The impact of type systems and test tooling on codified testing strategies: an exploratory multi-method approach

Patrick Anthony van Hesteren

The impact of type systems and test tooling on codified testing strategies: an exploratory multi-method approach

THESIS

submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER SCIENCE

by

Patrick Anthony van Hesteren
born in Capelle aan den IJssel, the Netherlands

TUDelft

Software Engineering Research Group
Department of Software Technology
Faculty EEMCS, Delft University of Technology
Delft, the Netherlands
www.ewi.tudelft.nl

The impact of type systems and test tooling on codified testing strategies: an exploratory multi-method approach

Author: Patrick Anthony van Hesteren
Student id: 4166132
Email: P.A.vanHesteren@student.tudelft.nl

Abstract

Software testing has been around for decades and many tools exist to aid developers in their testing process. However, little is known about the rate at which developers test their projects, the tools they use for these purposes and the impact of type systems on testing practices. Our work is the first of its kind to identify and classify available test tooling for programming languages Java, C, JavaScript and Ruby. By conducting a large scale automated analysis on open-source software projects, we show that both available test tooling and type systems have significant impact on codified testing strategies. Our observations are strengthened by targeted interviews and a large-scale survey among developers working with both statically and dynamically typed programming languages. The soft typing system described in [7] seems like a promising solution, allowing developers to work with the great flexibility less strict type systems provide, while simultaneously benefiting from strict type checks that help reduce the amount of testing required to ensure the correctness of a piece of software. However, future research is needed to estimate the feasibility of such a type system.

To help better aid developers in their testing process and to reduce some of the testing burden many developers seem to cope with, we propose a list of test tooling improvements based on our observations.

Preface

Before you lies my Master thesis, the work that concludes my 5 years as a Computer Science student at Delft University of Technology. Throughout this project I have been provided with a great deal of freedom, that has allowed me to grow both scientifically as well as personally. It has certainly been the most challenging and educational project that I have worked on as a student thus far. Looking back at the past months, I am most proud of what I have achieved, but it goes without saying that this work would never have been possible without the valuable advice and support of the many people that I would like to thank in this preface.

First of all, Andy Zaidman, thank you for offering me the opportunity to conduct this empirical study on testing practices. Throughout the project, you have trusted me and provided me with a great deal of freedom that allowed me to explore and get back to you with anything that seemed worthwhile to pursue. While we did not meet frequently, your advice was sound and most valuable when I needed it. You were most definitely as dedicated to this project as I was, as you managed to squeeze in time for our meetings even while being aboard in Canada.

I would like to thank Moritz Beller for his supervision and guidance while Andy Zaidman was absent. Moritz, your critical thoughts have certainly helped me grow and made my analyses more sound.

I am most grateful to all of the participants in the interviews and surveys that we conducted. Unfortunately, due to the unforeseen large number of responses, I am unable to thank each and everyone of you individually. Your perspectives have certainly helped shed light on new insights in this project and helped bring forward a large list of test tooling improvements. I certainly enjoyed all of our conversations and appreciated all of your enthusiasm.

Last, but certainly not least, I would like to thank the other two members of my thesis committee, Georgios Gousios and Cees Witteveen, for their efforts and feedback.

Patrick Anthony van Hesteren
Woerden, the Netherlands
August 23, 2017

Contents

List of Figures

Chapter 1

Introduction

Automated software testing (later on referred to as codified testing strategies) has been around for quite a while and many tools exist to help developer test their software projects. However, very little is known about the tools developers use to test their projects, the impact type systems have on testing approaches and the rate at which developers test. Some research exists on assert usage (e.g. [8]), usage of tests in Continuous Integration pipelines (e.g. [4]) or tools used for testing purposes (e.g. the annual JetBrains survey [1]), but thus far no study has been conducted to investigate the adoption rate of test tooling in practice. Little is known about the rationales for picking certain test tooling and the strengths and weaknesses of the tools involved in testing processes.

By deriving an overview of available test tooling, obtaining test data of a large number of software projects and confronting developers with our observations we hope to gain insight into these questions as we strive to answer the question ‘How are codified testing practices impacted by type systems and test tooling?’.

Our work is of an exploratory multi-method nature, which allows us to cross-check and validate our results as there is little literature available to compare our results against. Moreover, this multi-method approach allows us to target our research question from multiple angles and explain our observations using developer perspectives.

1.1 Research questions

The main research question ‘How are codified testing practices impacted by type systems and test tooling?’ consists of multiple parts and can therefore not be answered at once. It is therefore split up into 3 sub-research questions that can be answered on their own. By combining the results of all sub-research questions, the main question can be answered. The sub-research questions are defined as per the following:

- ‘What are state of the art codified testing strategies?’
- ‘What differences in codified testing strategies can we observe in software projects using different type systems?’

[1] https://www.jetbrains.com/research/devecosystem-2017/

- ‘How do developers reflect on the relationships of testing practices, type systems and test tooling involved?’

The first research question revolves around gaining insight in the codified testing spectrum, the approaches one can take in testing their system and the tools involved. The second research question uses this knowledge and helps us gain insight in the differences in codified testing strategies that we can observe in software projects. Unfortunately, pointing out the differences will not necessarily help us make sense of them. This is where the third research question comes in. It will look at type systems and test tooling from a different perspective by reflecting on developers perspectives and help us explain the results obtained in RQ2. Moreover, it can help us validate the approach taken in RQ2 by cross-checking the obtained data with perspectives of developers.

1.1.1 RQ1: What are state of the art codified testing strategies?

This research question will be answered by combining a literature study with an exploratory approach. First, an overview of the codified testing spectrum will be derived so that we can start to compile a list of available test tooling. This overview of tool classifications and their approaches can be obtained by summarizing available literature.

This overview of test tooling classifications can then be used to compile a list of available test tooling and its classification(s), which will serve as a basis for answering RQ2.

The deliverables for this research question include:

- An overview of the codified testing spectrum, based on literature
- A list of available test tooling per programming language, classified based on the derived overview of the codified testing spectrum

1.1.2 RQ2: What differences in codified testing strategies can we observe in software projects using different type systems?

In order to answer this research question, we first need a proper definition and overview of available type systems and their impact on programming practices. This definition and overview should be compiled based on available literature.

Moreover, a list of metrics should be compiled so that the adopted codified testing strategies can be weighted and compared for different programming languages. This list of metrics should also be defined based on available literature.

Once these overviews are in place, a selection of software projects should be made that will be analyzed. Once all this is in place, a mining tool should be build that is able to detect the list of test tooling derived in RQ1. The usage of test tooling should be tracked, so that it can be used with the derived metrics in RQ2.

Finally, all data should be used to derive measurements based on these metrics. The results should be split on type system level and be compared against each-other.

The deliverables for this research question include:

- A definition of type systems and their impact on programming practices, based on literature
- A list of metrics that can be used to compare test practices, based on literature
- A selection criterion for software projects that will be analyzed
- A mining tool that can detect test tooling identified in RQ1
- A sound analysis of the differences in codified testing strategies observed per type system

1.1.3 RQ3: How do developers reflect on the relationships of testing practices, type systems and test tooling involved?

In order to answer this research question and to make sense of the results on RQ2, a survey should be conducted to gain insights in how developers approach testing practices, how they choose their tooling, what the impact of type systems might be and how tooling helps (or hurts) them in their testing process. Moreover, in-depth interviews can be conducted with developers that take on out of the ordinary testing approaches, based on the data obtained in RQ2. By doing so, we can try to make sense of the rationales revolving around testing practices and explain the true impact of type systems and test tooling on codified testing practices.

The deliverables for this research question include:

- Interviews with developers that use out of the ordinary testing approaches
- A large scale survey among developers to gain insight into the relationship of testing practices, type systems and test tooling
- A sound analysis combining the interview and survey results with the data obtained in RQ2
- A list of possible test tooling improvements, based on the perspectives of developers and data obtained in RQ2

1.2 Thesis outline

Our work first describes type systems and codified testing strategies based on available literature in chapter 2. Based on the obtained literature, we conduct exploratory research to derive an overview of the state of the art codified testing strategies and classify these tools in chapter 3. Next, we describe the design of our automated mining approach in chapter 4, that is used in chapter 5 to describe the observed differences in testing approaches taken in various programming languages. We complement our mining approach with targeted interviews and a large-scale survey described in chapter 6 to cross-check our results with the perspectives of developers. Moreover, these perspectives help us explain the observed differences in chapter 5. We combine all data into a list of test tooling improvements in chapter 7, after which we reflect and discuss all results in chapter 8. We conclude our work by discussing the threats to validity in chapter 9 and summarizing our work, its main contributions and future research in chapter 10.

Chapter 2

Background

Before we can start to analyze the impact of type systems and test tooling on codified testing strategies, we need to gain a good understanding of type systems and their potential impact on programming practices. Moreover, we need to dig into the background of codified testing strategies and derive an overview of all possible strategies, so that we can make sure that we cover all available testing practices later on in our analysis. This chapter serves as the backbone of the thesis in the sense that available literature is structured and combined, which helps us in the process of conducting the analysis. This chapter contributes to both RQ1 (the overview of the codified testing spectrum, based on literature) and RQ2 (the definition of type systems and their impact on programming practices, based on literature).

2.1 Codified testing strategies

Codified testing strategies allow developers to automatically verify the correctness of a piece of software. Besides fault localization, assertions may also help developers better understand their code and avoid constructing faulty code in the first place [8]. Especially unit and integration testing prove to be efficient methods for early defect detection in the development phase of a piece of software [31]. Moreover, codified testing strategies can be used to improve confidence in the process of altering existing code without causing (parts of) the system to break [3]. According to [17], there is a negative correlation between assert density (number of asserts per 1000 lines of source code) and fault density, strengthening the importance of codified testing strategies.

2.1.1 Strategies

Software testing is involved in many stages in the software life-cycle. Moreover, codified testing strategies are applied to various levels of software development. Each strategy has a different nature, as well as a different purpose.

Codified testing techniques boil down to two different approaches. The first one is built on the assumption that similar input on a module will result in similar behavior of the piece of software, which is referred to as *partition testing*. It allows the tester to pre-define a test suite of manageable size. Moreover, it allows test coverage to be measured with regard to

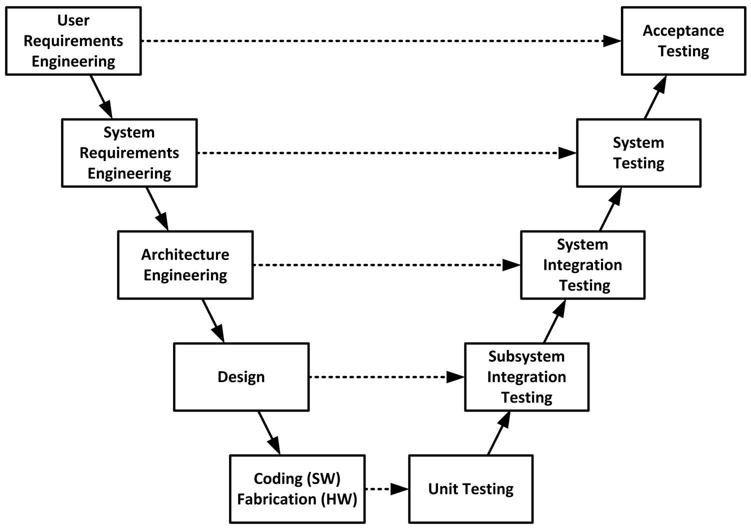

User Requirements Engineering
Acceptance Testing
System Requirements Engineering
System Testing
Architecture Engineering
System Integration Testing
Design
Subsystem Integration Testing
Coding (SW) Fabrication (HW)
Unit Testing

Figure 2.1: The V-Model, introduced by Paul Rook

the partition model that is used to create test scenarios for the piece of software. The other technique is called *random testing*, in which test cases (and their input) are chosen randomly from some sort of input distribution (e.g. a uniform distribution), without exploiting the specification of the piece of software or previous test scenarios [11]. In order to create a useful test suite, the chosen distribution may fit a certain operational profile, which describes the system its usage in production [24].

Partition testing is favoured over random testing in practice, since random testing was found to be less effective than partition testing methods [27, 12]. For this reason, the next sections will solely focus on codified partition testing techniques in the following chapters.

The V-model (figure 2.1) is a software development process which can be presumed to be an extension of the waterfall model. It was introduced by Paul Rook [28] in the late 1980s and is still used today. The V-model introduces relationships between the phases of the development life cycle and their associated testing phases. The purpose of V model is to improve efficiency and effectiveness of software development and to reflect the relationship between development and test activities.

The codified testing spectrum can be divided among three different categories: Unit testing, Integration testing, System (or functional) testing as described in the V-Model [19, 20]. The different categories will be explained and popular approaches for these strategies will be mentioned in the following subsections. **Note:** the V-model has been extended with software maintenance tests since its first introduction [20]. There is some overlap with the codified testing strategies spectrum (i.e. regression testing), but not all software maintenance tests mentioned in [20] are executed via codified tests, which is why they will not be covered in the next sections. **Note:** acceptance testing is also a part of the V-model. Usually, acceptance tests are executed to verify a product meets customer specified requirements. Typically, customers conduct these *manual* tests on a product that is developed externally [20]. Since these tests are of a manual nature, they fall outside of the codified testing spectrum and will therefore not be covered in the following sections.

Codified testing strategies can be of a white or black box nature. White box approaches take into account the internal logic and code structure of the system that is to be tested. Black box testing techniques examine the fundamental aspects of the system and have little or no relevance with the internal logical structure of the system [16]. Typically, the low level tests are of a white-box nature, whereas high level (abstract) tests are of a black-box nature.

Unit testing

The rationale behind unit testing is to test a basic part (unit) of a piece of software, that is the smallest part possible to test. These testable parts are often referred to as units, modules or components [19]. These type of tests happen at the lowest level possible and are run according to a strategy that takes the control structure of a program as the basis for developing test cases. This white box approach is particularly popular due to its simplicity and vast amount of tooling that allows for these tests to be ran [25].

Within unit tests, various ways exist to verify the correctness of a single unit. Well-known options include statement, branch and path testing. Statement testing requires each statement in the program to be executed by at least a single test case. Branch testing requires each option in a branch (e.g. an if-else statement) to be hit at least once by a single test case. Finally path testing requires all possible paths in a unit to be executed, but proves to be practically infeasible since even small programs can have a huge number of paths [25].

Other strategies try to breach the gap between branch testing and the unfeasible path testing, which try to structure path testing by grouping ’similar’ paths (e.g. iterating a loop 2 or 3 times is regarded to be a similar test and will therefore be grouped). These strategies include structured path testing [14] and boundary-interior path testing [13].

Finally there are data-flow strategies that randomly select variables and track back their usage to the beginning of the module and generate a path based on their flow.

Surely it is not always feasible to cover all statements of one or more testing approaches. Often, developers require a certain threshold to be reached in order for a build to succeed when running these tests. If the coverage of the testing approach falls below the threshold, the software build process will fail. This creates a need for additional tests to be created to verify the correctness of the unit.

In addition to coverage verification of unit tests, mutation testing can be included in the build process of a piece of software. Mutation testing introduces mistakes (mutations) into the source code that developers might be introducing to the system and verifies that these mistakes lead to failing tests. If the test fails, this means that the test is properly verifying the unit’s behavior. However, if the test succeeds, this either means that developer did not spend enough time developing the source code and/or test code, or the test is simply not covering the unit’s behavior. If test mutations pass, the software build process will fail. This requires the developer to re-assess the test and/or unit source code. While less commonly applied in practice, mutation testing can also be applied in other codified testing strategies.

Integration testing

Often, units are dependent on one another and must be combined into a larger structure within a piece of software. Integration tests aim to verify that different units are connected and communicating in a proper way. This black-box testing technique is often performed on both the interfaces between the different components, as well as the larger structure that has been constructed using these components [19]. Note that there is whole spectrum ranging from testing integration between two classes up to testing integration with the production environment.

Note: other strategies exist, but these can actually be regarded as sub-categories of integration testing. These other strategies include:

- Smoke testing, which are the first step in verifying a system's correct behavior. Smoke tests are similar to functional testing, except within a smoke test various systems may be mocked or stubbed - making them more easy to be executed. If a smoke test fails, there is no need to perform a full functional test, that is likely to have a much longer duration.
- Regression testing, in which tests are executed that verify that previously introduced bugs are no longer a part of the system - even after making modifications to a piece of software. They are quite similar to functional testing, except this time there is a strict focus on verifying that new functionality did not (re-)introduce inconsistent behavior. Regression testing is also part of the software maintenance tests categorization in [20]. The importance of regression testing was also shown in [3]. The interviews conducted with developers, described in chapter 6, indicated that combating regression is the most important gain from testing practices.

System (functional) testing

This approach aims to affirm the end-to-end quality of a piece of software [19]. These tests are often based on (formal) requirements and/or functionality of a system [6]. Non-functional requirements, e.g. response time, can also be verified. In this black-box approach, test data is constructed from the specification by using methods such as equivalence partitioning and boundary value analysis [2], after which the results of the tests are compared against the desired behavior of the system as described in the requirements.

2.2 Type systems

A type system defines how a language classifies similar types of values and expressions, how it can manipulate them, and how they interact with one another. This generally includes a description of the data structures that the language enables [26]. In general, static languages are strongly and statically typed. On the other hand dynamic languages are often weakly and dynamically typed. While uncommon, exceptions to these rules do exist: (i.e. Lisp is strongly typed).

[7] argues that programmers should be not be provided with a black or white choice between static or dynamic typing. Instead, softer type systems should be created. This means that static typing should be applied where possible and dynamic typing should be used where needed. However, there is a big gap between there is a discontinuity between statically and dynamically typed languages. Moreover there are large technical *and* cultural differences between the respective language communities [21].

2.2.1 Statically typed programming languages

Strong typing enforces certain rules, which define how programs may interact with various elements by detecting and correcting type-related errors at compile time. Statically typed programming languages allow earlier error checking, better enforcement of disciplined programming styles, and the generation of more efficient object code than languages where all type consistency checks are performed at run time[7]. On the other hand, [21] claims that static typing provides a false sense of safety, since it can only prove the absence of certain errors and does not guarantee that no run-time errors will occur.

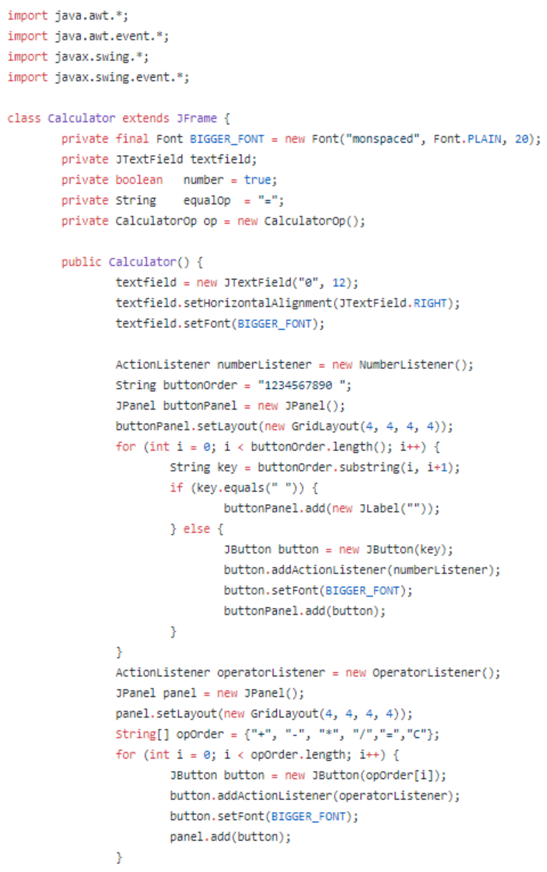
2.2.2 Dynamically typed programming languages

Weak typing does not enforce any of such explicit rules. However, it still uses a type-violation mechanism, for cases in which a rule violation will not cause problems for a program. Weak typing can correct correct the type-related errors for which there are no exceptions at run-time. This permits maximum programming flexibility at the potential cost of efficiency and security [1]. Software can be more compact because for example, there is no type-declaration overhead. Depending on the application, writing code in an dynamically language (e.g. Python) can reduce code by 5-10 times when compared to a static language (Java or C++) [26]. This is backed by an example calculator application (courtesy of [1]), written using Java (2.2) and Ruby (2.3) which showcases the strength of the flexibility dynamically typed programming languages can provide as there is much less boilerplate code.

As there is no need to explicitly declare variables and some other structural items before they are used, developers can introduce variable types, module names, classes and functions on the fly, thereby greatly improving flexibility. This also allows them to write code more quickly and efficiently, as it removes some of the detailed and repetitive work from programming and lets developers focus more on creative matters.

Since dynamically typed programming languages provide less of a safety net than statically typed programming languages, one might argue that projects written using dynamic languages require additional test effort (e.g. a higher assert density) than those written in static languages, to achieve the same level of confidence in a piece of software. [29] shows a study in which runtime error detection found only 11% of the faults, whereas assertions were able to identify another 53% of the faults missed by the basic runtime checker.

[1] https://github.com/kassisdion/Gui_calculators



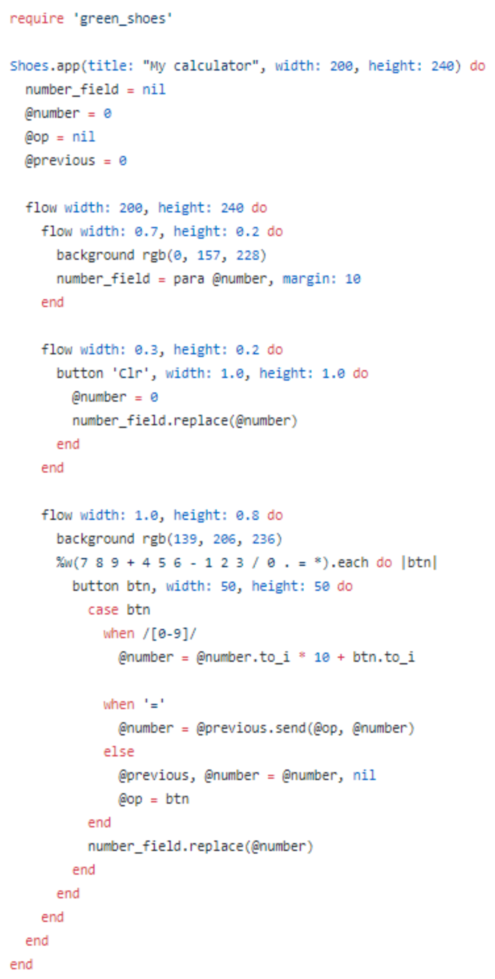
```java
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import javax.swing.event.*;

class Calculator extends JFrame {
        private final Font BIGGER_FONT = new Font("monspaced", Font.PLAIN, 20);
        private JTextField textfield;
        private boolean   number = true;
        private String    equalOp  = "=";
        private CalculatorOp op = new CalculatorOp();

        public Calculator() {
                textfield = new JTextField("0", 12);
                textfield.setHorizontalAlignment(JTextField.RIGHT);
                textfield.setFont(BIGGER_FONT);

                ActionListener numberListener = new NumberListener();
                String buttonOrder = "1234567890 ";
                JPanel buttonPanel = new JPanel();
                buttonPanel.setLayout(new GridLayout(4, 4, 4, 4));
                for (int i = 0; i < buttonOrder.length(); i++) {
                        String key = buttonOrder.substring(i, i+1);
                        if (key.equals(" ")) {
                                buttonPanel.add(new JLabel(""));
                        } else {
                                JButton button = new JButton(key);
                                button.addActionListener(numberListener);
                                button.setFont(BIGGER_FONT);
                                buttonPanel.add(button);
                        }
                }
                ActionListener operatorListener = new OperatorListener();
                JPanel panel = new JPanel();
                panel.setLayout(new GridLayout(4, 4, 4, 4));
                String[] opOrder = {"+", "-", "*", "/","=","C"};
                for (int i = 0; i < opOrder.length; i++) {
                        JButton button = new JButton(opOrder[i]);
                        button.addActionListener(operatorListener);
                        button.setFont(BIGGER_FONT);
                        panel.add(button);
                }
```

Figure 2.2: Part of a calculator program written using a statically typed programming language (Java)

2.3 Conclusion

There are many different codified testing strategies that revolve around testing software systems from different angles. Ideally, black- and whitebox testing approaches should be used to complement each-other to increase confidence in the correctness of a piece of software. The codified testing spectrum consists of unit-, integration and system testing.

Type systems appear to have significant impact on development practices. Dynamically typed programming languages can reduce code by 5-10 times and thereby speed up development compared to statically typed languages, but lack compile-time type checks that are present in statically typed programming languages. Codified testing practices (assertions) prove to be more thorough and accurate at detecting errors than runtime error detection. As developers using dynamically typed programming languages miss out on compile time type checks, they need to put in additional testing effort to gain the same confidence in the correctness in their software when compared to statically typed languages. The extent to which this is the case will need to be researched in the next chapters.



```ruby
require 'green_shoes'

Shoes.app(title: "My calculator", width: 200, height: 240) do
  number_field = nil
  @number = 0
  @op = nil
  @previous = 0

  flow width: 200, height: 240 do
    flow width: 0.7, height: 0.2 do
      background rgb(0, 157, 228)
      number_field = para @number, margin: 10
    end

    flow width: 0.3, height: 0.2 do
      button 'Clr', width: 1.0, height: 1.0 do
        @number = 0
        number_field.replace(@number)
      end
    end

    flow width: 1.0, height: 0.8 do
      background rgb(139, 206, 236)
      %w(7 8 9 + 4 5 6 - 1 2 3 / 0 . = *).each do |btn|
        button btn, width: 50, height: 50 do
          case btn
            when /[0-9]/
              @number = @number.to_i * 10 + btn.to_i

            when '='
              @number = @previous.send(@op, @number)
            else
              @previous, @number = @number, nil
              @op = btn
          end
          number_field.replace(@number)
        end
      end
    end
  end
end
```

Figure 2.3: A a calculator program written using a dynamically typed programming language (Ruby)

Chapter 3

State of the art codified testing strategies

Using the codified testing spectrum described in the previous chapter, we can now use this overview to classify the available test tooling per programming language. This chapter will first define the programming language scope of the thesis. After doing so, all available test tooling is identified and classified according to the codified testing spectrum. Finally, this chapter will lay out the differences in the availability of tooling based on their classifications per programming language and type system. This chapter contributes to both RQ1 (the list of available test tooling per programming language, classified based on the derived overview of the codified testing spectrum) and RQ2 (part of the analysis of the differences observed in codified testing strategies per type system). **Note:** Most codified testing strategy tooling comes in the form of frameworks, which is why the terms framework and test tooling are used jointly.

3.1 Scope definition

Due to the time constraint of this thesis, it is practically infeasible to investigate the codified testing strategies for all existing programming languages. Therefore, the list of top programming languages in 2016 as gathered from[9] is used to narrow down the scope of our research. This list is compiled using multiple sources, unlike other lists such as the well-known language trends on GitHub [1]. By not only looking at raw commit data, this list also includes trending / rising languages and languages that are not amongst the top languages, but are still frequently discussed amongst developers. It should also be noted both lists have a lot in common. [9] has compiled its list using the following sources [2]: Google Search results, Google Trends, Twitter, GitHub, Stack Overflow, Reddit, Hacker News, CareerBuilder, Dice and the IEEE Xplore Digital Library.

The programming language scope will consist of 2 statically typed and 2 dynamically typed programming languages, as this will reduce outliers (one language may be very suited

[1]https://github.com/blog/2047-language-trends-on-github

[2]http://spectrum.ieee.org/ns/IEEE_TPL_2016/methods.html

Table 3.1: Programming languages selected for this study, based on the list of top programming languages in 2016 or existing test behavior research

Dynamically typed languages	**Statically typed languages**
Ruby	C
JavaScript	Java

for testing, whereas the other may not) and allow us to compare the impact on testing differences by type system. These should ideally be languages that are popular among developers and should also have been analyzed with regard to their testing behavior before. There are two studies that have conducted similar research:

- [4] analyzed Java and Ruby projects, albeit focusing on tests in Continuous Integration pipelines.
- [8] analyzed the assert usage in C and C++ projects.

As the results of these thesis should be compared and used in conjunction with previous research, Java, Ruby and C will be added to the scope of the thesis. Moreover, one more dynamic programming language will be added. To the best of our knowledge, no further research has been conducted on other dynamic programming languages. For this reason, the most popular dynamic programming language will be added: JavaScript. The selected languages and their typing are shown in table 3.1

3.2 Available tooling for applying codified testing techniques

In order to empirically derive the way software projects are tested using codified testing strategies, a testing framework scope needs to be defined. To the best of our knowledge, no *up-to-date* overview of available testing tools exists on a programming language basis. The list of present day testing frameworks can be derived via various ways:

- Compile a list of what is used in practice by going through software repositories
- Compile a list of testing frameworks by conducting a questionnaire
- Derive a list of testing frameworks based on available literature
- Perform a random selection by searching for testing frameworks on the internet

Unfortunately, most of these options are not viable for the duration and purpose of this study. Going through repositories manually would be very ineffective and above all very time-consuming. It is likely that repositories would have to be sampled due to time constraints. Since the framework usage is likely to be skewed (some frameworks are widely used, but most are hardly used at all), only a limited set of frameworks would be retrieved.

Conducting a questionnaire would also be time consuming. Moreover, since the framework usage is assumed to be skewed, chances are that most participants will not be familiar the majority of the existing frameworks. Once again, this would result in a limited set of frameworks.

While quite some relevant literature exists on software testing, only few studies mention involved tools. To the best of our knowledge, not much literature is available on software testing tooling. This approach would once again result in a limited set of frameworks.

Finally, performing a random selection by searching for testing frameworks on the internet would likely also not result in a full comprehensive list of existing present day testing frameworks. However, the approach is very efficient time-wise. Moreover, by choosing a broad search query, it becomes likely that results of discussions on sites such as Stack Overflow will pop up, which would make the list of frameworks less skewed than other approaches.

Due to the time constraint of this study, a random selection will be performed in order to obtain the list of testing frameworks on the internet. The random selection will be performed using the following settings:

- Search on https://www.google.com
- Language settings: English
- All queries will be executed on the same machine in a private window to prevent user-tailored results
- Query the search engine for the following query: 'Test frameworks *X*', in which *X* represents the programming language for which the list should be obtained
- Take into account all findings on the first 3 result pages, while also following up on direct links to other pages that may be related to a testing framework
- Filter out any elusive results

The obtained list will be combined with a list of well-known frameworks that may not have popped up during the random selection, as the purpose of the list is to be as thorough as possible.

Frameworks are classified based on the nature of their tests and their domain within the codified testing spectrum: Unit, Integration or System tests. Frameworks have been classified based on the following publicly available information:

- Author statements claiming the framework should be used for a certain testing purpose
- Example code included in the framework, showcasing a certain testing purpose
- Discussions on popular developer websites, such as Stack Overflow, showcasing a certain testing purpose

Classifying a framework is rather easy when authors state the intended use or showcase excellent examples. Unfortunately, this is much harder when there is a lack of publicly available information. If the information provided was not sufficient, the classification was based our own judgment that included framework syntax, method scope and method naming conventions, as these can be good indicators for the intended usage of a framework. **Note**: while we tried to make the classification process transparent, some bias may have been introduced. Moreover, one might argue that the classification of unit, integration and system testing is not as straightforward was described in chapter 2 and the classification of tests may well overlap. One might even argue that a dedicated unit testing framework can be used for integration testing in some cases. While this is true, its syntax and features were not meant to do so and therefore these frameworks were not labeled for these purposes.

During the classification it was observed that some frameworks did not fit these categories well as they were of a different, more supportive nature. Therefore their categorization was extended:

- Unit tests
- Integration tests
- System tests
- Matcher framework
- Mocking / Stubbing / Fixture framework
- UI tests
- Performance tests
- Test (code) generator
- Replay framework (replay requests / responses)
- Test runner (can run tests, but does not support tests or asserts on its own)
- Monkey tests (generate random input / user behavior and verify that the system does not crash)

Note: some frameworks offer extensive functionality and may therefore be labeled with multiple classifications.

As the obtained list will also be used to infer test behavior as described in chapter 4, the obtained list of frameworks will be split up between frameworks that will be part of this setup and those that will not. The obtained results of frameworks that have been selected for this study have been summarized in the following tables: 3.2, 3.4, 3.6 & 3.8. Frameworks that have not been selected for this study and the reason for their exclusion have been summarized in the following tables: 3.3, 3.5, 3.7 & 3.9.

3.2.1 C Frameworks

Framework	**Classification**
Default assert	Unit tests
libcbdd [3]	Unit tests
AceUnit [4]	Unit tests
cfix [5]	Unit tests, Mocking / Stubbing / Fixture framework
Cgreen [6]	Unit tests, Mocking / Stubbing / Fixture framework
CHEAT [7]	Unit tests
Check [8]	Unit tests
Cmocka [9]	Unit tests, Mocking / Stubbing / Fixture framework
Cmockery [10]	Unit tests, Mocking / Stubbing / Fixture framework
CppUTest [11]	Unit tests, Mocking / Stubbing / Fixture framework
Criterion[12]	Unit tests, Mocking / Stubbing / Fixture framework
Ctest [13]	Unit tests
Cunit [14]	Unit tests
CuTest [15]	Unit tests
Cutter [16]	Unit tests
Embunit [17]	Unit tests
FCTX [18]	Unit tests, Mocking / Stubbing / Fixture framework
Kyua [19]	Unit tests
Lcut [20]	Unit tests, Mocking / Stubbing / Fixture framework
LibU [21]	Unit tests
MinUnit [22]	Unit tests
Mut [23]	Unit tests

[3] https://github.com/nassersala/cbdd
[4] http://aceunit.sourceforge.net/
[5] http://www.cfix-testing.org/unit-testing-framework/windows/
[6] https://github.com/cgreen-devs/cgreen
[7] https://github.com/Tuplanolla/cheat
[8] https://libcheck.github.io/check/
[9] https://cmocka.org/
[10] https://github.com/google/cmockery
[11] https://cpputest.github.io/
[12] https://github.com/Snaipe/Criterion
[13] https://cmake.org/Wiki/CMake/Testing_With_CTest
[14] http://cunit.sourceforge.net/
[15] http://cutest.sourceforge.net/
[16] http://cutter.sourceforge.net/
[17] http://embunit.sourceforge.net/embunit/
[18] https://github.com/imb/fctx
[19] https://github.com/jmmv/kyua
[20] https://github.com/bigwhite/lcut
[21] https://github.com/koanlogic/libu
[22] https://github.com/siu/minunit
[23] https://github.com/galvedro/mut

NovaProva [24]	Unit tests, Mocking / Stubbing / Fixture framework
Opmock [25]	Unit tests, Mocking / Stubbing / Fixture framework
RCUNIT [26]	Unit tests, Mocking / Stubbing / Fixture framework
SeaTest [27]	Unit tests, Mocking / Stubbing / Fixture framework
Sput [28]	Unit tests
STRIDE [29]	Unit tests, Mocking / Stubbing / Fixture framework
Unity [30]	Unit tests
tinytest [31]	Unit tests
xTests [32]	Unit tests
Cunit for Mr. Ando [33]	Unit tests
LibCut [34]	Unit tests
libtab [35]	Unit tests
GoogleTest [36]	Unit tests, Mocking / Stubbing / Fixture framework
CxxTest [37]	Unit tests
Cppunit [38]	Unit tests, Mocking / Stubbing / Fixture framework
Catch [39]	Unit tests, Mocking / Stubbing / Fixture framework
Fake function framework [40]	Unit tests, Mocking / Stubbing / Fixture framework
C2Unit [41]	Unit tests
Microsoft Unit Testing Framework for C++ [42]	Unit tests
Bandit [43]	Unit tests

[24]https://github.com/novaprova/novaprova
[25]https://sourceforge.net/p/opmock/wiki/Home/
[26]https://github.com/jecklgamis/rcunit
[27]https://github.com/keithn/seatest
[28]http://www.use-strict.de/sput-unit-testing/
[29]http://www.stridewiki.com/index.php?title=Test_Units
[30]http://www.throwtheswitch.org/unity/
[31]https://github.com/nmathewson/tinytest
[32]http://xtests.sourceforge.net/
[33]http://park.ruru.ne.jp/ando/work/CUnitForAndo/html/
[34]https://github.com/kirbyfan64/libcut
[35]https://github.com/zorgnax/libtap
[36]https://github.com/google/googletest
[37]https://github.com/CxxTest/cxxtest
[38]http://cppunit.sourceforge.net/doc/cvs/cppunit_cookbook.html
[39]https://github.com/philsquared/Catch
[40]https://github.com/meekrosoft/fff
[41]https://github.com/cwyang/c2unit
[42]https://msdn.microsoft.com/en-us/library/hh598953.aspx
[43]http://banditcpp.org/

Boost.Net [44]	Unit tests
CppUnitLite [45]	Unit tests
Unit++ [46]	Unit tests

Table 3.2: Testing frameworks selected for programming language C

Framework	**Reason for exclusion**
API Sanity Checker	Automatically generates tests, not to be included in a repository
Autounit (GNU)	Missing documentation
Parasoft C/C++test	Closed source / documentation
QA Systems Cantata	Automatically generates tests, not to be included in a repository
Catsrunner	Missing documentation
CU	Missing documentation
CUnitWin32	Missing documentation
Smarttester	Closed source / documentation
Test Dept.	Unclear documentation
TF unit test	Unclear documentation
TPT	Closed source / documentation
VectorCast/C	Closed source / documentation
Visual Assert	Not a codified testing tool, instead a plugin to aid in writing tests
HWUT	Not a real testing framework. Outputs logs to files for manual verification
Tessy	Closed source / documentation
ACT	Missing documentation
CeeUnit	Missing documentation
CUT - C Unit Tester system	Missing documentation
InTheGuard	Missing documentation
GoogleMock	Does not include test or assert methods
NanoCppUnit	Missing documentation

Table 3.3: Testing frameworks *not* selected for programming language C

3.2.2 Java Frameworks

Framework	**Classification**

[44]http://www.boost.org/
[45]https://github.com/smikes/CppUnitLite
[46]http://unitpp.sourceforge.net/

Default assert	Unit tests
Arquillian [47]	Unit tests, Integration tests
beanSpec [48]	Unit tests
Concordion [49]	Unit tests, Integration tests, Mocking / Stubbing / Fixture framework
Cucumber-JVM [50]	Unit tests, Integration tests, System tests
Cuppa [51]	Unit tests
DbUnit [52]	Unit tests, Integration tests
EasyMock [53]	Unit tests, Integration tests, Mocking / Stubbing / Fixture framework
EvoSuite [54]	Test generator
GrandTestAuto [55]	Unit tests, Integration tests, System tests, Performance tests
GroboUtils [56]	Unit tests
HavaRunner [57]	Unit tests, Integration tests
Jbehave [58]	Unit tests, Integration tests, System tests
JDave [59]	Unit tests, Integration tests
Jexample [60]	Unit tests
Jgiven [61]	Unit tests, Integration tests
Jmock [62]	Unit tests, Integration tests, Mocking / Stubbing / Fixture framework
Jmockit [63]	Unit tests, Integration tests, Mocking / Stubbing / Fixture framework
Jnario [64]	Unit tests, Integration tests, System tests
Jukito [65]	Unit tests, Integration tests, Mocking / Stubbing / Fixture framework
JUnit [66]	Unit tests, Integration tests
Mockito [67]	Unit tests, Integration tests, Mocking / Stubbing / Fixture framework

[47]http://arquillian.org/
[48]https://sourceforge.net/projects/beanspec/
[49]http://concordion.org/
[50]https://cucumber.io/docs/reference/jvm
[51]http://cuppa.forgerock.org/
[52]http://dbunit.sourceforge.net/
[53]http://easymock.org
[54]http://www.evosuite.org/
[55]http://grandtestauto.org/
[56]http://groboutils.sourceforge.net/
[57]https://github.com/havarunner/havarunner
[58]http://jbehave.org/
[59]http://jdave.org/
[60]https://github.com/akuhn/jexample
[61]http://jgiven.org/
[62]http://www.jmock.org/
[63]http://jmockit.org/
[64]http://jnario.org/
[65]https://github.com/ArcBees/Jukito
[66]http://junit.org/junit4/
[67]http://site.mockito.org/

Mockrunner [68]	Unit tests, Integration tests, Mocking / Stubbing / Fixture framework
Needle [69]	Unit tests, Integration tests, Mocking / Stubbing / Fixture framework
NUTester [70]	Unit tests
OpenPojo [71]	Unit tests
PowerMock [72]	Unit tests, Integration tests, Mocking / Stubbing / Fixture framework
SureAssert [73]	Unit tests, Mocking / Stubbing / Fixture framework
TestNG [74]	Unit tests, Integration tests
Unitils [75]	Unit tests, Integration tests, Mocking / Stubbing / Fixture framework
XMLUnit [76]	Unit tests, Integration tests
AssertJ [77]	Matcher framework
concurrentjunit runner [78]	Unit tests, Integration tests
Concurrent-JUnit [79]	Unit tests, Integration tests
JUnitPerf [80]	Unit tests, Integration tests, Performance tests
The Grinder [81]	Unit tests, Performance tests
ContiPerf [82]	Performance tests
HTTPUnit [83]	Unit tests, Integration tests
JWebUnit [84]	Unit tests, Integration tests, System tests
Spock [85]	Unit tests, Integration tests, Mocking / Stubbing / Fixture framework
JUnit Runners [86]	Unit tests, Integration tests
junit-dataprovider [87]	Unit tests, Integration tests
MockFtpServer [88]	Unit tests, Integration tests, Mocking / Stubbing / Fixture framework

[68]http://mockrunner.github.io/
[69]http://needle.spree.de/
[70]http://programming.nu/nutest
[71]https://github.com/oshoukry/openpojo
[72]https://github.com/powermock/powermock
[73]http://www.methodsandtools.com/tools/unittestingsureassert.php
[74]http://testng.org/doc/
[75]http://unitils.sourceforge.net/
[76]http://www.xmlunit.org/
[77]http://joel-costigliola.github.io/assertj/
[78]https://javadocs.com/docs/com.mycila/mycila-junit/1.4.ga/com/mycila/junit/concurrent/ConcurrentJunitRunner.java
[79]https://github.com/ThomasKrieger/concurrent-junit
[80]https://github.com/clarkware/junitperf
[81]http://grinder.sourceforge.net/
[82]https://github.com/lucaspouzac/contiperf
[83]http://httpunit.sourceforge.net/
[84]https://jwebunit.github.io/jwebunit/
[85]http://spockframework.org/
[86]https://github.com/NitorCreations/CoreComponents/tree/master/junit-runners
[87]https://github.com/TNG/junit-dataprovider
[88]http://mockftpserver.sourceforge.net/

HamCrest [89]	Matcher framework
Spring test [90]	Unit tests, Integation tests
Spring db test [91]	Unit tests, Integation tests, Mocking / Stubbing / Fixture framework
Twip [92]	Unit tests, Integation tests
jfcUnit [93]	Unit tests, Integation tests, System tests, UI tests
HtmlUnit [94]	Unit tests, Integration tests, System tests, UI tests
StrutsTestCase for JUnit [95]	Unit tests, Integration tests, Mocking / Stubbing / Fixture framework
Feed4JUnit [96]	Unit tests, Integation tests
Citrus Enterprise SOA Application Based [97]	Integration tests
CassandraUnit [98]	Unit tests, Integration tests
XTest [99]	Unit tests
UISpec4J [100]	Unit tests, Integration tests, System tests, UI tests
Thread Weaver [101]	Unit tests, Integration tests
Serenity [102]	Unit tests, Integration tests, System tests
Galen Framework [103]	Unit tests, Integration tests, System tests, UI tests
Gauge [104]	Unit tests, Integration tests
Abbot Java GUI Test Framework[105]	Unit tests, Integration tests, System tests, UI tests
Jersey [106]	Unit tests, Integration tests
ScalaTest [107]	Unit tests, Integration tests

[89]http://hamcrest.org/
[90]https://docs.spring.io/spring-framework/docs/current/spring-framework-reference/html/testing.html
[91]https://springtestdbunit.github.io/spring-test-dbunit/
[92]http://twip.sourceforge.net/
[93]http://jfcunit.sourceforge.net/
[94]http://htmlunit.sourceforge.net/
[95]http://strutstestcase.sourceforge.net/
[96]http://databene.org/feed4junit.html
[97]http://www.citrusframework.org/
[98]https://github.com/jsevellec/cassandra-unit
[99]http://msbarry.github.io/Xtest/
[100]https://github.com/UISpec4J/UISpec4J
[101]https://github.com/google/thread-weaver
[102]http://www.thucydides.info/
[103]http://galenframework.com/
[104]https://getgauge.io/
[105]http://abbot.sourceforge.net/doc/overview.shtml
[106]https://jersey.java.net/documentation/latest/test-framework.html
[107]http://www.scalatest.org/

Specs2 [108]	Unit tests, Integration tests, System tests
ScalaCheck [109]	Unit tests, Integration tests
Android UI Automator [110]	Unit tests, Integration tests, System tests, UI tests
Robolectric [111]	Unit tests
Robotium [112]	Unit tests, Integration tests, System tests, UI tests
Calabash for Android [113]	Unit tests, Integration tests, System tests, UI tests

Table 3.4: Testing frameworks selected for programming language Java

Framework	**Reason for exclusion**
ConcJunit / Concutest	Missing documentation
EtlUnit	Missing documentation
Instinct	Missing documentation
Java Server-Side Testing framework (JSST)	Server side tests, configuration is private and should not be included in repositories
Jtest	Closed source and documentation
Jwalk	Automatically generates tests, but does not output them into the repository
JUnitEE	Replaced by Cactus
TimeShiftX	Closed source and documentation
H2 Database Engine	Missing documentation
Cactus	Missing documentation
Jetif	Unclear documentation
p-unit	Missing documentation
Ejb3Unit	Inconsistent documentation
EastyTesting	Unclear documentation
Ripplet	Missing documentation
Jubula	UI testing, not a codified testing tool
JCrawler	Server side tests, configuration is private and should not be included in repositories
iValidator	Missing documentation

[108] https://etorreborre.github.io/specs2/
[109] http://www.scalacheck.org/
[110] https://developer.android.com/topic/libraries/testing-support-library/index.html
[111] http://robolectric.org/
[112] https://github.com/robotiumtech/robotium
[113] https://github.com/calabash/calabash-android

Lattu	Missing documentation
MockCentral	Missing documentation
JEasyTest	Unclear documentation
Jacareto	Tool for replaying UI interactions, not a codified testing tool
RedwoodHQ	Generates UI tests, these should not be included in repositories
SimpleJavaUnit TestFramework	Missing documentation

Table 3.5: Testing frameworks *not* selected for programming language Java

3.2.3 Ruby Frameworks

Framework	**Classification**
Test::Unit [114]	Unit tests
RSpec [115]	Unit tests, Integration tests
Shoulda [116]	Unit tests, Integration tests, Matcher framework
microtest [117]	Unit tests
Bacon [118]	Unit tests, Integration tests
minitest [119]	Unit tests, Integration tests, Mocking / Stubbing / Fixture framework
TMF [120]	Unit tests, Mocking / Stubbing / Fixture framework
mocha [121]	Unit tests, Mocking / Stubbing / Fixture framework
rr [122]	Unit tests, Mocking / Stubbing / Fixture framework
flexmock [123]	Unit tests, Mocking / Stubbing / Fixture framework
Cucumber [124]	Unit tests, Integration tests, System tests
Riot [125]	Unit tests, Mocking / Stubbing / Fixture framework
Shindo [126]	Unit tests, Integration tests
Testy [127]	Unit tests, Integration tests
assert [128]	Unit tests

[114] https://github.com/test-unit/test-unit
[115] http://rspec.info/
[116] https://github.com/thoughtbot/shoulda
[117] https://github.com/rubyworks/microtest
[118] https://github.com/chneukirchen/bacon
[119] https://github.com/seattlerb/minitest
[120] https://github.com/bowsersenior/tmf
[121] https://github.com/freerange/mocha
[122] https://github.com/rr/rr
[123] https://github.com/jimweirich/flexmock
[124] https://cucumber.io/
[125] https://github.com/thumblemonks/riot
[126] https://github.com/geemus/shindo
[127] https://github.com/ahoward/testy
[128] https://rubygems.org/gems/assert/versions/2.15.0

kintama [129]	Unit tests, Integration tests
test_inline [130]	Unit tests
Lemon [131]	Unit tests
Detest [132]	Unit tests, Integration tests
Exemplor [133]	Integration tests
Contest [134]	Unit tests
Turnip [135]	Unit tests, Integration tests, System tests
steak [136]	Unit tests, Integration tests, System tests
spinach [137]	Unit tests, Integration tests, System tests
coulda [138]	Unit tests, Integration tests, System tests
stella [139]	Unit tests, Integration tests
Unencumbered [140]	Unit tests, Integration tests, System tests
bewildr [141]	Unit tests, Integration tests, System tests, UI tests
Stories [142]	Unit tests, Integration tests, System tests
Filet [143]	Unit tests, Integration tests, System tests
saki [144]	Unit tests, Integration tests, System tests
mountain_berry_fields [145]	Unit tests, Integration tests
Attest [146]	Unit tests, Integration tests
yard-doctest [147]	Unit tests, Integration tests
ActiveMocker [148]	Mocking / Stubbing / Fixture framework
TestXml [149]	Matcher framework
WebMock [150]	Mocking / Stubbing / Fixture framework

[129]https://github.com/lazyatom/kintama
[130]https://github.com/eric1234/test_inline
[131]https://github.com/rubyworks/lemon
[132]https://github.com/sunaku/detest
[133]https://github.com/quackingduck-archive/exemplor
[134]https://github.com/citrusbyte/contest
[135]https://github.com/jnicklas/turnip
[136]https://github.com/cavalle/steak
[137]https://github.com/codegram/spinach
[138]https://github.com/elight/coulda
[139]https://github.com/solutious/stella
[140]https://github.com/atilaneves/unencumbered
[141]https://github.com/natritmeyer/bewildr
[142]https://github.com/citrusbyte/stories
[143]https://github.com/xing/filet
[144]https://github.com/ludicast/saki
[145]https://github.com/JoshCheek/mountain_berry_fields
[146]https://github.com/skorks/attest
[147]https://github.com/p0deje/yard-doctest
[148]https://github.com/zeisler/active_mocker
[149]https://github.com/alovak/test_xml
[150]https://github.com/bblimke/webmock

vcr[151]	Unit test, Integration tests, Replay framework
Capybara[152]	Unit tests, Integration tests, System tests, UI tests
factory_girl [153]	Mocking / Stubbing / Fixture framework
protest [154]	Unit tests, Integration tests
Kata [155]	Unit tests, Integration tests
RamCrest [156]	Matcher framework

Table 3.6: Testing frameworks selected for programming language Ruby

Framework	**Reason for exclusion**
testrocket	Not detectable due to its syntax
Micronaut	Missing documentation
dtf	Missing documentation
rubydoctest	Missing documentation
Relish	Closed source and documentation
Narf	Not a test framework, instead it is an assertion method that can be copied into existing projects

Table 3.7: Testing frameworks *not* selected for programming language Ruby

3.2.4 JavaScript Frameworks

Framework	**Classification**
Ava [157]	Unit tests, Integration tests
DOH [158]	Unit tests
Qunit [159]	Unit tests
UnitJS [160]	Unit tests
Mocha [161]	Unit tests, Integration tests, Mocking / Stubbing / Fixture framework
Intern [162]	Unit tests, Integration tests, System tests
YUI Test [163]	Unit tests

[151]https://github.com/vcr/vcr
[152]https://github.com/teamcapybara/capybara
[153]https://github.com/thoughtbot/factory_girl
[154]https://github.com/janko-m/protest
[155]https://www.codewars.com/docs/kata-test-framework
[156]https://github.com/hamcrest/ramcrest
[157]https://github.com/avajs/ava
[158]http://www.dojotoolkit.org/reference-guide/util/doh.html
[159]https://qunitjs.com/
[160]http://unitjs.com/
[161]https://mochajs.org/
[162]https://theintern.github.io/
[163]https://github.com/yui/yuitest/

Jasmine [164]	Unit tests, Integration tests
Screw-Unit [165]	Unit tests, Integration tests
Tape [166]	Unit tests
Test_it [167]	Unit tests
JS Test Driver [168]	Unit tests, Integration tests
Sinon.JS [169]	Mocking / Stubbing / Fixture framework
NodeUnit [170]	Unit tests, Mocking / Stubbing / Fixture framework
Tyrtle [171]	Unit tests, Mocking / Stubbing / Fixture framework
Wru [172]	Unit tests
Buster.JS [173]	Unit tests, Integration tests, Mocking / Stubbing / Fixture framework
Lighttest [174]	Unit tests
Chai [175]	Unit tests, Mocking / Stubbing / Fixture framework
Karma [176]	Test runner
google-js-test [177]	Unit tests, Mocking / Stubbing / Fixture framework
Pavlov [178]	Unit tests, Integration tests
Simpletest [179]	Unit tests
Enzyme [180]	Matcher framework
Jest [181]	Unit tests, Integration tests, Mocking / Stubbing / Fixture framework
Cucumber [182]	Unit tests, Integration tests, System tests
Selenium [183]	Test runner
Nightwatch [184]	Integration tests, System tests, UI tests
Cypress [185]	Unit tests, Integration tests, System tests, UI tests

[164]https://github.com/jasmine/jasmine
[165]https://github.com/nkallen/screw-unit
[166]https://github.com/substack/tape
[167]https://github.com/yihui/testit
[168]https://github.com/wesabe/JsTestDriver
[169]http://sinonjs.org/
[170]https://github.com/caolan/nodeunit
[171]https://github.com/spadgos/tyrtle
[172]https://github.com/WebReflection/wru
[173]http://docs.busterjs.org/en/latest/
[174]https://github.com/asvd/lighttest
[175]http://chaijs.com/
[176]https://github.com/karma-runner/karma
[177]https://github.com/google/gjstest
[178]https://github.com/mmonteleone/pavlov
[179]https://github.com/orenaksakal/js-simpletest
[180]http://airbnb.io/enzyme/
[181]https://facebook.github.io/jest/
[182]https://github.com/cucumber/cucumber-js
[183]https://www.npmjs.com/package/selenium-standalone
[184]http://nightwatchjs.org/
[185]https://www.cypress.io/

Lab [186]	Unit tests
Casperjs [187]	Unit tests, Integration tests, System tests, UI tests
Phantom [188]	UI tests
Nightmare [189]	UI tests
chimp [190]	Unit tests, Integration tests, System tests, UI tests
jsverify [191]	Unit tests, Matcher framework
testdouble [192]	Mocking / Stubbing / Fixture framework
gremlins [193]	Monkey tests
nemojs [194]	Test runner
dalekjs [195]	Unit tests, Integration tests, System tests, UI tests
spectacular [196]	Unit tests, Integration tests
testem [197]	Unit tests, Integration tests
painless [198]	Unit tests
supertest [199]	Unit tests, Integration tests
teaspoon [200]	Unit tests, Integration tests
chai as promised [201]	Unit tests, Mocking / Stubbing / Fixture framework
sinon as promised [202]	Unit tests, Mocking / Stubbing / Fixture framework
jfunit [203]	Unit tests
Node.js assert [204]	Unit tests
unit.js [205]	Unit tests
should.js [206]	Unit tests

[186] https://github.com/hapijs/lab
[187] http://casperjs.org/
[188] http://phantomjs.org/
[189] http://www.nightmarejs.org/
[190] https://chimp.readme.io/
[191] https://github.com/jsverify/jsverify
[192] https://github.com/testdouble/testdouble.js/
[193] https://github.com/marmelab/gremlins.js/
[194] http://nemo.js.org/
[195] http://dalekjs.com/
[196] http://abe33.github.io/spectacular/
[197] https://github.com/testem/testem
[198] https://github.com/taylorhakes/painless
[199] https://github.com/visionmedia/supertest
[200] https://github.com/jejacks0n/teaspoon
[201] https://github.com/domenic/chai-as-promised
[202] https://github.com/bendrucker/sinon-as-promised
[203] http://www.jsclasses.org/package/64-JavaScript-Test-the-quality-of-JavaScript-code.html
[204] https://nodejs.org/api/assert.html
[205] http://unitjs.com/
[206] https://shouldjs.github.io/

testr.js [207]	Unit tests, Mocking / Stubbing / Fixture framework
bunit.js [208]	Unit tests
squire.js [209]	Mocking / Stubbing / Fixture framework
grunt-castle [210]	Unit tests, Mocking / Stubbing / Fixture framework
FuncUnit [211]	Unit tests, Integration tests, System tests, UI tests
Preamble [212]	Unit tests, Integration tests, System tests
rx test runner [213]	Unit tests, Integration tests
taperun [214]	Unit tests, Integration tests
grunt-contrib-jasmine [215]	Test runner
phantomjs-yuitest [216]	Unit tests, Integration tests, System tests, UI tests
lotte [217]	Unit tests, Integration tests, System tests, UI tests
react test utils [218]	Unit tests
jsdom [219]	Matcher framework
mockery [220]	Mocking / Stubbing / Fixture framework
smokestack [221]	Test runner
chai-enzyme [222]	Matcher framework
Expect [223]	Matcher framework

Table 3.8: Testing frameworks selected for programming language JavaScript

Framework	**Reason for exclusion**
J3Unit	Missing documentation
JSNUnit	Missing documentation
JSNSPec	Missing documentation
UnitTesting	Missing documentation

[207]https://github.com/mattfysh/testr.js
[208]https://github.com/bebraw/bunit.js
[209]https://github.com/iammerrick/Squire.js
[210]https://github.com/walmartlabs/grunt-castle
[211]http://funcunit.com/
[212]https://jeffschwartz.github.io/preamble/
[213]https://github.com/gizur/rxtestrunner
[214]https://github.com/juliangruber/tape-run
[215]https://github.com/gruntjs/grunt-contrib-jasmine
[216]https://github.com/metafeather/phantomjs-yuitest
[217]https://github.com/StanAngeloff/lotte
[218]https://facebook.github.io/react/docs/test-utils.html#shallow-rendering
[219]https://github.com/tmpvar/jsdom
[220]https://github.com/mfncooper/mockery
[221]https://github.com/hughsk/smokestack
[222]https://github.com/producthunt/chai-enzyme
[223]https://github.com/mjackson/expect

JS Unity	Missing documentation
JSTest.NET	Not a JavaScript testing framework, instead meant for .NET
RhinoUnit	Unclear documentation
JSUS	Not a testing framework. Util framework
Wallaby	Paid testing tool, closed source and documentation
Protractor	Does not include asserts or tests, undetectable due to its setup
TestSwarm	Decommissioned
yolpo	Decommissioned
Chutzpah	A tool for running existing tests, not a codified testing framework
Jstestdriver	Not a codified testing framework. A plugin for js-test-driver
Env.js	Missing documentation
qooxdoo	Not a codified testing tool
testrunner	Decommissioned. Nowadays included in other frameworks
simulator	Missing documentation
htmlunit	Not a JavaScript testing framework, instead meant for Java
celerity	Missing documentation
jruby	Not a codified testing framework
watir	Not a JavaScript testing framework
elm	Not a testing framework
must.js	Part of unit.js
capybara	Not a JavaScript testing framework
Hiro	Missing documentation
Laika	Decommissioned. Not detectable since it is injected into Meteor tests
phantom-assert	Missing documentation
robot framework	Not a JavaScript testing framework
venus.js	A tool for running existing tests, not a codified testing framework
webdriver	Part of selenium
poltergeist	Not a JavaScript testing framework. Web driver for Ruby Capybara
terminus	Not a JavaScript testing framework. Web driver for Ruby Capybara
mocha-phantomjs	A tool for running existing tests, not a codified testing framework
guard-jasmine	Not a JavaScript testing framework. Meant for Ruby
phantom-jasmine	A tool for running existing tests, not a codified testing framework
js-test-driver-phantomjs	Missing documentation
qunit-phantom-js-runner	A tool for running existing tests, not a codified testing framework
js test runner	Missing documentation
qlive	A tool for running existing tests, not a codified testing framework
qunited	Not a JavaScript testing framework
phantomrobot	Missing documentation
ghostdriver	Not a JavaScript testing framework

grover	A tool for running existing tests, not a codified testing framework
Turing test	Missing documentation

Table 3.9: Testing frameworks *not* selected for programming language JavaScript

During the extensive classification process, we observed that many of the described frameworks extend or piggy-back on the success of well-known frameworks. Many frameworks copy part of the framework names into their own framework names or even publicly state they are clones of other frameworks. E.g., Bacon [224] publicly claims it is an RSpec clone. The following well-known frameworks are often extended or piggy-backed on:

- Java: JUnit
- Ruby: RSpec, MiniTest, Test::Unit
- JavaScript, Jasmine, Phantom, Chai, Selenium, Karma

In the case of C, available testing documentation appeared to be *very* limited and confusing, making it hard to adopt a testing strategy using these frameworks.

For programming languages JavaScript, Ruby and C (although to a lesser extent), many frameworks consist of a very limited and simple scope. These frameworks typically claim to have a more concise syntax and faster execution speed due to their simplicity (when compared to larger, well-known frameworks), e.g. Riot [225]. Many of these smaller frameworks appear to be clones or of a very similar nature and the reason for their existence is not immediately clear. Many of the framework authors compare their frameworks against larger frameworks and do not seem to be aware of the existence of similar, smaller testing frameworks.

3.3 Observed differences in available tooling on a programming language basis

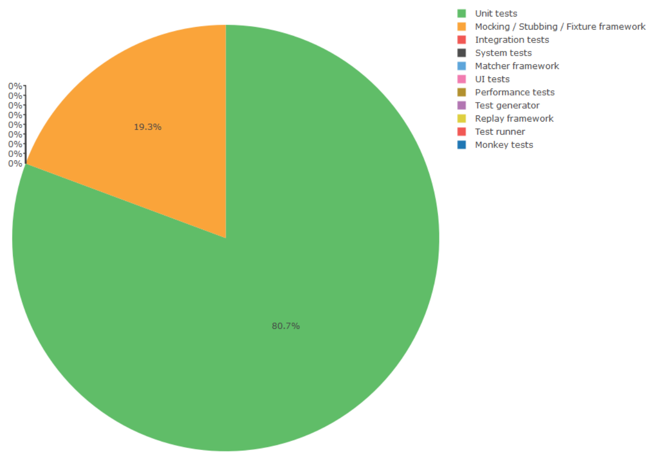
Now that all frameworks have been classified, we can compare the observed differences. The distribution of the classification of the tooling mentioned in the previous section is visualized in figures 3.1 & 3.2.

Most available tooling exists to aid in unit testing, followed by integration testing, mocking purposes and system tests.

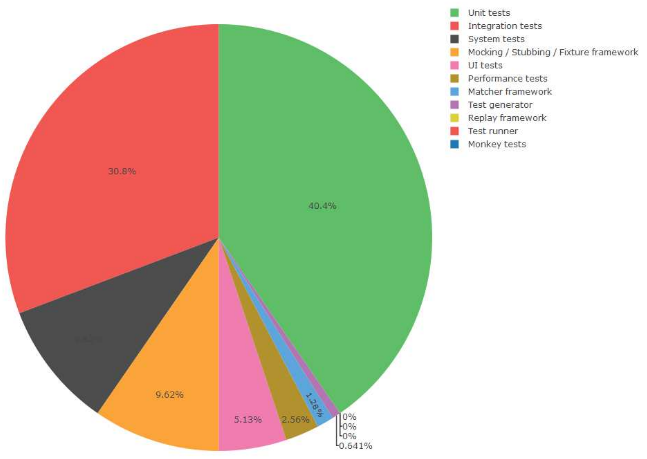
It immediately becomes clear that in the case of programming language C, there is a large gap as integration and system testing frameworks appear to be non-existent.

Only in the case of Java, tooling was found that was combining performance tests with assertions. Because of its focus on speed, one would expect these tools to be available for programming language C. This was however not the case. Note that several profiling tools

[224] https://github.com/chneukirchen/bacon
[225] https://github.com/thumblemonks/riot

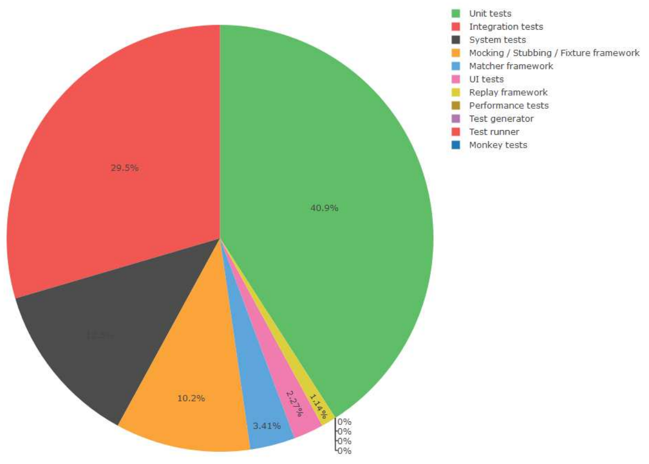


(a) Distribution of available categorized test tooling for programming language C

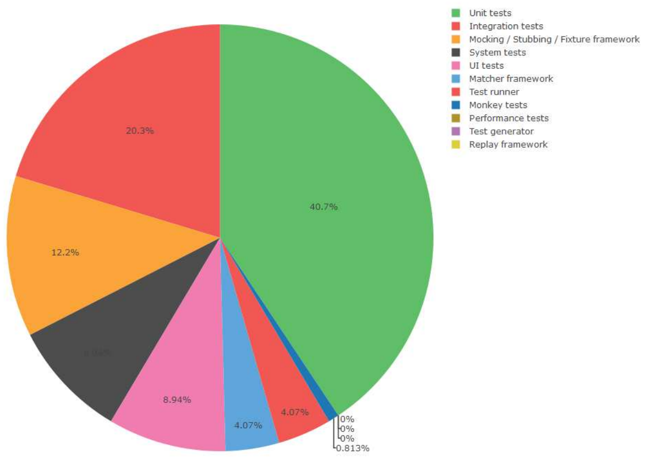


(b) Distribution of available categorized test tooling for programming language Java

Figure 3.1: Distribution of available categorized test tooling for statically typed programming languages



(a) Distribution of available categorized test tooling for programming language Ruby



(b) Distribution of available categorized test tooling for programming language JavaScript

Figure 3.2: Distribution of available categorized test tooling for dynamically typed programming languages

exist, yet these lack the assertion capabilities and do therefore not fall within the scope of this thesis.

In the case of JavaScript, an increased number of UI test tooling was observed. This is likely related to the fact that a lot of web-applications are developed using JavaScript.

No clear differences were observed between statically or dynamically typed programming languages with regard to available tooling.

An increased number of matcher frameworks was found for dynamically typed programming languages, however the difference was not significant.

In the case of Java, most of frameworks seem to support different types of testing by default (i.e. supporting both unit tests and integration tests). This allows developers to test their project using only a few number of frameworks. This is less so the case for C, Ruby and JavaScript.

3.4 Conclusion

While the scope of this chapter is limited to only 4 programming languages, to the best of our knowledge our work is the first to investigate and provide an overview of available tooling. Moreover, by classifying all available tooling we were able to point out that most available tooling aids in the process of unit testing, followed by integration testing, mocking purposes and system tests. JavaScript test tooling is somewhat more focused on UI testing, which is likely related to the large number of web-applications developed using JavaScript. In the case of Java, available test tooling seemed to cover a larger part of the codified testing spectrum by default (e.g. both unit testing and integration testing). This allows developers to test their software project from different angles using only a few frameworks. Finally, our classification approach showed that there is a clear lack of tooling available for C.

Chapter 4

Analysis setup

Now that have a good overview of the tooling involved in codified testing strategies, we can use it to analyze testing behavior in practice. In order to do so, we need to design a mining approach that can detect and analyze the testing tools used in software projects. This chapter will first describe the metrics that will be used to analyze and compare testing behavior of software projects. After that, the scope of the mining approach will be described, as it is practically unfeasible to analyze all software projects. The analysis process and design will be described and finally this chapter will reflect on the drawbacks and validity of the results obtained by the mining approach.

This chapter contributes to the basis of RQ2 and supplies the following deliverables for this research question: a list of metrics that can be used to compare test practices based on literature, a selection criterion for software projects and a mining tool that can detect test tooling.

The described analysis process is vital for the thesis, as the data this approach will gather will be used as input to compare software testing approaches between programming languages, test tooling and type systems, based on the metrics defined in the first section of this chapter.

4.1 Test metrics

In order to analyze and compare testing practices in software projects, we need to define a set of ‘test metrics’. This set of metrics will be used in the next chapter to compare the obtained test data by our mining approach.

4.1.1 Rate of testing

This metric consists of the percentage of software projects that adopt test tooling to test their software project. **Note**: is impossible to test a software project without the adoption of any frameworks, unless the authors wrote their own test framework. This metric provides basic insight into the adoption rate of testing practices and the rate at which developers test their software projects.

4.1.2 Assert density

[17] defines the assert(ion) density (the number of asserts per 1000 lines of code) as the following: $\frac{\#ofasserts}{\frac{LOC}{1000}}$. It is a widely adopted metric to measure the rate at which a project is testing. This metric takes the project size (lines of code) into account and is therefore more fair than measuring the absolute number of asserts of tests (e.g. the metric used in [4]), as software projects that are larger in size (lines of code) are likely to have more asserts of tests when applying testing approaches. This however, does not necessarily mean that these larger software projects test more thoroughly, hence the inclusion of density in this metric.

This metric provides insight into the extent to which developers test their applications.

4.1.3 Test density

While there exists no official metric for test density, the assert density metric can be slightly modified to obtain a similar metric (the number of tests per 1000 lines of code) for tests: $\frac{\#oftests}{\frac{LOC}{1000}}$.

This metric provides insight into the extent to which developers test their applications.

4.1.4 Adoption rate of frameworks

This metric consists of three separate metrics, namely the following:

- The number of frameworks used to test software projects
- The adoption rate of frameworks used to test software projects
- The composition of frameworks used to test software projects

This metric provides insight into the tools that are used to test by developers to cover their testing needs. We gain insight into the rate at which frameworks are used, frequent and popular compositions and the number of frameworks that is used to test software projects.

This metric can help us identify differences in the availability of testing tools and identify missing functionality that may be present in testing tools available for other programming languages.

4.1.5 Usage of available tooling

This metric uses the adoption rate of test tooling and their classification and combines it into an overview of the rate of which certain testing approaches are used on a programming language level. For example, integration testing may be prominent in programming language X, whereas mocking practices are more popular in programming language Y.

This metric provides insight into the approaches developers use to test their applications.

4.2 Scope definition

In order to obtain meaningful data on the way developers test their applications, we need to obtain a set of projects that can represent state of the art software projects. These projects should preferably adhere to the following criteria:

- The project should have been active recently, so that its testing approaches reflect present day testing activities.
  Since most projects on GitHub are inactive [15], most of the projects will not have been updated and developers will not have had the chance to include new (present day) testing technologies into their projects. By only looking at recently active projects, we can more accurately try to infer why certain testing frameworks were chosen over others as we know which testing frameworks developers could have used or considered recently.

- The project should have at least had some basic activity.
  Most projects have very low activity and are of a personal nature [15]. Many of these projects include ‘toy’ projects that are quickly assembled, which have a very different development cycle and project lifespan than the average software project. Because of the different lifespan, these projects are typically tested on a much smaller basis or not even tested at all. For this reason these projects should be excluded.

- The project should at least have received some form of traction by a community.
  Software projects with multiple stakeholders are usually set up in a more professional manner with respect to both their architecture as well as their testing. Not only does filtering by this criterion help us weed out ‘toy’ projects more accurately, this will also guarantee that the project is of a more serious nature.

- The project should not be forked from another project. Forked projects are usually meant for small personal changes or used for submitting Pull Requests. Since these changes are usually rather minor, the forked projects will not differ much from the original project and will only bias the results by multiplying the project results.

Pull Requests will not be taken into account for the filter criteria, as many active projects do not use GitHub exclusively [15]. For instance, code reviews and merges may take place outside of GitHub via one of the widely used Atlassian software development tools [1]. Moreover, if a project originated from another source control system (i.e. GitLab [2]), none of the Pull Request of merge related events would show up on GitHub, falsely indicating that the project did not make use of a strict process for codereviews or -changes.

Obtaining the thresholds for the criteria

Several related studies have obtained their filter criteria via intuitive reasoning:

[1]https://www.atlassian.com/
[2]https://gitlab.com

- [8] chose to investigate the top 100 C or c++ projects on GitHub
- [5] chose to investigate projects on GitHub with > 10 watchers and > 50 CI builds
- [4] chose to investigate projects that were not forked and had > 50 stars (GitHub changed this to watchers after this study)

Unfortunately, these values are too strict for this study. By using more strict criteria for the project selection, more professional software projects are targeted by these studies. Because of their more professional nature, these projects are likely to be tested in a different way than the average software project. Since the purpose of this study is to derive an overview of present day testing strategies and the tools involved, the criteria should be just sufficient to filter out any non-representative projects.

The criterion for recent activity in GitHub projects should be set in such a way that it will allow for the retrieval of a good amount of projects, while simultaneously making sure that projects should be able to include present day testing technologies in their development cycle. Projects will be filtered by looking at their latest commit date and it should be **within a time frame of maximum 2 years from today**.

In order to derive the other thresholds based on project data, we can utilize the GHTorrent dataset [10]. It contains the required commit & watcher data per project. The obtained data and its distributions are described in the paragraphs below were obtained from the GHTorrent dataset published on *September 5th, 2016* [3].

By taking a look at the distribution of the number of watchers of software projects on GitHub, it becomes clear that distribution is exponential. Figure 4.1 shows the distributions for both the languages that are in scope of this study, as well as the distribution for all programming languages. We see that the distribution for the languages in the scope of this study does not really differ from the distribution for all languages. As can be derived from figure 4.1, only a very slim amount of projects have watchers.

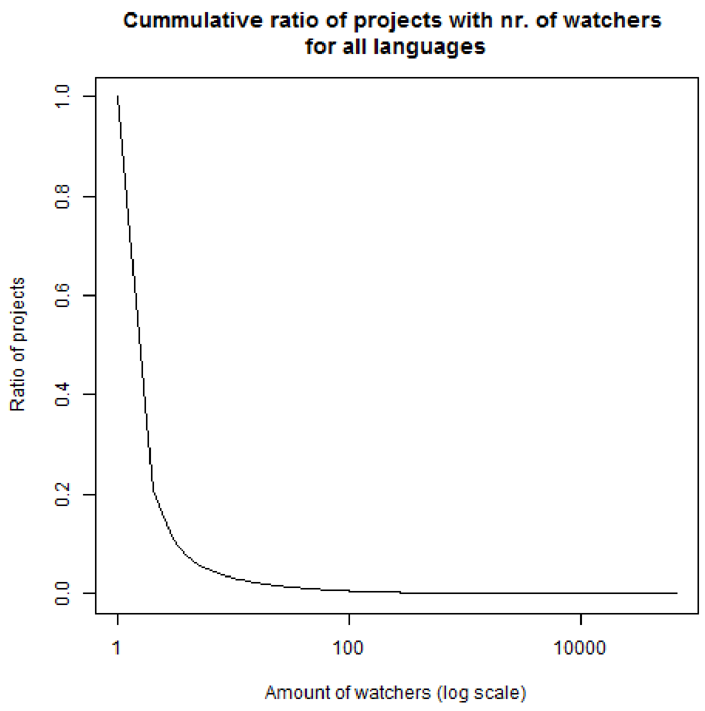
Figure 4.2 shows the distributions of the number of commits for both the languages that are in scope of this study as well as the distribution for all programming languages. Once again we observe that the distribution for our language scope does not really differ from the distribution for all languages. As can be derived from figure 4.2, most projects have very few commits as was also observed in [15]. Once again, the distribution is clearly exponential.

Since the goal is to filter out the outliers for both criteria (all projects that have a low amount of commits or low amount of watchers), the thresholds should be calculated based on the data distribution. [18] suggests to stay away from popular methods that detect outliers using standard deviation around the mean and instead suggests to use the absolute deviation around the median. [18] states that regular data should fall within the interval defined in equation 4.1.

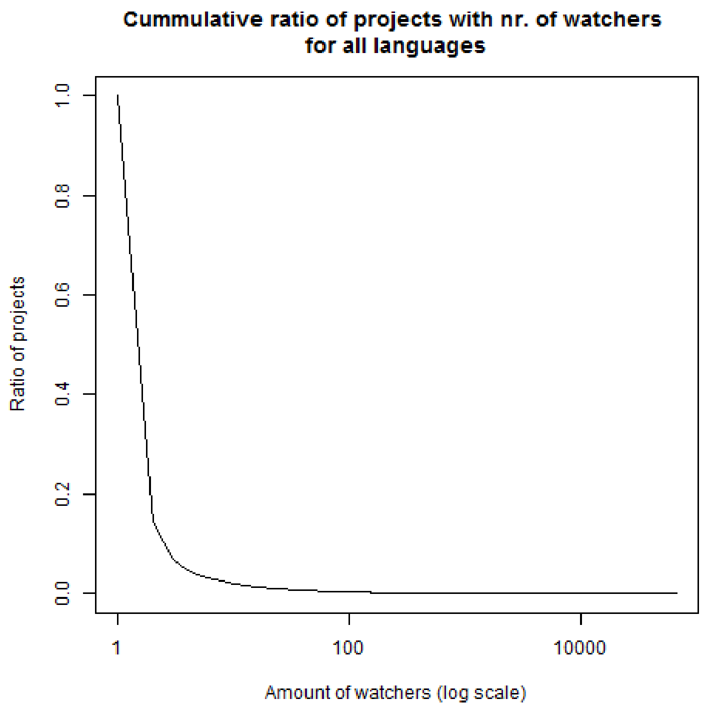
$$M - Y \cdot MAD < x_i < M - Y \cdot MAD \tag{4.1}$$

In this equation, Y corresponds to 3 (very conservative filtering), 2.5 (moderately conservative filtering) or even 2 (poorly conservative filtering), based on the findings of [23]. M

[3] https://ghtstorage.blob.core.windows.net/downloads/mysql-2016-09-05.tar.gz

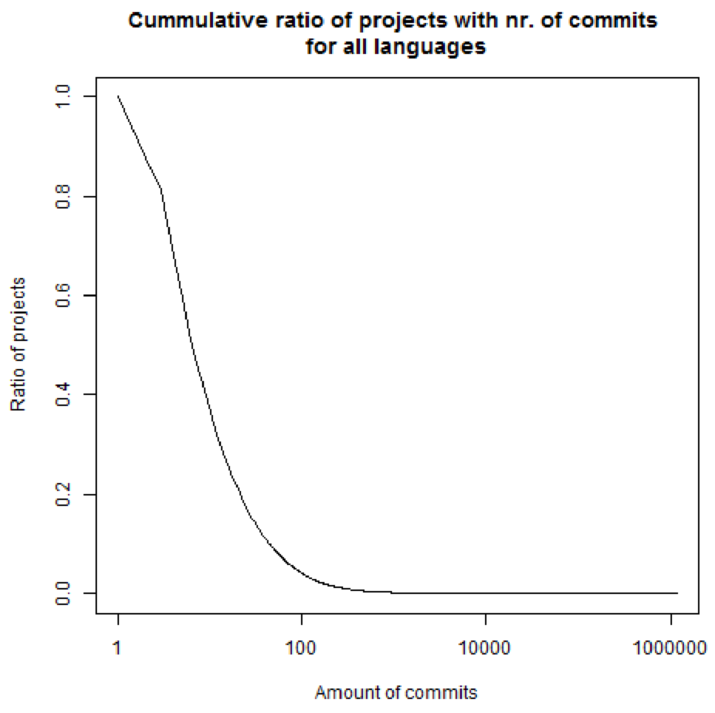


(a) Distribution of the number of watchers for the selected programming languages

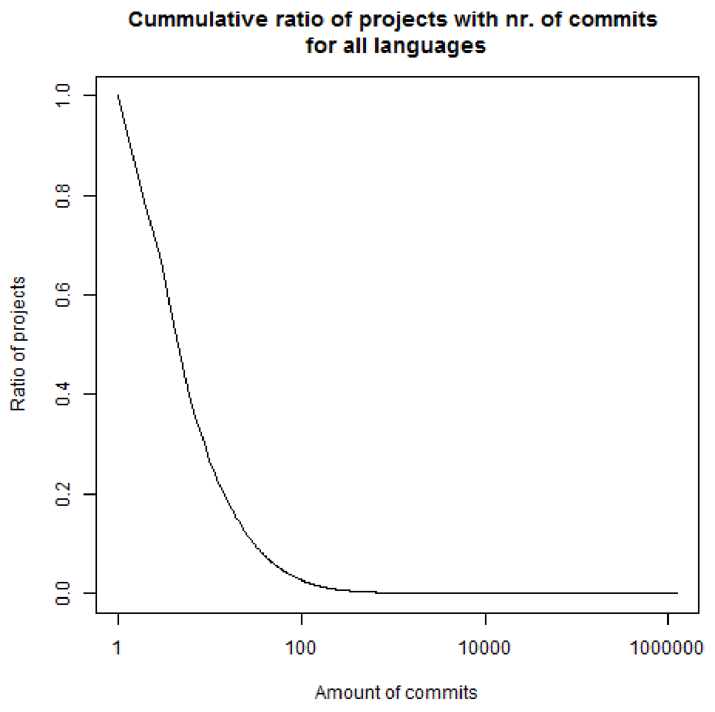


(b) Distribution of the number of watchers for all programming languages

Figure 4.1: Distribution of the number of project watchers



(a) Distribution of the number of commits for the selected programminglanguages



(b) Distribution of the number of commits for all programming languages

Figure 4.2: Distribution of the number of project commits

Table 4.1: Impact of criteria thresholds

Criterion	**Threshold**	**Projects matching**
Number of commits	23	17.98%
Number of watchers	7	3.89%
Last project activity	2 years ago	79.43%
Combined	N/A	1.82%

Table 4.2: Number of projects selected per language

Language	**Nr. of projects selected**
JavaScript	42648
Java	17589
Ruby	11801
C	9043

denotes the median of the dataset and *MAD* denotes the median absolute deviation of the dataset. Since toy projects should not be part of our project scope, there is a need for strict filtering. Therefore, value $Y = 3$ will be used.

To calculate the threshold for the number of commits and the number of watchers, only projects that meet the following criterion will be selected: $x_i > M + 3 \cdot MAD$. Using this formula, the following threshold values were obtained:

- Minimum number of commits : 23 (22.79)
- Minimum number of watchers : 7 (6.45)

Impact of criteria thresholds

The obtained criteria have been tested for their recall against the project dataset of GHTorrent. The results have been summarized and are shown in table 4.1.

A total of 81.081 software projects were selected based on these criteria. The number of projects selected per language are shown below in table 4.2.

4.3 Analysis process

With the selection of software projects laid out, we can design our analysis process. The next section will describe the analysis process of our designed mining approach. The data sources used and steps taken in each part of the process are mentioned separately per step.

4.3.1 Gathering the required project data

Using GHTorrent, we can obtain the list of projects that meet the criteria defined in table 4.1. Since the commit and watcher data are listed in separate tables, two separate queries are run.

The query for obtaining the project commit data was defined as per the following:

```
SELECT P.id as Projectid, P.url as ProjectUrl, P.language as Projectlanguage,
MAX(C.created_at) as LastActive, P.created_at as ProjectCreateDate,
COUNT(distinct C.id) as commits
FROM projects P
LEFT JOIN commits C ON C.project_id = P.id
WHERE P.deleted=0 AND P.forked_from is NULL
GROUP BY P.id
```

The query for obtaining the watcher data was defined as per the following:

```
SELECT P.id as Projectid, P.url as ProjectUrl, P.language as Projectlanguage,
COUNT(distinct W.user_id) as ProjectWatchers
FROM projects P
LEFT JOIN watchers W ON W.repo_id = P.id
WHERE P.deleted=0 AND P.forked_from is NULL
GROUP BY P.id
```

The results of these queries filter out any deleted and forked projects, but do not yet filter them on the criteria mentioned in 4.1. The results are stored into two seperate CSV files, which are joint together in an R script that filters out any projects that do not meet the above mentioned criteria and joins all available project data in a finalized CSV file. The data in this file consists of the following:

- Project id
- Project url
- Date the project had its last commit
- Number of watchers
- Project creation date

4.3.2 Automated project analysis

After having gathered the required project data, the CSV file containing the project information can be loaded into a Java application, tailored for this analysis. The purpose of this automated analysis is to gain insights into the testing behavior of software projects, its involved tooling and the factors it may be impacted by. The process of analyzing each of the projects consists of the following steps, which are described in more depth later in this chapter.

- Start up
- Enriching project data
- Cloning the project repository

- Removal of comments
- Calculating project size
- Framework, test and assert detection
- Result aggregation
- Cleanup

4.3.3 Analysis start up

Upon loading the list of projects available for analysis (obtained in the previous step), projects are selected on a language basis. All projects that were created using the same programming language are loaded into memory, after which they will get analyzed in parallel. By default the numbers of projects that are analyzed concurrently corresponds to the number of CPU cores on the machine that will be running the analysis. This greatly increases analysis speed since the cloning and analysis process are mostly dependent on disk and network delays.

4.3.4 Enriching the project data

After starting the analysis of a certain project, additional data is fetched to enrich the available project data. This will allow for an analysis on auxiliary features involved in testing practices, which will be performed in the next chapter. Projects are enriched with the following data:

- Project url
- Project main author
- Project name
- Project languages
- Date the project has had its last commit
- Number of watchers
- Project creation date
- Number of authors (enriched via Git)
- Number of commits (enriched via Git)
- Number of pull requests (enriched via a query on the GHTorrent database)
- Number of issues (enriched via a query on the GHTorrent database)
- Project age in days

If any of the enrichment tasks fail, the project is abandoned and will not be analyzed. Common problems such as a dropped SSH connection (to the GHTorrent server), database connection or network connection are automatically resolved and enrichment will continue as soon as the connection is available again.

4.3.5 Cloning the project repository

If the enrichment of the project data was successful, the widely adopted JGIT library [4] is used to clone the project repository locally. As the GHTorrent database - used for obtaining the project data - originates from the 5th of September 2016, the latest commit of the project may have been after this date. In order to ensure that all data is consistent, the latest commit before the 5th of September 2016 is cloned locally.

Note that for some repositories, the cloning process may not be completed due to repositories that have been made private since the 5th of September 2016. Moreover, projects may have removed all branches or code, deleted the project entirely or consist of a name that cannot be parsed on the machine whilst cloning. As quite some repositories include malicious code, repositories are cloned into a sand-boxed directory to prevent system crashes and infection.

4.3.6 Removal of comments

Some tests may have been commented out for the purpose of disabling them temporary due to possibly flaky tests that make the project build fail. These tests are not supposed to be running and should therefore not be detected. All comments are stripped from the project by applying replacement regexes using *sed* and *perl* for both single- and multiline comments, tailored to the file extensions (e.g. the Java comment syntax looks different from HTML comments). Each file type contains an entry in the setup configuration, that consists of one or multiple regexes targeting comments.

4.3.7 Calculating project size

After removing all comments from the source code, the CLOC [5] package is used to calculate the project size (source lines of code (SLOC) for the repository. The *–ignore-whitespace* flag is used to get rid of any white space that should not be considered source code. The project size is measured in overall SLOC (language independent), as well as SLOC for the prorgamming language in question (e.g. just Java code). This distinction is particularly useful when comparing just source code (e.g. java files) and not taking into account any (possibly large) configuration files that may bias the results.

4.3.8 Framework, test and assert detection

Frameworks, tests and asserts can be detected based on unique identifiers, i.e. like the framework used in figure 4.3.

[4]https://eclipse.org/jgit/

[5]https://github.com/AlDanial/cloc

```java
@Test
public void testUsingTempFolder() throws IOException {
  File createdFolder = folder.newFolder("newfolder");
  File createdFile = folder.newFile("myfilefile.txt");
  assertTrue(createdFile.exists());
}
```

Figure 4.3: Example identifiers for Java testing framework JUnit. In this case the @Test annotation and the assertion method assertTrue should be detected

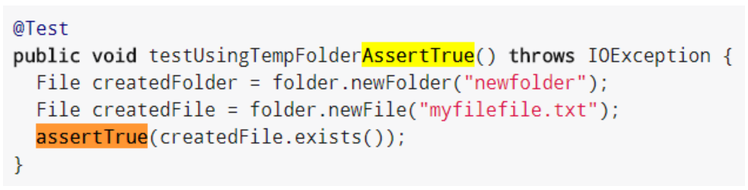
Depending on the nature of the framework and programming language involved, frameworks may be imported in the test class itself or elsewhere in the project. For example, the *@Test* annotation in JUnit requires the method to be loaded directly into the Java class, while Cucumber-JVM .feature files are not ran directly, but are instead run with a JUnit runner in another file. After detecting the unique identifier for the framework inclusion in a file, the file is scanned for any tests and asserts (once more using unique identifiers). Framework detection is configured on a programming language basis, consisting of a configuration file including all unique identifiers for a framework. Each framework described in chapter 3 consists of the following configuration entries:

- Framework name
- Framework test identifiers
- Framework assert identifiers
- Framework import identifiers
- Framework file extensions

First, a list of files within the cloned repository is compiled. This list of files is then checked against the entries of possible file extensions consisting frameworks. I.e. when looking for Java testing frameworks, it does not make sense to search for them in Ruby (.rb) files. Any non-matching files are removed from the list of files that should be analyzed. The list of frameworks is narrowed down by verifying that at least one or more files contains the import identifiers for any of the frameworks. The resulting list of frameworks and files are then loaded into memory and checked for test and assert identifiers on a file level basis. All test and assert methods configured for a framework (e.g. *@Test* and *assertEquals* for JUnit) are compiled into a large regex. This regex merges any test or assert method with patterns that should tail such a method to prevent false positives, i.e. as shown in figure 4.4.

4.3.9 Result aggregation

After obtaining the results of test and assert detection per framework per file, all results are joint together on a project level. Any duplicate matches due to overlapping framework functionality and identifiers are eluded. For example, many projects import both TestNG



```
@Test
public void testUsingTempFolderAssertTrue() throws IOException {
  File createdFolder = folder.newFolder("newfolder");
  File createdFile = folder.newFile("myfilefile.txt");
  assertTrue(createdFile.exists());
}
```

Figure 4.4: An example of a false positive that should not be detected. The orange assertion method should be detected, but the yellow part of the method name should not be detected.

and JUnit into the same test class with the *@Test* identifier. This process ensures that the test annotation is counted only once. The results are then aggregated and exported into a JSON file with the following structure:

- The enhanced project data
- A list of file results containing:
  - The file name
  - A list of framework results that contains the following:
    - The detected framework name
    - All asserts written using this framework
    - All tests written using this framework
- All frameworks that were detected within the project
- All asserts that were detected within the project
- All tests that were detected within the project

4.3.10 Cleanup

After the analysis of the project, the repository is deleted and all files are deleted from memory. The duration of the analysis is measured and used to calculate the estimated duration for the remainder of the analysis.

4.3.11 Configuration and extendability

The tool has been set up in such a way that is can easily be extended. As the scope of this thesis is limited, the tooling is published GitHub, so that others may broaden the scope using this tool [6]. All that is required for extending the scope is modification of a couple of configuration files:

- Adding a new property file for each new language that is to be supported as mentioned in section 4.3.8, including the framework unique identifiers

[6]https://github.com/Pvanhesteren/GitHubTestFrameworkAnalysis

- Extension of comment removal regexes in the comment removal configuration file for new file types that are to be supported for framework detection
- Modification of the setup configuration file to set up some basic configurations, such as the working directory of the machine
- Modification of the connection configuration file to connect to the GHTorrent database

The setup configuration enables the user to set the option to overwrite any existing results for a framework (e.g. when extending the framework scope of the analysis). By default, this flag is set to false and existing analysis will be read from the working directory. This also allows one to abort the analysis at any time and to resume it later on. This can prove to be particularly useful when performing a large analysis or when dealing with network or power issues.

4.4 Result validity

Now that the our mining approach has been laid out, we will assess the validity of our approach and reflect on any potential drawbacks in the next section.

4.4.1 Potential drawbacks

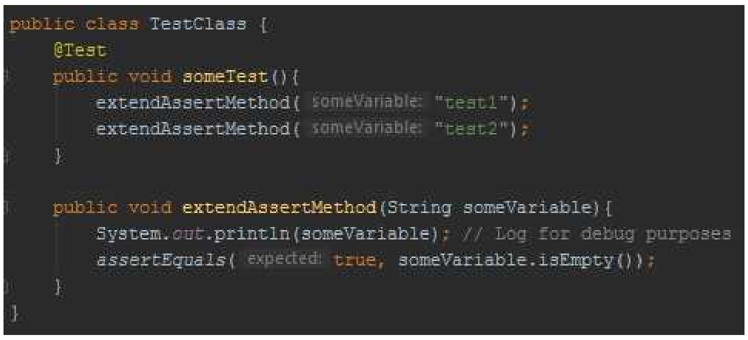
The scan based approach described in this chapter is much faster than a parsing / test-running approach. Because of this increased speed, the scale at which repositories can be analyzed is much larger. On the other hand there are several potential drawbacks that may impact the validity of the detection method, including the following:

- Non-configured frameworks will not be detected.
- Identifiers may not be unique and therefore introduce false positives.
- Extended or modified identifiers will not be detected.
- Some identifiers may not be detectable due to their nature.

The potential threats and their impact on the validity of this approach are reflected on in the next subsections.

Non-configured frameworks

Any framework that is not configured for the analysis, will not be picked up during the analysis. While this poses for a validity treat, its impact is likely to be limited. Many of the frameworks that were included in this analysis did not support IDE integration and quite some of them did not even include build tool integration. Since all well-known frameworks have been included, any left out frameworks must be not so well known and are likely to be unpopular. It is highly unlikely that such frameworks offer IDE / build tool integration, therefore limiting the impact of this potential threat.



```
public class TestClass {
    @Test
    public void someTest(){
        extendAssertMethod( someVariable: "test1");
        extendAssertMethod( someVariable: "test2");
    }

    public void extendAssertMethod(String someVariable){
        System.out.println(someVariable); // Log for debug purposes
        assertEquals( expected: true, someVariable.isEmpty());
    }
}
```

Figure 4.5: An an example of an extension method, logging the variable before calling the assertion method. As the assertion method is written only once, it will be detected 1 times, whereas it will be called two times.

Identifiers may introduce false-positives

In theory, any scan-based approach is prone to false-positives as projects may introduce assert or test methods that match identifiers and trigger regex matches falsely. However, since assert and test methods are only matched if the framework is imported into the project, the impact of this threat becomes limited.

Extended framework functionality won't be detected

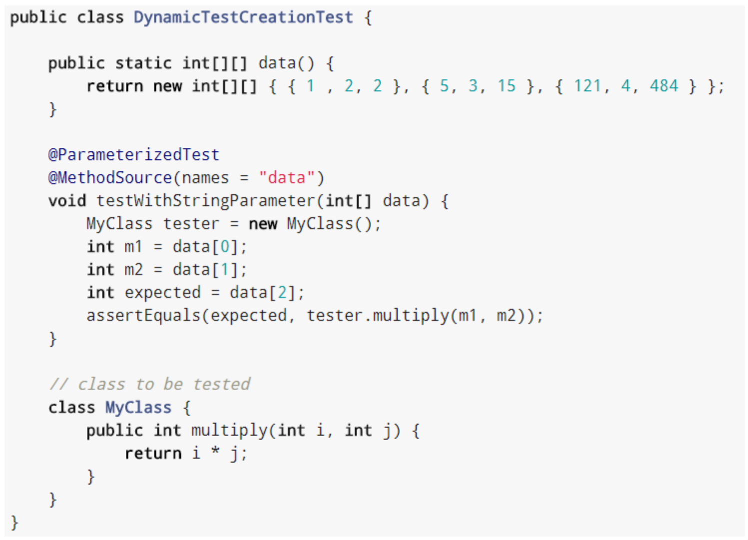
Should projects modify or extend an existing framework's functionality in such a way that the unique identifiers are no longer the same, the scan based approach will not pick up these tests or asserts, i.e. as shown in figure 4.5. This poses for a big threat of this approach, although it should be mentioned that IDE and possibly even build tool integration can break for more extreme examples, meaning that other approaches are vulnerable to this threat as well.

Identifiers may not be detectable due to their nature

Some frameworks allow for a dynamic setup of tests. This allows a user to write a single base test and to supply it with multiple inputs and expected outputs. As this scan based approach is not parsing the parameters, the parameters may be lists, objects or anything in between, i.e. as shown in figure 4.6. For this reason, it is unknown how many times the test will be executed, which poses for a threat of this approach that is not present in a parsing / test execution approach.

4.4.2 Validity analysis

Since there are several potential drawbacks of this approach, a manual validity check has been conducted on 20 random selected projects, distributed evenly over a statically typed language (Java) and a dynamically typed language (Ruby). The repositories were analyzed using the above described approach, but not deleted afterwards. All files were manually



```java
public class DynamicTestCreationTest {

    public static int[][] data() {
        return new int[][] { { 1 , 2, 2 }, { 5, 3, 15 }, { 121, 4, 484 } };
    }

    @ParameterizedTest
    @MethodSource(names = "data")
    void testWithStringParameter(int[] data) {
        MyClass tester = new MyClass();
        int m1 = data[0];
        int m2 = data[1];
        int expected = data[2];
        assertEquals(expected, tester.multiply(m1, m2));
    }

    // class to be tested
    class MyClass {
        public int multiply(int i, int j) {
            return i * j;
        }
    }
}
```

Figure 4.6: An an example of parameterized tests that pose a threat for this detection method

Table 4.3: Validity analysis for programming language Java

	Frameworks detected	**Tests detected**	**Asserts detected**
Automated analysis	18	485	1175
Manual analysis	18	484	1139
Recall	100%	99.18%	95.74%
Precision	100%	99.38%	98.77%

Table 4.4: Validity analysis for programming language Ruby

	Frameworks detected	**Tests detected**	**Asserts detected**
Automated analysis	17	960	1486
Manual analysis	17	904	1267
Recall	100%	91.35%	84.86%
Precision	100%	97.01%	99.53%

annotated and the results were compared against those obtained by the automated analysis, shown in 4.3 & 4.4. Due to time constraints, this analysis was not performed for programming languages C and JavaScript. Their results are expected to be similar to those of Java and Ruby due to their type systems, although this cannot be confirmed.

For both programming languages, all frameworks were correctly identified (100% recall and precision). One Java test case was marked with the *@Ignore* annotation. Moreover, some detected assertion methods were falsely detected as method names were overlapping with those of the included testing frameworks. The accuracy of the analysis for Ruby is significantly worse, as some String parameters were matching framework identifiers, impacting the precision. Moreover, some framework extensions were observed that were not picked up by the automated analysis, impacting the recall. This confirms the potential threats mentioned in the previous section, although its impact appears to be limited.

4.5 Conclusion

This chapter provides the setup required for a large scale analysis to gain insight into the way codified testing strategies are applied in practice and how they are impacted by type systems and available tooling. After carefully weeding out all toy-projects, 81.081 software projects remain to be analyzed for programming languages C, Ruby, Java and JavaScript. The mining tool described in this chapter is able to detect the categorized test tooling in the previous chapter. By manual comparison on 10 software projects for Java and Ruby, we show that the mining tool works with excellent recall and precision at detecting frameworks, assertions and tests. **Note**: Due to time constraints, the results for programming languages JavaScript and C were not manually verified, which poses for a threat to validity.

The mining tool can be used to obtain data on testing practices, which will be used to compare testing practices and the impact of type systems and available tooling using the following metrics:

- Rate of testing
- Assert density
- Test density
- Adoption rate of frameworks
- Usage of available tooling

The mining tool has been open-sourced [7], so that other researchers may extend the scope of our work.

[7]https://github.com/Pvanhesteren/GitHubTestFrameworkAnalysis

Chapter 5

Observed differences in testing approaches

Using the mining approach described in the previous chapter, we can now analyze the obtained data from the software projects that were analyzed. We compare the results based on a programming language and type system level in order to identify the differences and we try to explain them either by intuitive reasoning or by linking them to related literature (albeit it should be noted that the amount of related literature was rather limited). This chapter is the most vital chapter of the thesis as it delivers the sound analysis of the differences in codified testing strategies observed per type system, which helps us answer RQ2.

While this chapter helps us gain insight into the differences in testing approaches, it does not necessarily help us explain and / or gain insight into the reasons behind these differences, as not every difference can be explained by intuitive reasoning or related literature. The next chapter will focus on developer perspectives, cross-check the results with these perspectives and investigate the differences from a different angle (the human perspective).

This chapter distinguishes the metrics previously described into two separate sections: differences observed in testing behavior and the differences observed in usage of test tooling.

After analyzing the selected repositories for programming languages Java, C, Ruby and JavaScript (as described in chapter 4), we can compare the results per language and per their type system. *Note*: due to the following reasons not all projects could analyzed:

- Project repositories were too large to analyze (some projects exceeded several gigabytes of code)
- Project repositories were made private
- Project repositories were deleted
- Project repositories deleted their entire code-base

Table 5.1 shows the percentage of projects analyzed for each of the analyzed programming languages.

Table 5.1: Number of projects analyzed per language

Language	Nr. of projects selected	Nr. of projects analyzed	Projects analyzed %
JavaScript	42648	38408	90.06%
Java	17589	14192	80.69%
Ruby	11801	11410	96.69%
C	9043	7618	84.24%

5.1 Differences observed in testing behavior

First, we will have a look at the differences that were observed in testing behavior using the metrics described in the previous chapter, including the rate of testing, assert density and test density.

5.1.1 Rate of testing

The rate of using a test framework, differs per programming language. This rate is referred to as rate of testing as it is impossible to test without the adoption of any frameworks, unless the authors wrote their own test framework. **Note**: Default assertion methods in the case of JavaScript, C and Java have also been taken into account. The results have been grouped based on type system in figures 5.1 & 5.2

From these figures, there does not appear to be a clear relation between the typing of a programming language and the rate of testing. The rate of testing for C appears to be a big outlier, quite possibly related to the lack of framework support indicated in chapter 3. For all other programming languages, the rate of testing is above 50%, indicating that over half of the selected projects apply codified testing strategies.

5.1.2 Assert density

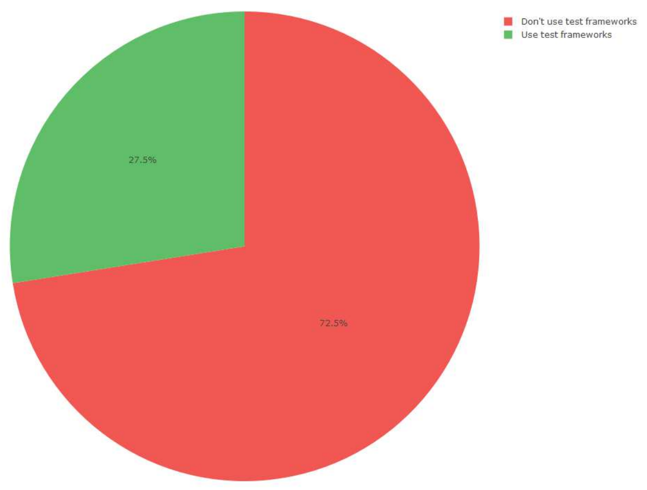
Figures 5.3 & 5.4 show the distributions of the assert density per programming language. These values have been calculated taking into account all lines of code (i.e. also taking into account configuration files). These values were also derived using just language code (i.e. only taking into account .java files), but no differences were observed.

The observed assert density of C (mean: 3.76) is much higher than the one observed in [8] (mean: $\frac{9.376}{21.909} = 0.43$). This is likely related to the fact that our study includes a much broader scope of frameworks and assertion methods, whereas [8] only extracted the *assert* keyword for all projects.

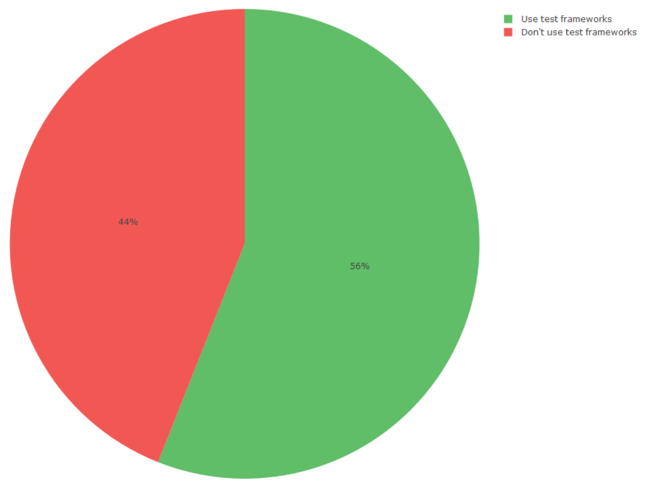
Clearly, assert density is much higher for those using dynamically typed programming languages than for projects using a statically typed programming language.

The assert density can be predicted using a Generalized Linear Model (GLM). A GLM works particularly well, since most data is not normally distributed. In this case, the assert density is the dependent variable and its independent factors include:

- Number of commits, rationale: asserts and their density can be modified via commits

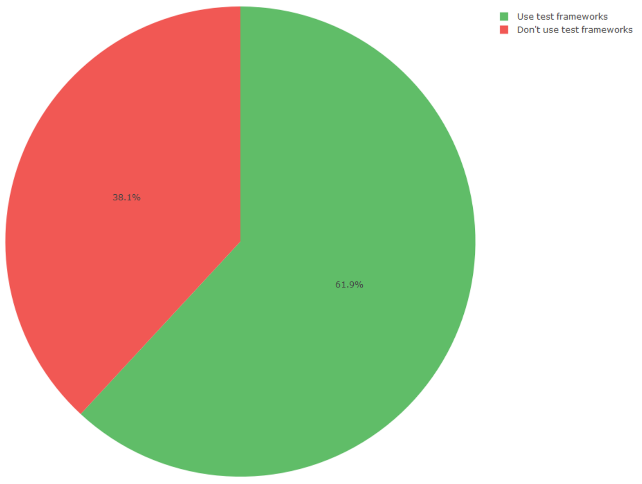


(a) Rate of testing for programming language C

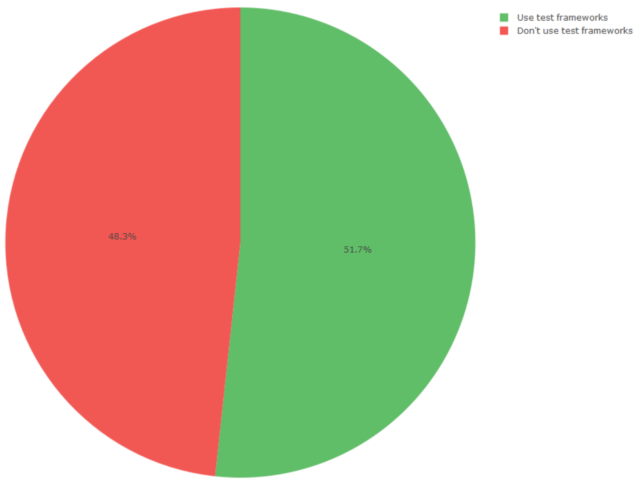


(b) Rate of testing for programming language Java

Figure 5.1: Rate of testing for statically typed programming languages

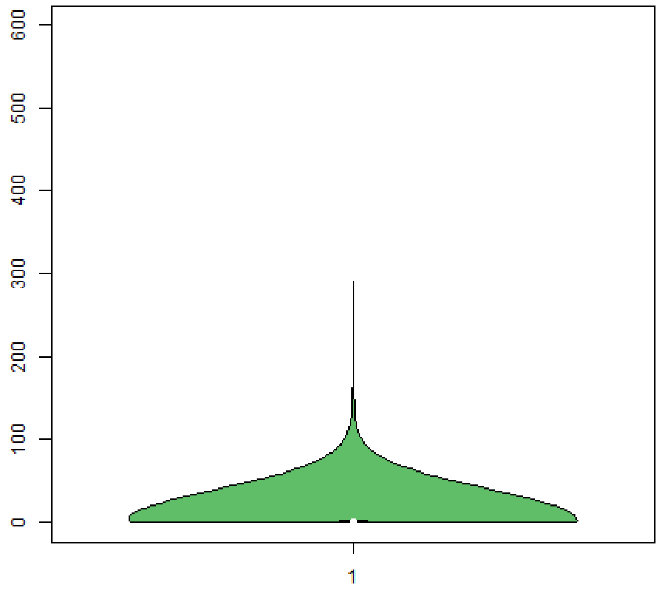


(a) Rate of testing for programming language Ruby

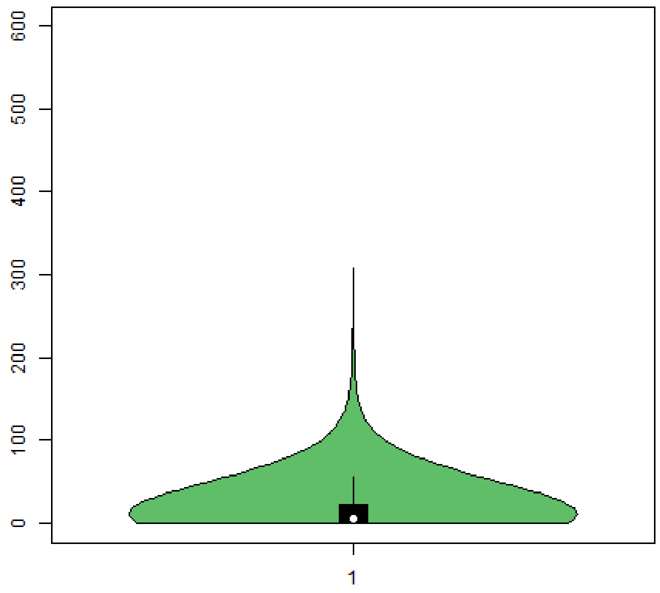


(b) Rate of testing for programming language JavaScript

Figure 5.2: Rate of testing for dynamically typed programming languages

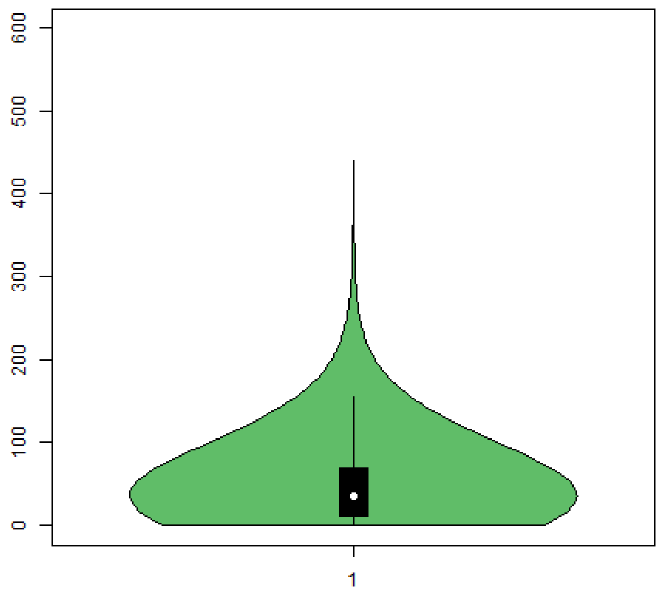


(a) Distribution of assert density in projects using programming language C. Mean: 3.76, Median: 0.13

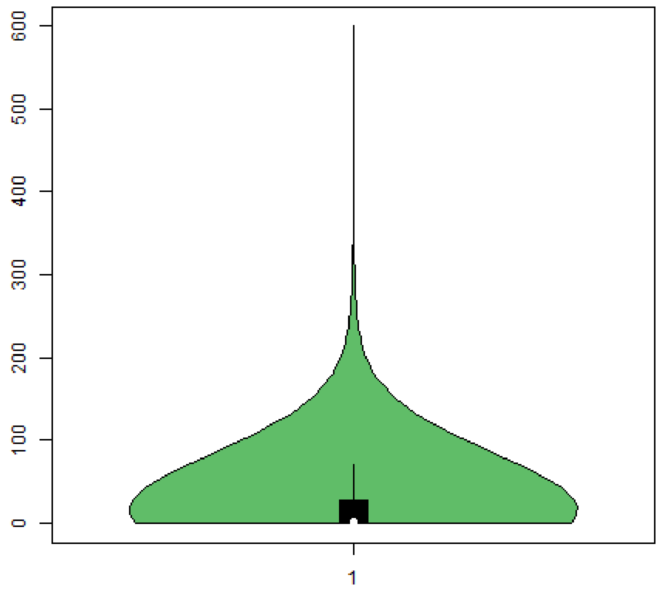


(b) Distribution of assert density in projects using programming language Java. Mean: 16.82, Median: 6.25

Figure 5.3: Distribution of assert density included in projects for statically typed programming languages



(a) Distribution of assert density in projects using programming language Ruby. Mean: 47.07, Median: 36.26



(b) Distribution of assert density in projects using programming language JavaScript. Mean: 20.85, Median: 3.08

Figure 5.4: Distribution of assert density included in projects for dynamically typed programming languages

- Number of authors, rationale: a larger number of authors requires a different testing approach
- Project age in days, rationale: more mature projects have to cope with regression and testing in a different way
- SLOC, rationale: larger projects may test in a different way than smaller projects
- SLOC language only (e.g. only .java files), rationale: larger projects may test in a different way than smaller projects
- Number of frameworks used, rationale: projects that use more frameworks may achieve higher code coverage, targeting different aspects of their project
- Language typing, rationale: projects using a programming language with dynamic typing need additional asserts to verify correct behavior
- Test density, rationale: a project that has a large number of tests may use more asserts

The obtained model has an R-squared fit of *1*, which is the highest possible fit. The obtained formula is the following:

1.06557276480396 * (Intercept) + -0.000191505885297395 * projectNrCommits + 0.177951833851141 * projectTestDensity + 0.00425569208842091 * projectNrAuthors + 0.00576744770081871 * projectNrAge + 9.17189349633422e-06 * projectNrLanguageCode + -1.03751168679971e-05 * projectNrAllCode + 5.5072262732081 * projectNrFrameworks + -7.20315464658722 * projectStaticDynamicStatic

(5.1)

The following independent variables appear to have the most impact on the assert density:

- -7.20: Static typing of the programming language
- +5.51: The number of frameworks used for testing the project

In other words, projects using a programming language with static typing are likely to have a much lower assert density than those developed using a dynamically programming language. As mentioned in chapter 2, this is likely related to the fact that the safety net of type checking at compile time is missing for dynamic languages. For this reason, additional asserts are required to provide the same level of confidence for the project. The benefits of assertions on top of the basic runtime checker were also shown in [29].

Moreover, an increased number of frameworks used in a project is bound to increase the assert density.

5.1.3 Test density

Figures 5.5 & 5.6 show the distributions of the test density per programming language. Once more, these values have been calculated taking into account all lines of code (i.e. also taking into account configuration files). These values were also derived using just language code (i.e. only taking into account .java files), but no differences were observed.

The test density results of Java and Ruby are in line with those observed in [4], once more confirming that more tests are executed in Ruby when compared to Java. **Note**: our study is different in the sense that it measured the test density as opposed to the absolute number of tests executed in the Continuous Integration pipeline in [4].

The test density can be predicted using a Generalized Linear Model (GLM). Once again, a GLM works particularly well, since most data is not normally distributed. In this case, the test density is the dependent variable and its independent factors include:

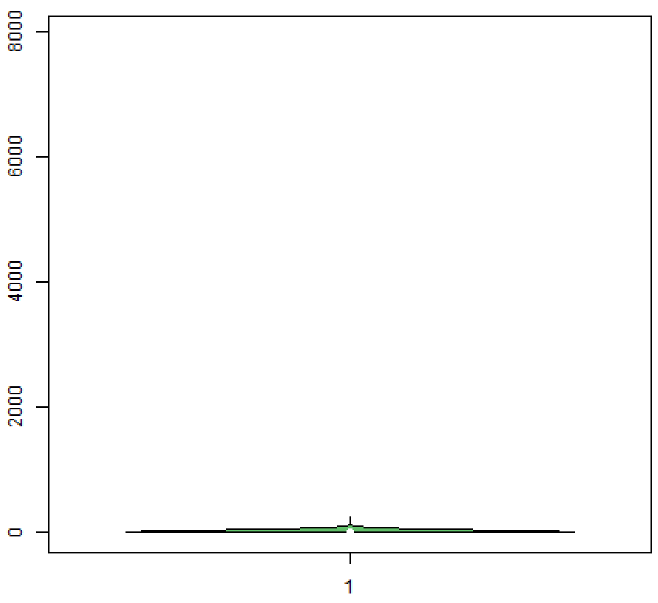
- Number of commits, rationale: tests and their density can be modified via commits
- Number of authors, rationale: a larger number of authors requires a different testing approach
- Project age in days, rationale: more mature projects have to cope with regression and testing in a different way
- SLOC, rationale: larger projects may test in a different way than smaller projects
- SLOC language only (e.g. only .java files), rationale: larger projects may test in a different way than smaller projects
- Number of frameworks used rationale: projects that use more frameworks may achieve higher code coverage, targeting different aspects of their project
- Language typing, rationale: projects using a programming language with dynamic typing need additional tests to verify correct behavior
- Assert density, rationale: a project that has a large number of asserts may use more tests

The obtained model has an R-squared fit of *1*, which is the highest possible fit. The obtained formula is the following:

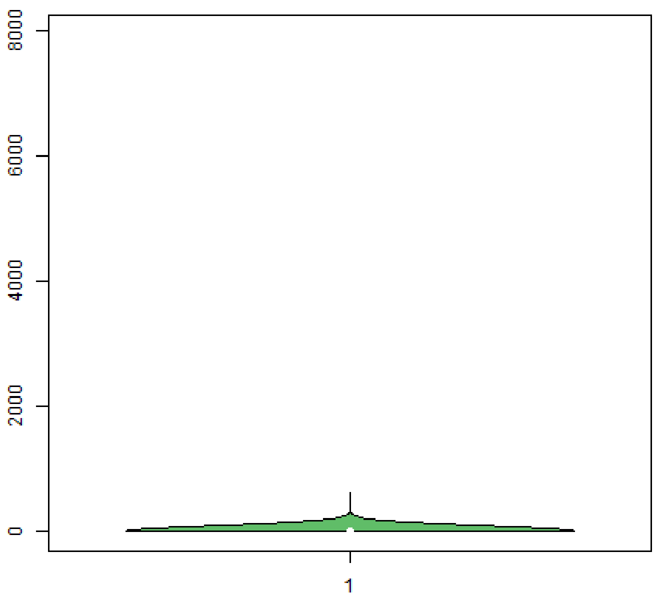
0.825725148814765 * (Intercept) + -1.7068299159107e-05 * projectNrCommits + 0.342923651225365 * projectAssertDensity + 0.00046283138115177 * projectNrAuthors + 0.000393063430055679 * projectNrAge + 2.46765657906822e-06 * projectNrLanguageCode + -2.86378552673201e-06 * projectNrAllCode + 1.64367174217076 * projectNrFrameworks + -2.05047063471444 * projectStaticDynamicStatic

(5.2)

The following independent variables appear to have the most impact on the test density:

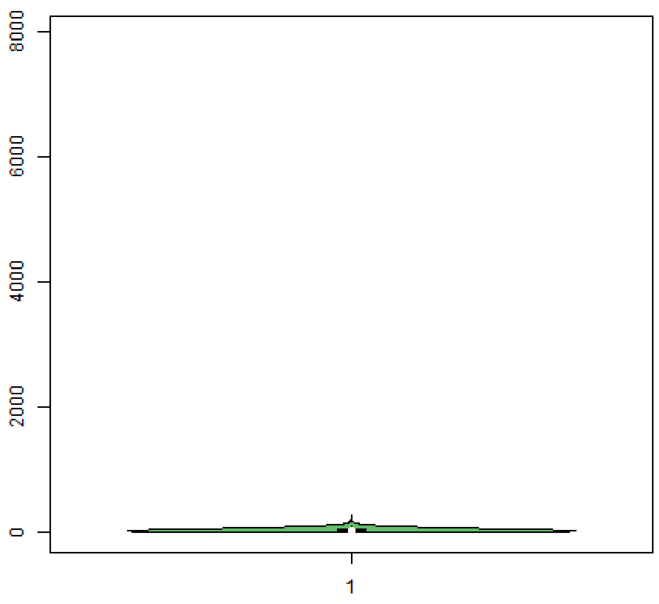


(a) Distribution of test density in projects using programming language C. Mean: 0.93, Median: 0

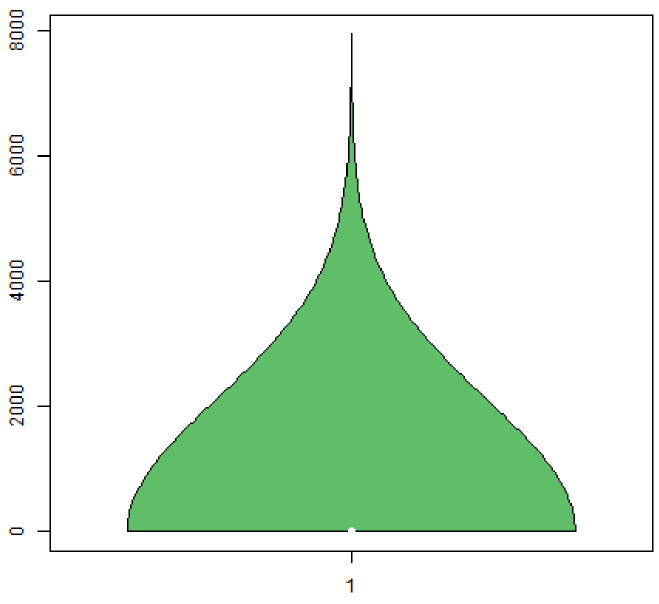


(b) Distribution of test density in projects using programming language Java. Mean: 7.73, Median: 2.89

Figure 5.5: Distribution of test density included in projects for statically typed programming languages



(a) Distribution of test density in projects using programming language Ruby. Mean: 28.37, Median: 23.00



(b) Distribution of test density in projects using programming language JavaScript. Mean: 10.48, Median: 0.34

Figure 5.6: Distribution of test density included in projects for dynamically typed programming languages

- -2.05: Static typing of the programming language
- +1.64: The number of frameworks used for testing the project

Once more, static typing of the programming language appears to have the largest (negative) impact on the test density. Moreover, an increased number of frameworks used in a project is bound to increase the test density. As mentioned in chapter 2, this is once more likely related to the fact that the safety net of type checking at compile time is missing for dynamic languages. For this reason, additional tests are required to provide the same level of confidence for the project. The benefits of assertions on top of the basic runtime checker were also shown in [29].

5.2 Differences observed in usage of available tooling

Next, we will have a look at the differences that were observed in usage of available test tooling using the metrics described in the previous chapter, including the adoption rate of codified test tooling and the usage of available tooling.

5.2.1 Adoption rate of codified test tooling

The adoption rate of testing frameworks consists of three parts: the number of frameworks that is used for codified testing in a project, as well as the usage of the frameworks that test the project and their composition. The following three sections lay out these parts. **Note:** since projects that do not apply testing should not impact this data, they have been filtered out and have not been taken into account.

Number of frameworks used for codified testing

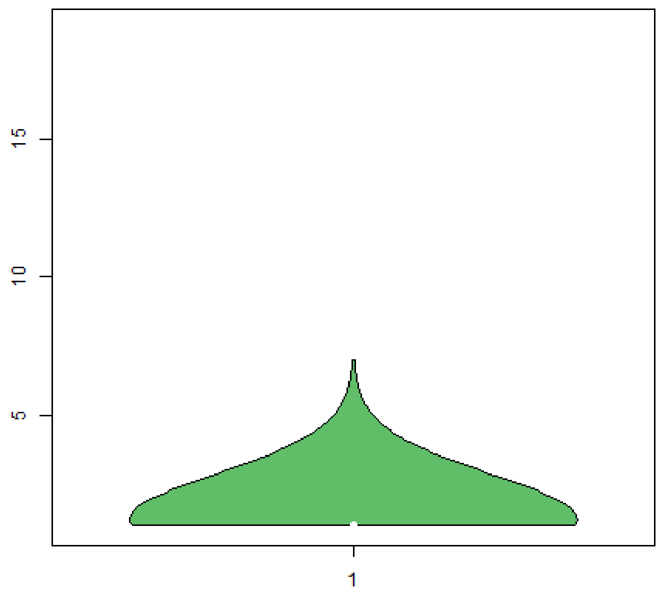
The distribution of the number of frameworks used is shown in figures 5.7 & 5.8.

While both values of the mean and median number of frameworks used for statically and dynamically typed programming languages appear to be the same, the distribution of the violin plots is somewhat different. The number of frameworks used in dynamically typed languages can grow larger than those observed in statically typed languages. Moreover, there is more deviation in the the distribution of the number of frameworks used than those of statically typed languages, although there is no apparent reason.

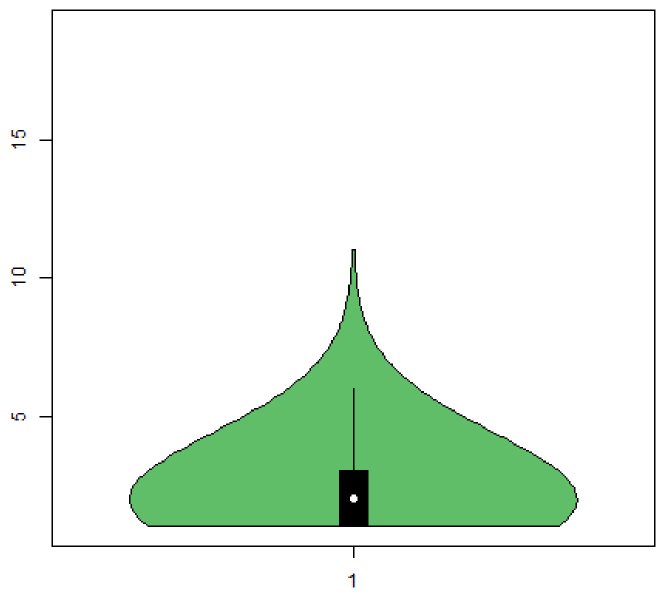
Once more, the number of frameworks used in C projects appears to be a big outlier, which was also observed during the interviews described in chapter 6. This outlier and its impact is described more in debt in chapter 6.

The number of frameworks used for testing purposes can be predicted using a Generalized Linear Model (GLM). A GLM works particularly well, since most data is not normally distributed. In this case, the number of frameworks is the dependent variable and its independent factors include:

- Number of commits, rationale: the number of frameworks used and their tests can be modified via commits

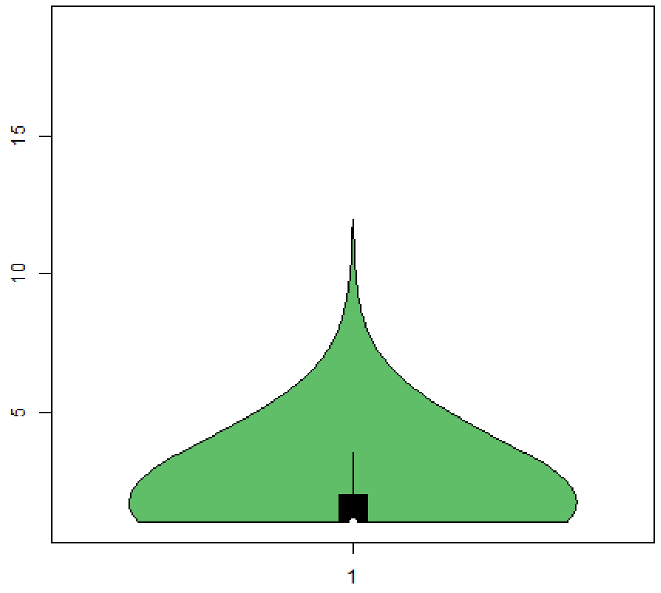


(a) Distribution of the number of frameworks used in projects using programming language C (Median: 1, Mean: 1.25)

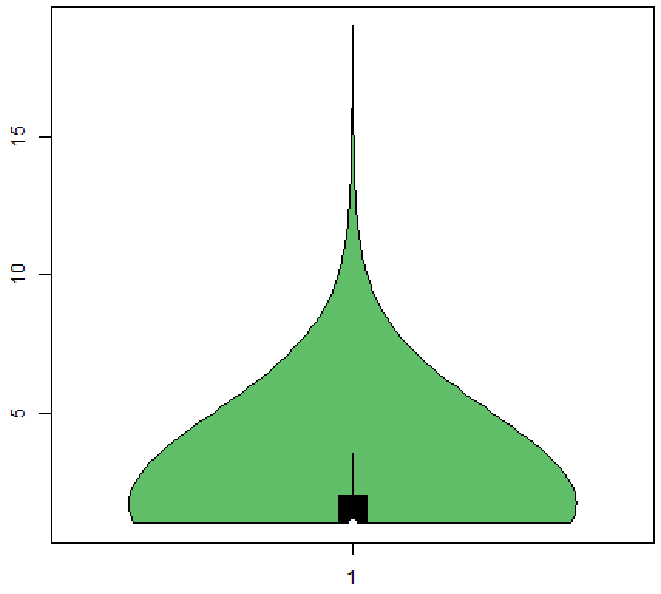


(b) Distribution of the number of frameworks used in projects using programming language Java (Median: 2, Mean: 2.18)

Figure 5.7: Violin plot of the number of frameworks included in projects for statically typed programming languages



(a) Distribution of the number of frameworks used in projects using programming language Ruby (Median: 1, Mean: 1.87)



(b) Distribution of the number of frameworks used in projects using programming language JavaScript (Median: 2, Mean: 1.80)

Figure 5.8: Violin plot of the number of frameworks included in projects for dynamically typed programming languages

- Number of authors, rationale: a larger number of authors requires a different testing approach
- Project age in days, rationale: more mature projects have to cope with regression and testing in a different way
- SLOC, rationale: larger projects may test in a different way than smaller projects
- SLOC language only (e.g. only .java files), rationale: larger projects may test in a different way than smaller projects
- Language typing, rationale: projects using a programming language with dynamic typing need additional / different tests and frameworks to verify correct behavior
- Assert density, rationale: a project that has a large number of asserts may use different types of tests (offered by different frameworks)
- Test density, rationale: a project that has a large number of tests may use different types of tests (offered by different frameworks)

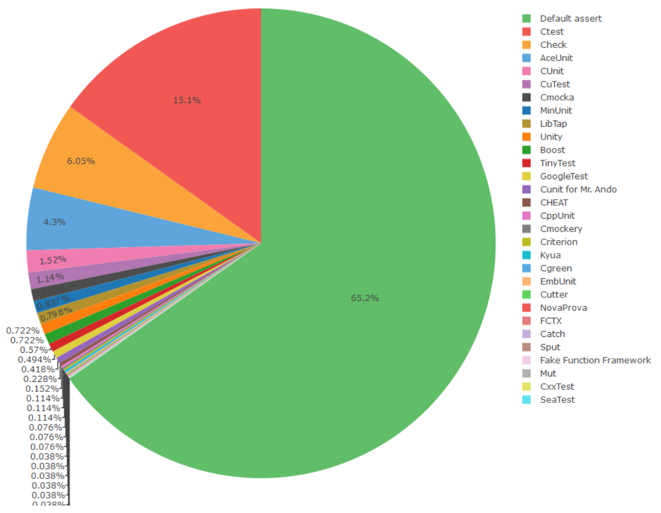
The obtained model has an R-squared fit of *0.21*, which is a very bad fit for the model. It appears that the number of frameworks used is not (in)directly impacted or correlated by the above mentioned factors and can therefore not be predicted using these factors.

Adoption rate of frameworks

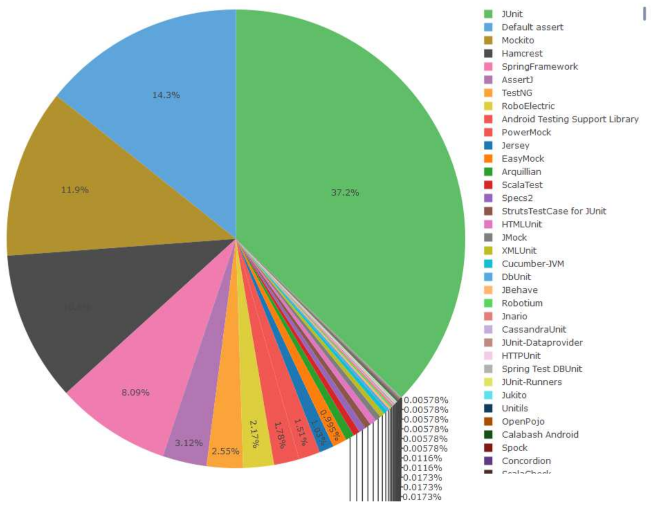
Figures 5.9 & 5.10 show the adoption rate of each of the frameworks per programming language. It immediately becomes clear that the assumption that the list of frameworks would be skewed (made in chapter 4), is true. **Note:** the percentages shown in the figures are not the absolute percentages of the adoption rate of the framework. These percentages indicate how often the framework occurs in the list of total frameworks that was detected for all projects in a particular programming language, therefore the percentages indicate the relative adoption rate to other frameworks. It should be noted that a lot of frameworks configured and mentioned in 4 were not used in practice and are therefore not shown in the figures. Instead, a list of tools that there not detected using this approach is shown in figures 5.2 (C), 5.3 (Java), 5.4 (Ruby) & 5.5 (JavaScript).

In the case of C, the default assert appears to be used by a large amount of projects. Other prominent frameworks include Ctest, Check and AceUnit. In the case of Java, JUnit appears to be the most widely used framework, followed by the default Java assert, Mockito, HamCrest and SpringFramework.

In the case of Ruby, RSpec is most often used, followed by MiniTest, Capybara, Test::Unit, factory_girl, WebMock, Mocha and Shoulda. Interestingly enough, Test::Unit - the default test suite for Ruby - is not the most popular testing framework, which is quite different from the default assert usage of functionality in NodeJS, Java and C, especially since this default assertion suite offers more extensive features than those of other languages. The results are

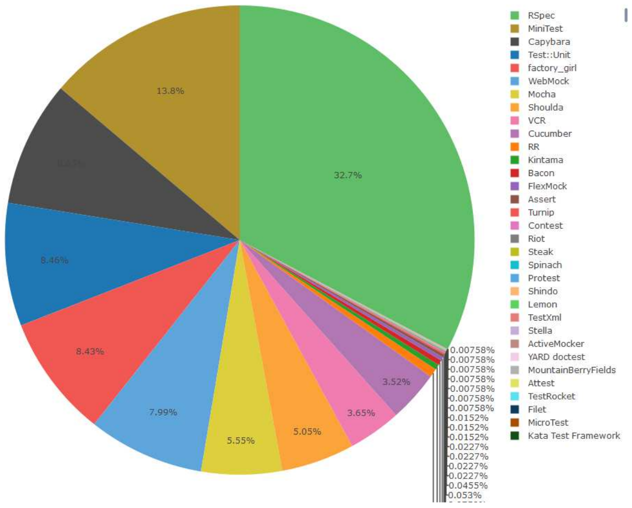


(a) Distribution of frameworks used in projects using programming language C

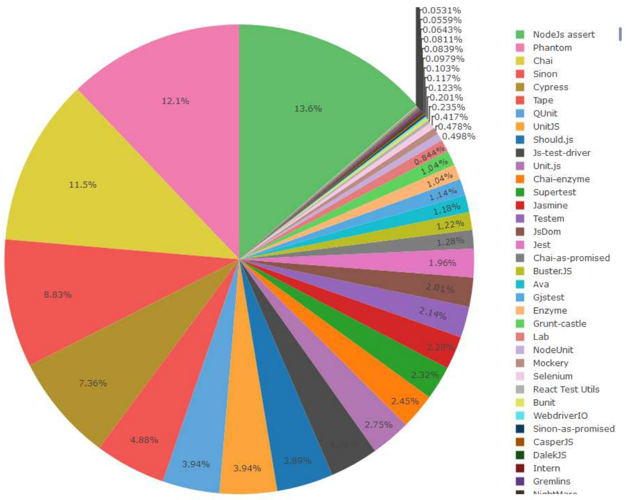


(b) Distribution of frameworks used in projects using programming language Java

Figure 5.9: Distribution of frameworks included in projects for statically typed programming languages



(a) Distribution of frameworks used in projects using programming language Ruby



(b) Distribution of frameworks used in projects using programming language JavaScript

Figure 5.10: Distribution of frameworks included in projects for dynamically typed programming languages

Table 5.2: An overview of test frameworks that were not detected for programming language C

Framework
LibCbdd
Cfix
CppUTest
Lcut
LibU
MinUnit
Mut
Opmock
RcUnit
STRIDE
XTests
LibCut
C2Unit
Unit++
Microsoft Unit Testing Framework for C++
Bandit
CppUnitLite

more or less in line with those observed in the 2017 JetBrains developer survey [1], which strengthens the validity of the mining approach described in the previous chapter.

In the case of JavaScript, there is no clear winner when it comes to the adoption rate of test tooling. The list consists of NodeJS' default assert, Phantom JS, Chai, Sinon, Cypress, Tape, QUnit, UnitJS and Should.js. The results are very different from the ones observed in the 2017 JetBrains developer survey [2]. While all of the tools listed in the JetBrains developer survey are shown in figure 5.10, the usage of these tools is very different. This is possibly related to the fact that subset of JavaScript developers participating in the JetBrains survey may not reflect the JavaScript testing community as a whole. However, it could also be seen as an indicator that the mining approach is not accurate.

In all cases, the default assert and test libraries are among the most widely adopted testing frameworks. The list of testing frameworks per language is rather skewed as only a handful are widely adopted. In the case of the adoption of testing frameworks for dynamically typed programming languages the adoption rate is spread over a lot of frameworks, indicating that developers have a lot of suitable or possibly even interchangeable frameworks to choose from. This is much less observed for statically typed programming languages.

[1]https://www.jetbrains.com/research/devecosystem-2017/ruby/

[2]https://www.jetbrains.com/research/devecosystem-2017/javascript/

Table 5.3: An overview of test frameworks that were not detected for programming language Java

Framework
SureAssert
beanSpec
Cuppa
EvoSuite
GrandTestAuto
GroboUtils
JDave
JExample
JMockit
NuTester
Concurrent-JUnit
ConcurrentJunitRunner
JUnitPerf
TwiP
jfcUnit
XTest
UISpec4J
Thread Weaver
Abbot

Table 5.4: An overview of test frameworks that were not detected for programming language Ruby

Framework
Testy
Exemplor
Test_inline
Detest
Coulda
Unencumbered
Bewildr
Stories
Saki
Ramcrest

Table 5.5: An overview of test frameworks that were not detected for programming language JavaScript

Framework
Mocha
Karma
Chimp
Spectacular
painless
JfUnit
RxTestRunner
Grunt-jasmine
Phantomjs-YUItest
Lotte
Smokestack
Expect

Composition of included frameworks

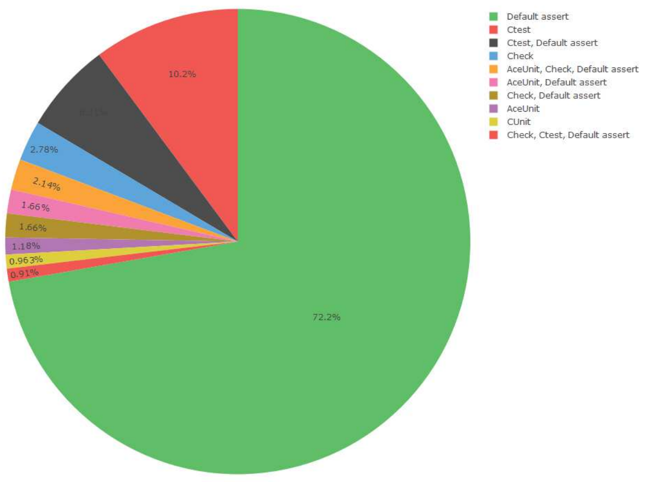
Figures 5.11 & 5.12 show the top 10 framework compositions per programming language. **Note**: many more compositions were found, but their adoption rate was very low and the figures would become unreadable when listing all of them.

For all programming languages, using one particular framework appears to be the most common composition. This may indicate that developers prefer sticking to a single framework with a common syntax or setup for all of their tests. Moreover, the default assert library appears to be popular both in terms of using it as a single framework or combining it with others, although in the case of Ruby Test::Unit is outperformed by RSpec. Mocking frameworks or frameworks with extended assert matchers appear to be favored over other frameworks in the top compositions. In the case of JavaScript, most top compositions consist of just one framework. This may indicate that JavaScript frameworks offer more features than other, therefore knocking down the need for additional frameworks.

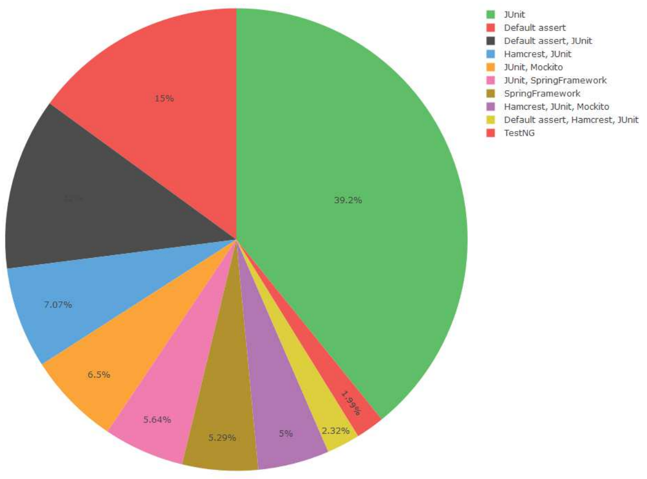
As is shown in figures 5.11 and 5.12, common / default assertion libraries are often used in conjunction with other - more extensive - frameworks that overlap with the default assertion libraries. This is particular, as the assertions used in the default assertion libraries could easily have been replaced with those of the more extensive frameworks. Instead, developers now have to maintain two different type of assertions with a different syntax and test runner. This increases the number of dependencies and knowledge required to maintain and run the tests and can be seen as a form of testing debt.

5.2.2 Usage of available tooling

With the adoption rate of available test tooling in the previous section and their classification in chapter 3, we can derive the classification of the most used frameworks per programming language. For every project using a framework, its corresponding classifications receive one

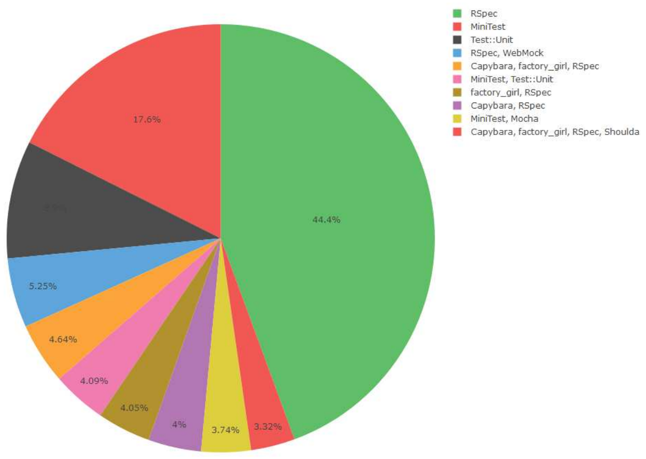


(a) Top 10 compositions of frameworks used in projects using programming language C

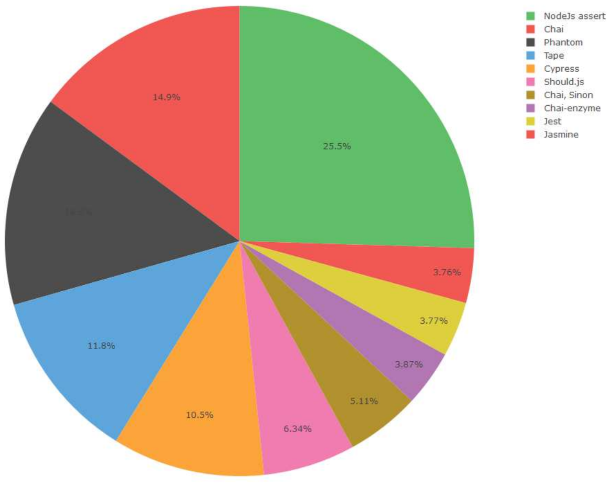


(b) Top 10 compositions used in projects using programming language Java

Figure 5.11: Top 10 compositions included in projects for statically typed programming languages

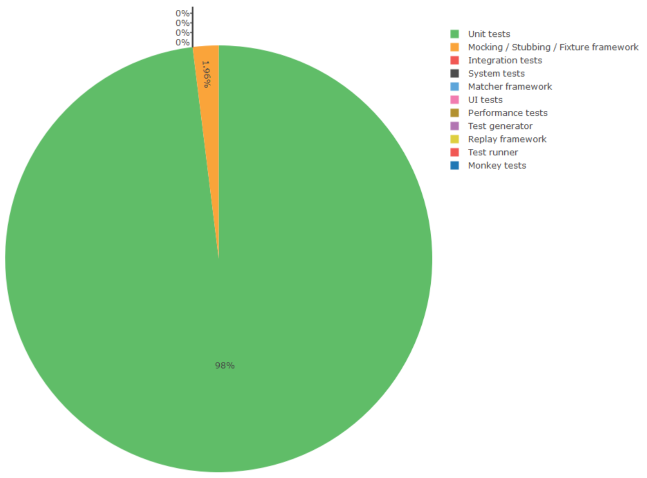


(a) Top 10 compositions used in projects using programming language Ruby

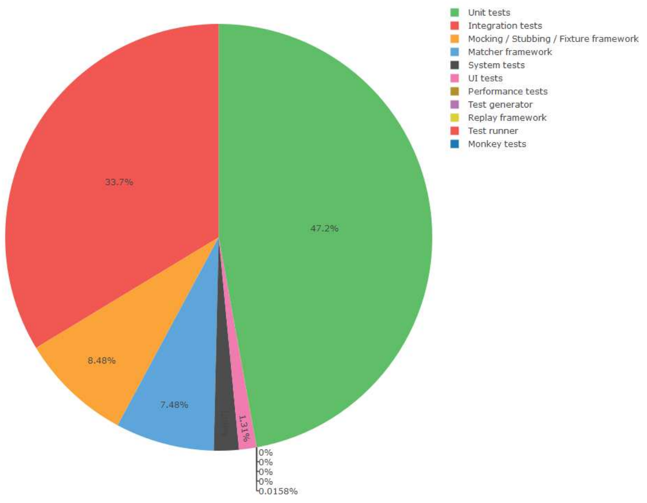


(b) Top 10 compositions used in projects using programming language JavaScript

Figure 5.12: Top 10 compositions included in projects for dynamically typed programming languages



(a) Distribution of used categorized test tooling for programming language C

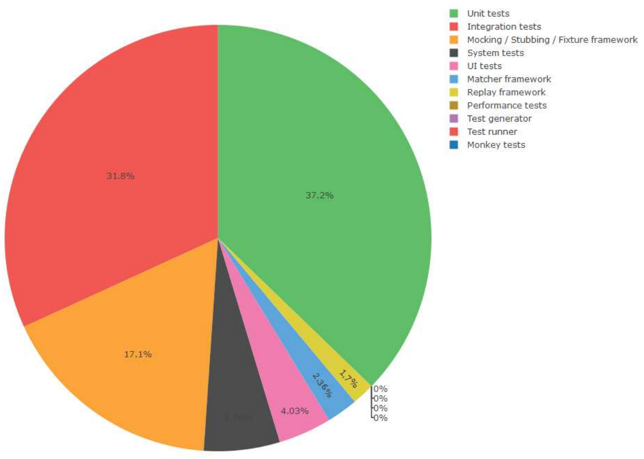


(b) Distribution of used categorized test tooling for programming language Java

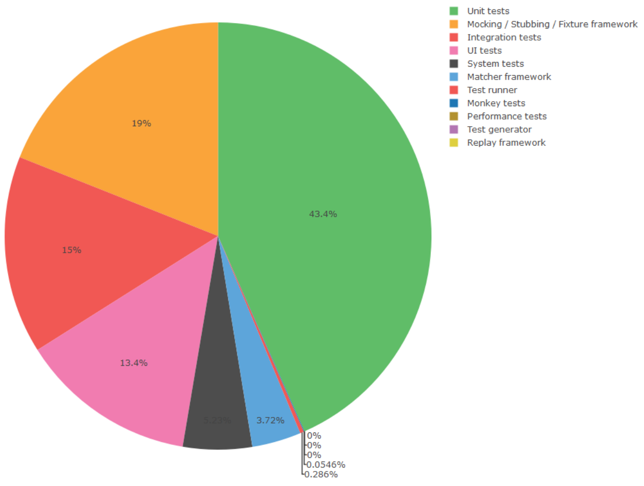
Figure 5.13: Distribution of used categorized test tooling for for statically typed programming languages

point and the results are summed and visualized per programming language in figures 5.13 and 5.14.

Regardless of the programming languages, all projects predominantly use unit testing for their codified testing strategies. Unit testing appears to be much more significant for statically typed languages when compared to dynamically typed languages. Mocking, System



(a) Distribution of used categorized test tooling for programming language Ruby



(b) Distribution of used categorized test tooling for programming language JavaScript

Figure 5.14: Distribution of used categorized test tooling for dynamically typed programming languages

testing and UI testing practices are much more common in dynamically typed languages than in statically typed languages. This might be related to the fact that statically typed languages provide a safety net by catching a lot of trivial implementation faults at compile time, as explained in chapter 2. For dynamically typed languages this is not the case, as the typing of variables is only known at run time, thus requiring for more extensive tests, ideally of a black box nature (i.e. integration and system tests).

Matching frameworks have a larger usage rate in Java when compared to other languages. This is likely caused by the language syntax, which limits readability and expressability in comparison to more flexible dynamically typed programming languages, as was also observed in [26]. These matcher frameworks combine the language type system with improved read- and express-ability. Integration testing appears to be limited in JavaScript, although this is probably made up for by an increase in UI and system testing when compared to other languages.

5.3 Conclusion

Using the mining tool described in the previous chapter, we obtained data on the testing approaches that were adopted by a total of 71.628 software projects written in Java, Ruby, JavaScript and C. The rate of testing for C is much lower than those observed in all other languages. This is likely related to the lack of tooling observed in chapter 3, indicating that there is a clear need for better test tooling for this programming language. Both violin plots (figures 5.3 and 5.4) and a generalized linear model show that assert density is largely impacted by type systems, as dynamically typed programming languages included a larger number of asserts when compared to statically typed languages. Moreover, the same applied for test density, albeit it should be noted that the impact was somewhat smaller. This is likely related to the fact that these programming languages are missing out on the safety net that type systems provide, as was explained in chapter 2. The obtained results are in line with the results obtained in [4]. Our approach observed a much larger assert density for programming language C than the approach taken in [8], which can be explained by the fact that the approach taken in [8] only included a basic regex on the word 'assert', which did not cover all frameworks that could be adopted for testing approaches.

When it comes to the number of frameworks used for testing purposes, this number can grow larger in the case of dynamically typed languages than those observed in statically typed languages. Moreover, there is more deviation in the the distribution of the number of frameworks used than those of statically typed languages, although there is no apparent reason. We observe that in the case of programming language C the number of frameworks used is much smaller, once more strengthening the need for better test tooling for this language.

The most popular test tools are listed as per the following:

- C: Default assert, Ctest, Check and AceUnit
- Java: JUnit, default assert, Mockito, HamCrest and SpringFramework

- Ruby: RSpec, MiniTest, Capybara, Test::Unit, factory_girl, WebMock, Mocha and Shoulda
- JavaScript: NodeJS' default assert, Phantom JS, Chai, Sinon, Cypress, Tape, QUnit, UnitJS and Should.js

These results are in line with those observed in the 2017 JetBrains developer survey [3] in the case of Ruby, but are different from the ones observed from JavaScript [4]. This is possibly related to the fact that subset of JavaScript developers participating in the JetBrains survey may not reflect the JavaScript testing community as a whole.

Regardless of the programming languages, all projects predominantly use unit testing for its codified testing strategies. Unit testing appears to be much more significant for statically typed languages when compared to dynamically typed languages. Mocking, System testing and UI testing practices are much more common in dynamically typed languages than in statically typed languages. This might be related to the fact that statically typed languages provide a safety net by catching a lot of trivial implementation faults at compile time, as explained in chapter 2.

Figures 5.11 and 5.12 show that extensive frameworks are often used in conjunction with default assertion libraries. This is particular, as the assertions used in the default assertion libraries could easily have been replaced with those of the more extensive frameworks, which can be seen as a form of testing debt as developers have to deal with an increased number of dependencies and different test syntaxes.

[3] https://www.jetbrains.com/research/devecosystem-2017/ruby/

[4] https://www.jetbrains.com/research/devecosystem-2017/javascript/

Chapter 6

Developer perspectives on testing approaches

The previous chapter provided insight into the observed differences in testing approaches and usage of test tooling from a ‘mining data’ perspective. However, this approach did not necessarily reveal how developers approach testing and what rationales they use for their testing decisions (e.g. why do developers adopt tool X?). This chapter will therefore focus on developer perspectives via two approaches:

- Interviewing developers that approach testing in a very different way than developers tend to do for that same programming language
- Conducting a large scale survey to gain insight in the testing approach developers take and the rationales involved

This approach will complement the previous chapter as the results can partly be used to verify the correctness of the previous chapter. Moreover, it provides a new angle to look at testing approaches that can provide new insights into testing practices. This chapter plays a vital role in answering RQ3 as it delivers the interviews with developers that use on out of the ordinary testing approaches, a large scale survey among developers to gain insight into the relationship of testing practices, type systems and test tooling and sound analysis combining the interview and survey results with the data obtained in RQ2. Finally, it lays the foundation for the next chapter that will provide an overview of possible test tooling improvements to better aid developers in their development process.

6.1 Interviewing developers

While analyzing the data obtained in chapter 5, some notable results were found. Some projects were of a significant size (> 10000 SLOC) and appeared to apply no codified testing strategies. Moreover, some projects adopted an exceptionally high number of testing frameworks (> 7). In order to gain a better understanding of the underlying ideas of the project authors, the authors of these projects were contacted and briefly interviewed to explain their point of view on these design principles. **Note**: All authors were asked the same set of questions. Not all authors were willing to respond to all questions and some authors shared their thoughts via email conversions beyond the scope of these questions. For this reason, some results were not taken into account as part of the results gathered from these questions and will be mentioned separately.

Note: percentages were calculated based on the amount of developers that answered the question, as not all respondents were willing to answer all questions. Therefore, percentages may differ per question. The number of respondents is mentioned at the beginning of every subsection.

6.1.1 Interviewing developers using a large number of test frameworks

As shown in chapter 3, many frameworks exist covering a wide set of features. Because of their extensive scope, projects should be able to cover the entire codified testing spectrum with +- 3 frameworks (e.g. a framework that covers unit & integration testing, a framework that covers mocking purposes and a framework that covers system or UI testing). This is also in line with the common number of frameworks used, as was shown in chapter 5. All project authors that were adopting 7 or more testing frameworks were contacted and asked the same set of questions, namely the following:

- How did you pick the frameworks used in your project?
- Do you allow contributors to your project to include any testing framework they like or should they adhere to certain guidelines?
- Arguably, there is quite some overlap in the frameworks you are using for testing purposes. Why did you choose to incorporate them next to each other in your project?
- Arguably, including many frameworks in an (open-source) project increases maintenance and perhaps even technical debt. What is your view on this?
- What measures have you taken to align test layouts, so that tests have a similar structure?
- What measures have you taken to ensure code coverage?

222 authors matched the criteria and were contacted via email. In total, 20 respondents cooperated and shared their thoughts on testing approaches via interviews that were conducted via email. **Note:** many email-addresses were invalid due to which emails bounced and authors were not reached. Therefore, the true response rate is hard to indicate, but it is assumed to be much higher.

Selecting suitable frameworks for a project

19 (95%) project authors responded to this question and elaborated on their selection process for testing frameworks. A large majority (73.68%) indicated to pick frameworks based on their functionality. Moreover, they indicate the number of frameworks is significantly larger as they require tailor-made functionality that is not covered in ‘common’ frameworks.

31.57% of the authors indicate to only adopt frameworks they are familiar with. A small number (21.05%) only adopts frameworks that are commonly-used and provide a readable syntax. Only one author indicated that the ease of writing tests is taken into account when choosing a set of frameworks. One author stressed the fact that one should not adopt frameworks based on just its popularity:

> Basically one of our mistakes in choosing a test framework was to trust trends, Basically you shouldn't trust trends, You need to choose your tool based on your case study not the trend. If many people are using a tool, it does not mean that it would be a good fit for you as well.

Adding new frameworks to a project

19 (95%) project authors responded to this question and elaborated on their guidelines of adding new testing frameworks to the project. A large number of authors (42.10%) indicated that all frameworks were chosen at the very beginning of the project and no new frameworks would ever be adopted. 47.37% had a similar view, albeit they would allow the adoption of a new framework after a thorough discussion (although most of them indicated this would probably never be the case). It seems that developers give serious thought to their test structure and process when creating a new project.

Overlap of frameworks

19 (95%) project authors responded to this question and elaborated on the possible overlap of the frameworks they were using in their project. A small number of authors (26.31%) were convinced that there was no overlap in the frameworks they were using. 42.11% argued there was some overlap in the frameworks they were using, but required additional functionality that was missing in other frameworks. Moreover, 42.11% argued that some testing practices could have been applied using a smaller set of frameworks, however the additional frameworks allowed them to develop tests more quickly as some practices can be applied more easily using these additional frameworks. 31.58% of the authors agreed to have significant testing debt as tests were written using different frameworks. Moreover, some authors were surprised that frameworks were detected in their project by our mining approach described in chapter 4 as they were convinced they were no longer being used. The frameworks turned out to be used by a small number of tests that were not yet refactored and were in fact a remainder of their project testing debt. This is a good indicator that a large number of frameworks may lead to an increase in technical (testing) debt and should be avoided were possible.

Impact of a large number of frameworks on technical debt

19 (95%) project authors responded to this question and elaborated on the possible increase of technical debt by the large number of frameworks used in their project. The majority of authors (73.68%) agreed to the fact that the large number of test frameworks was increasing their technical debt, as was also observed in the results of the previous question. 42.11%

argued that while there was an increase in technical debt, the trade-off was worth it as they were able to test more thorough. A small number of authors (10.53%) argued that an increase of technical debt could be avoided by taking precautions and setting up a proper test structure. One developer stressed the fact that one should be careful when adopting a large number of frameworks, as this may decrease test execution speed and the ease of writing tests:

> Yes, this hit us mostly during the last two years. We feel that many changes to the code require extensive changes to the test suite. To give you an idea: Our test suite runs for over 20 Minutes on Travis CI for a code coverage of just over 60%.

Aligning test layout throughout the project

18 (90%) project authors responded to this question and elaborated on their test layout alignment throughout the project. Many authors (44.44%) use pull requests to verify that tests are aligned throughout the project. Some (27.78%) have taken additional measures and written extensive templates to which all tests must adhere. 11.11% of the authors mention that they do not have a real structure, but try to adhere to test framework ‘conventions’. A significant amount of authors (38.89%) have taken no measures to ensure test alignment throughout their project.

Ensuring code coverage

19 (95%) project authors responded to this question and elaborated on how they ensured code coverage throughout their project. The majority (73.69%) of project authors indicated to ensure code coverage in their testing process. Popular tooling included:

- SimpleCov (15.79%)
- Jacoco (10.53%)
- IDE code coverage (10.53%)
- Sonar (10.53%)
- Pitest (5.26%)
- Coveralls (5.26%)
- rcov (5.26%)

26.31% of the authors indicated they did not measure or ensure code coverage.

6.1.2 Interviewing developers that do not apply testing

In the case of programming language C, a large number of projects was observed that was of a significant size (> 10000 SLOC) that were not using any of the frameworks mentioned in chapter 3. As a piece of software grows in terms of its size and features, regression testing becomes all the more important, which is why it is noteworthy that these projects did not use any of these frameworks. All project authors of projects with > 10000 SLOC for which no frameworks were detected were contacted and asked the same set of questions, namely the following:

- What is the reason you have decided not to include any codified testing strategies in your project?
- What measures have you taken to ensure code quality and validity?
- Do you feel that there is an increased need for thorough code reviews when a project does not incorporate codified testing strategies?
- Arguably, distributed (open source) software development such as projects like yours on GitHub introduce a distribution of project knowledge / ownership. Since the project is no longer developed by one single person/team on site, it becomes likely that not all contributors are aware of any code change implications. How do you ensure that there are no breaking changes without adopting codified testing strategies?
- Arguably, adopting codified testing strategies in a project like yours would consume a large portion of time that could have been spent on further development / improvement of the project. Do you feel testing is a burden and moreover, did this influence your decision to not adopt codified testing strategies in your project?
- Do you have any repulsion against codified testing strategies in general or did you feel that the adoption of such strategies was simply unnecessary / not suitable for this project? If you have any distaste against codified tests, would you care to elaborate on that?

356 authors matched the criteria and were contacted via email. In total 63 authors responded to the interview requests. 38 (60.32%) respondents cooperated and shared their thoughts in testing approaches via interviews that were conducted via email. 25 respondents (39.68%) indicated they created their own testing framework and did in fact apply testing approaches. The opinion of these respondents is separated from those that did not apply testing approaches in the following subsections. **Note:** many email-addresses were invalid due to which emails bounced and authors were not reached. Therefore, the true response rate is hard to indicate, but it is assumed to be much higher.

Developers writing their own test harness

A large number of authors (25, 39.68% out of the total responses) indicated that they had chosen to implement their own (private) test harness, resulting in a false-positive that the project did not apply codified testing strategies by the setup described in chapter 4. After several email conversations with these authors, it became clear that they had chosen to write their own test harness as the available testing frameworks did not suit their needs for the following reasons:

- Lack of performance (speed of test execution)
- Frameworks lack functionality
- Frameworks are too complex
- Frameworks are too generic

It is noteworthy that many developers claimed that there is a lack of simple, straight-to-the-point, test tooling. In the case of JavaScript and Ruby, many of such frameworks exist, but these are not used on a significant scale in practice (as was observed in chapter 5). While compiling the list of frameworks described in chapter 3 and their unique identifiers required for the setup described in 4, we observed that documentation for C frameworks appeared to be very scarce, confusing and poor. This may explain the lack of adoption of these frameworks.

Reasons for not applying codified testing strategies

37 (97.37%) project authors responded to this question and elaborated on the reasons they did not apply codified testing strategies throughout their project. 35.14% of the respondents indicated that they did not test the project due to a lack of time to do so. One developer stated:

> Perhaps governments are large companies have nothing to do but spend 5 times longer on a project. It's not practical for many smaller companies.

29.73% of the respondents argued that the project was of a private nature and did not require robustness. Some respondents (13.14%) indicated that testing is inefficient and is not worth the required effort. Quite some respondents indicated that there was a lack of tooling available for the nature of their project, including graphical user interfaces (GUI) (13.51%) and firmware (5.41%). 8.11% indicated that the project included code of other authors and therefore did not require testing. This indicates that tests are linked to code ownership as was also observed in [8], claiming that asserts tend to be added to methods by developers with a higher ownership of that method.

8.11% of the respondents indicated that the tools available did not provide the required functionality and were unpractical in their usage. A stunning 21.62% indicated they were unable to test the project, as they were not familiar with software testing.

Measures taken to ensure code quality and -validity

20 (52.63%) project authors responded to this question and elaborated on the measures taken to ensure the quality and validity of their code. A large majority (75%) of the respondents

relies on manual testing to ensure code quality and -validity. 30% of the authors have a strict code review process and keep tabs on all code changes. 10% of the respondents turn on all compiler warnings and improve code quality by addressing all issues that arise during compile time. 5% of the respondents took no measures to ensure code quality or -validity.

Importance of code reviews

18 (47.37%) project authors responded to this question and reflected on the importance of code reviews with the lack of tests in their project. The majority of the authors agreed (77.78%) that code reviews are of greater importance in project that do not apply codified testing strategies, although many authors indicated that code reviews are important regardless of testing practices:

> I believe that code reviews and testing are complements, rather than substitutes; they generally find distinct (albeit overlapping) classes of errors.

Ensuring code stability

19 (50%) project authors responded to this question and reflected on their process of combating software regression. A large number of authors (36.84%) indicated they did not have any process in place to combat software regression. 26.32% relied on Pull Requests, whereas manual testing was applied by 21.05% of the authors (mostly authors combined both practices).

While only one developer (5.26%) indicated this by answering the question, many others backed him up via email conversations indicating that they had stopped the development of the project as changes would likely result in build failures or instability. Most of them indicated this could have been prevented by applying proper testing practices and would have done so from the beginning had they known this would have been the result of their development process.

Test burden

19 (50%) project authors responded to this question and reflected on their view on whether or not testing is a burden. A large majority (73.69%) of the respondents indicated testing is a burden to them, while simultaneously indicating this was the reason they did not apply codified testing practices in their project. The largest indicator for this burden (26.32%) appeared to be that testing practices are too time consuming. 10.53% of the respondents indicated that testing is too difficult. Another 10.53% of the authors indicated that there is a lack of available tooling to support their test practices. This once more stresses the need for proper test tooling. One developer (5.26%) indicated that fixing any of the existing bugs would likely introduce more bugs as the software was in a very unstable state, since they had not applied testing in their project.

Many developers indicated they saw the need for testing, but simply discarded testing practices due to the large burden. It should be noted that these developers indicated that they applied testing practices at their daily job as they were enforced:

> It's important when you need to ensure quality. I was making the emulator for fun, I didn't need to meet any quality standards, so I focused on what I thought it was more interesting. If I had to actually sell this or try to get people to use it, for example, I would have created a test framework from the start.

Repulsion against testing

19 (50%) project authors responded to this question and reflected on any distaste against testing practices. A large number of authors (73.69%) indicated they had no repulsion against testing and stated they would have applied testing practices if the project were to be of a different nature. 26.31% indicated that testing practices are never worth the effort spent, due of the reasons mentioned in the previous subsection. Moreover, a large number of authors (31.58%) indicated that testing practices are far too time consuming.

Notable results

Some developers shared their testing thoughts via email conversions outside of the questions. A few of them stated that projects should start adopting testing practices as soon as possible as they are unlikely to be adopted afterwards:

> It is a burden in the sense that I left it for so long before I started building tests. If I had done it from day one, it would have been almost no burden at all. Again, I convinced myself when I started the project that I do not need any tests because the library would be very limited in scope.
> ...
> However, once QEMU support was finished, the scope of libnfs grew in size, orders of magnitude, and finally covered a reasonably full set of the nfs protocol families. Unfortunately, as I never had any tests to start with, I never added any tests once the project started growing in scope.

As some developers stated, this may be related to the fact that not every project structure is suitable for testing purposes and projects may therefore require a large refactoring effort before they can adopt testing practices.

Moreover, quite some authors indicated that their testing approaches and the extent to which they apply testing in their project could be improved.

Finally, many authors - even those that have repulsion against testing - indicate that combating software regression is the most important gain from applying testing strategies throughout a project.

6.2 Large scale testing approach survey

The previous chapter showed insights into the usage of test tooling, but is lacking insight into the development processes in which they are used and the rationales behind them. By conducting a large-scale survey, we hope to gain insight into these development processes and the rationales behind the usage of test tooling. Participants for this survey were selected based on the following criterion:

- Must be the author of at least one project developed using a dynamically typed programming language in the scope of this survey (Ruby or JavaScript)
- Must be the author of at least one project developed using a statically typed programming language in the scope of this survey (C or Java)

This criterion was invented, as all participants are likely to have a good understanding of type systems and their impact. Moreover, they will have worked with multiple languages and are therefore likely more able to properly reflect on the pro's and con's of type systems and test tooling.

The following query was executed on the GHTorrent database to obtain the email-adresses of all GitHub authors matching the criterion:

```
SELECT u.login, u.name, u.email FROM users u
WHERE u.name IN (
SELECT u.name
FROM users u, projects p
WHERE p.owner_id = u.id
AND (p.language = "java" OR p.language = "c")
)
AND u.name IN (
SELECT u.name
FROM users u, projects p
WHERE p.owner_id = u.id
AND (p.language = "ruby" OR p.language = "javascript")
)
GROUP BY u.name
```

This query resulted 366.276 authors that matched the selection criterion. A total of 380 authors responded to the survey. Due to a high number of email bounces, exceeded daily outgoing email thresholds and invalid email-addresses, it is hard to estimate the number of authors that were contacted via email. Regardless, we expect that 5.000 - 10.000 authors were contacted.

6.2.1 Demography of participants

Participants were asked to rate their testing- and programming experience on a scale of 1-5 (1 indicating they have little experience, 5 indicating they have a large amount of experience). The results are shown in figures 6.2 & 6.1. The average programming experience was rated at 4.12 (median: 3), whereas the testing experience was rated somewhat lower with an average of 3.31 (median: 3).

Testing experience showed a strong relationship with programming experience (an increased testing experience was a good indicator for an increased amount of programming

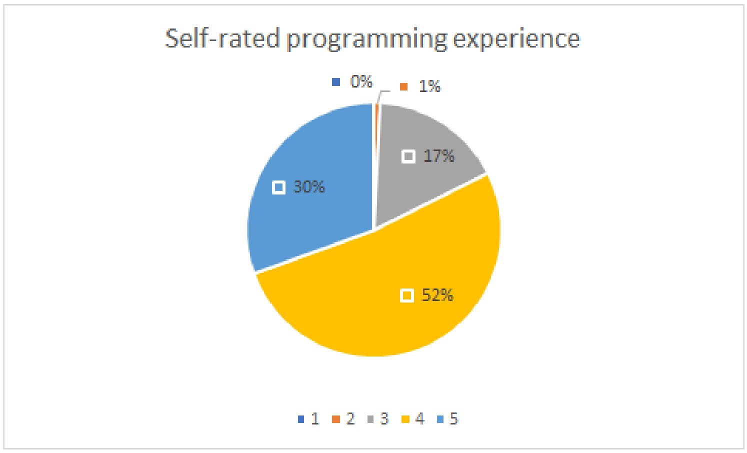


Figure 6.1: Distribution of self-rated programming experience as indicated by the survey respondents

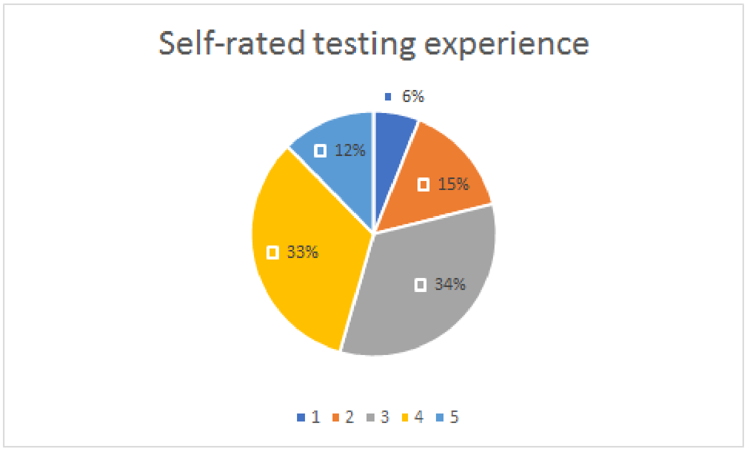


Figure 6.2: Distribution of self-rated testing experience as indicated by the survey respondents

experience), but programming experience did not appear to be related to testing experience. The majority of respondents had a significant amount of programming experience in terms of years (>8 %), followed by a large group of respondents that had between 2 and 5 years experience, as is shown in figure 6.3. The programming experience in terms of years showed a strong relationship with self-rated programming- and testing experience (an increase in years of experience would increase the self-rated experience in both cases). This shows that testing and programming skills are acquired over time.

Almost all respondents indicated they were familiar with JavaScript (352, 92.63%) and Java (332, 87.37%). 267 (70,26%) of the respondents were familiar with C and 155 (40.79%) respondents were familiar with Ruby. The distribution of familiarity with these languages in shown in figure 6.4. On average, respondents were familiar with 2.91 (me-

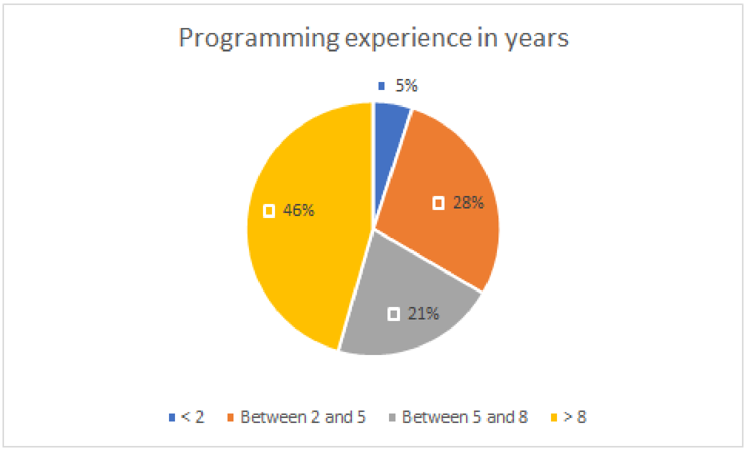


Figure 6.3: Distribution of programming experience in years as indicated by the survey respondents

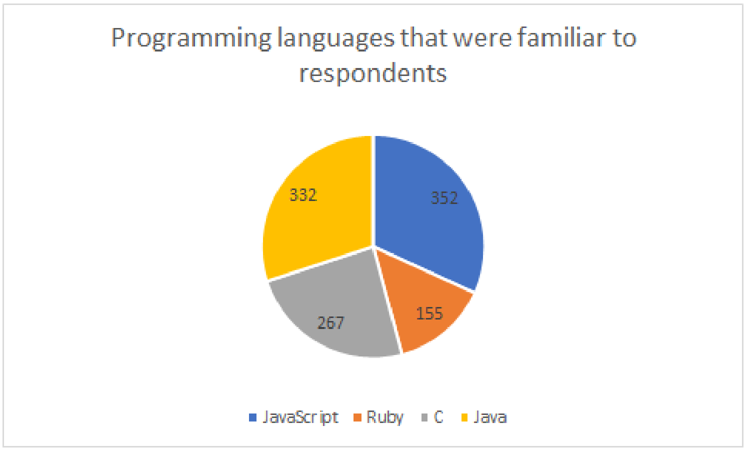


Figure 6.4: Distribution of familiarity with languages as indicated by the survey respondents

dian: 3) of the 4 selected programming languages (meaning that they had worked with the language on multiple occasions).

Most respondents indicated that they were primarily working with Java (38.95%), followed by JavaScript (37.63%), C (13.68%) and Ruby (10%), as is shown in figure 6.5. Both the average and median programming experience was the same for all programming languages. The average and median testing experience of Ruby developers appeared to be higher than those of other languages (average 3.68 vs. 3.31, median 4 vs. 3).

6.2.2 Impact of language typing on applied testing practices

All respondents were asked the following question: ‘Do you feel that writing tests for a dynamically typed programming language versus a statically typed one requires a different testing strategy? Why (not)?’. This question was not mandatory to answer, as not all de-

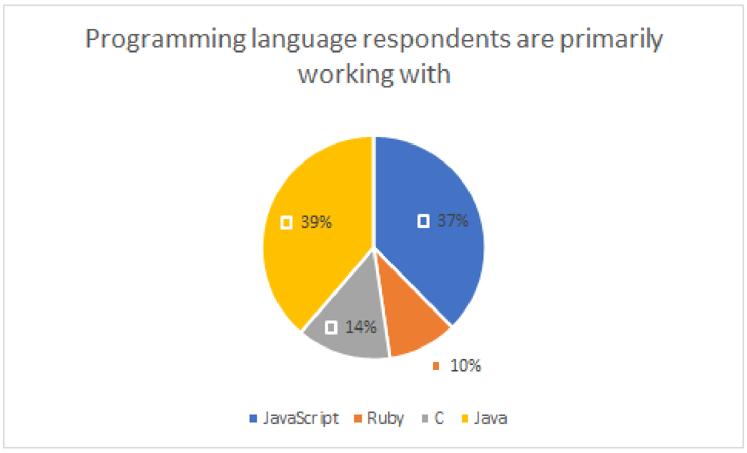


Figure 6.5: Distribution of primarily worked with programming languages as indicated by the survey respondents

Table 6.1: Results on whether or not typing influences testing practices, split on language level

Programming language	Has influence	Has no influence	Not sure
JavaScript	63.11%	33.01%	3.88%
Ruby	65.63%	34.37%	0%
C	57.14%	42.86%	0%
Java	60.98%	35.77%	3.25%
All	61.77%	35.50%	2.73%

velopers might be aware of type system impact. **Note**: as mentioned in the beginning of this subsection, all respondents were selected based on their programming experience as all of them had worked with both statically and dynamically typed languages. 293 respondents (77.11%) answered this question. The results are shown in table 6.1. The majority (61.77%) of the respondents that answered this question indicated that type systems influence their testing practices in the sense that they need to write additional tests and asserts to make up for the missing compiler type checks that are provided by default in statically typed programming languages. It appears that respondents that are primarily working with a dynamically typed language are more aware of type systems and their impact, as a larger percentage of these respondents indicated that type systems influence their testing practices.

Of the respondents that indicated that type systems influenced their testing practices, the following rationales were mentioned most often:

- 75.14% indicated that there was a need to write additional tests and asserts, as they needed to make up for the missing compiler type checks.

- 6.08% indicated that mocking practices were far easier and less time-consuming in dynamically typed languages, due to the flexibility the language provided.

- 2.21% indicated that the flexibility of a dynamically typed language allowed them to write higher level abstraction tests with greater ease.

Respondents that indicated that typing did *not* influence their testing practices, mentioned the following rationales:

- 63.46% indicated that testing strategies should validate input and output regardless of typing and practices should therefore be the same.

- 4.81% indicated that they did not write any tests related to type checking.

While not officially reflected via the survey, numerous respondents reached out to us via email and mentioned that black-box (i.e. high level integration- and system tests) testing practices can also ensure type checking:

> Generally we test functional and code coverage, which doesn't relate to inner struct of language (dynamic/static types, etc.)

For example, if a web application system test is able to submit a form and verify that this form was successfully stored in the database, it is likely that all modules that were touched are functioning well and no type errors occurred. It should be noted that this approach provides a false sense of validity as not all modules or code paths may have been touched during this system test.

The beliefs of whether or not typing influences testing practices did not appear to be related to testing experience. In other words, the amount of testing experience did not influence the respondents opinion on this topic.

6.2.3 Frequency of working with tests

All respondents were asked to indicate how frequently they work with test via the following mandatory question: 'How frequently do you write / modify tests?'. The distribution of the responses is visualized in figure 6.6. A significant number (31%) of the respondents indicated to work with tests on a monthly basis or even less frequent (never).

The frequency at which developers work with tests appears to have a strong relationship with the self-rated testing experience. An increased self-rated testing experience leads to an increase in the frequency at which the developer works with tests. This likely has to do with the fact that developers gain more testing experience when frequently working with tests.

As can be derived from table 6.2, there appears to be no relationship between the typing of a programming language and the frequency at which tests are worked with. Once more, Ruby developers stand out as 55.27% indicated that they were working with tests multiple times per day, which is significantly larger than the average (20.79%). C developers indicated they were working less frequently with tests as only 26.92% of them were working with tests on at least a daily basis, compared to the average of 42.89% of the respondents.

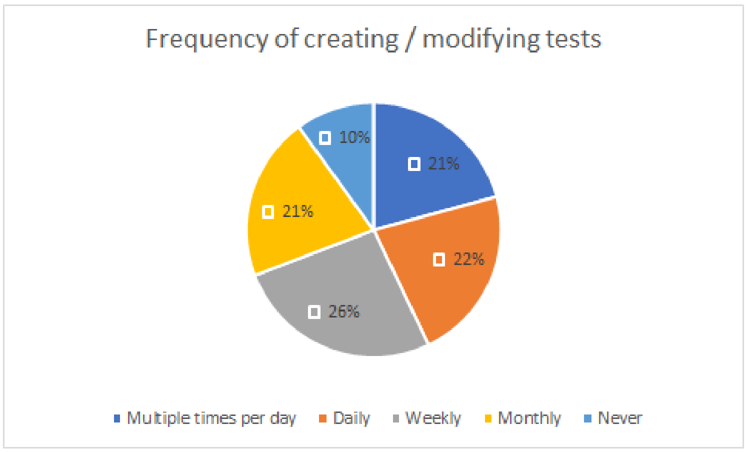


Figure 6.6: Distribution of the frequency of which developers work with tests

Table 6.2: Frequency of working with tests, split on language level

Frequency of working with tests	**JavaScript**	**Ruby**	**C**	**Java**
Multiple times per day	18.18%	55.26%	15.38%	16.33%
Daily	17.48%	18.42%	11.54%	31.29%
Weekly	24.48%	15.79%	28.85%	29.93%
Monthly	27.27%	10.53%	32.69%	12.93%
Never	12.59%	0%	11.54%	9.52%

6.2.4 Selection criteria for the adoption of test frameworks

All respondents were asked to indicate based on what criteria they pick a test framework to use for testing practices in their projects. This mandatory question included a set of pre-defined answers, including the following:

- Application domain of the system you are contributing to
- Amount of documentation / examples available regarding the testing tool
- Familiarity of the testing tool of all (active) contributors to the system
- Readability / syntax of tests written using the testing tool
- Ease of writing tests written using the testing tool
- Compatibility of the testing tool with existing tools
- Compatibility and integration with your IDE (e.g. Eclipse)

Respondents were able to select multiple answers and were also able to enter their own criteria, albeit it should be noted that only a handful entered their own criteria. No notable custom answers were observed. The distribution of the most important criteria is shown

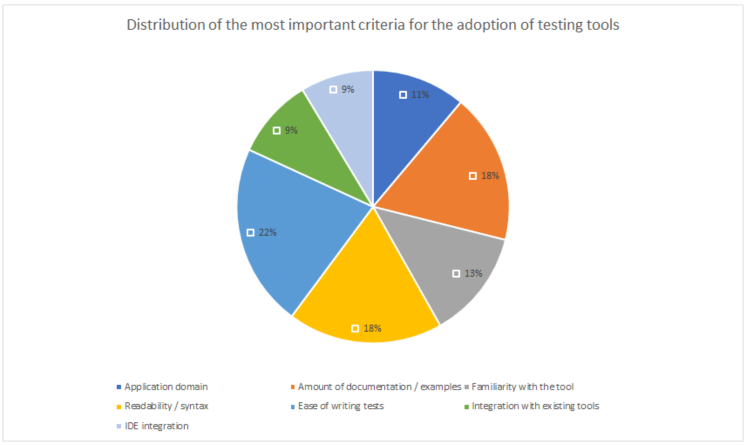


Figure 6.7: Distribution of the criteria developers use for the selection of test frameworks

Table 6.3: Criteria developers use for the selection of test frameworks, split on language level

Language	Domain	Documentation	Familiarity	Readability
JavaScript	37.76%	72.73%	42.66%	67.83%
Java	39.46%	63.95%	46.94%	69.39%
Ruby	39.47%	81.58%	65.79%	84.21%
C	57.69%	42.31%	51.92%	53.85%
All	41.32%	66.05%	47.90%	68.16%

Language	Ease of writing tests	Tool integration	IDE integration
JavaScript	78.32%	30.77%	18.18%
Java	80.95%	42.18%	53.74%
Ruby	94.74%	31.58%	10.53%
C	75.00%	30.77%	25.00%
All	80.53%	35.26%	32.11%

in figure 6.7. All results have also been summarized in table 6.3 and have been split on programming language level. The ease of writing tests (80.53%) is the most important criteria for the adoption of a test framework, followed by readability of tests (68.16%) and the amount of documentation on the framework (66.05%). This is in line with the responses we received via our interviews described earlier in this chapter. It indicates that developers are looking for a framework that can easily be set up (indicated by the need of documentation) and allows them to quickly write and modify clean, readable tests.

Developers working with statically typed languages were less likely to indicate that documentation was a criterion for the adoption of a test framework for them, although the reason for this cannot directly be explained. It may be related to the fact that we observed that there was less extensive documentation available for test frameworks aimed at statically typed languages when compared to dynamically typed languages. Developers might simply be used to the fact that there is less documentation available. **Note**: C developers complained about the lack of documentation in the interviews previously described, whereas they seem to value documentation less when compared to other languages based on the survey results. This may be related to the fact that the target audience was different (C developers working on large projects that do not test vs. developers working with both statically and dynamically typed languages). It may also be an indicator that the question was phrased incorrectly.

IDE integration seemed to be of bigger importance to statically typed languages than for dynamically typed languages. This may be related to the fact that developers working with dynamically typed languages appear to be working less with rich-feature IDE's and instead opt for light-weight code editors such as Sublime, Atom, Vim or Notepad++, as was observed in the 2017 JetBrains developer survey with $>$ 5000 respondents [1,2,3,4]. Since these developers are not using extensive IDE's, they are unlikely to select a framework based on its IDE integration support. **Note**: These code editor results for Ruby were not mentioned in the JetBrains survey.

Once more, Ruby developers stand out as they set clear and high expectations for testing frameworks compared to other languages for the following criteria:

- Amount of documentation / examples (81.58% vs 66.05%)
- Familiarity with the testing tool (65.79% vs 47.90%)
- Readability / syntax of tests (84.21% vs 68.16%)
- Ease of writing tests (94.74% vs 80.52%)

During our test tooling investigation in chapter 3, we observed that many testing practices (e.g. Behavior Driven Development (BDD) and test tooling (e.g. Cucumber) were first adopted in and / or created for Ruby and later ported to other programming languages. Combining these observations with the high expectations Ruby developers indicated to have for their test tooling, it seems that the testing community for Ruby is much more mature than those for the other languages within the scope of this thesis.

6.2.5 The burden of testing practices

Respondents were asked to reflect on their experience with testing practices and whether or not these practices were a burden to them via the following mandatory open question: 'Do you feel that testing is a burden to your development process? Why (not)?'. The majority (63.85%) indicated that testing was not a burden to their development process, whereas

[1] https://www.jetbrains.com/research/devecosystem-2017/ruby/
[2] https://www.jetbrains.com/research/devecosystem-2017/javascript/
[3] https://www.jetbrains.com/research/devecosystem-2017/java/
[4] https://www.jetbrains.com/research/devecosystem-2017/clang/

Table 6.4: Survey testing burden results, split on testing experience

Testing experience	Testing is a burden	Not a burden	Don't know
1	47.83%	43.48%	8.70%
2	49.12%	49.12%	1.75%
3	39.37%	59.06%	1..57%
4	23.02%	76.98%	0.00%
5	29.79%	68.09%	2.13%
All	34.83%	63.85%	1.58%

Table 6.5: Survey testing burden results, split on language level

Language	Testing is a burden	Not a burden	Don't know
JavaScript	34.97%	61.54%	3.50%
Ruby	26.32%	73.68%	0.00%
Java	34.01%	65.31%	0.68%
C	42.31%	57.69%	0.00%
All	34.83%	63.85%	1.58%

34.83% felt it was a burden to some extent. 1.58% of the respondents was not sure of their answer.

Developers with less self-rated testing experience showed a higher burden rate than those that had more testing experience, as indicated by table 6.4. This may in an indicator that more experienced developers know how to efficiently write and debug tests. It might also be the case that more experienced developers tend to be convinced of the need for testing, therefore valuing their testing practice efforts.

The results were also analyzed on a language level, as shown in table 6.5. There appears to be no relationship between the type system of a language and the testing burden. However, once more we observe that Ruby developers stand out in the sense that they do not observe as much of a burden as other languages. Moreover, C developers tend to feel much more of a burden compared to other languages. This is likely related to the lack of tooling observed in the previous chapters.

Developers that indicated testing was not a burden to them provided the following rationales:

- Testing ensures correctness and combats regression (77.89%).
- Testing is an investment that pays off and can save significant effort over time (25.14%).
- Testing ensures that refactoring existing code is safe (14.01%). This is backed by the refactoring activities described in [22].
- Testing is simply part of our development cycle and therefore not a burden (12.36%).
- Testing provides a form of code documentation (3.30%).

However, they also indicated that:

- It can be hard to add tests to a project of significant size that does not test at this time (3.78%).
- Not all code can easily be tested (3.78%).
- A (too) strict testing process (e.g. breaking builds when falling below a certain code coverage threshold) can hurt the agility and efficiency of the development team (2.06%).

On the other hand, respondents that did notice a burden from testing practices argued:

- Testing practices are too time consuming (45.34%).
- Refactoring existing tests when modifying source code takes too much effort (9.82%). This is likely related to the test smells that come up during the process of refactoring test code, observed in [30].
- It is hard to implement proper mocking practices (5.29%).
- UI testing can be unstable (flaky tests that sometimes fail or pass) (4.53%).
- Compilers should cover more test cases by default and reduce the time to write 'real' tests (3.78%).
- All tests should be inferred and generated by test tooling (0.76%).

It appears that both developers that feel and do not feel the burden of testing practices believe that testing practices take a significant amount of time and for the minority (34.83%) the effort is not worth it. There appears to be a lack of tooling for properly executing UI tests and mocking objects. Some developers argue that 'dull' tests like null checks should automatically be generated to save them time.

6.2.6 Co-existence of testing frameworks

Respondents were asked to reflect on why they were using multiple testing frameworks in case they were doing so. This question was on a non-mandatory open nature. 164 out of the 380(43.16%) chose to answer this question. The majority of users indicated they adopted multiple frameworks so that they could test their application form multiple angles (64.02%). A large number of respondents (45,73%) indicated that the other testing frameworks were lacking functionality. 10,98% of the developers indicate their project scope extents beyond one programming language and therefore required an increased number of testing frameworks to fully test their system. 6,10% of the respondents indicated they adopted another framework because of its syntax and the way of writing tests was more readable.

Developers seem to value the fact that they have a wide range of frameworks to choose from. Moreover, extend-ability and integration with other frameworks are valued:

They are bricks that build on top of one another, they're not independent.

Many tools deliberately only offer part of the solution to allow users to mix and match pieces they prefer (e.g., one could easily use Mocha instead of Jasmine).

6.2.7 Usage of test tooling

Respondents were asked to indicate what testing tools they were using the following mandatory question: 'What test tooling do you use in the system you are contributing to most?'. Unfortunately, we are unable to link this to the programming language of the project they were contributing most to, as many developers indicated all of the testing frameworks they were using or had ever used before (regardless of programming language). It appears that the questions was misinterpreted, which means that the results cannot be split on a language level. The results are shown in table 6.6.

Testing tooling	**Percentage used**
Junit	31.05%
Mocha	12.37%
Rspec	10.79%
Selenium	9.21%
Jasmine	8.95%
Mockito	6.32%
Custom framework	6.05%
Karma	5.26%
Jest	5.00%
Cucumber	4.21%
Chai	3.68%
TestNG	2.89%
Jenkins	2.63%
Capybara	2.63%
travisci	2.11%
googletests	2.11%
Protractor	1.84%
PhpUnit	1.84%
Spock	1.58%
unittest	1.58%
ScalaTest	1.32%
AssertJ	1.05%
nUnit	1.05%
Qunit	0.79%
Powermock	0.79%
Phantomjs	0.79%
spring-test	0.79%
Jmeter	0.79%

Xunit	0.79%
nosetests	0.79%
SoapUI	0.53%
Supertest	0.53%
Android studio	0.53%
ScalaCheck	0.53%
Gatling	0.53%
Roboelectric	0.53%
Appium	0.53%
xctest	0.53%
RestAssured	0.53%
Wiremock	0.53%
ava	0.53%
cppunit	0.53%
MiniTest	0.53%
Perl Test::More	0.53%
pytest	0.53%
Teamcity	0.53%
CasperJs	0.53%
Maven	0.53%
test::unit	0.53%
Nightwatchjs	0.53%
Catch	0.53%
BOOST	0.53%
SonarQube	0.53%
Debugger	0.26%
Jwalk	0.26%
A/B experimentation tool	0.26%
Json compare	0.26%
SBT	0.26%
Jgiven	0.26%
Jacoco	0.26%
Sinon	0.26%
Fluenlenium	0.26%
ctest	0.26%
Robotests	0.26%
spring-boot-starter-test	0.26%
QT-test	0.26%
ruby specs	0.26%
scalaspec	0.26%
homegrown	0.26%

Chrome dev tools	0.26%
Intellij	0.26%
Pycharm	0.26%
Cmocka	0.26%
Cwrap	0.26%
Frisby	0.26%
Hamcrest	0.26%
go.cd	0.26%
Enzyme	0.26%
FactoryGirl	0.26%
Vagrant	0.26%
Calabash	0.26%
Espresso	0.26%
CDIUnit	0.26%
UnitJS	0.26%
Docker	0.26%
clojure.test.check	0.26%
clojure.spec	0.26%
diff	0.26%
Node assert	0.26%
QTP	0.26%
arduinounit	0.26%
BusterJS	0.26%
MS test	0.26%
Visual studio	0.26%
ScalaUnit	0.26%
festassert	0.26%
Jmockit	0.26%
Specflow	0.26%
Rstudio	0.26%
pgtap	0.26%
GitLabCI	0.26%
Istanbul	0.26%
Groovy	0.26%
Runit	0.26%
Cunit	0.26%
simplecov	0.26%
vowsjs	0.26%
Cmake	0.26%
Bespoke	0.26%
Tape	0.26%

Cerberus	0.26%
Quicktest	0.26%
fj.test	0.26%
SourceMeter	0.26%
Arquillian	0.26%
Unity	0.26%
Packt	0.26%
Testem	0.26%
Ember	0.26%
ember-cli-mirage	0.26%
ember-cli-page-object	0.26%
mochitest	0.26%
istec Testcenter	0.26%
valgrind	0.26%
Gradle	0.26%
FindBugs	0.26%
AceUnit	0.26%
labjs	0.26%

Table 6.6: Test framework adoption rates, as indicated by the survey respondents

As the results cannot be split on language level, due to the nature of the responses, we are unable to properly compare the observed percentages against the results in chapter 5. However, we observe that the most widely adopted tools (Junit, Mocha, Rspec, Selenium Jasmine, Mockito, Karma), match results in chapter 5. This once more strengthens the results obtained using the mining approach, described in chapter 4. It is peculiar that the high usage of default assert functionality observed in chapter 5 of programming languages was not reflected in this survey question. Moreover, it should be noted that we cannot reflect on the extent to which they are used since the survey results cannot be split on language level. Finally, we note that a large number (6.05%) of respondents designed their own custom framework, once more indicating that existing tooling is lacking functionality developers need to properly test their software.

6.2.8 Tracking test quality

Respondents were asked to indicate how they measure and track test quality (if applicable) in the mandatory open question ‘If you are actively tracking test quality (e.g., with test coverage)? What tool(s) are you using?’. As is shown in table 6.7, 40.79% of all respondents is tracking test quality. Once more we observe that C and Ruby are outliers, as developers using C were less likely to track test quality and Ruby developers tend to track their test quality on a much higher basis. In the case of C, this is likely related to the lack of proper test tooling (you cannot track the quality of non-existing tests). In the case of Ruby, it is likely related to the fact that the Ruby community is very test-focused. The adoption rates

Table 6.7: Percentage of respondents tracking test quality, split on language level

Language	Percentage tracking test quality
JavaScript	34.27%
Ruby	60.53%
Java	46.94%
C	26.92%
All	40.79%

of test quality tools have been split on language level and can be observed in tables 6.8, 6.9, 6.10 and 6.11.

The nature of a programming language (static or dynamic) does not appear to be related to tracking test quality.

In the case of JavaScript, Istanbul is the clear winner when it comes to measuring test quality.

In the case of Ruby, Simplecov and Code climate are widely adopted tools for tracking test quality. Moreover, Ruby developers sometimes integrate them with CI tooling (e.g. Jenkins).

In the case of Java, SonarQube and Jacoco are widely adopted tools for tracking test quality. It is particular that JavaScript tools were mentioned for testing Java projects, which may be related to the fact that respondents were working with multiple programming languages in a single project.

In the case of C, Gcov is most often used to track test quality, albeit it is not widely used in practice.

6.2.9 Suggested tooling improvements

All respondents were asked to indicate what improvements they would like to see in available test tooling via the following non-mandatory open question: 'What improvements would you like to see in available test tooling to aid you better in your development cycle?' This question was answered by 155 respondents (40.79%).

Table 6.12 shows all suggestions that were backed by more than 1 respondent ($> 1\%$ support).

A large number of respondents (16.77%) indicated that frameworks should be able to automatically generate tests for them, i.e. in the same way that modern IDE's are able to provide suggestions (e.g. 'this variable might not be initialized'). Some respondents (6.45%) indicate that the generation of boiler template test code would also be sufficient for them.

Moreover, many respondents (10.97%) indicated their tests take too much time to run is hurting their development process efficiency. As was also observed previously in table 6.1, quite some respondents (10.32%) also indicate that mocking framework tooling should be improved.

Many respondents indicate that some frameworks can improve their syntax (9.03%) and readability (8.39%). It is worth a mention that quite a few respondents indicated that the

Table 6.8: Adoption rate of test quality tools for JavaScript

Test quality tool	**Adoption rate**
Istanbul	11.19%
Karma	2.80%
Jest	2.80%
Jenkins	2.10%
Coverage	2.10%
SonarQube	1.40%
Pitest	1.40%
NYC	1.40%
Jacoco	1.40%
Code Climate	1.40%
TeamCity	0.70%
Stryker	0.70%
Rspec	0.70%
Rcov	0.70%
PHPUnit	0.70%
Nosetest	0.70%
Mocha	0.70%
Lcov	0.70%
JSDOC	0.70%
Jasmine	0.70%
IDE	0.70%
Emma	0.70%
Ember	0.70%
Custom	0.70%
Coverity	0.70%
Coveralls	0.70%
Corbertura	0.70%
Blanket	0.70%

Table 6.9: Adoption rate of test quality tools for Ruby

Test quality tool	**Adoption rate**
Simplecov	31.58%
Code Climate	15.79%
Jenkins	7.89%
Coveralls	5.26%
Istanbul	2.63%
Karma	2.63%
Rcov	2.63%

Table 6.10: Adoption rate of test quality tools for Java

Test quality tool	**Adoption rate**
SonarQube	15.65%
Jacoco	10.20%
Corbertura	6.12%
IDE	4.76%
TeamCity	2.72%
Jenkins	2.72%
Codecov	2.04%
Findbugs	2.04%
Karma	1.36%
Emma	0.68%
Istanbul	0.68%
Codecov	0.68%
Code Climate	0.68%
Jest	0.68%
Scoverage	0.68%
Checkstyle	0.68%
Pmd	0.68%
Spock	0.68%
HP ALM	0.68%
Gcov	0.68%
Jasmine	0.68%
Clover	0.68%
Testcenter	0.68%

Table 6.11: Adoption rate of test quality tools for C

Test quality tool	**Adoption rate**
Gcov	7.69%
Coverity	3.85%
Gcovr	3.85%
Cdash	1.92%
Rebar3	1.92%
VSO	1.92%
Gitlab	1.92%
XC Test	1.92%
Googletest	1.92%

Table 6.12: Suggested test tooling improvements by the survey respondents

Suggestion improvement	**Backing rate**
Automatically generate tests	16.77%
Improve speed of test execution	10.97%
Simplified mocking	10.32%
Easier syntax for tests	9.03%
Improved readability	8.39%
Ease writing of tests	7.74%
Simpler setup	7.74%
Generate boiler template test code	6.45%
Improved static type checks	5.16%
Improved integration with other tools / dependencies	4.52%
Improved error display	3.87%
Better documentation with examples	3.87%
Automated bug detection	2.58%
Choose how to run code (concurrent, etc.)	2.58%
Visualize code analysis (where did tests fail?)	2.58%
Easier UI testing	2.58%
Improved code quality / coverage tools	2.58%
Profilers (e.g. load times, memory usage, cpu usage)	1.94%
Easier debugging	1.94%
Embedded programming test tooling	1.94%
Web-based online validation	1.29%
Real world tests	1.29%
Frameworks should be merged into one	1.29%
Extend CI features	1.29%
Automatically run tests after changes	1.29%
Improved mutation testing	1.29%
Smaller, extendable frameworks	1.29%
Better hints at where tests might have failed	1.29%

Ruby test framework RSpec sets an excellent example for these improvements in the field. A few respondents also indicated that RSpec is an excellent framework for writing tests with great ease and would like to see a similar process in other frameworks (7.74%).

Quite a few respondents (7.74%) indicate that the effort of setting up the testing architecture is too difficult and would like to see a simple out-of-the-box setup process. This may be related to the suggested improvement to improve integration with other tools and dependencies (4.52%).

Finally, quite a few respondents indicated that the visualization and feedback of failing and passing tests can be improved, including an improved error display (possibly including stack traces) (3.87%), visualizing test reports (2.58%) and possibly even providing suggestions where and why tests may have failed (1.29%).

It is worth a mention that some (1.29%) would like frameworks to be even smaller and more extendable, whereas others (1.29%) indicated that there are too many frameworks and they should simply be merged into one. It seems that there is no *one size fits all* approach that will please all developers.

6.3 Conclusion

After conducting interviews with developers that do not apply testing practices in their software projects, it became clear that the top reason for not testing a piece of software of the lack of time to do so. Instead, these developers rely on manual testing and strict code reviews to ensure the quality of their software. Developers that apply testing practices tend to pick their testing tools at the very beginning of the project and are unlikely to change their setup afterwards. Adopting a large number of testing frameworks is likely to introduce overlap in framework functionality and testing debt. However, developers feel that this trade-off is worth it as they are able to test more thorough (testing their application different angles).

Our survey results showed that a large majority of the respondents indicated that software projects developed using dynamically typed programming languages require additional tests to ensure correctness, strengthening the results observed in chapter 5. Testing tools are picked based on the following criteria: the ease of writing tests, readability of tests and the amount of documentation describing the tool. A large number of developers tracks their test quality via tooling.

Many developers feel that testing is a burden to them, mostly because testing practices are too time consuming. Moreover, refactoring existing tests when modifying source code takes too much effort, which may be related to the test smells observed in [30]. This burden was less so observed by developers that had more experience testing software, indicating that more experienced developers know how to efficiently write and debug tests. Developers working with C felt much more of a burden, likely related to the lack of tooling observed in chapter 3.

Most developers feel that this burden can partly be taken away by improving test tooling, mostly by automatically generating tests, generatingboiler template test code, improved speed and mocking support and improving the syntax and readability of tests.

The Ruby testing community seemed much more mature than those for the other languages within the scope of this thesis, whereas the C testing community appeared to be falling behind.

Chapter 7

Suggested tooling improvements

The previous chapters have helped us gain insight into testing practices from different perspectives. We can now use the obtained data to compile a list of suggested tooling improvements to answer RQ3. This chapter aims to combine all previous chapters and will deliver a list of possible test tooling improvements that can be used by test tool creators to help bring forward the software testing field.

7.1 Increase available test tooling for C

As was already noted in chapter 3, there is a clear lack of available test tooling for programming language C. Only a small part (unit testing) of the codified testing spectrum described in chapter 2 is covered with the available tooling. This claim is supported by the results in chapter 5, showing that a large number of projects do not apply testing techniques. Moreover, our claim is strengthened by the interview results described in chapter 6, in which we showed that many developers were complaining about the available test tooling. Many of these developers were forced to create their own test frameworks or decided to abandon testing strategies as the available tools did not support their testing needs.

Developers created their own (private) test frameworks for the following reasons:

- Lack of performance (speed of test execution)
- Frameworks lack functionality
- Frameworks are too complex
- Frameworks are too generic

These are pointers that should definitely be taken into account when building a new test framework to support the testing needs for these developers. Moreover, we strongly suggest to build tooling that covers new parts of the codified spectrum, e.g. integration tests and system tests as these are not yet supported. Finally, our survey in chapter 6 showed that developers would value profiling tools to gain insights into their application behavior (e.g. memory usage).

7.2 Mocking support

The survey in chapter 6 clearly indicated that mocking practices need to be improved. Improving mocking practices was among the top improvements mentioned by developers in this survey. Moreover, many of the developers argued that testing was a burden, as properly applying mocking practices was too difficult for them.

As mocking practices are not required for all testing purposes, we suggest to build dedicated mocking frameworks (or to improve existing ones) that ease the setup of mocks and limit the amount of boilerplate code that is required for them to set up. These frameworks should support integration with existing frameworks, as many developers indicated this was a requirement for them via the conducted survey described in chapter 6. This burden may be related to the 'General Fixture' test smell observed in [30], as developers may set up fixtures / mocks incorrectly. Test tooling could help developers overcome these smells by better aiding them in the creation of fixtures or mocks.

It should be noted that developers working with dynamically typed programming languages coped less with the mocking burden due to the flexibility these programming languages provide by default.

7.3 Clear documentation

During the compilation of the overview of available test tooling in chapter 3, we observed that documentation describing the usage and required setup steps for adopting a test tooling were rather limited and unclear in most cases, especially for tooling revolving around programming language C. The survey described in chapter 6 showed that many developers indicated that the amount of available documentation and the ease of setting up a framework within a software project is an important criterion for the adoption of test tooling. Moreover, quite some developers indicated they would like to see clearer and more extensive documentation (especially with example code).

While this is stating the obvious, we cannot help but stress the importance of clear documentation covering at least the bare essentials, including the framework setup and basic test code.

7.4 Improve readability of tests and framework syntax

Many of the survey respondents indicated that readability of tests is of great importance to them. This is backed by test smells described in [30], as many test smells are introduced by unclear test code. New approaches like Behavior Driven Development (BDD) and dedicated tools (e.g. Cucumber) aim to improve readability of tests, but not all available testing tools have taken measures to improve their readability yet. Many survey respondents indicated that they would like to write tests with a more user-friendly syntax that features better readability. Quite some respondents argued that Ruby testing framework RSpec is an excellent example, so it makes sense to adopt some of its features when modifying an existing or creating a new testing framework aimed at improving the readability of tests.

7.5 Automatic test generation

Quite some developers indicated that testing felt like a burden to them, most of which argued that writing and maintaining tests was too time consuming. Moreover, many developers indicated that test tooling should be able to automatically generate tests for them, i.e. in the same way that modern IDE's are able to provide suggestions (e.g. 'this variable might not be initialized'). Some respondents indicated that the generation of boiler template test code would also be sufficient for them. Both of these suggestions would speed up the testing process, therefore reducing some of the testing burden. This should allow developers to solely focus on applying proper testing practices through well written test scenarios, which should in turn increase the developers' confidence in their software projects.

7.6 Faster test runners

Many developers indicated that their development process is slowing down due to the large number of test cases and / or slow speed of execution of these tests. While this may be related to improper test setups (as indicated in [30]), many developers feel that test tooling lack the performance they need to run their tests. Moreover, many frameworks claim to be significantly faster than others (e.g. Riot [1]), indicating there is room for improvement in a lot of cases. Some viable solutions include only running tests that touch code that was modified and running tests in parallel.

7.7 Conclusion

It is hard to estimate the impact the improvements mentioned in this chapter will have on the speed and accuracy at which developers are able to write, modify and debug their tests. However, one cannot deny the fact that a large number of developers feels that testing is a burden. The suggested improvements will definitely reduce quite some of this burden and thereby hopefully make testing more convenient and joyful, which should in turn increase the rate at which software projects are tested.

[1] https://github.com/thumblemonks/riot

Chapter 8

Discussion

In this chapter, we will reflect on the observations made in all previous chapters and combine these to help determine the impact of type systems and test tooling on codified testing strategies. We will combine the results of all 3 different research questions to answer the main research question. This chapter is different from the 'conclusion' chapter, as it will derive new insights by combining different pieces of data, whereas the 'conclusion' chapter will merely underline the main contributions and results of the thesis. The next to sections will distinguish between the impact of type systems and the impact of test tooling on codified testing strategies.

8.1 Impact of type systems on codified testing strategies

Using our mining approach described in chapter 4, we obtained data on the testing approaches taken in software projects in chapter 5. Figures 5.3 & 5.4 show clear assert density differences when comparing statically and dynamically typed programming languages. This is once more strengthened by the Generalized Linear Model (GLM) (equation 5.1) with an R-squared fit of 1 - the highest possible fit - showing that type systems are the largest factor impacting the assert density of software projects.

Moreover, the systems appeared the be the largest factor impacting the test density of software projects, as shown in figures 5.5 & 5.6. This was strengthened by the Generalized Linear Model (GLM) (equation 5.2) with an R-squared fit of 1 - the highest possible fit.

As explained in chapter 2, the differences in assert- and test density can be explained by [29]. This study shows that runtime error detection is much less effective than assertion error detection. As dynamically typed programming languages miss out on compile time type checks, software projects working with these programming languages ensure their correctness via more extensive test and assert approaches. By doing so, developers tend to spend additional time and effort on testing their software to ensure the correctness of their software projects.

These results were backed by the survey described in chapter 6, showing that a majority of the developers feel that type systems impact testing approaches. Moreover, a large majority of the developers that believes that type systems influence testing approaches indicated that they have to write additional tests and asserts to make up for the missing compile type checks that are present in statically typed programming languages.

Finally, many developers feel that test tooling should provide more extensive type- and common checks (e.g. 'nill'-checks) automatically (without needing to write these themselves), as many of them feel that this makes testing feel like a burden to them. This once more indicates the importance of type systems and the extent to which they are valued by developers.

Based on these observations, we conclude that type systems have a large impact on codified testing strategies. It appears that more strict type systems reduce the need for software projects asserts and these project require less testing effort when compared to those written using less strict type systems. The soft typing system described in [7] seems like a promising solution, allowing developers to work with the great flexibility less strict type systems provide (therefore reducing the time spent writing software), while simultaneously benefiting from strict type checks that help reduce the amount of testing required to ensure the correctness of a piece of software.

8.2 Impact of test tooling on codified testing strategies

Using the overview of available test tooling described in chapter 3 and the codified testing spectrum described in chapter 2, we showed the distribution of the classification of available test tooling per programming languages in figures 3.1 & 3.2. It became clear that in the case of programming language C, much of the codified testing spectrum was not covered using available tooling. Interviews with developers (described in chapter 6) showed that developers resorted to not test their project or wrote their own testing tooling. The amount of available tooling appears to have significant impact on testing approaches, as figures 5.1 & 5.2 showed that developers using C were less likely to adopt testing strategies in their projects due to the lack of available test tooling. Moreover, test- and assert density were heavily impacted in the case of C, once more indicating that there is a need for good test tooling in order to properly test a software project.

Chapter 5 showed that developers tend to rely on multiple test tools for their testing process. This was strengthened by the interview and survey results shown in chapter 6. Our survey showed that developers pick the tools involved based on the ease of writing tests, readability of tests and the amount of documentation available, describing the framework. Moreover, this survey showed developers valued the fact that test tools are extendable and integrate with other tooling, so that they can pick the tools they see fit for their testing purposes. Developers adopt multiple test tools as they allow them to test their application from different angles, as they posses functionality that other tools lack or because their project is not written in just one programming language.

While much test tooling is available to choose from, many developers still feel that testing is a burden to them. A large number of developers feels that testing is cumbersome as writing and/or modifying tests is too time consuming. Moreover, they feel that part of the testing process should be automated and they should not have to write boilerplate test code. Moreover, test syntax and readability should be improved. The same applies for the amount of available documentation. Many developers indicated test tooling should be improved. Some even mentioned that the process is such a burden to them that they have not applied testing practices in their software project. We believe that overcoming these issues and implementing the suggestions mentioned in chapter 7 will better aid developers in their testing process, reduce the time required to properly test a project and thereby increase the rate at which software projects are tested.

Based on these observations, we conclude that test tooling has a significant impact on codified testing strategies as they are enablers for the testing process. By improving the available test tooling, test tooling can better aid developers in their testing process.

Chapter 9

Threats to validity

Unfortunately, as our work was confined in its time frame, so was the scope of this thesis. We have tried our very best to make our analysis as sound as possible, by cross-checking our results using multiple methodologies (data mining, interviews and a survey), but because of the exploratory nature of this research there are some threats to its validity nonetheless. This chapter will provide an overview of these threats and reflect on their possible impact on the results of this thesis, which helps place our results in their proper context.

9.1 Limited coverage of programming languages

Due to time constraints, the programming language scope was limited to 2 dynamically and 2 statically typed programming languages. In chapter 5, we observe great differences when it comes to the impact of type systems and test tooling. These claims are later on backed by our interviews and survey, described in chapter 6. Even so, we also observed that in the case of C testing is much less common. As this programming language makes up for half of the statically typed programming languages, this poses a plausible threat to the validity of our study. We therefore recommend to re-use our open-sourced mining tool to extend our work to broaden the scope of programming languages to confirm our findings.

9.2 Possibility of excluding test tooling

To the best of our knowledge, very few literature is available that describes available test tooling. Therefore, the approach taken in chapter 3 is of an exploratory nature, in which we may have missed out on available test tooling. While the available tooling described in chapter 3 and our obtained data in chapter 5 seems to match the obtained data via interviews and our survey in chapter 6, this should still be considered a threat to the validity of our work. By excluding test tooling in our mining approach, present testing strategies implemented using these tools would be excluded and therefore not be counted as testing practices. This would in turn bias the results described in chapter 5.

9.3 Subjective classification of test tooling

As we observed in chapter 3 - which was later backed by developers in our interviews and survey described in chapter 6 - documentation on test tooling was often limited. In the case that frameworks, their available documentation or example pages on the internet did not describe which part of the codified testing spectrum was covered using this framework, we classified the test tool ourselves based on the nature and syntax of the framework. By doing so, we introduced a possible subjective bias into our results that poses a threat to the validity of our research.

9.4 Limited validity check on mining approach

As the list of test tools described in chapter 3 used in our mining approach described in chapter 4 grew rather large, manually verifying the results of our mining approach was rather time consuming. As our work was confined in its time frame, we only verified the results for 10 Java projects (table 4.3) and 10 Ruby projects (table 4.4). While the precision and recall appears to be good, it should be noted that this was a rather small sample that may simply worked well with our mining approach. Moreover, the mining approach was not checked for its validity against programming languages JavaScript and C. While the results appear to be in line with the 2017 JetBrains developer survey [1,2,3,4] and the results by our interviews and survey, this still poses for a threat to the validity of our work. We therefore recommend to verify the validity of our mining approach once more when broadening the scope of programming languages, to ensure the correctness of our work.

9.5 Possible self-selection of survey and interview respondents

While we targeted our interview and survey respondents based on their actions (using many frameworks, not testing and working with both dynamically and statically typed programming languages), it may be the case that the respondents were self-selecting in the sense that developers that have greater affinity with testing their software were more likely to participate in our study as the topic was of greater interest to them. Therefore, it may be the case that the testing burden described in chapter 6 is even bigger than we observed, as the affinity with testing is likely to be related with the extent to which they feel the burden of testing. In this case, non-participating developers might have less affinity with testing because they feel much more of a testing burden. Moreover, via similar reasoning the frequency of working with tests may also be lower than we observed. It is hard to estimate the impact on our work by this possible self-selecting participant process, but it poses a treat to our work nonetheless.

[1]https://www.jetbrains.com/research/devecosystem-2017/ruby/

[2]https://www.jetbrains.com/research/devecosystem-2017/javascript/

[3]https://www.jetbrains.com/research/devecosystem-2017/java/

[4]https://www.jetbrains.com/research/devecosystem-2017/clang/

9.6 Analyzing open-source software projects only

It should be noted that GitHub was used as our primary source for our mining approach described in chapter 4. The open-source nature of these projects allowed us to conduct a large scale analysis on testing practices, but simultaneously introduced bias into our work as open-source projects may have a different nature from those developed for personal or commercial use. This assumption is strengthened by the results of our survey described in chapter 6, as quite some developers indicated they saw the need for testing, but simply discarded testing practices due to the large burden. These developers indicated that they applied testing practices at their daily job as they were enforced.

Therefore, the results of our work only reflect testing practices applied in open-source projects, meaning that they do not reflect testing practices applied in personal or commercial software projects. We suggest to conduct additional research investigating the differences in testing practices applied in open-source, personal and commercial software projects, so that our results can be put into their proper context.

9.7 Conclusion

While we applied various methodologies to make our analysis as sound as possible, the nature of our work is exploratory and is bound to introduce some threats to its validity. We have tried to minimize the impact of these threats by matching our results against related work, the JetBrains 2017 developer survey and by cross-checking our mining results with those of our interviews and conducted survey. Due to time constraints, the programming language scope was limited. Moreover, our work is only a reflection of testing practices applied in open-source projects. We suggest to broaden the scope of our work using our open-sourced mining tool and to verify the validity of the tool once more to ensure the correctness of our work.

Chapter 10

Conclusion

This chapter serves the purpose of concluding the thesis. It will reflect on our research objectives and our main research question. Moreover, some parts of the thesis included contributions to the field that were not directly part of the research objectives, which will therefore be highlighted separately. Finally, possible future research is laid out, so that one may extend the scope of our research and reduce some of the threats to validity, mentioned in chapter 9.

10.1 Research objectives

Our main research question 'How are codified testing practices impacted by type systems and test tooling?' was divided into three separate research questions. We will first answer these three separate questions, after which we will reflect on our main research question.

10.1.1 RQ1: What are state of the art codified testing strategies?

Chapter 2 showed that there are many different codified testing strategies that revolve around testing software systems from different angles. Ideally, black- and whitebox testing approaches should be used to complement each-other to increase confidence in the correctness of a piece of software. The codified testing spectrum consists of unit-, integration and system testing. Our exploratory work described in chapter 3, classified and listed all available test tooling for programming languages JavaScript, Java, Ruby and C. To the best of our knowledge, our work is the first of its kind to list available test tooling.

By classifying all available tooling we were able to point out that most available tooling aids in the process of unit testing, followed by integration testing, mocking purposes and system tests. Available tooling tended focus slightly more on UI testing in the case of JavaScript, whereas we identified a clear lack of available test tooling for C.

Based on the mining approach described in chapter 4, we obtained the following list of most popular test tooling per programming language:

- C: Default assert, Ctest, Check and AceUnit
- Java: JUnit, default assert, Mockito, HamCrest and SpringFramework

- Ruby: RSpec, MiniTest, Capybara, Test::Unit, factory_girl, WebMock, Mocha and Shoulda
- JavaScript: NodeJS' default assert, Phantom JS, Chai, Sinon, Cypress, Tape, QUnit, UnitJS and Should.js

These results are in line with those observed in the 2017 JetBrains developer survey [1] in the case of Ruby, but are different from the ones observed from JavaScript [2]. This is possibly related to the fact that subset of JavaScript developers participating in the JetBrains survey may not reflect the JavaScript testing community as a whole.

10.1.2 RQ2: What differences in codified testing strategies can we observe in software projects using different type systems?

Our mining approach described in chapter 4 helped gather testing data of 71.628 open-source projects. Violin plots (figures 5.3 and 5.4) and a Generalized Linear Model (GLM) (equation 5.1) show that assert density is largely impacted by type systems, as dynamically typed programming languages included a larger number of asserts when compared to statically typed languages. The same applied for test density, albeit to a lesser extent. This is likely related to the fact that these programming languages are missing out on the safety net that types as was explained in chapter 2. The obtained results are in line with the results obtained in [4].

Chapter 5 showed that the rate of testing and number of test tools used for projects using C is much lower, strengthening the lack of available test tooling observed in chapter 3.

Unit testing appears to be much more significant for statically typed languages when compared to dynamically typed languages. Mocking, System testing and UI testing practices are much more common in dynamically typed languages than in statically typed languages. This might be related to the fact that statically typed languages provide a safety net by catching a lot of trivial implementation faults at compile time, as explained in chapter 2.

10.1.3 RQ3: How do developers reflect on the relationships of testing practices, type systems and involved test tooling?

In order to gain insights into the perspectives of developers on testing approaches, targeted interviews and a large-scale survey were conducted. Interviews with developers (described in chapter 6) showed that developers resorted to not test their project or to writing their own testing tooling should available tooling not suit their needs. The amount of available tooling appears to have significant impact on testing approaches, as figures 5.1 & 5.2 showed that developers using C were less likely to adopt testing strategies in their projects due to the lack of available test tooling. Moreover, test- and assert density were heavily impacted in the case of C, once more indicating that there is a need for good test tooling in order to properly test a software project.

[1] https://www.jetbrains.com/research/devecosystem-2017/ruby/
[2] https://www.jetbrains.com/research/devecosystem-2017/javascript/

A large majority of survey respondents indicated that software projects developed using dynamically typed programming languages require additional tests to ensure correctness, strengthening the results observed in chapter 5. Testing tools are picked based on the following criteria: the ease of writing tests, readability of tests and the amount of documentation describing the tool.

A large number of developers feels that testing is a burden to them, mostly because testing practices are cumbersome and too time consuming. In chapter 7, we propose a lists possible improvements that should ease the testing process. Some of the improvements include: automatic test generation, reducing the need for writing boilerplate code, improved readability and syntax of tests and more extensive documentation on test tooling.

10.1.4 Impact of type systems and test tooling on codified testing strategies

Our work shows that both type systems and available test tooling appear to have a significant impact on the codified testing strategies that are adopted in the testing process of software projects. The flexibility less strict type systems provide is valued by developers as it allows them to write software with much more freedom, while simultaneously reducing the time spent writing the software [26]. On the other hand, our work shows that these less strict type systems increase the time and effort required for properly testing the project to ensure its validity and code quality. The soft type system described in [7] seems like a promising solution, allowing developers to work with the great flexibility less strict type systems provide (therefore reducing the time spent writing software), while simultaneously benefiting from strict type checks that help reduce the amount of testing required to ensure the correctness of a piece of software. However, this soft type system requires research when it comes to its feasibility.

Moreover, we have shown that test tooling is an enabler for codified testing strategies. Many developers choose to write their own test tooling or abandon testing processes in the case that available tooling falls short. Quite some developers indicate that test tooling should significantly be improved to reduce the burden due to which many developers choose to not adopt testing practices. In order to bring forward the field of testing, we have compiled a list of improvements that should address these testing issues and thereby help test tooling better aid developers in their testing processes.

10.2 Contributions

Due to the exploratory nature of this work, some contributions to the software testing research field were made that fall outside of the direct scope of our main research question. These contributions are shortly mentioned in the next paragraphs.

10.2.1 Extendable open-source test data mining tool

Much of our work is based on our mining tool described in chapter 4. As the programming language scope is limited, we recommend to broaden the scope in future research over time.

With this possible extension on our work, we have designed the tool in such a way that it is easily extendable. The tool has been open-sourced and is freely available at [3].

10.2.2 Overview of available test tooling

To the best of our knowledge, no up to date test tooling overview is available. While limited to only 4 programming languages, we have compiled a list of all available test tooling and classified the tools based on existing literature so that developers may easily pick the tools as they see fit. The overview per programming language is described in chapter 3. Moreover, the figures in chapter 3 show the most widely adopted test tooling.

10.2.3 Suggested test tooling improvements

Based on our various methods, we have combined our obtained data into a list of suggested test tooling improvements that are laid out in chapter 7. We high recommend taking these suggestions into account when extending existing or building new test tools, as many developers indicated that testing is a serious burden to them.

10.3 Future research

Unfortunately, our work was confined within its time frame and we therefore lay out several suggestions for future research in the next paragraphs.

10.3.1 Testing practices outside of open-source projects

As indicated in chapter 9, our work focused on open-source projects only. As developers indicated they were enforced to adopt certain testing strategies at their employer, we suggest to research the testing practices adopted outside of open-source project, as this is an indication that these practices may be different from those observed in our work. Due to copyright and confidentiality it seems improbable to get access to software projects employed by corporations. However, developers seemed very willing to share their views and experiences via interviews and surveys, which may pose for a viable substitute that may grant insight into testing practices outside of open-source projects.

10.3.2 Broadening the programming language scope of our work

Our work was limited to only 4 programming languages and while our results show a clear relation between type systems, available test tooling and codified testing strategies, additional programming languages should be taken into account to confirm our findings. The effort required is not significant as our mining tool described in chapter 4 can easily be extended for this purpose.

[3]https://github.com/Pvanhesteren/GitHubTestFrameworkAnalysis

10.3.3 Feasibility of soft typing

Soft typing, described in [7], seems like a promising solution to reduce testing effort by applying compile-time type checks of statically typed programming languages, while simultaneously allowing developers to work with the great flexibility in dynamically typed programming languages. However, as described in chapter 2, there are large technical *and* cultural differences between the statically and dynamically programming language communities [21]. The feasibility of introducing soft typing should therefore be researched.

Bibliography

[1] Alexander Aiken, Edward L Wimmers, and TK Lakshman. Soft typing with conditional types. In *Proceedings of the 21st ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 163–173. ACM, 1994.

[2] Victor R Basili and Richard W Selby. Comparing the effectiveness of software testing strategies. *IEEE transactions on software engineering*, (12):1278–1296, 1987.

[3] Kent Beck. *Test-driven development: by example*. Addison-Wesley Professional, 2003.

[4] Moritz Beller, Georgios Gousios, and Andy Zaidman. Oops, my tests broke the build: An explorative analysis of travis ci with github. In *Proceedings of the 14th International Conference on Mining Software Repositories*, pages 356–367. IEEE Press, 2017.

[5] Moritz Beller, Georgios Gousios, and Andy Zaidman. Travistorrent: Synthesizing travis ci and github for full-stack research on continuous integration. In *Proceedings of the 14th International Conference on Mining Software Repositories*, pages 447–450. IEEE Press, 2017.

[6] Antonia Bertolino. Software testing research: Achievements, challenges, dreams. In *2007 Future of Software Engineering*, pages 85–103. IEEE Computer Society, 2007.

[7] Robert Cartwright and Mike Fagan. Soft typing. In *ACM SIGPLAN Notices*, volume 26, pages 278–292. ACM, 1991.

[8] Casey Casalnuovo, Prem Devanbu, Abilio Oliveira, Vladimir Filkov, and Baishakhi Ray. Assert use in github projects. In *Proceedings of the 37th International Conference on Software Engineering-Volume 1*, pages 755–766. IEEE Press, 2015.

[9] Nick Diakopoulos and Stephen Cass. Interactive: The top programming languages 2016. http://spectrum.ieee.org/static/interactive-the-top-programming-languages-2016/, January 2016. Last accessed on Nov 29, 2016.

[10] Georgios Gousios. The ghtorrent dataset and tool suite. In *Proceedings of the 10th Working Conference on Mining Software Repositories*, MSR '13, pages 233–236, Piscataway, NJ, USA, 2013. IEEE Press.

[11] Mats Grindal, Jeff Offutt, and Sten F Andler. Combination testing strategies: a survey. *Software Testing, Verification and Reliability*, 15(3):167–199, 2005.

[12] Dick Hamlet and Ross Taylor. Partition testing does not inspire confidence. *IEEE Transactions on Software Engineering*, 16(12):1402, 1990.

[13] William E Howden. Methodology for the generation of program test data. *IEEE Transactions on computers*, 100(5):554–560, 1975.

[14] William E Howden. *Symbolic Testing: Design Techniques, Costs and Effectiveness*. National Technical Information Service, 1977.

[15] Eirini Kalliamvakou, Georgios Gousios, Kelly Blincoe, Leif Singer, Daniel M German, and Daniela Damian. The promises and perils of mining github. In *Proceedings of the 11th working conference on mining software repositories*, pages 92–101. ACM, 2014.

[16] Mohd Ehmer Khan, Farmeena Khan, et al. A comparative study of white box, black box and grey box testing techniques. *International Journal of Advanced Computer Sciences and Applications*, 3(6):12–1, 2012.

[17] Gunnar Kudrjavets, Nachiappan Nagappan, and Thomas Ball. Assessing the relationship between software assertions and faults: An empirical investigation. In *Software Reliability Engineering, 2006. ISSRE'06. 17th International Symposium on*, pages 204–212. IEEE, 2006.

[18] Christophe Leys, Christophe Ley, Olivier Klein, Philippe Bernard, and Laurent Licata. Detecting outliers: Do not use standard deviation around the mean, use absolute deviation around the median. *Journal of Experimental Social Psychology*, 49(4):764–766, 2013.

[19] Lu Luo. Software testing techniques. *Institute for software research international Carnegie mellon university Pittsburgh, PA*, 15232(1-19):19, 2001.

[20] Sonali Mathur and Shaily Malik. Advancements in the v-model. *International Journal of Computer Applications*, 1(12):29–34, 2010.

[21] Erik Meijer and Peter Drayton. Static typing where possible, dynamic typing when needed: The end of the cold war between programming languages. OOPSLA, 2004.

[22] Tom Mens, Arie Van Deursen, et al. Refactoring: Emerging trends and open problems. In *Proceedings First International Workshop on REFactoring: Achievements, Challenges, Effects (REFACE)*, 2003.

[23] Jeff Miller. Short report: Reaction time analysis with outlier exclusion: Bias varies with sample size. *The quarterly journal of experimental psychology*, 43(4):907–912, 1991.

[24] Vijay N Nair, DA James, Willa K Ehrlich, and J Zevallos. A statistical assessment of some software testing strategies and application of experimental design techniques. *Statistica Sinica*, pages 165–184, 1998.

[25] Simeon C. Ntafos. A comparison of some structural testing strategies. *IEEE Transactions on software engineering*, 14(6):868, 1988.

[26] Linda Dailey Paulson. Developers shift to dynamic programming languages. *Computer*, 40(2), 2007.

[27] Stuart C Reid. An empirical analysis of equivalence partitioning, boundary value analysis and random testing. In *Software Metrics Symposium, 1997. Proceedings., Fourth International*, pages 64–73. IEEE, 1997.

[28] Paul Rook. Controlling software projects. *Software Engineering Journal*, 1(1):7–16, 1986.

[29] Kavir Shrestha and Matthew J Rutherford. An empirical evaluation of assertions as oracles. In *Software Testing, Verification and Validation (ICST), 2011 IEEE Fourth International Conference on*, pages 110–119. IEEE, 2011.

[30] Arie Van Deursen, Leon Moonen, Alex van den Bergh, and Gerard Kok. Refactoring test code. In *Proceedings of the 2nd international conference on extreme programming and flexible processes in software engineering (XP2001)*, pages 92–95, 2001.

[31] Xusheng Xiao, Suresh Thummalapenta, and Tao Xie. Advances on improving automation in developer testing. *Advances in Computers*, 85:165–212, 2012.