

### 7.4.4 Joda-time Pull Request

I created the Joda-time pull request in response to the idea from the first Gson pull request feedback to find refactoring cases involving many test methods. Knowing that these refactoring cases can be rare, Joda-time seemed like a great candidate since it had the most refactoring cases out of all the benchmarks.

So the approach for this implementation was to run the JTestParametrizer tool first and then going over steps 1, 2, and 3 of the process of selection pull request in Section 7.3.2. Then I would go over every refactoring case in the output of step 3 and for each of the template methods, look for other test methods in the code that share the same body with that template method. After finding a case that more than two or more unrefactored methods shared the same body with a template method, I would manually refactor those methods and use that example for this pull request.

The reasoning behind the idea was that since this helper function is shared between at least four methods, it decreases a higher level of duplication. Also, since at least four test methods were sharing the body of this helper function, this might mean that this helper function might have been a semantic routine that all of those cases were using, and extracting it will not the understandability since it was a shared semantic routine that could have been extracted in the first place.

However, before we submitted this pull request, we figured out that the Joda-time is in maintenance mode, and they will not be accepting any PRs that are essentially cosmetic. Unfortunately, we concluded that we would not be able to get any feedback for this pull request either. We decided not to submit the pull request as it would have been unlikely to help anyone.

### 7.4.5 Bootique Pull Request

After failing to submit the Joda-time pull request due to it being in maintenance mode, we decided to try that idea with another benchmark. We created the Bootique pull request based on finding refactoring cases involving multiple test methods.

The main reason that this idea would be valuable is that if a part of the code is shared between multiple test methods, then there is a chance that that part is doing a discrete task, one that can be given a name, and that makes sense to abstract into a procedure. This leads to not having a substantial adverse effect on the understandability of the tests and can potentially even increase the readability of the test.

To implement this idea, I did the same work I did to create the Joda-time pull request. First, I ran the JTestParametrizer tool on the benchmark. Then I went over steps 1, 2, and 3 of the process of selection pull request in Section 7.3.2, and for every remaining case, I tried to find unrefactored test methods in the code that share some part of their body with one of the template methods. After that, when I found such test methods, I would manually modify those unrefactored tests, that template method, and its two refactored tests to make it a refactoring case involving all of those test methods. Finally, I studied the new template method and tried to determine if it is doing a semantic process based on the context of that benchmark.

One of the examples that I used for implementing this idea in this pull request was a refactoring nominee between the "testLastPathComponent_ArrayRootValue" and "testLastPathComponent_ArrayValue" test methods. Figure 7.12 shows the test methods before running the JTestParametrizer tool. Figure 7.13 shows the test methods and the template method after running the JTestParametrizer tool but before the manual modification. Finally, Figure 7.14 shows the final version that we used in this pull request after we manually modified it to implement the effect of our idea in the code.

The extracted template method "assertCanCreateValidPathSegment" in Figure 7.14 checks the possibility of creating a valid path segment for a given JsonNode and path. It follows a standard procedure that is semantically sound, and it encapsulates that procedure.

By comparing Figure 7.12 and Figure 7.14, we can argue that even the readability and understandability of the final version is higher since the helper function does a straightforward semantic procedure that checks the possibility of creating a valid path segment for a given JsonNode and path. The name of the helper function is appropriate to what it does, which improves the understandability of the tests. Not only that, these modifications improve the extensibility and reduce duplication of the code. Overall, in our opinion, this example was a great response to the feedback that we got for the first Gson pull request.

I submitted this pull request on 22nd of July 2021. The pull request includes four refactoring cases involving 14 test methods. We checked cases and ensured that all the tests passed successfully and that, based on our opinion, every refactoring case was considerably increasing the overall quality of the code. Unfortunately, we have not received any feedback regarding this pull request yet. However, we believe that a reasonable number of projects could accept refactorings like this one, and that it would be helpful to modify JTestParametrizer to facilitate such refactorings.

```java
//  PathSegmentTest.java

public void testLastPathComponent_ArrayRootValue() {
  JsonNode node = YamlReader.read("- 1\n- 2");

  Optional<PathSegment<?>> last0 =
    PathSegment.create(node, "[0]").lastPathComponent();
  assertTrue(last0.isPresent(), "Couldn't resolve '[0]' path");
  assertNotNull(last0.get().getNode(), "Couldn't resolve '[0]' path");
  assertEquals(1, last0.get().getNode().asInt());

  Optional<PathSegment<?>> last1 =
    PathSegment.create(node, "[1]").lastPathComponent();
  assertTrue(last1.isPresent(), "Couldn't resolve '[1]' path");
  assertNotNull(last1.get().getNode(), "Couldn't resolve '[1]' path");
  assertEquals(2, last1.get().getNode().asInt());
}
public void testLastPathComponent_ArrayValue() {
  JsonNode node = YamlReader.read("a:\n  - 1\n  - 2");

  Optional<PathSegment<?>> last0 =
    PathSegment.create(node, "a[0]").lastPathComponent();
  assertTrue(last0.isPresent(), "Couldn't resolve 'a[0]' path");
  assertNotNull(last0.get().getNode(), "Couldn't resolve 'a[0]' path");
  assertEquals(1, last0.get().getNode().asInt());

  Optional<PathSegment<?>> last1 =
    PathSegment.create(node, "a[1]").lastPathComponent();
  assertTrue(last1.isPresent(), "Couldn't resolve 'a[1]' path");
  assertNotNull(last1.get().getNode(), "Couldn't resolve 'a[1]' path");
  assertEquals(2, last1.get().getNode().asInt());
}
public void testLastPathComponent_ArrayObject() {
  JsonNode node = YamlReader.read("a:\n  - b: 1\n  - b: 2");

  Optional<PathSegment<?>> last =
     PathSegment.create(node, "a[1].b").lastPathComponent();
  assertTrue(last.isPresent(), "Couldn't resolve 'a[1].b' path");
  assertNotNull(last.get().getNode(), "Couldn't resolve 'a[1].b' path");
  assertEquals(2, last.get().getNode().asInt());
}
```

Figure 7.12: Fourth idea example: before running the JTestParametrizer tool

```java
//  PathSegmentTest.java

public void testLastPathComponent_ArrayRootValue() {
  this.pathSegmentTestTestLastPathComponent_ArrayValueTemplate(
    "- 1\n- 2", "[0]", "Couldn't resolve '[0]' path",
    "Couldn't resolve '[0]' path", "[1]", "Couldn't resolve '[1]' path",
    "Couldn't resolve '[1]' path");
}
public void testLastPathComponent_ArrayValue() {
  this.pathSegmentTestTestLastPathComponent_ArrayValueTemplate(
    "a:\n  - 1\n  - 2", "a[0]", "Couldn't resolve 'a[0]' path",
    "Couldn't resolve 'a[0]' path", "a[1]",
    "Couldn't resolve 'a[1]' path", "Couldn't resolve 'a[1]' path");
}
public void testLastPathComponent_ArrayObject() {
  JsonNode node = YamlReader.read("a:\n  - b: 1\n  - b: 2");

  Optional<PathSegment<?>> last =
     PathSegment.create(node, "a[1].b").lastPathComponent();
  assertTrue(last.isPresent(), "Couldn't resolve 'a[1].b' path");
  assertNotNull(last.get().getNode(), "Couldn't resolve 'a[1].b' path");
  assertEquals(2, last.get().getNode().asInt());
}
public void pathSegmentTestTestLastPathComponent_ArrayValueTemplate(
    String string1, String string2, String string3, String string4,
    String string5, String string6, String string7) {
  JsonNode node = YamlReader.read(string1);

  Optional<PathSegment<?>> last0 =
    PathSegment.create(node, string2).lastPathComponent();
  assertTrue(last0.isPresent(), string3);
  assertNotNull(last0.get().getNode(), string4);
  assertEquals(1, last0.get().getNode().asInt());

  Optional<PathSegment<?>> last1 =
    PathSegment.create(node, string5).lastPathComponent();
  assertTrue(last1.isPresent(), string6);
  assertNotNull(last1.get().getNode(), string7);
  assertEquals(2, last1.get().getNode().asInt());
}
```

Figure 7.13: Fourth idea example: before manual modification

```java
//  PathSegmentTest.java

public void testLastPathComponent_ArrayRootValue() {
  JsonNode node = YamlReader.read("- 1\n- 2");
  this.assertCanCreateValidPathSegment(node, "[0]", 1);
  this.assertCanCreateValidPathSegment(node, "[1]", 2);
}

public void testLastPathComponent_ArrayValue() {
  JsonNode node = YamlReader.read("a:\n  - 1\n  - 2");
  this.assertCanCreateValidPathSegment(node, "a[0]", 1);
  this.assertCanCreateValidPathSegment(node, "a[1]", 2);
}

public void testLastPathComponent_ArrayObject() {
  JsonNode node = YamlReader.read("a:\n  - b: 1\n  - b: 2");
  this.assertCanCreateValidPathSegment(node, "a[1].b", 2);
}

public void assertCanCreateValidPathSegment(
    JsonNode t, String path, int expectedValue) {
  Optional<PathSegment<?>> last =
    PathSegment.create(t, path).lastPathComponent();
  assertTrue(last.isPresent(), "Couldn't resolve '" + path + "' path");
  assertNotNull(
    last.get().getNode(), "Couldn't resolve '" + path + "' path");
  assertEquals(expectedValue, last.get().getNode().asInt());
}
```

Figure 7.14: Fourth idea example: after manual modification

## 7.5 Learning From our Experience

This chapter used three different processes to get qualitative feedback. This feedback included both the feedback on the current version of the JTestParametrizer tool and early feedback on what the JTestParametrizer tool could do. By studying this feedback, we realized that developers care about certain critical factors when refactoring test methods. We tried to address some of these factors by adding new configurations to the JTest-Parametrizet tool. Finally, we tried manual modification to get feedback on some non-existent features, which helped us better understand the potential best next step for the JTestParametrizer tool.

In this section, I will briefly compare the three processes that we used for getting feedback. Then I will have a quick review of the factors that we learned that are most important for developers when it comes to refactoring test methods. Then I will quickly explain the one configuration we added and some configurations that we could add to address some of the quality shortcomings noted in the feedback from developers. Finally, considering all manual modifications to pull requests and our feedback, I will explain what we think the potential best next step is for the JTestParametrizer tool.

### 7.5.1 Processes for Getting Feedback

Out of the three processes that we used to get qualitative feedback, submitting pull requests was the one that benefited us the most and provided us with the most helpful feedback. However, based on our considerations for submitting a pull request, it is impossible to select pull requests without first practicing manual quality evaluation.

Furthermore, although using a questionnaire is a great way to get feedback, questionnaires do not guarantee quality results. Therefore, we decided that using a questionnaire would not be as helpful as the other two processes for our research.

### 7.5.2 Factors Learned

Based on the feedback for the pull requests, I concluded that the understandability and readability criteria in a test method are much more valuable than the same criteria in the production code since it is an advantage to understand a test method in isolation without looking elsewhere in the test class.

I also learned that even though reducing duplication is generally good, it is not as crucial in test methods since duplication in tests can make the expected behavior clearer. A reduction of duplication in tests is only valuable if it does not damage the understandability of the tests.

Furthermore, refactorings that add to the complexity of the code by creating a new exception handling scenario or creating less useful type parameters will not be welcome by the developer, even if they reduce the code duplication and improve extensibility.

### 7.5.3 Potential Configurations for the Tool

After manual quality evaluation, we realized that the refactoring cases that use behavioral parametrization have the lowest rating and damage the code's overall quality. With that in mind, we added a new configuration to the JTestParametrizer tool that discards the behavioral nominees.

Next, we got feedback on the complexity of the refactoring cases due to creating new exception handlings and less useful type parameters. Now, even though the adverse effect of these two transformations is not as apparent as the adverse effect of behavioral parametrization, it might be a good idea to add two new configurations to the JTestParametrizer tool. One configuration would discard the refactoring cases that introduce new exception handling cases, and the other would discard the refactoring cases with type parametrization.

### 7.5.4 Potential Best Next Step for the Tool

Based on the feedback that we got so far, it appears that the best next step would be to do refactorings for cases that have many test methods involved because there would be a higher chance that the template method created for that case be an encapsulated semantic procedure in the domain of that code. If this happens, then using this helper function will not necessarily decrease the understandability of the test. Also, the amount of reduction of duplication and increase in extensibility will be higher with this feature, which means that the advantages of the current version would be more substantial.

We have submitted a pull request to an open-source project based on this idea, and when we get the feedback for that pull request, we will better understand the value of this potential feature.

# Chapter 8

# Conclusion

In this work, we did a quantitative and qualitative evaluation of the JTestParametrizer tool. Furthermore, we modified and extended JTestParametrizer to enhance its overall quality. Here, I will summarize the work, list the lessons learned, and mention some ideas about extending the JTestParametrizer tool to be more useful based on the feedback that we got from the developers.

## 8.1 Summary of the Work

To reiterate the work, first, we ran the JTestParametrizer tool on the 18 open-source benchmarks projects that we chose. After studying the quantitative feedback, we found several conceptual and non-conceptual bugs in the tool. We followed a debugging procedure to determine the source of those bugs, and we spent a great time fixing a portion of those bugs. Subsequently, we decided that it would be more logical to put fixing the rest of the bugs on hold and work on the quantitative side of the tool first. The intuition behind this decision was that when we are working on the quantitative side of the tool, we might decide to discard a feature, and then we will not need to deal with the bugs related to that feature, which can save much time.

Next, we designed a questionnaire to evaluate the quality of the refactoring cases. However, due to several reasons, such as the difficulty of finding the right engaged participants for a very technical questionnaire and that we wanted detailed feedback, we did not pursue this method to the end. We then manually evaluated the quality of all the 33 refactoring cases in the Gson, Jimfs, and Bootique benchmarks. We rated each case based on qualitative factors on a scale of 0 to 10, with 5 representing cases that had the same overall quality before and after refactoring. By investigating these ratings, we determined two critical learnings. First, the overall average rating was slightly below five, meaning that, on average and based on our opinion, the tool was not enhancing the overall quality of the tests. Second, we noticed that the eight refactoring cases with behavioral parameterization had the lowest ratings out of all the 33 cases. We assessed the quality of the behavioral parametrization as a technique, and we decided that overall it would decrease the quality of the tests. Consequently, we added a new configuration to the tool that discards the behavioral parametrization. Next, we ran the tool with the new configuration on the 18 benchmarks, and we investigated the comparison between the new results and previous ones. Even after applying the new configuration, there was no significant enhancement in the overall quality of the refactored tests.

Afterward, we created and submitted proposed refactorings upstream using Pull Requests to get the developers' feedback on the quality of the tool. We created a Pull Request for the Jimfs benchmark. For this Pull Request, we selected some representative refactoring cases that we rated higher than five. Based on the feedback, we learned that our earlier hypothesis was correct and that the current version of the JTestParametrizer tool was not significantly enhancing the overall quality of the tests. We also learned some critical factors about developers' preferences regarding refactoring tests.

Knowing that the current version of the JTestParametrizer is not making significant enhancements in the overall quality of the tests, we decided that we should extend the JTestParametrizer tool and add new features. However, the question was which feature would be a feature that changes the tool to something that developers would want to use. We employed a technique similar to the Minimum Viable Product technique to conclude which potential feature of the JTestParametrizer tool will have the best feedback before implementing that feature. We came up with two ideas (potential features) that might improve the tool's overall quality. Then we selected two sets of refactoring cases for two Pull Requests for the Gson benchmark. Then, we manually implemented the effects of the first idea on the refactored cases that we selected for the first Pull Request, and similarly, we manually implemented the effects of the second idea on the refactored cases that we selected for the second Pull request. Based on the feedback from developers, we learned some new factors that the developers care about a lot regarding refactoring test methods. We also learned a heuristic about choosing the benchmarks for this methodology. Overall, we figured that the ideas that we used for these two Pull Requests were not making a significant improvement in the quality of the tool.

Based on the factors that we learned from the feedback for the previous Pull Requests, we came up with a new idea (potential feature) that, in theory, would provide all the things that the developers wanted from a test refactoring case. The idea was to refactor cases involving multiple test methods (preferably over three) instead of cases with two test methods. We looked for these cases in the Joda-time and Bootique benchmarks and manually implemented the effects of this idea on the cases that we found. We quantitatively evaluated those cases. Based on our opinion, there was a noticeable improvement in the quality of those cases as they got the highest ratings of all the refactoring cases rated to that point. Then we created two Pull Requests for the Joda-time and Bootique benchmarks using the manually modified cases that we had for this idea. In doing so, we learned more heuristics about choosing the benchmarks for this methodology.

As mentioned, throughout this work, we had several learnings. We learned some heuristics for determining which benchmarks would suit which processes the best. We utilized three methodologies for quantitative evaluation and found out the strengths and weaknesses of each of these processes. We compared the priority of the qualitative evaluation with quantitative evaluation. We learned critical factors that developers care about a lot regarding refactoring tests. We determined the ideas (potential features) that will provide the best overall quality based on our learned factors. I will explain all these learnings in the subsequent sections of this chapter.

## 8.2 Heuristics for Selecting Benchmarks

As explained in Section 4.7, we learned some non-trivial heuristics for selecting the benchmarks for different methodologies throughout the work. Here I will list three of these critical heuristics.

### 8.2.1 Familiarity With the Benchmark's Domain

In both the "Manual Quality Evaluation" and "Submitting Pull Requests" processes we used for qualitative evaluation, we assessed the refactoring cases based on qualitative factors such as understandability and readability in the first step of the process. However, to have a fair assessment of the changes in the overall quality of the refactored tests, we had to understand the behavior and the purpose of those tests completely. Having this knowledge also provided us with more detailed feedback for each refactoring case.

The learned heuristic was that when we choose a benchmark for qualitative evaluation of our tool, it might be better to choose the benchmarks where we are familiar with their domain. Otherwise, we need to spend some time learning and understanding the benchmark's behavior, or our qualitative assessment will not be accurate.

### 8.2.2 Estimated Response Time

For evaluation processes requiring external feedback such as "Questionnaire" and "Submitting Pull Requests", considering estimated response time is essential for selecting the benchmarks. It helps to decide whether using a benchmark would fit our timing schedule.

We put a considerable amount of effort into creating a manually modified Pull Request for the Bootique benchmark, but after over 80 days, we still have not received any feedback for that pull request. At the same time, we received feedback in less than an hour for the Gson Pull Request, and for the Jimfs Pull Request, we got feedback in three days. After studying the previous Pull Requests sent to these repositories, we determined a pattern, and we figured that we could have estimated the response time for each repository to a certain extent.

### 8.2.3 Maintenance Mode

When selecting benchmarks for the "Submitting Pull Requests" process, determining if the benchmark is in the maintenance mode or not is extremely important. If a project is in maintenance mode, the reviewers would generally be reluctant to accept any essentially cosmetic Pull Requests, especially if they are not trivially obvious.

## 8.3 Quantitative vs. Qualitative Evaluation

In this work, we started with a quantitative evaluation which led us to discover several conceptual and non-conceptual bugs in the tool. However, after spending a notable amount of time on fixing a portion of the discovered bugs, we decided to put the rest of that task on hold and work on the qualitative evaluation of the tool. Our hypothesis was to ensure that we had the right tool that developers would want to use first and then worry about the quantitative side and fixing bugs. Consequently, based on the qualitative evaluation, we decided to discard the Behavior Parameterization, which meant that we no longer needed to deal with the errors related to Behavior Parameterization.

Our learning based on this work was that when we are trying to modify and extend our artifact based on the feedback, it might be better to do the quantitative evaluation prior to the qualitative evaluation. This is because it would be more logical to use the feedback from the qualitative evaluation and enhance the overall quality of the artifact first to make sure that it is desirable for developers, as opposed to fixing quantitative issues of an artifact that might not be desirable to anyone and then trying to make that artifact into something desirable for developers.

## 8.4 Processes for Quantitative Evaluation

We used three processes for the qualitative evaluation of this work. Here, I will compare these three processes and discuss their strengths and weaknesses based on our experience.

### 8.4.1 Questionnaire

Using a questionnaire is inexpensive, it is easy to analyze the results, and it is scalable. However, it is hard to find the right participants for a very technical quality assessment, such as our work.

As we mentioned, to have a fair qualitative assessment of the cases, we need to be familiar with the domain of the benchmark and have complete understanding of the behavior of the cases. It is tough to force this constraint on the participants for a questionnaire, and if we do not force this, the results will not be as accurate. Engagement is another concern with the questionnaires. A technical qualitative assessment can take a long time, and it could be hard to keep the general participants engaged when they do not benefit from the process. Overall, a technical questionnaire does not come with any guarantee of quality feedback.

A questionnaire would be more scalable, and the results easier to analyze, compared to the "Manual Quality Evaluation" and "Submitting Pull Request" processes. However, it would be much harder to find the right participant group, and the overall quality of the feedback would be lower than the other two processes. Finally, the feedback would not be as reliable or valuable.

### 8.4.2 Manual Quality Evaluation

Using the "Manual Quality Evaluation" process for qualitative evaluation has the critical advantage of not needing any external feedback and can be done without the need for participants. Hence, it does not have the main problem that the "Questionnaire" process had. Furthermore, we can have as much detail as we want in the evaluation since we are in charge of this process's level of detail. Also, since we are the people doing the evaluation, and because we benefit from the evaluation, we will not be concerned about the engagement of participants. Besides, we can force the constraints that we could not force on participants of the “Questionnaire” process, such as the familiarity with the benchmarks’ domain. Consequently, by forcing these constraints and controlling the level of detail, we can guarantee the high quality of the feedback using this process.

The main weakness of the “Manual Quality Evaluation” process is that it will take a long time to do it, and it is not scalable since we are the only participants. Also, even though the quality of the feedback using this process is higher than the quality of the feedback using the “Questionnaire” process, like any other self-assessment, it still is vulnerable to unconscious bias. Furthermore, even though we force ourselves to be familiar with the domain of the benchmarks, we might still not consider some of the non-trivial factors that the developers of those benchmarks care about a lot. This is why even though this process’s feedback is valuable, it will still not be as valuable and reliable as the feedback we get from the “Submitting Pull Requests” process.

### 8.4.3 Submitting Pull Requests

The main advantage of using the “Submitting Pull Requests” process is that its feedback has the highest value and reliability. This is because the reviewer who provides the feedback is one of the repository’s maintainers and has complete familiarity and understandability over the domain of that repository. The reviewers would know the non-trivial factors related to that benchmark that we might not consider using the “Manual Quality Evaluation” process. Furthermore, examining the changes in Pull Request can be beneficial for the reviewer, so we would not be concerned about the engagement.

The main weakness of this process is that it is even more time-consuming than the “Manual Quality Evaluation” process. To be ethical, we should only include cases judged to be a net benefit in our Pull Requests. This means that this process requires the “Manual Quality Evaluation” as a prerequisite, making it even less scalable. This process also requires external feedback. The positive thing is that we do not need to look for participants as opposed to the “Questionnaire” process. However, this process still forces us to consider factors such as estimated response time.

## 8.5 Deduplication in Tests vs. in Main Code

After investigating the feedback we got for the Pull Requests, we learned some non-trivial critical factors that the developers cared about regarding refactoring test methods. Reviewers believed that reducing duplication is generally good but not as much so in test methods as some amount of duplication can make the expected behavior of the test more straightforward, which is valuable. Also, they did not welcome complex helper functions that introduced new exception handling cases to the code. They believed that it is an advantage to understand a test method without looking elsewhere.

Overall, we learned that regarding refactoring test methods, readability and understandability factors have a higher value than the code repetition factor, which is not necessarily valid for production code. Based on this learning, we concluded that to make the tool desirable for developers, we should increase the amount of deduplication to compensate for the loss of understandability and, more importantly, try minimizing the loss of readability and understandability.

## 8.6 Best Potential Extensions

We concluded that to extend the JTestParametrizer tool to a tool that developers would want to use, we need to add features to minimize the loss of readability and understandability. Increasing the amount of deduplication would be good too, but not if it harms the readability and understandability of the refactored cases. Here, we will explain two potential features that could potentially achieve this goal.

### 8.6.1 Method Pairs vs. Cases With Multiple Methods

The first idea is to only do the refactoring for the cases that involve multiple (preferably more than three) test methods.

The primary intuition behind this idea is that if the extracted template method from the refactored test cases does an encapsulated semantic procedure in the domain of that benchmark, and if the name of that extracted method accurately represents the encapsulated semantic procedure that it is doing, then this refactoring case will not decrease the readability or understandability of the code. This is because we can understand the refactored test method without looking elsewhere, since the name of the template method properly represents its behavior, and we do not need to look at the body of the template method to understand its behavior.

That said, we hypothesized that if a piece of code is repeated in five or six test methods, there is a higher chance that that piece of code would represent an encapsulated semantic behavior than when it is repeated only in two test methods. We manually assessed this idea when creating the Pull Requests for Joda-time and Bootique benchmarks, and in most cases, our hypothesis was correct. In most cases where the refactoring involved four or five test methods, the extracted template method was doing an encapsulated semantic procedure that could be described using a great name. After manual evaluation, the refactoring cases that we got using this idea had the highest ratings out of all the cases that we rated. The other minor improvement of this idea is that it also enhances the amount of deduplication for each refactoring case.

### 8.6.2 Pairing With Inliner IDE

The second idea is to change the tool's behavior to provide refactoring suggestions on specific trigger points instead of its current mechanism.

The problem with code duplication is that when we try to change one of the instances of the repeated codes, we most probably need to apply that change in all the other instances of that repeated code as well. This makes it harder to keep all these repeated codes consistent with each other, which would lead to less maintainability. The primary intuition behind this idea is to provide the test refactoring suggestion whenever a developer is modifying one of the test codes with code duplication. This way, the developer will completely understand that refactoring case and can decide whether to accept it. This idea will not misuse developers' time, and they would be in charge of every refactored case, and they can decide for themselves which refactoring cases would increase the overall quality.

Duala-Ekoko et al. [16] proposed a technique for tracking clones in evolving software. They described a complete clone tracking system that can produce clone region descriptors from a clone detection tool's output and notify developers of modifications to those clone regions, and support simultaneous editing of clone regions. Although their approach does not refactor the clones, it could be used in the development of this potential feature because it describes a suggestion-based system that works with evolving software.

# References

[1] Standard performance evaluation corporation. SPECjvm98 documentation, August 1998. Online version at https://www.spec.org/jvm98/jvm98/doc/.

[2] Standard performance evaluation corporation. SPECjvm2008, 2008. Online version at http://www.spec.org/jvm2008/.

[3] David B Allison, Richard M Shiffrin, and Victoria Stodden. Reproducibility of research: Issues and proposed remedies. *Proceedings of the National Academy of Sciences*, 115(11):2561–2562, 2018.

[4] Vaibhav Bajpai, Mirja Kühlewind, Jörg Ott, Jürgen Schönwälder, Anna Sperotto, and Brian Trammell. Challenges with reproducibility. In *Proceedings of the Reproducibility Workshop*, pages 1–4, 2017.

[5] Tibor Bakota, Péter Hegedűs, Péter Körtvélyesi, Rudolf Ferenc, and Tibor Gyimóthy. A probabilistic software quality model. In *2011 27th IEEE International Conference on Software Maintenance (ICSM)*, pages 243–252, 2011.

[6] Stefan Bechtold, Sam Brannen, Johannes Link, Matthias Merdes, Marc Philipp, Juliette de Rancourt, and Christian Stein. JUnit 5 user guide. https://junit.org/junit5/docs/current/user-guide/#writing-tests-parameterized-tests Accessed 6 Oct. 2021.

[7] Emery D. Berger, Celeste Hollenbeck, Petr Maj, Olga Vitek, and Jan Vitek. Reproducing "A large-scale study of programming languages and code quality in GitHub: A reproducibility study". https://nextjournal.com/PRL-PRG/toplas-analysis/ Accessed 11 Sept. 2021.

[8] Tegawendé F. Bissyandé, Ferdian Thung, Shaowei Wang, David Lo, Lingxiao Jiang, and Laurent Réveillère. Empirical evaluation of bug linking. In *2013 17th European Conference on Software Maintenance and Reengineering*, pages 89–98, 2013.

[9] Stephen M. Blackburn, Amer Diwan, Matthias Hauswirth, Peter F. Sweeney, José Nelson Amaral, Tim Brecht, Lubomír Bulej, Cliff Click, Lieven Eeckhout, Sebastian Fischmeister, Daniel Frampton, Laurie J. Hendren, Michael Hind, Antony L. Hosking, Richard E. Jones, Tomas Kalibera, Nathan Keynes, Nathaniel Nystrom, and Andreas Zeller. The truth, the whole truth, and nothing but the truth: A pragmatic guide to assessing empirical evaluations. *ACM Trans. Program. Lang. Syst.*, 38(4), October 2016.

[10] Stephen M. Blackburn, Robin Garner, Chris Hoffmann, Asjad M. Khang, Kathryn S. McKinley, Rotem Bentzur, Amer Diwan, Daniel Feinberg, Daniel Frampton, Samuel Z. Guyer, Martin Hirzel, Antony Hosking, Maria Jump, Han Lee, J. Eliot B. Moss, Aashish Phansalkar, Darko Stefanović, Thomas VanDrunen, Daniel von Dincklage, and Ben Wiedermann. The DaCapo benchmarks: Java benchmarking development and analysis. In *Proceedings of the 21st Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications*, OOPSLA '06, page 169–190, New York, NY, USA, 2006. Association for Computing Machinery.

[11] Steve Blackburn, Matthias Hauswirth, Emery Berger, Michael Hicks, and Shriram Krishnamurthi. ACM SIGPLAN: Empirical evaluation guidelines. https://www.sigplan.org/Resources/EmpiricalEvaluation/ Accessed 7 Oct. 2021.

[12] Carl Boettiger. An introduction to Docker for reproducible research. *ACM SIGOPS Operating Systems Review*, 49(1):71–79, 2015.

[13] Monica Chin. How a university got itself banned from the Linux kernel — the University of Minnesota's path to banishment was long, turbulent, and full of emotion. https://www.theverge.com/2021/4/30/22410164/linux-kernel-university-of-minnesota-banned-open-source Accessed 12 Sept. 2021.

[14] Jens Dietrich, Henrik Schole, Li Sui, and Ewan Tempero. XCorpus – an executable corpus of Java programs. *Journal of Object Technology*, 16(4):1:1–24, August 2017.

[15] F.J. Domínguez-Mayo, M.J. Escalona, M. Mejías, M. Ross, and G. Staples. Quality evaluation for model-driven web engineering methodologies. *Information and Software Technology*, 54(11):1265–1282, 2012.

[16] Ekwa Duala-Ekoko and Martin P. Robillard. Tracking code clones in evolving software. In *29th International Conference on Software Engineering (ICSE'07)*, pages 158–167, 2007.

[17] Anna Maria Eilertsen and Gail C. Murphy. The usability (or not) of refactoring tools. In *2021 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 237–248, 2021.

[18] Hugo Estrada, Alicia Martínez Rebollar, Oscar Pastor, and John Mylopoulos. An empirical evaluation of the i* framework in a model-based software generation environment. In Eric Dubois and Klaus Pohl, editors, *Advanced Information Systems Engineering*, pages 513–527, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg.

[19] Philip J. Guo. CDE: Run any Linux application on-demand without installation. In *Proceedings of the 25th International Conference on Large Installation System Administration*, LISA'11, page 2, USA, 2011. USENIX Association.

[20] Lingxiao Jiang, Ghassan Misherghi, Zhendong Su, and Stephane Glondu. Deckard: Scalable and accurate tree-based detection of code clones. In *29th International Conference on Software Engineering (ICSE'07)*, pages 96–105, 2007.

[21] Shriram Krishnamurthi and Jan Vitek. The real software crisis: Repeatability as a core value. *Commun. ACM*, 58(3):34–36, February 2015.

[22] Bettina Laugwitz, Theo Held, and Martin Schrepp. Construction and evaluation of a user experience questionnaire. In *Symposium of the Austrian HCI and usability engineering group*, pages 63–76. Springer, 2008.

[23] Karine Mordal, Nicolas Anquetil, Jannik Laval, Alexander Serebrenik, Bogdan Vasilescu, and Stéphane Ducasse. Software quality metrics aggregation in industry. *Journal of Software: Evolution and Process*, 25(10):1117–1135, 2013.

[24] Baishakhi Ray, Daryl Posnett, Vladimir Filkov, and Premkumar Devanbu. A large scale study of programming languages and code quality in GitHub. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 155–165, 2014.

[25] Caitlin Sadowski, Edward Aftandilian, Alex Eagle, Liam Miller-Cushon, and Ciera Jaspan. Lessons from building static analysis tools at Google. *Commun. ACM*, 61(4):58–66, March 2018.

[26] Ewan Tempero, Craig Anslow, Jens Dietrich, Ted Han, Jing Li, Markus Lumpe, Hayden Melton, and James Noble. The Qualitas corpus: A curated collection of Java code for empirical studies. In *2010 Asia Pacific Software Engineering Conference*, pages 336–345, 2010.

[27] Rongxin Wu, Hongyu Zhang, Sunghun Kim, and Shing-Chi Cheung. ReLink: Recovering links between bugs and changes. In *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*, ESEC/FSE '11, page 15–25, New York, NY, USA, 2011. Association for Computing Machinery.

[28] Chuan Yue. A projection-based approach to software quality evaluation from the users' perspectives. *International Journal of Machine Learning and Cybernetics*, 10(9):2341–2353, 2019.

[29] Zhao, Jun. Automatic refactoring for renamed clones in test code. Master's thesis, University of Waterloo, 2018.