Property-Based Testing in the Wild!

Exploring Property-Based Testing in Java: An Analysis of jqwik Usage in Open-Source Repositories

Harald Toth

Supervisor(s): Andreea Costea, Sára Juhošová

EEMCS, Delft University of Technology, The Netherlands

A Thesis Submitted to EEMCS Faculty Delft University of Technology,
In Partial Fulfilment of the Requirements
For the Bachelor of Computer Science and Engineering
27 June 2025

Name of the student: Harald Toth
Final project course: CSE3000 Research Project
Thesis committee: Andreea Costea, Sára Juhošová, Marco Zuñiga

Abstract

Property-based testing (PBT) verifies software correctness by checking that specific properties hold across a wide variety of randomly generated inputs. Despite its apparent usefulness, we lack an overview of how PBT is utilized in the Java ecosystem. In this study, we investigated seven repositories using the jqwik framework. We analyzed 84 PBTs to understand the types of properties developers typically test, their use of generators and shrinkers, and the role of PBT in broader testing strategies. Our findings show that PBT remains a small part of most test suites, often representing less than 2% of the total number of tests. Developers tend to focus on MUTATION, INVARIANT, and ROUND TRIP as the properties they test, frequently using custom generators combined with filtering but never implementing custom shrinkers. We conclude that property-based testing is not being utilized to its full potential in Java projects and highlight areas for future research, including the impact of filtering, the potential of custom shrinkers, and the overall effectiveness of property-based testing.

1 Introduction

Property-based testing (PBT) is a testing strategy that verifies if code is correct by checking that specific properties always hold for a wide range of automatically generated inputs. When a test fails, the framework starts a process called shrinking, where it systematically simplifies the failing input to produce the smallest or simplest version that still triggers the failure, helping the programmer find the error.

Listing 1 shows an example of a property-based test written in Java using functionality from the jqwik library. It generates a list of integers and verifies that it has the same size after reversing it.

The QuickCheck [10] framework, developed for Haskell, introduced the concept of property-based testing. PBT has since spread to other programming languages, and it is now used for testing automotive software [1, 11], operating systems [12], and even data generated by machine learning models [7]. It also serves as a form of documentation for the code's behavior by specifying the properties it should satisfy [10].

While property-based testing has been the topic of multiple research papers, none of them focus on the Java ecosystem or the jqwik framework. Given Java's widespread use in the enterprise and its role in powering secure and scalable applications across nearly every industry, understanding how property-based testing is applied within this ecosystem is crucial.

Our research focuses on the properties that PBTs aim to test and the use of generators and shrinkers. It builds on the results of four other papers.

Specifically, our project consisted of five papers that analyzed open-source repositories using property-based testing. The papers examined five different programming languages and environments in detail: Java with jqwik, Haskell with QuickCheck [14], Python with Hypothesis [4], Rust with PropTest [2], and Rust with Quickcheck [5]. This paper focuses on Java and aims to gain insights into the kinds of properties developers test with the long-term goal of improving bug detection. By breaking down complex properties into simpler ones, identifying correlations or dependencies between them, and examining the software that causes failures, it may be possible to find bugs more effectively and gain deeper insights into how and why certain failures occur.

```java
@Property(tries = 1000)
void sameSizeAfterReverse(
    @ForAll List<Integer> x
) {
    assert x.size() == ListUtils.reverse(x).
        size();
}
```

Listing 1: Example of a jqwik property-based test (PBT) in Java for list reversal. The "`@Property`" annotation declares the method as a PBT, while the "`tries`" attribute specifies how many times the test will run. The "`@ForAll`" annotation indicates that values should be generated for the parameter.

To guide our investigation, we formulated the following research questions, split into two categories:

Properties:

RQ1. What sort of properties do PBTs generally check?

RQ2. How are these properties usually expressed?

RQ3. What role does PBT play in the project's correctness guarantees and bug-finding strategies?

Generators and Shrinking:

RQ4. How and when are generators implemented?

RQ5. In which cases is shrinking support explicitly added?

2 Related Work

Despite its apparent usefulness on paper, we still lack a general view of how developers use PBT in real-world projects. We still do not know much about the properties they test and how they integrate PBT into their workflows. The existing literature explores the experiences developers had with property-based testing in Python and OCaml, first in a preliminary study [9] and then in a follow-up study [8] conducted by the same authors. Both studies consisted of interviews with experienced developers. The first one had seven participants, and the second one had 30. They both focused on the reasons behind the low usage of PBT in practice, extracting and classifying data such as properties, bugs, and experiences from transcripts. In these cases, the interviewee may have left out important details or may have been biased towards using PBT in the first place.

Another study examined the use of Hypothesis for testing open-source machine learning projects [13]. Like the other studies, they highlighted the issues developers faced when writing PBTs. We found only one study [3] that analyzed the types of properties developers test and the features they use

to create PBTs. This study also focuses on the Hypothesis library.

So far, none of the studies we found examined PBT usage in other programming languages, which could play a significant role in people's testing strategies or programming styles. We aim to fill this gap in the existing knowledge by analyzing PBT usage in the Java ecosystem, specifically through the jqwik framework.

3 Methodology

This section outlines our approach to identifying, selecting, and analyzing repositories relevant to our study. We also define the key terms we used throughout the paper.

3.1 Terminology

Before presenting our methodology for data collection and analysis, it is important to define a few key terms that we use throughout the paper to ensure consistency and avoid ambiguity when interpreting our findings.

System Under Test (SUT) refers to the specific piece of code being evaluated by a test. This can range from a single function or method to an entire class, module, or system, depending on the scope of the test.

Filtering refers to setting constraints or rules for the automatically generated inputs (e.g., setting boundaries or excluding certain values).

Generators are components of property-based tests that provide randomized inputs to the SUT. In jqwik, these are typically defined using **Arbitraries**, which provide control over the types of data produced. Developers can use the default generators provided by jqwik for common types or implement custom generators for more specific data.

Shrinkers are also components of property-based tests, but they minimize the input that failed a test to the smallest version that still reproduces the failure. Shrinking is an automatic part of jqwik PBTs, though developers can implement custom shrinkers for more control over the process.

3.2 Finding Repositories

One of the most critical aspects of our work was selecting the right repositories to analyze. Poorly chosen repositories can lead to low-quality, misleading, or insufficient data, compromising the validity of our findings. For this reason, we first needed to define the criteria for examining a repository. We prioritized popular repositories, which we measured by the amount of GitHub stars they had. While not a definitive measure of code quality, GitHub stars often serve as an early indicator of community interest and potential relevance, helping us focus on repositories that are more likely to provide meaningful insights.

We used several tools to find repositories. The first one was GitHub's advanced search engine. Our initial search targeted repositories that listed jqwik as a dependency. Within this list, we looked for the ones that also imported the "`@Property`" annotation specific to jqwik PBTs. This two-step approach helped us isolate projects that referenced jqwik and used its testing features.

The second tool we used was SourceGraph, a code search and navigation tool designed for exploring large-scale codebases across multiple repositories. We applied the same process as the first search but got significantly more results this time.

The last source we checked was Maven Repository, a widely used public index for Maven artifacts. On jqwik's page, we found a list of projects and libraries that depend on it. We reviewed the most popular ones after checking if they were open-source, as the index also lists closed-source libraries and frameworks. Maven Repository gave us the most promising results, with repositories that had up to 30,000 stars.

Another way of measuring popularity could have been through download statistics - both monthly and all-time. However, these statistics about Java libraries are not available to the general public. Unlike platforms like crates.io for Rust, a community-made registry that shows the number of all-time downloads a crate[1] has, Java lacks a platform with this level of transparency. Public indexes like Maven Repository only list officially indexed libraries that used jqwik as a dependency, but they do not provide download counts or usage metrics. This limitation strengthened our decision to use GitHub stars to measure popularity.

In the end, we found seven repositories that met our criteria: `kafka`, `crate`, `simple-binary-encoding`, `Mekanism`, `graph-data-science`, `java-storage`, and `Poset`. Among these, `kafka` was by far the most popular, with 30.1k GitHub stars, followed by `crate`, with 4.3k stars, and `simple-binary-encoding`, which had 3.2k stars. The remaining repositories had fewer stars but were still relevant due to their usage of jqwik and their open-source availability. An overview of these repositories can be found in Table 1.

It is worth mentioning that `Poset` had no GitHub stars. Despite its lack of apparent popularity and the fact that it had only one contributor, we decided to include it in our analysis. The long-term goal of this research is to improve debugging, and the vast majority of coders do not have enough experience to write code without making mistakes. Analyzing this repository allowed us to gain insights into more isolated, less experienced, and less refined environments. We chose a well-structured repository with a clearly defined purpose to ensure the results are still relevant, making it suitable for our study.

3.3 Coding Techniques

We conducted a qualitative analysis of the repositories we found, using inductive coding to categorize the properties identified in each PBT. This method involves creating labels directly from the data instead of making a list beforehand. As we examined each repository, we iteratively refined the existing codes and added new ones to highlight existing patterns. The process required multiple passes through the data to ensure that the resulting categories were accurate and meaningful in representing the diverse uses of property-based testing.

Given the nature of this research topic, which spans multiple languages and frameworks, it was essential to have con-

[1] A "crate" is the Rust community's term for a library or package, similar to a Java library.

Repository	Version	Stars	LOC	PBT Number
kafka	3.9.1	30.1k	1.5M (1.2M, 84.9%)	19 (0.08%)
crate	5.10.7	4.3k	881k (804k, 91.23%)	2 (0.02%)
simple-binary-encoding	1.34.1	3.2k	111k (68.3k, 61.52%)	5 (0.74%)
Mekanism	1.20.1	1.5k	495k (259k, 52.23%)	3 (3.19%)
graph-data-science	2.16.0	674	1.1M (529k, 46.21%)	86 (1.2%)
java-storage	2.52.2	116	332k (316k, 95.31%)	24 (1.87%)
Poset	1.0.0	0	1.7k (1.6k, 91.94%)	8 (16.67%)

Table 1: Overview of analyzed repositories. "Stars" indicates the number of GitHub stars at the time of analysis. "LOC" shows the total lines of code per repository, with the number and percentage of Java lines in parentheses. "PBT Number" indicates the number of property-based tests (PBTs) from each repository, along with the percentage of PBTs relative to the total number of tests in that repository.

sistency across the analyses. After completing the inductive coding process separately for each language and framework, all the papers converged toward a set of shared terms and labels. This common vocabulary served as a standard dictionary, which we then used as a basis for the individual analyses.

The terms we found for our dictionary are:

- INVARIANT, a property that must hold throughout the execution of a test,
- IDEMPOTENCE, the principle that running your code more than once is equivalent to executing it once,
- HARD TO PROVE, EASY TO VERIFY, a concept implying that while it may be hard to formally prove the code's correctness, it is easy to verify that the result it gives is correct,
- ROUND TRIP, which implies modifying the generated input and attempting to return to the original value, such as encoding and then decoding a value that should yield the same value at the end,
- STRUCTURAL INDUCTION, a technique for verifying code by first showing that it works for simple inputs to prove that it also works for larger and more complex inputs,
- TEST ORACLE, an alternative implementation or trusted reference of the algorithm used to verify the code's output against, and
- DIFFERENT PATHS, which describes executing methods in varying orders while still arriving at the same correct result.

Starting with this dictionary as a foundation, each paper added language-specific codes or terms to help capture unique aspects of their domain in their analysis. The one label we added was MUTATION, which represents a valid or intended change in the state of the SUT.

3.4 Other Measures

When analyzing the selected repositories, we considered several other measures beyond the properties to gain insights into how PBTs are used as a testing strategy. These measures helped us compare property-based testing to other types of testing and answer our questions related to generators and shrinkers. Specifically, we used the following measures:

- **lines of code (LOC)**, both overall and Java-specific, to help understand the role of Java in large codebases,
- **PBT density**, calculated as the ratio of property-based tests to total tests, to see how often developers implement property-based testing and why,
- **property decomposability**, which refers to whether properties could be broken down into smaller properties and is meant to give insights into possible bug origins,
- **number of tries**, which represents the amount of times a PBT is executed with different inputs and is measured to assess the thoroughness of each test,
- **input filtering**, which can overly restrict the input, potentially leading to missed edge cases,
- **use of custom generators and/or shrinkers** to answer our research questions, and
- **exception assertions** to determine whether tests explicitly verified failure conditions or exceptional behaviors.

We measured the lines of code in each repository using a tool called Tokei, which reports statistics such as the number of files, lines of code, and comments for all programming languages it detects within a project.

In addition, we documented the release version of each repository we analyzed to ensure our results were replicable and valid.

All the results are available on the 4TU.ResearchData platform [6].

4 Results

This section presents the results of our analysis of open-source repositories, including the use of PBT in their testing strategies, label distributions, the use of generators and shrinkers, and other relevant measures. We also provide observations regarding these statistics and any conclusions that we drew from them.

4.1 PBT as a Testing Strategy

We analyzed seven repositories, listed in Section 3.2. As shown in Table 1, these repositories contained a total of 147 property-based tests. We initially aimed to analyze 50 PBTs in total, but we surpassed our target, stopping at 84 tests due to time constraints. In particular, we decided to omit some of

the PBTs from graph-data-science and kafka because they were similar to those we had already examined. We made this decision to form a more diverse sample space from a larger number of repositories.

As shown in Table 1, Java is the primary language across all the analyzed repositories. Five of the seven repositories have over 60% of their codebase written in Java, while graph-data-science and Mekanism, though below this threshold, still use Java more than any other language. The repositories also use Java to program the primary logic of their codebases, as opposed to languages like TypeScript or Python, which are used to generate user interfaces and documentation, or JSON configuration files that Tokei still counts. These statistics imply that Java plays a central role in the projects and suggest that the overall testing strategy should prioritize the correctness and reliability of Java over those of any other language.

With this context in mind, we can now examine the role of PBT within these testing strategies in more detail. Also detailed in Table 1, PBTs represent a small fraction of the total tests. In six out of seven cases, they represent less than 2%, with the only exception being the repository with no stars. This trend suggests that property-based testing has a supplementary or supportive role rather than being the primary testing method in projects.

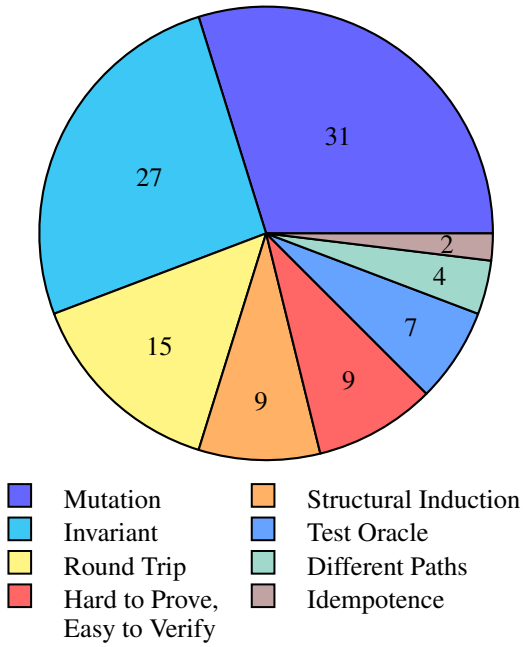

Figure 1: Distribution of assigned labels among analyzed property-based tests. Tests may have multiple labels. Total labels assigned: 104

4.2 Label Distributions

The distributions of labels across repositories can reveal meaningful patterns in how developers use property-based tests in practice. As shown in Figure 1, the most frequent labels we found were MUTATION and INVARIANT, followed by ROUND TRIP. Together, these three categories account for more than 60% of all labels assigned, indicating that developers tend to focus on properties that promote consistency and functional integrity.

When analyzing the repositories individually, we observed that most of them focused on one type of property-based test more than others. For example, java-storage predominantly used ROUND TRIP tests, kafka focused on STRUCTURAL INDUCTION, and graph-data-science primarily used MUTATION tests, with a smaller number of INVARIANT tests. Moreover, eight of the nine tests labeled with STRUCTURAL INDUCTION originated from the kafka repository alone. These trends suggest that developers often begin using property-based tests with a specific purpose in mind, typically aligned with the behavior and intentions of the system under test. Once they fulfill this purpose, they also look at other ways PBT can be applied in their codebase.

However, it is essential to note that since we only examined a subset of the PBTs in graph-data-science and kafka, the label distributions observed in these repositories may not accurately represent their overall use of property-based tests. For this reason, the previous conclusion may not hold if we include additional tests in the analysis.

4.3 Generators and Shrinkers

Two of our research questions focused on the use of generators and shrinkers. As summarized in Figure 2, out of 84 analyzed tests, 33 used custom generators, while none of them used custom shrinkers. In most cases, developers relied on jqwik's built-in generators, which were sufficient because the required input was simple. When inputs consist of basic types such as numbers, strings, booleans, or lists of primitive types, the default generation strategy combined with filtering proved sufficient for most tests.

Another notable pattern we observed is that when developers did implement custom generators, they sometimes created helper classes used exclusively for testing purposes. For example, the java-storage repository defines a static `TestData` class, which is instantiated only within a custom generator for a PBT in `ChunkSegmenterTest.java`. This class also had one of its formatting methods used in three unit tests but nowhere else. Such classes can increase the risk of introducing bugs because their behaviors are not independently verified. Any logic errors they contain can compromise the validity of the tests and hide bugs in the production code.

As previously stated, none of the tests used custom shrinkers for their failing inputs. We concluded from this that coders either don't take shrinking into consideration because their tests are already passing or they believe that jqwik's default shrinker is good enough for their needs. Another possibility is that developers are generally unaware of shrinkers and the possibility of implementing a custom one. However, upon inspecting each PBT, we also found no apparent reason to use custom shrinkers.

The only mentions of shrinkers we found were in the kafka repository, which was the only one to use jqwik's `AfterFailureMode` option. Specifically, 13 of their tests used `AfterFailureMode.SAMPLE_ONLY`, which configures

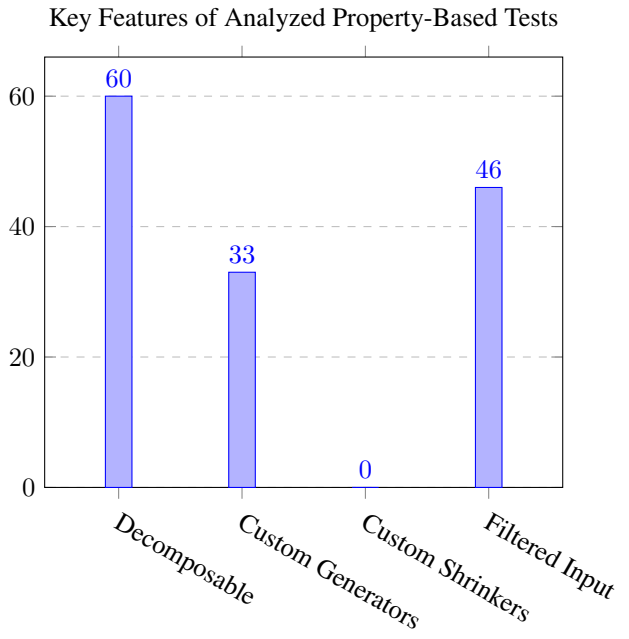

Figure 2: Distribution of analyzed property-based tests by four key characteristics. "Decomposable" refers to a test that can be broken down into multiple smaller, independently testable properties. "Custom Generators" and "Custom Shrinkers" represent ways for developers to define how input is generated or how failing input is minimized, respectively. "Filtered Input" refers to constraints set on the automatically generated inputs.

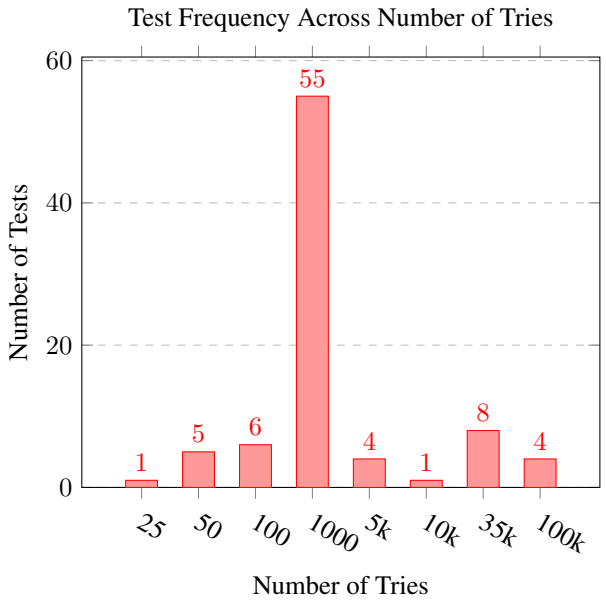

Figure 3: Distribution of tests by number of tries. This value represents how many times a property-based test will run. 1000 is jqwik's default number of tries.

jqwik to only rerun the final shrunk input that caused the failure, rather than re-executing the entire test with new input samples. This option could indicate that developers prioritize seeing the minimal failing output over the shrinking process, which helps in the debugging process.

4.4 Other Measures

Filtering is a key element of generators as it helps guide the input generation process. As shown in Figure 2, 46 tests used this technique. These tests include both the ones that implemented custom generators and those that did not. While filtering helps exclude unpermitted inputs, it can also reduce input diversity and even introduce bias toward narrower regions of the input spaces. We left these aspects as future work due to the time constraints associated with our research and the complexity of the tests.

Property-based tests are designed, as the name suggests, to verify specific properties that the SUT should satisfy. However, in practice, developers test multiple properties within a single test. This approach can complicate debugging, as errors can then originate from different properties, making it harder to identify the exact cause of the bug. 60 of the PBTs we analyzed can be further decomposed into smaller tests that focus on individual properties. Doing so would simplify tests, making them more maintainable and ensuring that failures are easier to trace to their cause. At the same time, decomposition also increases the total number of tests, which may result in redundant executions of the same code to verify different properties. In large-scale projects, this overhead can significantly increase implementation and execution times for tests, which is a big compromise for what might be a small benefit.

An interesting observation from our analysis was the variation in the number of tries configured for different property-based tests, as presented in Figure 3. Developers can set this option in the library configuration file or as shown in Listing 1. Most tests used jqwik's default of 1,000 tries, but some of them had different values. The lowest we noted was just 25 tries in `java-storage`, and the highest was 100,000 in the same repository, as well as in `Mekanism`. Many tests used small counts, such as 25, 50, or 100 tries, likely to speed up test execution. However, lowering this number can be risky, as it may prevent generators from reaching edge cases, reducing test effectiveness.

In `kafka`, this behavior seems to be justified for some of the tests by the fact that the generator only gives two values: an enum and a "seed" value. The enum does not have many values, but the seed represents a number that can be easily generated more than 50 times without finding duplicates. In contrast, this repository also includes tests with 5,000 tries to check string prefix matching. Conversely, `Poset` overwrites the default with 35,000 tries, which is unnecessarily large. We found no apparent reason for this behavior.

Another notable observation we made was that `kafka` and `crate` were the only repositories to use jqwik's assumption feature. Specifically, the `Assume.that()` method evaluates a given condition and skips the test execution without counting it as a try if the condition fails. `kafka` had five such tests, and `crate` had one. They were the only repositories to do so. `crate` used this functionality to ignore the PBTs that had one of its integer parameters larger than another. `kafka` used these assumptions to verify that the generated strings had a

valid structure for the SUT. The strings had an interface for a generator attached to them, but the interface was empty, and no method implemented it. This was unusual, as it appeared to serve no purpose.

The final aspect we examined was the scope of the PBTs: whether they tested isolated functionality, elements of the environment, or integration between systems, modules, or multiple classes. We found four integration tests and no environment-level tests. The others were more similar to unit tests, as they only tested small parts of code functionality. This behavior suggests that developers primarily use property-based testing to focus on the properties behind small parts of their code rather than higher-level interactions.

5 Discussion

In this section, we outline the limitations of our research, identify potential areas for improvement, and explore any new research questions that have emerged from the current findings.

5.1 Threats to Validity

One of the most significant limitations of our research was the time constraint. We conducted this project over 10 weeks, during which we gathered data from repositories, analyzed the data, drew conclusions, and wrote this paper. Given more time, we could have examined a larger number of repositories, resulting in a more representative overview of how jqwick is used in practice and, therefore, have more accurate results for our answers to the research questions. Despite this limitation, we still believe that the sample size we currently have is large enough to consider our results and insights relevant and meaningful.

A second threat to the validity of our analysis lies in the complexity of some of the PBTs we encountered. In multiple cases, it was difficult to understand the purpose of a test or the property it was designed to verify. As a result, we may have misclassified some tests, or some subtle errors may have gone unnoticed. While we interpreted each test as accurately as possible, this possibility remains a potential source of error in our analysis.

Our last limitation for our research stemmed from the fact that we only had access to open-source repositories. For this reason, we could not examine larger and more widely used projects that may potentially have more structured and effective testing strategies. Moreover, such projects could use property-based testing as a more significant part of their testing suites, making the results from such an analysis more representative and accurate for jqwik usage.

5.2 Potential Improvements and Future Work

We aimed to make our research as thorough and well-justified as possible, but there are areas that can still be improved. One clear improvement to our research would be gathering input from developers. Understanding how and why developers use property-based testing in their testing strategies can validate or challenge the assumptions we made based on our results. Although we base them on the patterns we observed and concrete results, they are still external observations that lack context, such as the motivations and limitations of developers.

As mentioned in Section 4.4, filtering can introduce bias when generating data by converging towards some areas of the input sample space and overlooking others. This aspect can be concerning during testing, as it may cause developers to miss edge cases or even typical cases that could fail the tests. We suggest further examining this part of property-based testing by conducting a study on how input filtering affects input diversity and test effectiveness. Additionally, exploring automated methods for detecting excessive filtering could be a step forward in automating bug finding in general.

Although our study included a high-level analysis of shrinking in property-based testing, we still believe it is a topic worth investigating in more detail. Our sample space did not include any tests that implemented custom shrinkers, and as a result, it only provided limited insights into how Java developers use shrinking in practice. We recommend a focused study that specifically targets repositories that use custom shrinkers to gain a deeper understanding of their motivations, implementation patterns, and potential benefits or challenges associated with shrinking.

While our investigation answered our original research questions, it also raised new questions for us to consider. Notably, we focused on the structure and categorization of property-based tests rather than their effectiveness. We did not evaluate how well these tests detect bugs, how often they fail, or whether they improve code quality. These areas could provide a better understanding of the impact PBTs have on a project's quality and correctness.

6 Responsible Research

In this section, we present the aspects that make our research reproducible and replicable, and address any potential ethical concerns that may come up during our investigation.

6.1 Reproducibility

We designed our research to be fully reproducible. We based our analysis on open-source repositories that use the jqwik library for property-based testing. Open-source repositories, by their nature, are public and accessible to anyone for review. For each repository, we specified the version we analyzed, ensuring that anyone can reexamine the same codebases we did. To further reinforce the idea of reproducibility, we also archived the GitHub pages of the repositories, ensuring they remain accessible even if the repositories are removed. The tools we used to find the repositories are also publicly available, and we clearly describe the process we followed in selecting codebases for our dataset.

Due to time constraints, we could not examine all the PBTs in two of the repositories. However, to ensure that there is no bias or selective sampling in our dataset, we chose the tests we examined from these repositories entirely at random. We made our results available on the publicly available 4TU.ResearchData platform [6] to allow others to reproduce our findings.

6.2 Replicability

Our findings are also replicable. We based our results and conclusions on the trends we found in our dataset, which rep-

resents an overview of the current state of PBT usage in practice. The margin between the three most common labels in our results and the other labels is significantly large, and any new data should follow these trends.

However, one area that may affect the replicability of our results is the use of custom shrinkers. Since we did not find any repositories that used them, our conclusions are limited by their absence. Any findings of shrinking implementations could disprove our conclusions while also providing new insights on this matter. We acknowledge this as a limitation in Section 5.1 and note that future research can improve our findings by expanding the dataset with more significant codebases from this perspective.

6.3 Ethical Considerations

We also examined other potential ethical concerns that may arise from our research. We did not store or access any sensitive or personal data, and we did not involve human interactions in our research, eliminating any concern related to privacy or consent. Furthermore, we did not use AI tools to collect or analyze data. The only AI tool we used was Grammarly, an AI-based writing support tool, to assist with grammar and spelling corrections during the writing of this paper. To maintain transparency in our process, we also stated all our decisions and the motivations behind them, including our limitations and any compromises we made. Therefore, our work can be verified and extended by anyone.

7 Conclusions

This section presents our findings from the research, including an overview of the current state of property-based testing in practice and answers to the research questions we posed in our study.

7.1 The Current State of PBT Usage

Property-based testing is not yet being utilized to its full potential by developers in open-source Java projects. As our results revealed, most repositories utilized PBT for less than 2% of their tests, indicating a low interest in adopting PBT as their primary testing strategy. Additionally, the lack of repositories with high GitHub star counts in our dataset also supports this result. We were only able to find one repository with a relatively high number of stars, while the rest had fewer than 4,000 stars.

Beyond its limited adoption, our analysis also highlights how PBT, when used improperly, can introduce minor logic issues into test suites. These issues may arise from tests that check multiple properties simultaneously, the use of custom classes for testing, or excessive filtering of generated input data. They can introduce problems such as unclear failure cases, overlap, omission of problematic code areas, and even the introduction of new bugs. We consider these issues to be relatively minor because, as mentioned in Section 4.4, more than 95% of the tests we found are similar to unit tests rather than integration or environment tests. Despite these risks, the bugs appear to be localized and only related to logic, making them relatively straightforward to debug once detected.

7.2 Answering our Questions

In our investigation of property-based testing usage in open-source Java projects, we focused on identifying the types of properties that developers commonly test and understanding how these properties are typically expressed. We also considered PBT's role in improving the project's correctness guarantees and bug-finding strategies. Additionally, we analyzed scenarios where developers implemented custom generators and shrinkers to support their tests.

To answer **RQ1.** and **RQ2.**, our analysis reveals that developers most often test properties related to state transformations and behavioral consistency. The most common categories we found were MUTATION, INVARIANT, and ROUND TRIP, which together accounted for the majority of labeled tests. These properties are commonly expressed by mapping inputs to outputs, sometimes using test oracles and other times using predefined values, either generated or hardcoded within the test.

Additionally, we observed that developers tend to prioritize one type of property in their projects. This preference is likely influenced by the testing goals they had in mind or the characteristics of the system under test. This behavior suggests that developers focus more on isolated tasks rather than using property-based testing in an exhaustive manner.

Regarding **RQ3.**, as a testing strategy, property-based testing typically accounts for a minor portion of test suites for large-scale Java projects. Developers adopt PBT selectively, applying it to isolated use cases where this technique offers clear advantages. While effective in specific scenarios, developers do not integrate PBT enough in their projects to significantly improve their testing strategies.

In response to **RQ4.** regarding generators, we found that developers typically implement custom generators when working with custom classes. These classes may either be part of the system under test or created specifically to support test scenarios. The need for custom generators typically arises from their reusability, ease of use, and the fact that the default jqwik generators are unable to create instances of the SUT on their own. In most cases, developers pair them with filtering and implement custom logic to control how values are created, ensuring that the data respects the structures required for testing.

On the other hand, to answer **RQ5.**, custom shrinkers are never implemented in the repositories we examined. This finding suggests that jqwik's default shrinkers are either sufficient for most testing needs or that developers are unaware they can implement their own. In either case, the absence of custom shrinkers suggests that shrinking is not considered a concern, as no problems arise from using the default ones.

References

[1] Thomas Arts, John Hughes, Ulf Norell, and Hans Svensson. Testing autosar software with quickcheck. In *2015 IEEE Eighth International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, pages 1–4, 2015.

[2] Antonios Barotsis. *Property-Based Testing in Open-Source Rust Projects: A Case Study of the proptest*

Crate. Bachelor Thesis, Delft University of Technology, 2025.

[3] Arthur Lisboa Corgozinho, Marco Tulio Valente, and Henrique Rocha. How developers implement property-based tests. In *2023 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 380–384, 2023.

[4] David de Koning. *Property-Based Testing in Practice using Hypothesis: In-depth study on how developers use Property-Based Testing in Python using Hypothesis*. Bachelor Thesis, Delft University of Technology, 2025.

[5] Max Derbenwick. *Property-Based Testing in Rust, How is it Used?: A case study of the quickcheck crate used in open source repositories*. Bachelor Thesis, Delft University of Technology, 2025.

[6] Max Derbenwick, Harald Toth, David de Koning, Antonios Barotsis, Ye Zhao, Andreea Costea, and Sára Juhošová. Property-based testing in the wild! 4TU.ResearchData, 2025. doi: 10.4121/368f63ab-10fc-4603-a15a-bde25e72e778.

[7] Vinicius Durelli, Ricardo Monteiro, Rafael Durelli, Andre Endo, Fabiano Ferrari, and Simone Souza. Property-based testing for machine learning models. In *Proceedings of the 9th Brazilian Symposium on Systematic and Automated Software Testing*, pages 39–48, Porto Alegre, RS, Brasil, 2024. SBC.

[8] Harrison Goldstein, Joseph W. Cutler, Daniel Dickstein, Benjamin C. Pierce, and Andrew Head. Property-based testing in practice. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, ICSE '24, New York, NY, USA, 2024. Association for Computing Machinery.

[9] Harrison Goldstein, Joseph W Cutler, Adam Stein, Benjamin C Pierce, and Andrew Head. Some Problems with Properties. 2022.

[10] John Hughes. Quickcheck: A lightweight tool for random testing of haskell programs. In *Proceedings of the 5th ACM SIGPLAN International Conference on Functional Programming (ICFP '00)*, pages 268–279. ACM, 2000.

[11] John Hughes. *Experiences with QuickCheck: Testing the Hard Stuff and Staying Sane*, pages 169–186. Springer International Publishing, Cham, 2016.

[12] André Santos, Alcino Cunha, and Nuno Macedo. Property-based testing for the robot operating system. In *Proceedings of the 9th ACM SIGSOFT International Workshop on Automating TEST Case Design, Selection, and Evaluation*, A-TEST 2018, page 56–62, New York, NY, USA, 2018. Association for Computing Machinery.

[13] Cindy Wauters and Coen De Roover. Property-based testing within ml projects: an empirical study. In *2024 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 648–653, 2024.

[14] Ye Zhao. *Property-Based Testing in the Wild!: A Study of QuickCheck Usage in Open-Source Haskell Repositories*. Bachelor Thesis, Delft University of Technology, 2025.