Schaapi

Early detection of breaking changes based on API usage

J.S. Abrahams, G. Andreadis,
C.C. Boone, F.W. Dekker

TU Delft

Schaapi

Early detection of breaking changes based on API usage

by

J.S. Abrahams, G. Andreadis,
C.C. Boone, F.W. Dekker

to obtain the degree of Bachelor of Science
at Delft University of Technology

Project duration:	April 2018 – July 2018	
Thesis committee:	Dr. M. F. Aniche,	TU Delft, client
	Dr. A. Katsifodimos,	TU Delft, coach
	Dr. H. Wang,	TU Delft, bachelor coordinator
	Ir. O. W. Visser,	TU Delft, bachelor coordinator

An electronic version of this thesis is available at `http://repository.tudelft.nl/`.

TU Delft

Contents

Preface

This thesis documents our final project of the Bachelor of Computer Science at Delft University of Technology. Over the past 10 weeks, we have built a tool for early detection of breaking changes based on API usage, called Schaapi. The name of our tool is an acronym for Safe CHAnges of APIs, but also contains the Dutch word for sheep ("schaap"). Even though the tool is still in a prototype stage, it has shown promising results in its ability to find real-world library usages and to generate tests for them. To the authors' knowledge, the tool is novel in its aim and approach, extending the state of the art with the automated detection of semantic breaking changes.

This project would not have been possible without the help of a number of people. We would like to take this opportunity to extend our gratitude to dr. Sebastian Erdweg (TU Delft) for his advice on bytecode analysis and the Soot framework. We would like to thank Yassin Hajaj (Tobania, Belgium) for his insights on Java bytecode instrumentation. A sincere thank you to the group of students at the TU Delft SERG research group gathering for their valuable feedback on our first project proposal. We also thank the core Soot developers for the insightful discussion we had with them on a pull request we submitted to their repository. Furthermore, we thank Anand Sawant (TU Delft) for pointing us in the direction of frequent itemset mining. We would like to thank the Software Improvement Group for their helpful feedback on the code quality of our software.

Our sincere thanks to our faculty coach, dr. Asterios Katsifodimos (TU Delft), for his insightful feedback on our work, especially on the data mining challenges we faced. Last but not least, we would like to express our gratitude to our supervisor and client, dr. Maurício Aniche (TU Delft), for his continued support and feedback throughout the project.

J.S. Abrahams, G. Andreadis,
C.C. Boone, F.W. Dekker

Delft, June 2018

Summary

Library developers are often unaware of how their library is used exactly in practice. When a library developer changes the internals of a library, this may unintentionally affect or even break the working of the library users' code. While it is possible to detect when a syntactic breaking change occurs, it is not as easy to detect semantic breaking changes, where the implicit contract of a functionality changes, sometimes unbeknownst to the library developer. Because library users rarely test the behaviour they expect of the library, neither the library developer nor the library user will be aware of the new behaviour.

As a library developer, you want to be able to see how a change in your library will affect your users before a new version of the library is deployed. More specifically, you want to gain insight into how users use the library, and want to see if and how changes affect users. This will allow you to determine whether the new version of the library is backwards compatible. Finally, after deploying the breaking changes, you want to notify the affected users of the changes and of a solution to the issue.

Schaapi, a tool for early detection of breaking changes based on API usages, addresses these needs. It mines public repositories for projects using a given library, analyses their usage of the API of that library, and generates tests that capture this behaviour. Finally, it offers a continuous integration service that automatically executes these tests against new versions of the library and warns developers of any potentially breaking changes in functionality. The tool has also been validated against real-world data to demonstrate its performance in realistic usage scenarios and to answer a selection of related research questions.

List of Figures

List of Tables

1 Introduction

Reusing software modules that other people have written is a core part of the software engineering process. This reuse enables software developers to focus their efforts on functionality that is unique to the scenario they are addressing. However, because the library user typically has no control over the code in external dependencies, library users need to adapt if the library changes.

As a library developer, deploying a new version of the library brings with it the risk that it will break the users' code in unforeseen ways. This concern may hold back users from upgrading, even when the new version fixes critical issues. As a library user, the decision to upgrade is often not an easy one to take: If one does not upgrade, end users may suffer from issues in the library code persisting; and if one does upgrade, end users may suffer from issues in the integration with the new library.

Existing tools check only for *syntactic* compliance of new library versions. This already helps in finding a number of types of incompatibilities, such as parameters being added to functions. However, this is also detectable at compile time, even when ignoring the context that single library usages are in. A tool for detection of *semantic* incompatibilities could make library developers aware of such breaking changes in advance. Even more useful would be if the tool could detect how popular the affected part of the library is, to give library developers an indication of how many library users would actually be affected.

We envision a tool that detects breaking changes of libraries based on real-world API usage. In this thesis, we have designed, implemented, and validated such a tool, called Schaapi. Schaapi consists of a number of smaller components, arranged into two pipelines: the *mining* pipeline and the *validation* pipeline.

In the *mining* pipeline, we first mine open-source repositories that use a certain library version. We then build graph-based models from their program structures to understand how the library is being used. These library usages are then mined for common patterns to generate tests that capture the usage of the library.

In the *validation* pipeline, the generated tests are run against each new library revision as a continuous integration service. Library developers can then be informed directly on the effects of their proposed changes.

We have written a reference implementation of Schaapi that runs on the JVM. It downloads Java projects that use a specific library version from GitHub, compiles these projects into an intermediate representation, detects patterns that are common to multiple user projects, turns these patterns into valid bytecode, and finally uses an evolutionary algorithm to automatically generate tests for the bytecode.

Schaapi also features an implementation of the validation pipeline. This implementation functions as a web service that automatically runs when a pull request is made on the GitHub repository of the library. It runs the previously generated tests against the updated version of the library and leaves a report on the test results at the pull request.

We also set out to validate our reference implementation. In particular, we looked at the amount of patterns found, the lengths of these patterns, the influence of changing the minimum required support of detected patterns, and finally whether Schaapi is actually able to detect breaking changes when these are inserted into a library.

1.1. Related Work

We first investigate existing work related to the aims of Schaapi. This can be split into two parts: mining of *Application Programming Interface (API)* usage patterns, and detecting breaking changes within APIs.

1.1.1. Mining API Usage Patterns

The *MAPO* tool (Mining API usage from Open source repositories) by Xie [23] aims to mine frequent API usage patterns and disclose these to the API developer. They aim to mine projects, produce call sequences, and find frequent sequences within this set of call sequences. Although we aim to also generalise this to include more complex constructs in our usage patterns, there are overlaps with our approach. One thing of note is that the tool analyses source code directly using the *PMD* [1] tool, meaning that they do not require the code to be compilable. For mining frequent sequences they use the *BIDE* algorithm by Wang and Han [20]. We provide our own analysis of the BIDE algorithm in Subsection 2.1.4.2. It should be noted that MAPO does not generate tests for these patterns; it simply displays them as-is to the user.

Authors Acharya et al. develop a framework which automatically extracts patterns of API usage from source code [1]. Similar to MAPO, the framework constructs traces of library calls and extracts frequent sequence calls. Also, just like MAPO, it does not use these patterns to generate tests. However, it operates on partial orders instead of consecutive sequences, meaning that the sets statements in frequent partial orders can have many other statements between them, in reality.

Uddin et al. mine the temporal usage of API patterns based on version control differences [19]. This means that they look at how developers introduce new API usages over time, analysing what library usages are added at each new revision. Although they attack a different problem, their design of data processing stages can serve as an inspiration to our system design.

To the knowledge of the authors, none of these three related projects has a publicly available software artifact. This means that we can not reuse existing code bases for many aspects of this project, and instead have to rebuild them or look elsewhere for partial implementations.

1.1.2. Breaking Changes

Software libraries often change over time. These changes can bring new features and resolve issues, but they can also introduce new flaws or break compatibility with previous versions of the library, to the detriment of the users of the library. A 2013 study shows that faulty and change-prone APIs can degrade the quality of Android applications relying on them, ultimately adversely impacting the success of API users' applications [11].

An analysis of 317 real-life Java libraries in 2017 shows that roughly 15% of API changes break syntactic compatibility with previous versions, and that 2.5% of clients or library users are impacted by these changes [22]. In addition to this, it has been shown that, without compatibility, library users face increased risk and cost when upgrading their dependencies [15].

Along the same lines, Raemaekers et al. conclude that roughly a third of all releases introduce at least one syntactic breaking change, both during minor and major releases [15]. Although these findings are based on a less recent snapshot of the *Maven Central Repository* (dated July 2011), they do indicate that breaking changes are to an extent still commonplace. It should be noted, however, that they focus on syntactic compatibility. They do this through use of the *Clirr* tool[2], which can be used to detect breaking changes such as the removal of methods, classes, or fields from the interface of a library. It should also be noted that we uncovered other tools which can be used to compare library versions, outlined below:

- *Java API Compliance Checker* [3], a tool which is used for checking backwards compatibility of both binary and source code.
- *japicmp* [4], a tool which can perfom an in-depth analysis of the difference between two jars, for instance by checking if private methods have been added or removed.
- *SigTest Tool* [5], a tool which, given two versions of the same API, can verify that all members are present, reports on new members, and can check the specified behaviour of each member.

1.2. Outline

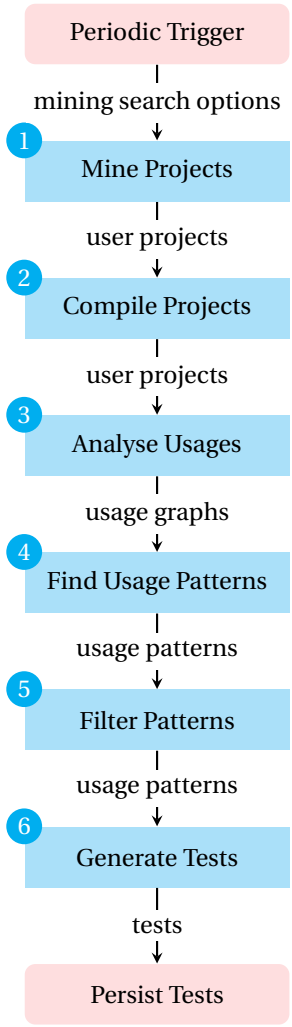
In this thesis, we first outline our approach in chapter 2. We then elaborate on our design, implementation, and quality assurance strategies in chapter 3. In chapter 4, we validate the functionality of our tool in a number of experiment setups and analyse its performance in real-world scenarios. Finally, we close this thesis with a discussion of our system and with concluding remarks in chapters 5 and 6.

[1] `http://pmd.sourceforge.net`, last accessed June 25, 2018.
[2] `http://clirr.sourceforge.net`, last accessed June 25, 2018.
[3] `https://lvc.github.io/japi-compliance-checker`, last accessed June 25, 2018.
[4] `https://github.com/siom79/japicmp`, last accessed June 25, 2018.
[5] `http://www.oracle.com/technetwork/java/javame/javamobile/sigtest-137088.html`, last accessed June 25, 2018.

2

Approach

Schaapi is a tool for early detection of semantic breaking changes in software libraries. We split this detection into two phases: a periodic *mining* step, establishing how users currently use the library, and a continuous *validation* step, testing whether a new version of the library breaks existing usage patterns. Each phase is designed as a pipeline of components, each component solving a part of the problem in a sequential fashion. In this chapter, we first describe our high-level approach and then elaborate more on each part of the process.



Mines version control or library distribution platforms (such as GitHub or the Maven Repository) for projects using a library version.

Compiles user projects, producing compiled output, if needed.

Creates library-usage graphs of user projects.

Finds common library-usage patterns across all given usage graphs.

Filters library-usage patterns, matching certain criteria of validity.

Generates regression tests for library-usage patterns.

Figure 2.1: Visualisation of the mining pipeline.

2.1. Mining Pipeline

A first step in identifying breaking changes is the gathering of real-world library usages and the generation of tests describing frequent patterns in these usages. The *mining* pipeline should answer this question, as visualised in figure 2.1.

2.1.1. Mine Projects

This component is responsible for gathering a set of software projects which use a given library. The process of mining should be reliable, meaning that mined projects actually depend on the desired version of a library, and should be reproducible to a certain extent. To this end, we explored several options for project mining, which are discussed below.

Maven Repository One option is to use the *Maven Repository* [1]. It has the functionality to query which Maven projects use a given Maven library. However, to our knowledge, this functionality is not accessible by a public API. This means that it is very difficult to use this functionality, and the only option we could find was to build a web crawler for retrieving projects. Such a web crawler would likely be very brittle and highly dependent on the website layout, which could change at any time. Therefore, we opted not to take this approach.

Maven Central Repository The *Maven Central Repository* [2] has an API[3] which can be used to search repositories based on certain attributes such as tags or checksums of projects. However, it does not offer the ability to search for projects which depend on a given library such as offered by the *Maven Repository* site. Searching on projects with the "guava" tag, for instance, gives only 2 results, as opposed to the roughly 15 000 results given on Maven Central when searching for projects which have guava as a dependency. This indicates that tags are rather inaccurate, making them ill-suited for a reliable mining process.

RepoDriller Another option we found was *RepoDriller* by Aniche [2], which is a software tool that aims to help researchers mine software repositories on GitHub. However, it has a heavy emphasis on the versioning aspect of GitHub, focusing on the history of code revisions of repositories. It does not support searching GitHub for repositories using a certain library, making it unfit for our purposes.

GitHub Search API *GitHub* is becoming one of the most important sources of software artifacts, with over 50 million git repositories being hosted on the site[4], with about a third of projects being public [9]. This indicates that GitHub is a popular tool amongst developers making it ideal for mining purposes. It also provides a search API[5] to traverse these repositories. Whilst it does not have the functionality to search repositories based on library usage, string matching within code bases can be performed to find library usages. Better still, build tools such as *Maven* [6] often have a configuration file, making it possible to check whether a project has a dependency on a given library through a textual search of said configuration file.

In addition to this, it is also possible to filter repositories by the language they use and when they have last been updated, limiting the scope of the search. We wish to focus mainly on active software projects, as a community of active developers is often a good indicator that a software project is being utilised [9]. To this end, we find that how recently a project was pushed to is a good metric for determining how active a project is [9]. Unfortunately, the amount of allowed requests to the API is limited, and the data is not curated [7], making this method of gathering data unreliable at times.

We eventually chose to implement this option. Despite its limitations, the simplicity of implementing such a miner and the fact that it operates within the same environment that we wish to deploy the tool in– namely GitHub–have been the main deciding factors for using this API.

[1] `https://mvnrepository.com`, last accessed June 25, 2018.
[2] `http://search.maven.org`, last accessed June 25, 2018.
[3] `http://search.maven.org/#api`, last accessed June 25, 2018.
[4] `https://blog.github.com/2017-04-10-celebrating-nine-years-of-github-with-an-anniversary-sale/`, last accessed June 25, 2018.
[5] `https://developer.github.com/v3/search`, last accessed June 25, 2018.
[6] `https://maven.apache.org`, last accessed June 25, 2018.

2.1.2. Compile Projects

At this stage, the mined projects need to be processed to be ready for usage analysis. If the project is present in source format, it may need to be compiled to a binary format. For Maven projects, the information necessary to be able to compile the project is already included in the build configuration file of the project. If the project already is in a compiled format (e.g. a JAR), its contents only need to be inspected to know which files to analyse later.

2.1.3. Analyse Usages

Once a sample set of projects using the library has been collected, a way of analysing their library usages is needed. To this end, we seek a way to model the code in user projects. Pezze and Young give an introduction to the modelling of control in software using graphs in their work on software testing and analysis [14, §5]. They describe an *intraprocedural control flow graph*, in which nodes represent parts of the code and directed edges between nodes represent the possibility that execution transitions from one to the other. We have adopted this model in our analysis, combined with a model found in a related publication surrounding the *JADET* project.

JADET The JADET tool shares our aim of modelling API usages [21]. It also models such usages, except that it does so using so-called *Object Usage Models*. Temporal properties can be extracted from these models, meaning it is possible to derive in what order methods are called.

JADET builds models only for the following objects that can occur as library usages:

- Objects that are created using the *new* keyword
- Parameters of methods
- Return values of methods
- Exceptions

Our approach is inspired by the JADET tool. However, instead of constructing control flow graphs for single objects, we construct control flow graphs for methods in user projects and filter all library usages. This results in a library-usage graph. This allows us to not only analyse the interaction between method calls on a single object, but also method calls across multiple library objects and their interaction.

2.1.3.1. Representations of Source Code

Regarding the actual generation of control flow graphs, choosing the abstraction on which to work here is critical: How the instructions of the program are modelled significantly influences the process of mining library usages. The goal is to build a graph of software instructions, ideally containing only elements using the library (or enabling those usages).

We considered plain text, abstract syntax tree, bytecode, and intermediate representations. Our analysis of each is outlined below.

Plain Text Assuming that the gathered library projects include their source code, one could do textual searches for names from the library and collect surrounding lines of code. While this first attempt could definitely yield some correct results, it would likely also produce a number of false positives as the naming of functions and classes does not have to be unique. Recognising duplicate patterns would have to be limited to exact (or approximate) string matches. This is not compatible with cases where variables are named differently (as they commonly are). Moreover, building a control flow graph from plain text would require the parsing of the syntax of the language.

Abstract Syntax Tree It seems clear that a higher level of abstraction is needed. The Abstract Syntax Tree (AST) representation could fit this description: It is a static, parsed tree representation of the source code. While looking for library usages would still require a string search of nodes in the AST, code between projects can now be compared more easily based on their structure. However, there are still some issues with this approach. First, shadowing and similar edge cases make it more difficult to find out which variable is being used. Second, it is not possible to find the type of a variable or to find which overload of a method is being used without building a custom data structure next to the AST because the AST does not keep track of which variables are in scope. Third, ASTs are specific to a particular version of a language, which means that a program written to interpret the AST of a Java 7 program may not be able to understand the AST of a Java 8

program. Therefore, the AST approach would result in a less maintainable program. Finally, there is only a relatively small body of literature and software artifacts using ASTs for the generation of control flow graphs.

Bytecode These issues could be worked around by performing additional processing steps after parsing the code into an AST. However, these additional processing steps are already found in compilers, which may indicate that compiling the code may be a more appropriate solution. As our target platform is the Java Virtual Machine (JVM), bytecode could be considered its own data structure. Libraries such as ASM [6] and Soot [10] are able to read bytecode into an internal graph data structure which can then be analysed. It should be noted that the JADET tool also uses bytecode analysis for its construction of method models [21]. See figure 2.2 for an example of how this analysis could look.

A potential downside of this approach is the need for compilation of user projects, which is both time-intensive and carries security risks. More importantly, however, bytecode operates on a very low level, deconstructing method calls into sequences of stack pushes and retrievals, amongst others. When analysing library usage, however, the registers and memory locations are not relevant, and deviations between different library users are often acceptable. Analysis on the level of bytecode would therefore make the comparison of different pieces of code significantly more difficult, and a solution that is in between ASTs and raw bytecode seems to be more desirable.

Intermediate Representation One solution in between these previous solutions is an Intermediate Representation (IR), which contains information on the types of variables and links back instances to their classes. The Soot framework offers Jimple as such an IR. It has a limited instruction set (similar to the Java bytecode specification, although less limited in scope), but binds occurrences of the same variable. We consider this level of abstraction to be more suitable to our needs. Therefore, we have opted to use the Soot framework, with Jimple as the central source code representation.

2.1.3.2. Collection Scope

Another issue that needs to be taken into consideration is whether we generate control flow graphs based only on the bodies of methods within the user project or based on the entire application. The latter would require considering all possible chains of method calls within the user project and in turn all calls to the library within those chains. We opted for considering only method calls within the body of a method, as considering the entire application would result in much larger and potentially unmanageable graphs. It also negates the need to analyse the behaviour of the user project. It should be noted, however, that methods are often small, and often call other methods which may in turn make calls to the library. Therefore, this approach may miss some patterns which may be commonplace, but are split across multiple methods.

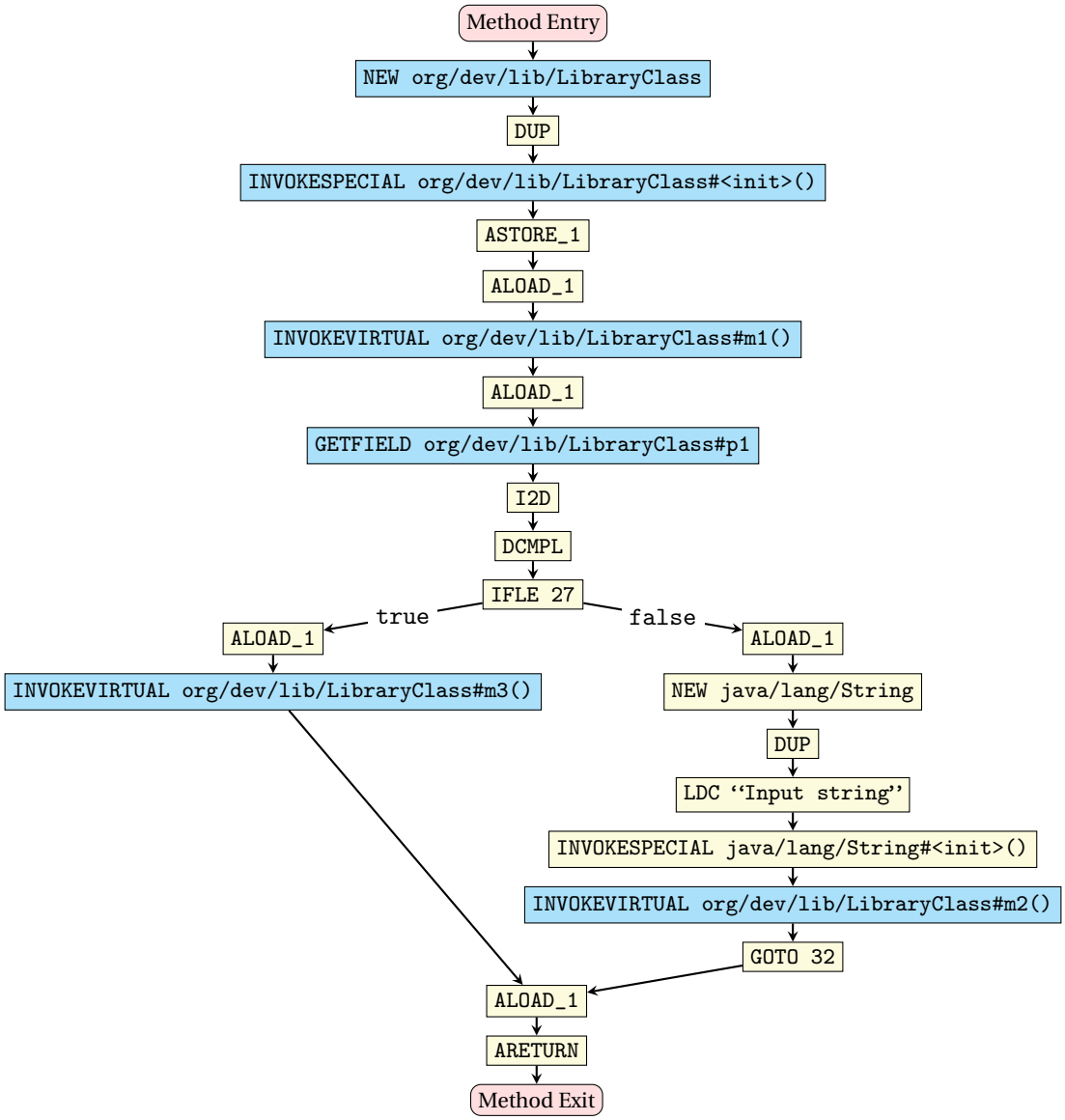
```java
import org.dev.lib.LibraryClass;

public class UserClass {
    public LibraryClass foo() {
        LibraryClass libraryObject = new LibraryClass();

        if (libraryObject.m1() > 2) {
            libraryObject.m2(new String("Input string"));
        } else {
            libraryObject.m3();
        }

        return libraryObject;
    }
}
```

(a) A source code example of a user method using library functionality.



(b) The corresponding bytecode Control Flow Graph. Library usages are highlighted with a blue background.

Figure 2.2: A control flow graph for a user method using library functionality.

2.1.4. Find Usage Patterns

The eventual goal is to generate test cases based on common library usages. To this end, we wish to find frequent sequences within the generated library-usage graphs. When searching for frequent sequences, we consider two options. We can either mine the generated library-usage graphs directly for frequent sequences of vertices as shown in figure 2.3, or first convert the graph into the set of possible paths and then find frequent sequences in this set of paths, as shown in figure 2.4. The nodes in these figures represent statements that perform calls to the library.

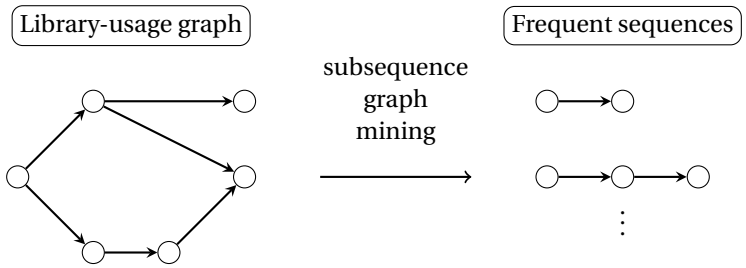


Figure 2.3: The process of mining patterns in graphs.

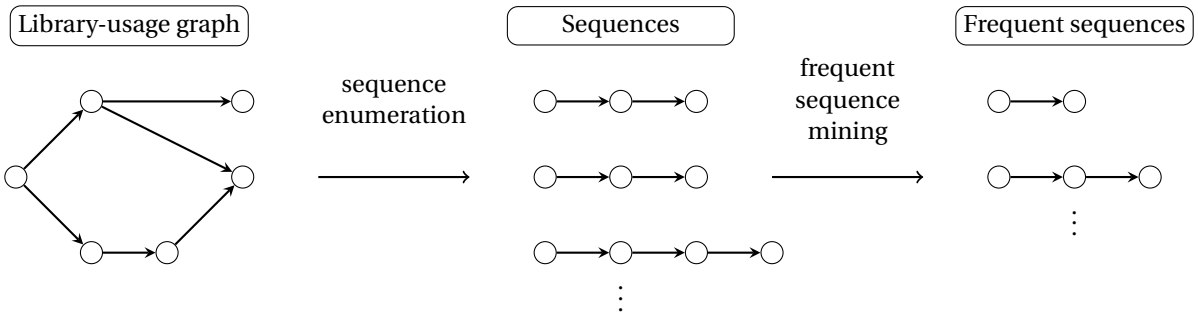


Figure 2.4: The process of mining patterns in sequences.

When analysing these options we considered computational and space complexities, their accuracy, and to a lesser extent their ease of implementation. We report our findings below.

2.1.4.1. Graph Mining

Frequent subgraph mining means directly mining the directed graph for frequently occurring sequences of nodes, as visualised in figure 2.3.

TreeMiner The *TreeMiner* algorithm by Zaki aims to find frequent subtrees in a forest [24]. Its efficiency is derived from the fact that candidate patterns are never generated twice. It does this by using a pattern pruning technique, by making sure all subtrees of a tree are frequent before adding it to the candidate set of frequent trees, and by using a prefix tree data structure for fast support counting. For a detailed explanation of the algorithm we refer the reader to [24].

A key problem with this approach, however, is that it generates subtrees, not sequences. In isolation, this might not seem to be a problem, and might even make it easier to deal with branching statements. However, to generate tests later down the line we wish to use common sequences of statements. Therefore, we would still have to enumerate over every single produced subgraph to generate the sequences of statements for test generation purposes.

It should be noted that early on we decided to switch to sequence mining, which means that we have less background research on subgraph mining. The main motivation was that it seems more logical and straight-forward implementation-wise to mine a set of sequences of statements as opposed to mining graphs. Another motivating factor, as stated before, is that we would still have to enumerate the generated subgraphs afterwards, meaning that mining subtrees directly might be more computationally expensive.

2.1.4.2. Sequence Mining

Sequence mining means mining a set of sequences for frequently occurring (sub)sequences, as visualised in figure 2.4. This carries less computational cost than subgraph mining, but does require enumerating over the graph and extracting all possible sequences first.

Most of the surveyed algorithms started with finding the most commonly occurring nodes in a graph. To achieve this, one first conducts a single pass of the data, counting the occurrence of each node. Only nodes that occur often are then used during the pattern-detecting algorithms, with all other nodes being ignored.

Note that in each of the following algorithms, we add the additional constraint that subsequences must be consecutive, even if not explicitly stated by the authors. This is done to prevent the generation of patterns which are in reality spread over large portions of the code, and to significantly reduce their number.

Table 2.1: Comparison of detected patterns between different sequence mining algorithms.

(a) An artificial database of sequences.

Sequence
`ABCD`
`BCD`
`ACD`

(b) Patterns produced by the sequence mining algorithms.

Algorithm	Generated Patterns
PrefixSpan	`A`, `B`, `C`, `D`, `BC`, `CD`, `BCD`
SPAM	`A`, `B`, `C`, `D`, `BC`, `CD`
CCSpan, BIDE	`A`, `CD`, `BCD`

Elements `A`-`D` represent four different statements. The sequence database in table 2.1a represents three methods consisting of the listed statement sequences. In table 2.1b, the algorithms described in section 2.1.4.2 are virtually applied in turn to the database.

PrefixSpan One such algorithm is the *PrefixSpan* algorithm by Pei et al. [13].

Traditionally, an *a priori-based* approach is used for mining frequent sets, or, in this case, sequences. This approach builds larger and larger sequences based on frequently occurring items. Each of these sequences is considered a candidate (i.e. not definitive) sequences, until it is verified that they actually occur in the dataset. Generating all these intermediate candidate sequences, many of which are potentially false positives which will be discarded, is computationally expensive.

The PrefixSpan algorithm improves upon the a priori-based approach by not continually generating all possible candidate sequences. In fact, it reduces the search space during each iteration of the algorithm by searching only suffixes of sequences which all contain the same prefix, reducing the amount of needed comparisons.

SPAM Due to performance concerns we also considered algorithms which are guided by heuristics, which might be less accurate as they only return some subsequences, but in theory can provide considerable speedups. One such algorithm is the *Sequential PAttern Mining (SPAM)* algorithm by Ayres et al. [3]. The algorithm uses a vertical bitmap representation of the input and intermediate sequences for fast candidate generation.

Ayres et al. claim that SPAM outperforms the PrefixSpan algorithm speed-wise by an order of magnitude on large datasets. They claim that the bitmap representation, used for efficient counting, is the main contributing factor to the gain in speed.

CCSpan A problem with the PrefixSpan algorithm is that it not only finds frequent sequences, but also outputs all possible subsequences of the found frequent sequences. We did not find this to be useful for our use case, where we simply want to find common chains of method calls to a library. If the longer sequence captures the same behaviour as the subsequences and is equally popular, then generating tests for the smaller subsequences has two drawbacks: First, it unnecessarily increases the size of the test suite, and second, it misses the context that the omitted statements of the larger sequence form. We therefore opted to use the *CCSpan* algorithm by Zhang et al. [25]. The CCSpan algorithm generates only *closed* frequent sequences. If a sequence is popular, the subsequences contained within are not given as output, except in the situation where these subsequences are found to be more common than the sequence(s) that contain them.

BIDE The *BIDE* algorithm by Wang and Han is another sequence mining algorithm that finds frequent sequences without the use of candidate generation [20]. It uses a novel sequence closure checking scheme called BI-Directional Extension. Like CCSpan, BIDE has the advantage that it generates only frequent *closed*

sequences, meaning that subsequences are not generated unless they are more common than the sequences containing them. The authors of BIDE also claim that it is more efficient than the PrefixSpan algorithm in terms of runtime. However, due to its similarity to the CCSpan algorithm with regards to the expected output and due to time constraints, we opted not to implement it. Future research could address this and compare its performance to that of the CCSpan.

2.1.5. Filter Patterns

The pattern detector may have identified frequent patterns that are trivial or invalid and thus will not lead to useful tests being generated. The pipeline therefore contains a filtering step that removes invalid patterns. Patterns could be filtered because they are too short, contain empty or infinite loops, or because they are simply invalid. While the interface is meant to remove invalid patterns, implementations could try to repair invalid patterns where possible to prevent otherwise useful patterns from being thrown away.

2.1.6. Generate Tests

The goal of the test generation step is to generate regression tests that fail when a new library version "breaks" old user behaviour. Our approach here is to first convert the gathered patterns into runnable code and then automatically generate tests for them. From each pattern we generate a function, which takes as input the unbound variables of that pattern. This is then passed to a test generator which chooses suitable input values for the functions and forms a test oracle from their observed behaviour.

2.1.6.1. Bytecode Generation

As in the library usage analysis step, two main frameworks are relevant here. *ASM* can be considered one of the main bytecode analysis and manipulation frameworks [6]. However, because it operates directly on Java bytecode, generation of *working* programs is challenging, especially while maintaining stack integrity.

This is where *Soot* [10] brings significant benefits to the table with its *Jimple* IR. It significantly reduces the complexity of bytecode generation, allowing us to generate entire *statements* rather than just single bytecode *instructions*. Furthermore, as library usage graphs consist of Jimple *statements* (see section 2.1.3.1), it is easier and less error-prone to use the same framework that was used to generate said statements to convert patterns to Java bytecode than to use a different framework to convert it to another (intermediate) representation before converting that to Java bytecode. Given the above, we opted to use the Soot framework to convert the generated sequences into Java bytecode.

2.1.6.2. Test Generation

It has been stated that creating regression tests is one of the most expensive activities in software development [18]. The act of writing unit tests themselves has also been described as difficult and time-consuming [12]. To this end, several tools have been created over the years to automatically generate tests.

In literature, we find that a heavy emphasis is placed on regression testing. Software often changes over time, and it is important to verify that functionality is not unintentionally broken when changes are made. Automated testing tools usually generate tests based on the current behaviour of software, therefore making them ideal for regression testing. That is, when changes are made, it is easy to run these tests and verify that old functionality has been not been broken. At the very least, developers should be made aware that functionality has been broken, ideally making it easier to make an informed decision on how to proceed.

Randoop One such tool is the *Randoop* tool created by Pacheco and Ernst [12]. This tool generates tests based on a codebase and a supplied API contract. It generates tests using *feedback-directed random testing*. This means that method chains are constructed incrementally, with method calls being selected at random and arguments constructed from previous sequences. Sequences that adhere to the rules of the supplied contract are regarded as regression tests, whereas tests that violate the contract are output to the user. A major problem with this approach is that it requires a specification of the desired behaviour in the form of an API contract, but because Schaapi does not have access to such a contract it would have to automatically generate one. We therefore opted not to use this approach to avoid unnecessary complexity.

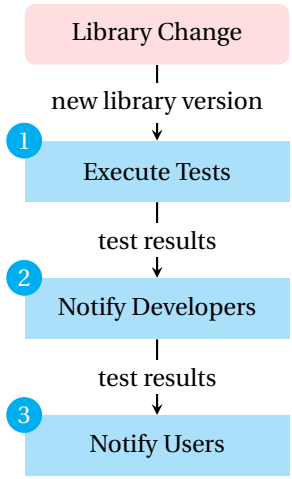
TestFul Another tool is the *TestFul* tool by Baresi et al. [4], which functions at both the method and class level. The authors of the tool state that it improves upon other testing methods by taking into account the internal states of objects and that it explores state configurations and reuses states to exercise different behaviours. An internal empirical analysis of the tool by Baresi et al. shows that in general, TestFul achieved

higher coverage than Randoop within the same timeframe. In addition to this, it was mentioned that compared to Randoop, TestFul creates tests that are far smaller and better suited for regression testing.

Orstra The *Orstra* tool by Xie [23] supports testing by running test suites and adding extra assertions based on expected behaviour. However, as we are not so much interested in augmenting (automatically) generated tests as much as generating tests from the ground up we decided not to use this tool. However, future work could investigate how this tool could enhance the generated tests.

EvoSuite Our final choice was the *EvoSuite* framework by Fraser and Arcuri [8], which seems to be the most stable and effective tool for the job. It generates test suites in an evolutionary fashion, with coverage of the class under test as maximisation target. One significant advantage is that no formal specification is required for the generation of tests. EvoSuite automatically generates full test suites for given code, and only requires the dependencies used in the given code. Furthermore, EvoSuite is also able to handle any type of object, as it will recursively search for dependencies of objects under test. Another advantage of the EvoSuite framework is that it aims to reduce the amount of assertions produced. This can help increase the maintainability of tests whilst at the same time improving their performance.

2.2. Validation Pipeline



Runs generated tests against a new library version.

Notifies library developers of broken usage patterns, along with the number of projects they occur in.

Notifies affected library users of possibly breaking changes (if desired by the library developers).

Figure 2.5: Visualisation of the validation pipeline.

Given the regression tests generated by the mining pipeline and a new version of the library, the system should identify regressions in functionality employed by users. This is the responsibility of the *validation* pipeline. By running the generated tests against the new version, breaking changes can be reported to both developers and users of the library. This should happen in a *continuous integration* fashion: Each new change request on the Version Control System of the library should trigger a run of this validation pipeline. Figure 2.5 visualises the steps that form this second pipeline.

2.2.1. Execute Tests

Using the tests generated in the mining pipeline, this stage assesses whether any functional regressions have occurred compared to a previous version of the library. The tests are executed against the new library version, with test results being recorded for later inspection.

2.2.2. Notify Developers

The test results from the previous pipeline component need to be relayed to the library developer. These results should contain information on which tests failed, how they failed, and to which usage patterns the failing tests belong. Based on that information, the developer can make an informed decision on whether to roll back a certain breaking change.

2.2.3. Notify Users

If the library developer decides to release a breaking changes (after being notified of this by Schaapi), this components offers the possibility of directly notifying affected users. This could happen through an issue being submitted to the issue tracker system of affected user repositories.

3

Implementation

We have developed a prototype of the Schaapi system in an open-source fashion. This chapter describes the design, implementation, and testing of this system. Please refer to our GitHub repository[1] for the full source code of the implementation.

3.1. Design

In Schaapi, we envision an extensible tool chain that is applicable to various languages and detection mechanisms. In order to achieve this goal, we spent considerable effort designing the generic structure of the system. This chapter describes our system design, starting with the motivating principles followed by an architectural description.

3.1.1. Design Principles

The design process of the Schaapi system was led by two main principles: modularity and configurability. These stem from the existing body of software architecture design theory, established by Bass et al. [5] and Rozanski and Woods [16].

Modularity As argued by Rozanski and Woods, an architecture should be composed of *architectural elements* with clearly defined responsibilities, boundaries, and interfaces [16]. This decomposition into different modules is a crucial step to enable focused and independent development of features. Bass et al. especially emphasise the need for *information hiding* and *separation of concerns*, to ensure that local changes in one module do not lead to an excessive amount of changes in the rest of the system [5].

Configurability Enabled by the modularity of the design, individual components can have a number of alternative implementations [5]. This configurability enables the system user to make fine-grained choices for each step. It should therefore be straightforward to create a new pipeline configuration, using both existing and new implementations for different steps, as long as they assume consistent data schemata and adhere to the functional interface of the relevant components.

3.1.2. Pipelines

We follow the *Pipes and Filters* model by Schmidt et al. as leading architectural design pattern [17]. Individual components are connected by *pipes*, each component performing a processing step on previously given data and passing it on to the next one. Such a processing step can be a filtering, mapping, or in-place manipulation of input data. The steps are grouped into two pipelines: A *mining* pipeline, executed periodically to gather and evaluate user projects, and a *validation* pipeline, used as continuous integration service for new library versions. See chapter 2 for a visualisation of the pipeline designs.

[1] `https://github.com/cafejojo/schaapi`, last accessed June 25, 2018.

3.2. Component Implementations

Following the design described in section 3.1, we set out to write a reference implementation of each component in both pipelines, as well as the framework necessary for executing them.

The system was implemented in Kotlin[2] (version 1.2.50), using the *Gradle* build system[3]. We follow a monolithic repository model with modules for each component implementation (see figure 3.1 for the module hierarchy). This modular layout allows for easy future reuse of individual components, as well as a clear separation between interfaces and their implementation.

3.2.1. Module Structure

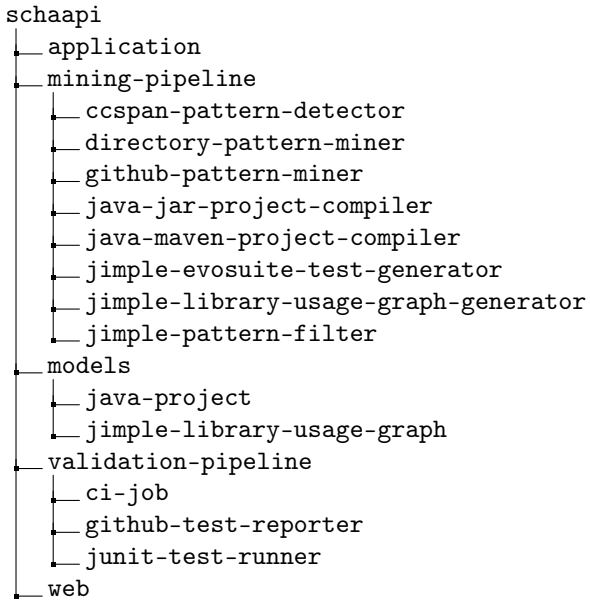


Figure 3.1: The Schaapi module structure

3.2.2. Mining Pipeline

3.2.2.1. Mine Projects

GitHubProject Miner *Mines projects from* GitHub *using its code search REST API.* To be initialised, this module implementation requires a GitHubtoken (needed for API access), a storage directory for downloaded projects, and the type of projects it should expect. The main component functionality takes an object containing search options, indicating what options to pass to the API, such as the specification of the *groupId*, *artifactId*, and *version*. It also ensures that all discovered projects contain a `pom.xml` file in the top-level directory with a dependency on the library indicated in the search options.

Directory Project Miner *"Mines" projects from the file system, finding all projects in one folder.* This is most suitable for local, reproducible setups where total control is needed over the set of projects fed to the system.

3.2.2.2. Compile Projects

Java Maven Project Compiler *Compiles Java Maven projects by fetching their dependencies and compiling the source code.* Using a version of the Maven build tool bundled with Schaapi, this module runs `maven clean install` with test execution turned off. It then traverses the directory containing the generated `class` files to collect a list of all their fully qualified names. As downloaded external dependencies are not needed for later steps of the pipeline, we do not copy their files to the project target directory. This leads to a significant reduction in the time and space complexity of the process.

[2] `https://kotlinlang.org`, last accessed June 25, 2018.
[3] `https://gradle.org`, last accessed June 25, 2018.

Java JAR Project Compiler *Compiles Java JAR projects by inspecting the classes contained in the JAR.* As the program in the JAR is already compiled, this module implementation only needs to collect the names of the classes contained within the JAR. It traverses the entries of the JAR archive and stores the fully qualified names of `.class` files.

3.2.2.3. Analyse Usages

Jimple Library-Usage Graph Generator *Generates library-usage graphs for Java using Soot's Jimple control flow graphs.* Given a library and a user project, this implementation first sets up the static Soot environment. It then goes over all methods in all classes contained in the user project and lets Soot generate a Control Flow Graph (CFG) in its own data structure. This CFG is then processed to filter out statements unrelated to the library usage and is converted into our internal data structure (a library-usage graph).

Detecting which statements should be filtered out was a challenge on its own because there are many different scenarios that need to be taken into account. For this reason, the library-usage graph generator first filters out individual statements, and then proceeds to look at larger constructs.

The library-usage graph generator filters out individual statements if they satisfy one of following three criteria. First, types of statements that are either not relevant or too difficult to simulate are removed. Examples of such types are some of Jimple's internal constructs that should not normally occur in generated Jimple, and static locks. Second, statements that use classes from the user's own project are removed because such statements cannot be executed in the CI pipeline where there is no access to the user code. Branching statements that use the user's project are not filtered out by this step, however, because the control flow may be an essential part of the usage pattern. Instead, the conditions of these branching statements are replaced with unbound conditions that will be randomised in a later step. Third, statements that do not use the library are removed because these are not relevant when looking at how the library is used. This means that statements invoking a method on an object of a library class or assigning a field to an instance of a library class are retained, while statements that only manipulate strings are filtered out.

The library-usage graph generator may also filter out entire branching constructs. This decision is based on both the branching condition and the conditions of the branches, because even if the condition does not explicitly use the library, the various branches may use the library in different ways, making it essential that the construct is retained. Therefore, the branching construct is removed only if neither the condition nor the branches use the library.

3.2.2.4. Find Usage Patterns

PrefixSpan Pattern Detector *Identifies frequent sequential patterns in graphs, using the PrefixSpan algorithm*[4]. As explained in section 2.1.4.2, this algorithm returns a complete set of frequent (sub)sequences. This leads to a large amount of patterns being output by this algorithm, which carries consequences both for space and time complexity of the test generation phase (as well as the test execution phase, later on).

SPAM Pattern Detector *Identifies frequent sequential patterns in graphs, using the SPAM algorithm*[5]. This algorithm returns significantly less patterns than PrefixSpan, which means that it may miss crucial longer sequences that occur frequently. However, it is also less computationally intensive than the PrefixSpan implementation, with informal tests on synthetic benchmark sets showing an order of magnitude of speedup.

CCSpan Pattern Detector *Identifies frequent sequential patterns in graphs, using the CCSpan algorithm.* This algorithm turned out to be the best fit for our situation, as it placed an emphasis on long (closed) sequences, which minimises the number of patterns and maximises their length. Initially, this implementation took much longer to execute than the other two. With a number of optimisations, however, we were able to significantly reduce its runtime and make it scale to more realistic numbers of patterns.

3.2.2.5. Filter Patterns

Jimple Pattern Filters *Collection of filters that work with Jimple-based library-usage graphs.* By default, we filter out patterns that have a length less than 2. The main motivation is that we are mainly interested in the interaction between method calls, and not so much whether individual library calls still function. It is however still possible to set this minimum length to 1 if the user wishes to investigate this. For instance, a single library call might still throw an exception.

[4] `https://github.com/cafejojo/prefix-span-pattern-detector`, last accessed June 25, 2018.
[5] `https://github.com/cafejojo/spam-pattern-detector`, last accessed June 25, 2018.

3.2.2.6. Generate Tests

Jimple EvoSuite Test Generator *Generates testable classes based on Soot Jimple based patterns, and generates tests for those using EvoSuite.* For each pattern, all statements are checked for undefined variables (defined out of the scope of the pattern). The implementation of this module then creates a static method with those variables as method parameters and the statements of the pattern as body. Once each pattern has been converted into a method, a class is generated containing all methods. This class is then passed to a subprocess running the EvoSuite generator, to let it generate tests for the patterns.

3.2.3. Validation Pipeline

The implementation of the validation pipeline is written around GitHub's new Checks API [6] (in public beta at the date of publication of this thesis). After a new PR is created, Schaapi executes the tests generated during the mining pipeline process, and reports the status back to GitHub. Examples of test reports by Schaapi on GitHub can be seen in figure 3.2.

3.2.3.1. CI Trigger

GitHub Interactor *Receives web hooks from GitHub.* For each new branch, GitHub sends a web hook to our application. Once we have received this web hook, we start a CI job run based on the information about the project present in the body of the web hook request.

3.2.3.2. CI Runs

CI Job *Specification of tasks to be executed in a single CI run.* In each CI job, a few tasks are executed. CI jobs are executed on a separate thread. First, a current version of the library project is downloaded, unzipped, and compiled (using the compilers from the mining pipeline 3.2.2.2). Then, tests are executed using the JUnit test runner. Finally, the system fires an internal event indicating the results of the test run. Such an event can also contain information about, for instance, a compilation error.

3.2.3.3. Execute Tests

JUnit Test Runner *Executes a JUnit test suite and reports on the results.* Given a test class, the JUnit test runner passes this class to the JUnit runner to execute the tests and then collects the results. These results are then passed on to the next pipeline step.

3.2.3.4. Notify Developers

GitHub Interactor *Reports the test results back to the developers of the library, through the GitHub Checks API.* The GitHub interactor listens for CI Job completed events. As soon as such an event is received, the test results are reported back to GitHub.

3.2.3.5. Notify Users

As discussed in more detail in appendix D on ethical implications, this step had controversial aspects to it. If we were to implement this, we would have to spend considerable effort on ensuring correctness and safety. With these considerations and the tight time constraints of this project in mind, we decided to assign lower priority to this component. As a result, there currently does not exist an implementation for it.

3.3. Quality Assurance

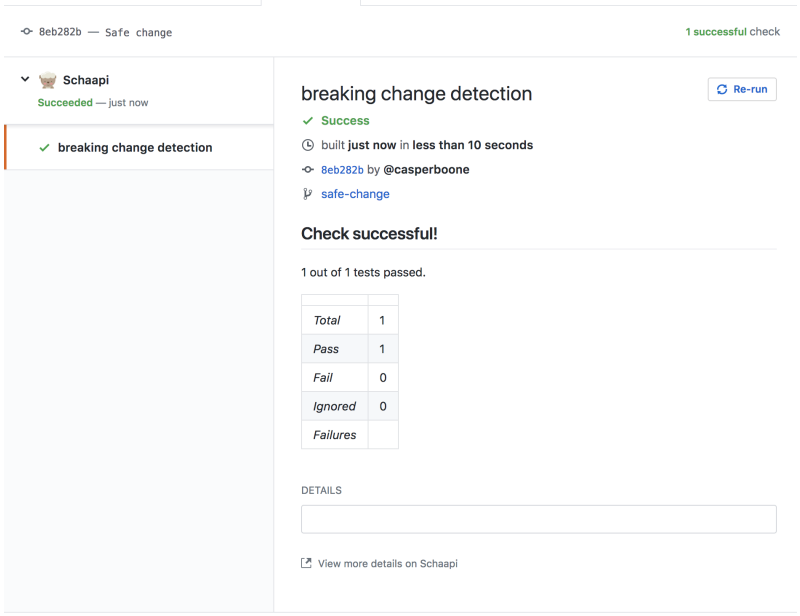
We employed a variety of strategies to gain confidence in the functional and non-functional adequacy of our system. These efforts were complemented by a code quality assurance workflow designed to ensure maintainability. In this chapter, we describe our approaches to both these facets of the software engineering process.

3.3.1. Functional Testing

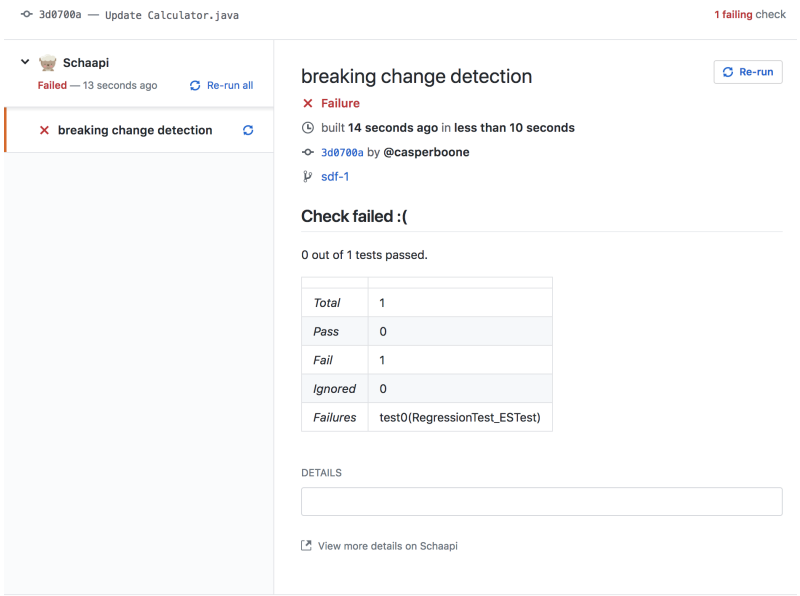
3.3.1.1. Testing Hierarchy

We tested the system at a variety of levels, from individual classes all the way up to a full system run-through. The following paragraphs explain how each of these levels in the system hierarchy was tested.

[6] `https://blog.github.com/2018-05-07-introducing-checks-api`, last accessed June 25, 2018.



(a) A successful check.



(b) A failed check.

Figure 3.2: GitHub Checks from Schaapi as visible on PR pages.

Unit Testing We wrote unit tests which test only a single unit (most often this was a single public method) and tried to mock out or otherwise rid of any unnecessary dependencies that are not part of the unit under test. This gives us information on whether the unit works in isolation, and is a prerequisite for bigger groups of functionality: If the units themselves do not work as expected, little can be expected of the combination of units.

Intra-Module Testing We also wrote intra-module integration tests, which test whether the different classes in a single module integrate well. The interactions between different units often give rise to new behaviour and unforeseen problems, which these tests attempt to detect.

Inter-Module Testing On a larger scale, we created a number of inter-module integration tests, which test whether the different modules integrate well. These tests were written for selected tuples of consecutive modules in the pipelines to see if the input of one module would be transformed to correct output by a second module.

End-to-End Testing Finally, we wrote a small number of tests which simply execute the pipeline from start to end and verify that the program produces valid output. These smoke test does not so much test that the program works as it is used to verify that the program's most basic execution is still valid, which makes for a useful baseline if more complex inputs seem to fail. While a useful baseline, we were constrained by its long duration – the smoke test was the longest running test, and adding more would significantly increase build times of our continuous integration builds (see section 3.3.3 for more information on our development workflow).

3.3.1.2. Testing Framework

We used *Spek* [7] to specify our tests, with *AssertJ* [8] for readable assertions. While these Spek tests are run with the *JUnit 5* [9] testing framework, the Spek framework allows for more expressive test specifications. Among these advantages is the ability to express the purpose of a test in a full sentence, which makes it easier to recognise which functionality is affected when there are failing tests. which significantly improves the readability of the test code base. Spek still misses a number of features that are available in the JUnit framework such as test class inheritance and parameterised tests, but luckily we did not find any cases where we needed that functionality. Overall, our tests provide a runnable and readable body of documentation to readers of the code base, combined with more traditional commenting strategies.

3.3.1.3. Coverage

At the publication of this thesis, the coverage as calculated by Codecov is 80.49%. This coverage is calculated as the ratio of the number of lines of which all branches have been executed to the total number of lines. The full coverage report can be inspected online[10].

3.3.2. Non-Functional Testing

In initial exploratory tests, we encountered severe performance limitations when working with input datasets of significant sizes. Especially the initial implementations of components such as of the pattern detector and the test generator turned out to require unreasonable amounts of time and space. In addition to addressing these issues with optimisations, we also designed benchmark scenarios that we executed periodically in order to evaluate the time and space requirements of the system. Chapter 4 describes these efforts in detail.

3.3.3. Change Management

Our code base is hosted on GitHub using a Git branching workflow for the proposal of changes. Each suggested change is submitted in the form of a Pull Request (PR) with a combination of dynamic and static checks providing automated feedback on functional and non-functional code quality aspects. We employ two Continuous Integration (CI) services as platforms for test and static analysis execution: *Travis* [11], representing Linux execution environments, and *AppVeyor* [12], representing Windows environments. These environments run all functional tests and report their results to the PR page. The coverage recorded during these runs is sent to *Codecov* [13] for coverage visualisation. *Codecov* then reports coverage differences and totals per file in the PR comments to inform us on which changed sections of code might need better testing.

Next to the dynamic analyses these CI services provide, we also let them execute two static analysis steps to ensure uniform code style and automatically detect a number of potential bugs. *ktlint* [14] verifies that our code style adheres to the Kotlin style guide[15]. *detekt* [16] checks for a number of statically verifiable code quality

[7] `http://spekframework.org`, last accessed June 25, 2018.
[8] `https://joel-costigliola.github.io/assertj`, last accessed June 25, 2018.
[9] `https://junit.org/junit5`, last accessed June 25, 2018.
[10] `https://codecov.io/github/cafejojo/schaapi`, last accessed June 25, 2018.
[11] `https://travis-ci.org`, last accessed June 25, 2018.
[12] `https://www.appveyor.com`, last accessed June 25, 2018.
[13] `https://codecov.io`, last accessed June 25, 2018.
[14] `https://github.com/shyiko/ktlint`, last accessed June 25, 2018.
[15] `https://kotlinlang.org/docs/reference/coding-conventions.html`, last accessed June 25, 2018.
[16] `https://github.com/arturbosch/detekt`, last accessed June 25, 2018.

indicators, comparable to the Java *CheckStyle* plugin[17].

We also required each PR to have at least two approving reviews by collaborators. These additional manual inspections have often led to suggestions that improved readability and corrected mistakes not caught by the toolchain. It also enforced a shared ownership and knowledge of the code.

3.3.4. Software Improvement Group Evaluation

In the context of this final Bachelor project, we submitted our code twice to the Software Improvement Group[18] (SIG) for static analysis. After submitting our code half-way through the project, we received a first assessment (included in appendix C). Our code base was rated 3.6 (out of 5) stars and was described by the reviewer as having "industry-average" maintainability. The feedback by SIG could be split into three main areas of improvement: (i) unit size, (ii) duplication, and (iii) test code volume. In the following paragraphs, we address each concern in turn.

Unit Size Long methods are harder to understand and test, which is why one should generally aim for small unit sizes. The SIG review mentioned one case of a particularly long method (`generalizedValuesAreEqual`, in `GeneralizedNodeComparator`), which could be refactored into smaller units.

In response to both this concrete feedback and the general feedback on average unit size, we combed through the code base and refactored longer methods wherever possible. We found several methods that operated on heterogeneous levels of abstractions. This is confusing to readers of the code and generally leads to longer methods. We addressed these cases by moving out parts of the functionality to helper methods, making the main function easier to read.

Duplication The feedback also mentions redundant code, which should be eliminated to improve the maintainability of the code base. One example given is the `PatternDetector` class, which the reviewers see appearing in three places in the code base. While we acknowledge that this can lead to confusion, we would like to elaborate on our design considerations behind this.

In our component architecture, each component implementation has a class with the same name as the interface it is conforming to in order to facilitate consistent interaction patterns with the component implementations. This means that we created three `PatternDetector` classes; one for each pattern detector component implementation that we made. While they share the same name, they are located in different packages, meaning that their fully qualified names (including the package structure) are unique. We understand that this can confuse readers of the code, and have renamed the classes to reflect their unique aspects (e.g. the main CCSpan pattern detection implementation class has been renamed from `PatternDetector` to `CCSpanPatternDetector`).

Test Code Volume The third area of improvement that is identified by SIG is the number of tests. According to the reviewer, the amount of testing code is lacking behind the amount of production code. However, the assessment text does not specify any specific areas of the code that could benefit from more tests. We have continued to extend our test suite, guided by the insights our code coverage tool provided, increasing both our coverage and the number of tests. At the date of publication of this thesis, 56% of the lines of code in our code base is test code.

[17] `https://github.com/checkstyle/checkstyle`, last accessed June 25, 2018.
[18] `https://www.sig.eu`, last accessed June 25, 2018.

4

Empirical Validation

Through empirical validation, we wish to see how well our tool functions in real-world scenarios. To this end, we ask the following questions:

RQ1 How many usage patterns are found for commonly used libraries?
RQ2 What are the lengths of the usage patterns found for commonly used libraries?
RQ3 How does the minimum support affect the amount of patterns generated?
RQ4 Does the tool detect injected breaking changes?

In this chapter, we first outline our research method. Then, for each research question, we describe our efforts to answer it and any results we have obtained.

4.1. Research Method

To answer our research questions, we run our tool on a set of projects.

We first construct two datasets, each capped at 150 user projects, all downloaded from GitHub. The first dataset consists of projects having the *Google Guava* library[1] as a dependency. The second set consists of projects that have a dependency on the *Apache Core Spark* project[2]. The minimum support that patterns need to have to be considered frequent is 2, which is the default.

```
--flavor "github"
-o ".../GuavaOutput"
-l ".../guava-25.1-jre.jar"
--library_group_id "com.google.guava"
--library_artifact_id "guava"
--library_version "25.1-jre"
--max_projects 150
--test_generator_timeout 120
```

```
--flavor "github"
-o ".../SparkOutput"
-l ".../spark-core_2.11-2.3.1.jar"
--library_group_id "org.apache.spark"
--library_artifact_id
    "spark-core_2.11"
--library_version "2.3.1"
--max_projects 150
--test_generator_timeout 120
```

Figure 4.1: Command line arguments used to retrieve the projects using two different libraries. From left to right: Guava, Spark.

The datasets acquired by running Schaapi with the above configurations can then be used to answer the stated questions. This can be achieved by using the directory mining tool and mining the respective output directories.

As there are many downloaded projects that do not compile, we ensure that our initial sample of projects is large enough by capping their number at 150.

Both datasets were created on June 20, 2018.

[1] `https://mvnrepository.com/artifact/com.google.guava/guava/25.1-jre`, last accessed June 25, 2018.
[2] `https://mvnrepository.com/artifact/org.apache.spark/spark-core_2.11/2.3.1`, last accessed June 25, 2018.

4.2. RQ1: Usage Patterns Found for Commonly Used Libraries

How many usage patterns are found for commonly used libraries?

We first aim to uncover how many patterns are found by our tool. This is both to evaluate our tool, and, to a lesser extent, the difference in usage of the Guava and Spark libraries.

The amount of projects found is shown in table 4.1. Under *Found*, the total amount of projects found by the GitHub search API is shown, which are all projects which (likely) use the relevant library as a dependency. We also find that some projects do not contain any production class files, which we will characterise as *empty* from now on. The number of remaining projects is shown under *Non-Empty*.

Table 4.1: Number of non-empty projects found.

Project Name	Found	Downloaded	Compiled	Non-Empty
Google Guava	581	149	103	85
Apache Spark	4 851	150	123	62

Of all downloaded projects that depend on Guava, 57.04% are not empty and are compilable, whereas for projects that depend on Spark this amount is 41.33%. These numbers indicate that it is important to download a large amount of projects, as many do not compile or do not contain relevant source files, something also previously mentioned by Kalliamvakou et al. [9].

Based on the amount of projects, we show the total amount of valid usages in table 4.2. We count the amount of statements found within a library usage and the amount remaining after filtering out non-relevant statements. We show the total amount of library-usage graphs, with each such graph representing a single method which contains at least one valid library usage. For comparison, we also include the total amount of concrete methods found, which are non-abstract methods which have a body, but may not have library usages for comparison.

Table 4.2: Number of library-usage graphs and statements found.

Project Name	Concrete Methods	Usages	Valid Usages	Library-Usage Graphs
Google Guava	9 542	105 232	10 470	243
Apache Spark	1 363	18 794	3 301	147

These statistics also show that, as to be expected, only a small amount of methods contain a usage of one of the respective libraries. We find that 39.26% of concrete methods in Guava projects produces a library-usage graph, meaning that only a small percentage of concrete methods uses the library. For Spark, this number is 9.27%. We aim to uncover the relationship between the amount of library-usage graphs, amount of patterns found, and amount of tests. In table 4.3, we see the relationship between the number of valid usages identified and the amount of patterns generated.

Table 4.3: Number of patterns found from valid usages, with default minimum support 2.

Project Name	Valid Usages	Patterns	Valid Patterns
Google Guava	10 470	272	205
Apache Spark	3 301	267	189

We observe that a large portion of patterns is filtered out. For projects depending on Guava, 75.36% of patterns is considered valid, whereas for projects depending on Spark this value is 70.78%. This is in part because generated patterns often contain only a single node—which we discard—or lack an initialisation step for the library object.

We also see that the amount of patterns found is not much greater than the amount of downloaded projects. In fact, our findings show that the ratio between downloaded projects and eventual pattern count is

1:0.72 and 1:0.79 for Guava and Spark projects respectively, meaning that it also seems to depend heavily on the library and how it is used.

In figure 4.2, we showcase the relationship between the amount of library-usage graphs and the amount of tests generated. We do not yet look at the size of the library-usage graphs and how this relates to the amount of tests generated; this relationship is further explored in section 4.3. The tests are run using a 120 second timeout, with a 72% coverage for the generated Guava patterns and 0% coverage for the generated Spark patterns, meaning that not all generated patterns are in fact covered by tests.

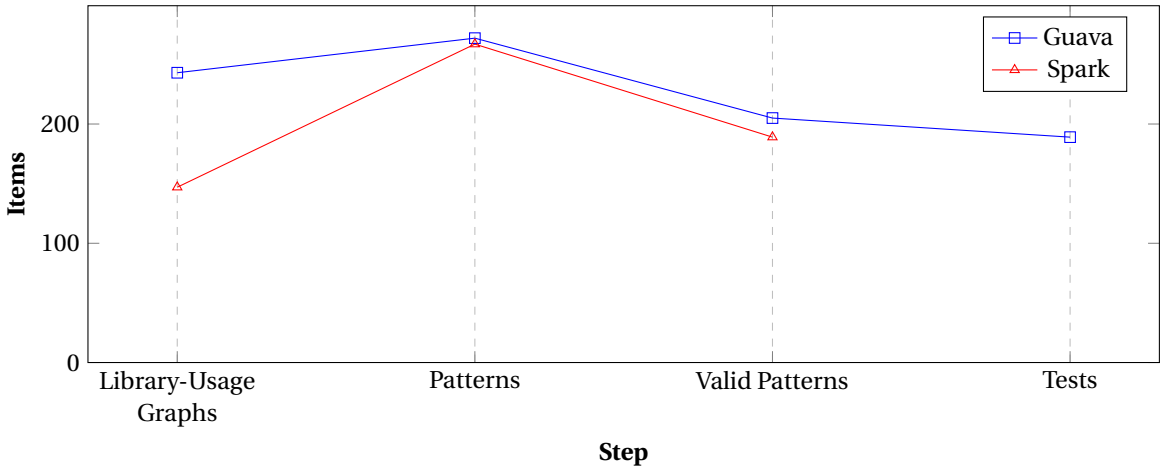


Figure 4.2: Visualisation of the amount of items per pipeline step, starting with the amount of valid usages found.

While our tool successfully detects usages of Spark, the test generation tool, EvoSuite, cannot handle Scala projects. As such, we unfortunately do not have any test data on Spark. However, we choose to still use Spark to show how its usages contrast with those of Guava.

Findings

1. It is important to download a large amount of projects to still end up with a sizeable amount of projects that are non-empty and compilable.
2. Often, only a small portion of the methods extracted from user projects contain library usages. This means that few methods can be used for library-usage graph generation.
3. The amount of patterns found is not much greater than the amount of projects, likely in part due to the above reasons.
4. Finally, due to the evolutionary nature of the test generation tool Evosuite, not all patterns are covered due to the test generation timout.

4.3. RQ2: Lengths of Found Usage Patterns

What are the lengths of the usage patterns found for commonly used libraries?

We wish to investigate whether there is a relationship between the amount and sizes of library-usage graphs and the amount and lengths of the found patterns. To investigate this, we take a look at the pattern generation stage of the pipeline, starting with the generation of library-usage graphs. The respective amounts of usages, library-usage graphs, and patterns are all shown in table 4.2.

Table 4.4: Amount of library-usage graphs and eventual amount of patterns.

Project Name	Valid Usages	Library-Usage Graphs	Patterns	Valid Patterns
Google Guava	10 470	243	272	205
Apache Spark	3 301	147	267	189

One thing of note is that there are many more usages than library-usage graphs. This implies that there are several library-usage graphs with multiple nodes. We also note that, even though there being many more valid usages of the Guava library and larger library-usage graphs, the amount of patterns generated is roughly equal. This implies that the amount of ways Spark is used is more varied.

The distribution of library-usage graph sizes can be seen in figure 4.3, with larger graph sizes implying larger—and potentially more complex—methods.

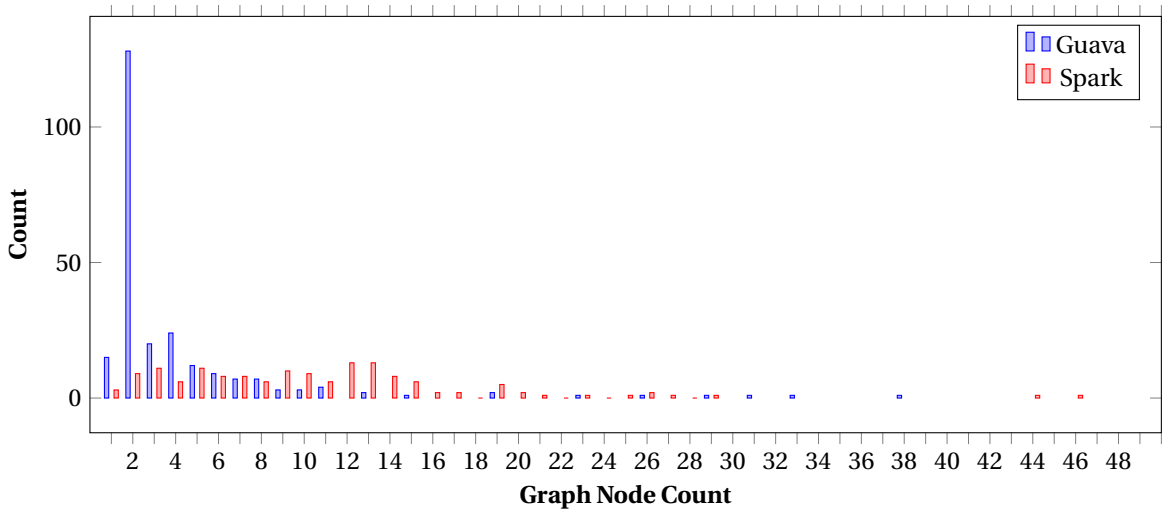


Figure 4.3: Distribution of library-usage graph node counts.

The lengths of the sequences extracted from these graphs are shown in figure 4.4.

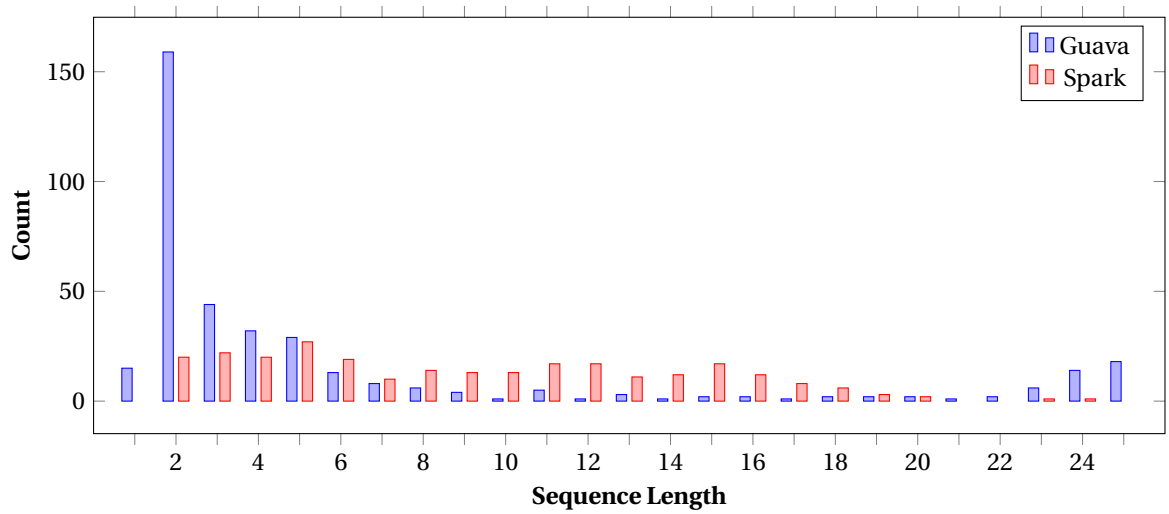


Figure 4.4: Distribution of lengths of extracted sequences.

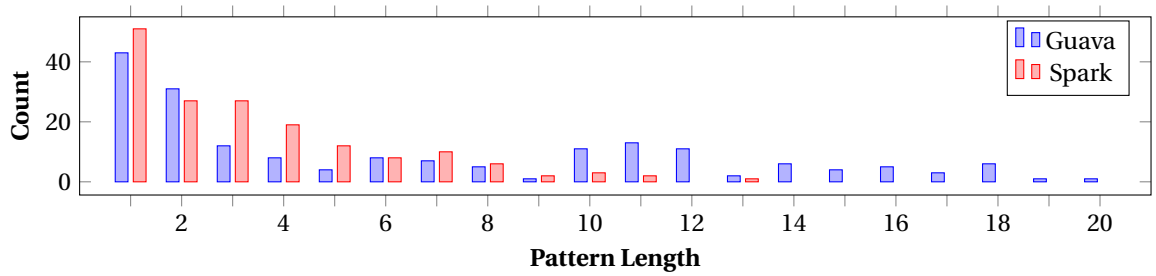
We observe that the sequence lengths have a Poisson-like distribution, with short sequences being more common than long ones, similar to the distribution of graph sizes. In addition to this, we see that most

sequences found are of length two. This suggests that in most situations, no more than two consecutive calls are made to the Guava library within a single method.

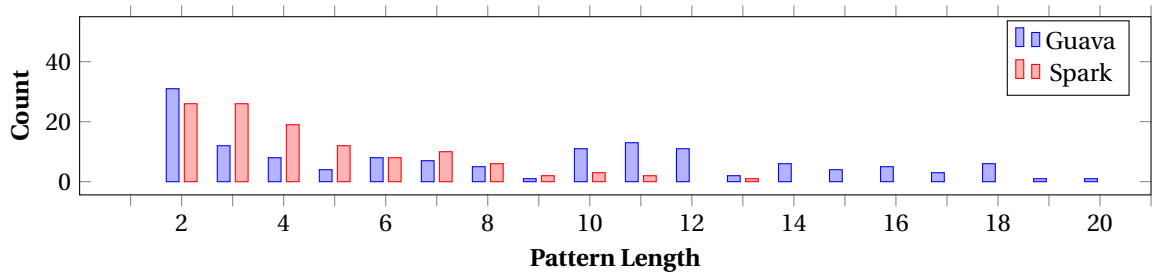
If the sequence of calls to the Spark library are more varied when compared to Guava, this could further explain why the ratio between the amount of found patterns and amount of usages is greater, as was found to be the case in section 4.2. Naturally, we would expect that it is more likely that two short sequences are much like one another, whereas for two long sequences the odds of them being identical is likely much smaller.

Of note, however, is that for Guava, sequence lengths of 23 and greater seem to be common, though we would expect this to taper off again at some point. This is likely because graphs with a large amount of nodes are more complex and have more possible paths running through them. With Spark usages, however, we do not observe such a trend.

We also wish to see how large the resulting patterns are based on these sequences. The sizes of patterns before the filter process are shown in figure 4.5a, and the sizes of patterns after the filter process are shown in figure 4.5b. The patterns that remain after the filtering are also referred to as *valid* patterns. Note that we also look at the amount of patterns generated when the minimum support is 1. A minimum support of 1 results in every node found being used for the generation of patterns, meaning that every single found library usage is regarded as statically significant.



(a) Before filtering.



(b) After filtering.

Figure 4.5: Distribution of lengths of patterns with default minimum support 2.

We see another Poisson-like distribution, with short patterns being more common than long ones. The figure also shows us that a significant portion of patterns are of length 1. These are filtered out by default. Aside from these, no patterns containing more than one node have been filtered.

Despite a large amount of long sequences, we observe no long patterns are found for Spark. However, these long sequences may contain statistically significant short sequences, which our tool does detect. This again suggests that Spark usages often consist of long method chains that are unique, but likely do contain short sequences of method calls that are themselves statistically significant.

Findings

1. The ratio between the lengths of patterns and the lengths of sequences depends heavily on the library itself.
2. When comparing Spark to Guava, Guava produces sequences of calls to library methods that are relatively short, whereas for Spark the distribution of lengths is more evenly distributed.
3. Despite the lengths of sequences of calls to the library being longer on average for Spark, the amount of short patterns found is greater than Guava. This suggests that long chains of calls to the library, whilst likely unique, may still contain statistically significant subsequences.
4. Pattern filters, as they are now, have little effect on the set of patterns, as they do little else besides removing patterns of length one.

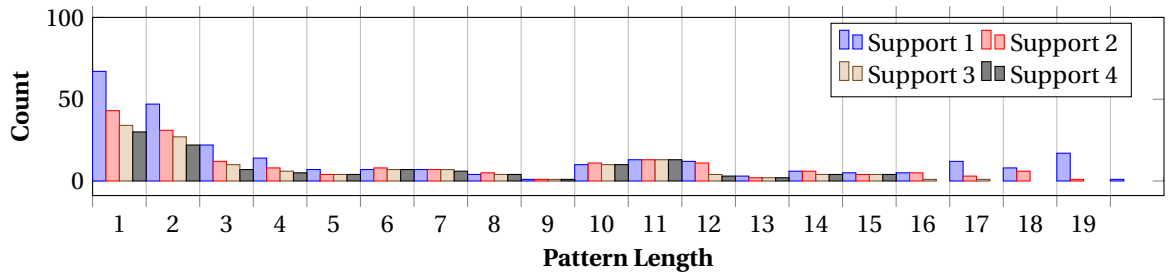
4.4. RQ3: Effect of Minimum Support on Patterns

How does the minimum support affect the amount of patterns generated?

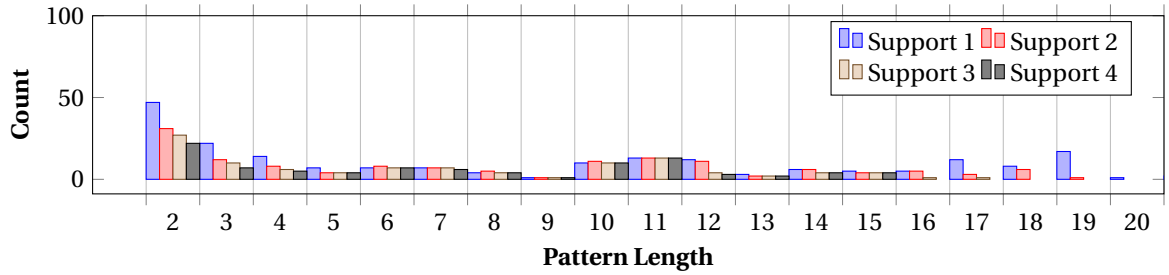
We wish to see how the minimum support affects the pattern generation stage of the tool. We run the tool several times on the dataset of Guava projects, varying the minimum support value for the pattern detector by passing a new value to `pattern_detector_minimum_count` argument. The resulting pattern lengths are shown in figures 4.6 for Guava and 4.7 for Spark.

We see that the change in minimum support has a large effect on the amount of short patterns that are generated. For patterns of length one, this due to all nodes being used for pattern generation, even those that are used only once. This effect is again seen when looking at long patterns, meaning that longer patterns are more likely to be unique.

We also observe again that the filter process removes only patterns of length one and leaves all others untouched.

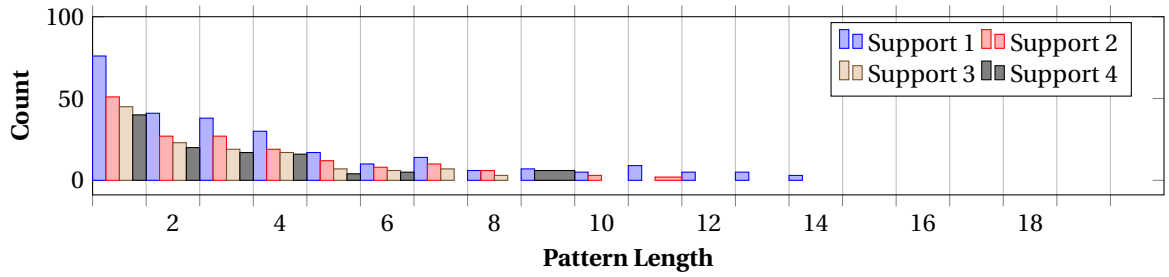


(a) Guava: Before filtering.

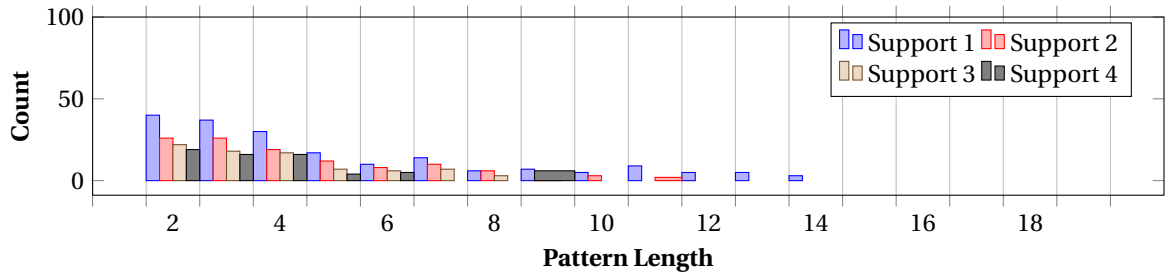


(b) Guava: After filtering.

Figure 4.6: Distribution of lengths of patterns for different minimum support choices for Guava.



(a) Spark: Before filtering.



(b) Spark: After filtering.

Figure 4.7: Distribution of lengths of patterns for different minimum support choices for Spark.

Findings

1. A change in minimum support has the most significant effect on the amount of short patterns being generated. The same is true for long patterns.
2. For patterns of medium length, the minimum support seems to have little effect on the amount being generated.

4.5. RQ4: Detecting Injected Breaking Changes

Does the tool detect injected breaking changes?

Lastly, we wish to see whether our tool can successfully detect semantic breaking changes. To do this, we create a dummy library and dummy user. We create two versions of the library, where the second version introduces semantic breaking changes. The code for these projects can be found in figures 4.8 and 4.9.

```java
package org.cafejojo.schaapi.test.library;

public final class LibraryClass {
    public String neverEmptyString() { return "string"; }
    public char alwaysBChar() { return 'b'; }
    public int alwaysPositiveInt() { return 10; }
    public double alwaysPositiveDouble() { return 100.0; }
    public boolean alwaysTrueBoolean() { return true; }

    public static void checkStringNotEmpty(String s) {
        if (s.isEmpty()) throw new RuntimeException(); }
    public static void checkCharIsB(char c) {
        if (c != 'b') throw new RuntimeException(); }
    public static void checkIntIsPositive(int i) {
        if (i < 0) throw new RuntimeException(); }
    public static void checkDoubleIsPositive(double d) {
        if (d < 0) throw new RuntimeException(); }
    public static void checkBooleanIsTrue(boolean b) {
        if (!b) throw new RuntimeException(); }
}
```

Figure 4.8: The example library source code before changes are applied.

```java
package org.cafejojo.schaapi.test.examplelibrary;

public final class LibraryClass {
    public String neverEmptyString() { return ""; }
    public char alwaysBChar() { return 'z'; }
    public int alwaysPositiveInt() { return -10; }
    public double alwaysPositiveDouble() { return -100.0; }
    public boolean alwaysTrueBoolean() { return false; }

    public static void checkStringNotEmpty(String s) {
        if (s.isEmpty()) throw new RuntimeException(); }
    public static void checkCharIsB(char c) {
        if (c != 'b') throw new RuntimeException(); }
    public static void checkIntIsPositive(int i) {
        if (i < 0) throw new RuntimeException(); }
    public static void checkDoubleIsPositive(double d) {
        if (d < 0) throw new RuntimeException(); }
    public static void checkBooleanIsTrue(boolean b) {
        if (!b) throw new RuntimeException(); }
}
```

Figure 4.9: The example library source code after changes have been applied.

We then create a dummy user project which uses the library, as can be seen in figure 4.10. The code depicted here can also be found on GitHub[3].

It should be noted that we do not track non-library objects, meaning that any consecutive checks using these objects will be discarded by the tool. As a workaround, we add methods to the library which check the values of these objects, and throw exceptions if they do not adhere to the contract. This simulates a user creating a library object and then passing it on to another library function.

We set the minimum support to 1 by setting the command-line option `pattern_detector_minimum_count` to 1 because each statement occurs only once in the user codebase.

[3] `https://github.com/cafejojo/schaapi-injected-changes-study`, last accessed June 25, 2018.

```java
package org.cafejojo.schaapi.test.user;

import org.cafejojo.schaapi.test.library.LibraryClass;

public class UserClass {
    public void usageOne() {
        String s = new LibraryClass().neverEmptyString();
        LibraryClass.checkStringNotEmpty(s);
    }
    public void usageTwo() {
        char c = new LibraryClass().alwaysBChar();
        LibraryClass.checkCharIsB(c);
    }
    public void usageThree() {
        int i = new LibraryClass().alwaysPositiveInt();
        LibraryClass.checkIntIsPositive(i);
    }
    public void usageFour() {
        double d = new LibraryClass().alwaysPositiveDouble();
        LibraryClass.checkDoubleIsPositive(d);
    }
    public void usageFive() {
        boolean b = new LibraryClass().alwaysTrueBoolean();
        LibraryClass.checkBooleanIsTrue(b);
    }
}
```

Figure 4.10: The example user code used for test generation.

We report the results in table 4.5. For each method, we note whether a pattern is generated which contains the statements in the method under *Pattern Generated*. We also note whether a test was generated for said pattern under *Test Generated*. Next, we run the generated test suite using the new version of the library as seen in figure 4.9. For each test, we check whether it fails. If a test fails, it means the change was indeed detected. This is shown under *Change Detected*.

Table 4.5: Whether a breaking change was detected in an example library.

Library Method	Return Type	Version 1	Version 2	Pattern Generated	Test Generated	Change Detected
neverEmptyString	String	"string"	""	✔	✔	✔
alwaysBChar	char	'b'	'z'	✔	✔	✔
alwaysPositiveInt	int	10	-10	✔	✔	✔
alwaysPositiveDouble	double	100.0	-100.0	✔	✔	✔
alwaysTrueBoolean	boolean	true	false	✔	✔	✔

We observe that our tool successfully detects all breaking changes in the given setting. This means that a test was generated for every situation where the contract was violated. Furthermore, the default test generation timeout of 60 seconds proves to be enough time to generate tests that cover all usages in this simple situation. It should be noted, however, that this setup is a very simple example, and for more complex cases a longer test generation timeout might be necessary (see coverage percentages in section 4.2).

Findings

1. The tool detects breaking changes in certain situations.
2. The default test generation timeout of 60 seconds is sufficient to generate tests that cover all library-usage patterns in the given simple synthesised situation.

5 Discussion

Detecting semantic breaking changes of library usage has proven to be difficult in several regards. Be it collecting the data itself, finding relevant patterns in the data, or verifying the correctness of the tool itself. Due to the approach we have taken being novel, to the best of our knowledge, and the fact that the tool was developed in a relatively short time frame, there is still much room for improvement regarding validating and testing of the tool.

5.1. Threats to Validity

Limited Validation One effect of the time constraints of this project is that we had little time to perform extensive validation testing on the output of the tool. Whilst the tool does output tests based on found patterns, we have not thoroughly verified if other common patterns have been missed or if the found patterns are indeed common.

We do perform an empirical analysis that indicates that the tool functions to a certain extent. This analysis, however, is limited in its scope and is likely biased due to the choice of library and limited dataset used. It might therefore be interesting to perform a historical analysis and see how the tool functions over time in the future.

Limited Scope of Analysis To limit the scope, we decided early on in the design process to limit ourselves to patterns that are completely contained within methods. While doing this limits the complexity of generated control flow graphs, it may also result in patterns that are spread over multiple methods simply not being detected. A trivial example would be multiple methods making calls to a given library, with a public method calling them sequentially. A large scope of analysis would allow us to detect intraprocedural library-usage patterns such as these.

Generated Patterns out of Context The patterns that our tool finds may not be relevant because the pattern detector may choose subsequences that do not occur in that form in actual code. A supersequence of such a pattern may be a common use case, but the pattern itself may never occur without certain statements preceding and following it.

A related issue arises in the generation of tests from patterns. As we rely on the EvoSuite test generator to derive input values for our patterns, there is a high likelihood that it generates input values that are not commonplace. We see this as one of the fundamental disadvantages of our approach. Due to the fact that we rely purely on static analysis of user code, we cannot accurately derive what values will be chosen at runtime. Alternative approaches, such as tracing real-world executions of user projects or mining test code, may help address this issue.

Choosing a Representative User Project Collection When mining projects from GitHub, we attempt to order them by amount of stargazers and watchers, which was stated by Kalliamvakou et al. to be a good metric for detecting whether projects are active and actively being used [9]. In addition to this, they mention that another good metric is to take only projects which have been recently committed to, ideally within the last month.

However, we find that all projects we mine at the moment do not have any stargazers or watchers. In addition to this, the current implementation we fails to sort projects by last commit date and does not sort them by last modified date which may include modification of for instance the project wiki. We should therefore look into better practices of GitHub mining, which could improve our sampling and perhaps the ratio of projects that compile.

5.2. Lessons Learnt

Emphasising Integration Testing We find that, with tools such as Schaapi that rely on complex APIs and incredibly varied and complex input data, it becomes of great value to perform more integration and end-to-end testing. Unit testing, whilst important, covers only a small amount of functionality and cannot effectively address the communication between several components. Such unit tests often have artificial input that does not necessarily reflect (common) real-life situations, reducing their usefulness. Most faults in the application were uncovered only after the tool was run several times on several real-world projects. Therefore, in the future, we should more thoroughly test the integration of different components. It would also be useful to run such a tool on a large number of synthetic or real-world projects early on in the process to validate that the tool functions as intended early on.

Balancing Research and Development As this tool was meant to be both a working product and a prototype of a novel system, we needed to balance development with research throughout the course of the project. The reality of the short timeframe of this project has lead to a number of difficult decisions regarding which parts to prioritise. We would have liked to empirically validate our tool earlier in the course of the project. When building such a prototype, it is difficult to foresee how it will perform in realistic scenarios.

We *did* run parts of the mining pipeline on real-world data early on, which helped us find a number of edge cases that we otherwise would not have come across. However, a full empirical validation would have required the entire mining pipeline to be working in at least a basic fashion, which took considerable effort and time to accomplish, especially when faced with the diverse nature of real-world code fragments. Nevertheless, we believe that we should have placed a more pronounced emphasis on this aspect early on to give it a higher priority.

Using the Kotlin Language Overall, our experience with Kotlin as a language was very positive. We would definitely recommend it over Java for future projects targeting the JVM.

Kotlin's expressive nature made it easy to write concise, expressive code devoid of excessive boilerplate structures. Features such as its non-null types and the inherent streaming lambdas on collections also improved the developer experience, both in writing and reading code. We also found Kotlin to have a shallow learning curve for developers coming from a Java background, with many of our team members not having any prior experience in Kotlin and quickly gaining familiarity over the course of the first few weeks.

However, the concise nature of the language also lead to a risk of over-engineering portions of the code, making them too concise to be easily readable. We attempted to point this out to each other during code reviews, but it remains a pitfall to be wary of.

5.3. Recommendations

Over the course of the project, we had to limit the scope of our aims in order to produce a working product at the end. However, we encountered a number of adjacent project ideas that could be pursued in future work. In this section, we describe future work that could extend our initial Schaapi prototype.

Communicate Common Patterns Proactively We find common usage patterns, but we do not relay this information to the user. As the tool stands now, it functions as a black box that generates tests, and the only feedback users get is whether the tests pass or not, without the ability to manually read the tests or the found patterns. Given that we collect this information, it might be useful to relay this information to library developers directly. This could make them more aware of what the common usage patterns actually are, even before potentially breaking them and being informed by Schaapi afterwards.

It should be noted that this information is stored as Java bytecode. Therefore, whilst not the most elegant solution, it is still possible to decompile this bytecode and display this to the user. However, we do not store how often a pattern has been found, meaning the library developer still has no way of uncovering how often a pattern occurs.

More Sophisticated Filtering Techniques Generating bytecode from isolated sequences of instructions without the full context of variables can lead to undesirable constructs in the generated patterns. Take, for instance, the problem of infinite loops in generated patterns. While the tool has a simple filter built in for empty loop constructs, an infinite loop with only a single statement in it is presently not removed. This is because infinite loops are in general difficult to detect. There are, however, heuristics that could be applied here to identify more types of infinite loops in the generated patterns, making them more useful to the library developer.

Generate Names for Test Cases Future work may also investigate the readability of the generated test cases. We envision that the tool automatically generates a name and potentially a comment description for each test case that describes the content of the pattern it tests, as well as the condition it is testing. Instead of seeing the number of test that failed (along with potentially the full source code), the generated description could be read to gain a quick understanding of what pattern is being broken.

Extend Scope from Method-Level to Global We recommend that functionality is added to look at the global scope of the application, or, failing that, a scope that is at least larger than method level. We suspect that a number of patterns is not detected as a result of only looking at the method level, and extending the scope of analysis may resolve this issue.

Support More Advanced Language Constructs A set of more advanced language constructs is not yet explicitly supported by our tool. Among these are exceptional control flow graphs, which supports exceptions and their `catch` clauses. Another such construct is the inline lambda, which brings with it a different set of challenges. We also do not support the mining of library annotation usages at this moment. Schaapi could be extended to support these constructs in order to more accurately cover the library usages of users in the tests it generates.

Isolate CI Execution Environments The tests generated by our tool are presently run in the same execution environment as the server. Although these tests are generated from patterns which we filter in the mining pipeline, potential security vulnerabilities in this approach may allow an attacker to exploit this lack of isolation to disrupt the web service. Future work should isolate the CI job execution environment to prevent this.

Conduct a Detailed Case Study To fully validate the Schaapi system, an in-depth validation of the detection of breaking changes in a real-world environment would be a useful extension of the research in this work. We envision this being conducted in two directions: In a historical analysis, and in a real-world case study. In the historical analysis, one could look at the versioning history of both a library and its users, and see if a previous breaking change in the library that affected real user projects would have been detected by Schaapi. In a real-world case study, Schaapi could be integrated into the workflow of a library development team to help further validate the tool.

6

Conclusion

Deploying a change to a library used by thousands of projects is not straight-forward. Changes may break user projects at compile time or during execution, to the detriment of the users. Awareness of how a library is used exactly in practice would help library users maintain backwards compatibility, or at least proactively warn users if a breaking change is about to be deployed.

To this end, we have designed Schaapi, a tool for detection of semantic breaking changes based on library usage. For this thesis, we have designed a generic system design for the detection of these types of changes with a mining pipeline and a validation pipeline. We have implemented the full mining pipeline for Java-based projects and built a continuous integration server that offers feedback on pull requests on GitHub for library developers.

We have also validated the tool against a number of real-world projects. It has shown promising results for the detection of library-usage patterns in real-world user projects, as well as for the generation of tests capturing these patterns. Schaapi has also detected a number of artificially injected breaking changes in a test setup. Future work could extend Schaapi in both functionality and validation. We envision enhancements of the communication of usage patterns to library developers, improvements to the analysis of user code, and more realistic case studies.

Appendices

A

Project Overview

See the next page for a one-page information sheet for this project. →

SCHAAPI

Early detection of breaking changes based on API usage

SERG

Software Engineering Research Group
To be presented on *July 3rd, 2018*

TU Delft

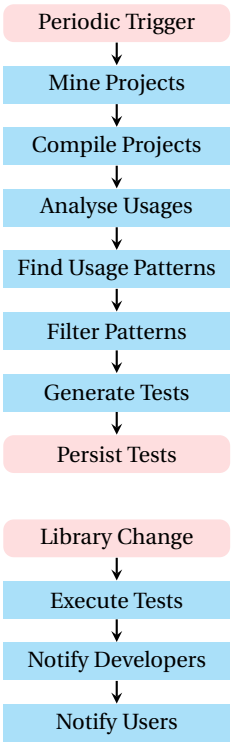
Schaapi ensures Safe Changes for APIs of libraries. It focuses on detection of semantic breaking changes, but provides a general-purpose pipeline that is also applicable to the detection of other types of breaking changes.

Problem Library developers are often unaware of how exactly their library is used in practice. When a library developer changes the internals of a library, this may unintentionally affect or even break the working of the library users' code. While it is possible to detect when a *syntactic* breaking change occurs, it proves to be more difficult to detect *semantic* breaking changes, where the implicit contract of a functionality changes, sometimes unbeknownst to the library developer. Because library users rarely test the behaviour they expect of the library, neither the library developer nor the library user will be aware of the new behaviour.

Product Schaapi allows library developers to see how a change in their library will affect their users before a new version of the library is deployed. It mines public repositories for projects using the library, analyses their usage of the library, and generates tests that aim to capture this behaviour. The tool also offers a continuous integration service that runs these tests against new versions of the library and warns developers of potentially breaking changes.

Research We have conducted an empirical validation of the tool with real-world data. This has shown that the tool can find a substantial number of patterns of non-trivial complexity in real-world user repositories. We also find that it is able to identify at least simple, artificially injected breaking changes.

Outlook Although Schaapi has already shown promising results, there are many possibilities for future work to extend the tool; from more extensive pattern filters to improved communication with library developers, and from historical to current real-world case studies with several libraries.



The Schaapi pipelines for mining (top) and continuous integration (bottom).

J.S. Abrahams
pattern detection, project mining, report

G. Andreadis
pattern detection, test generation, report

C.C. Boone
CI backend, library usage graph analysis

F.W. Dekker
bytecode analysis, pattern matching

Client: Dr. M. Aniche, *Software Engineering Research Group, TU Delft*
Coach: Dr. A. Katsifodimos, *Web Information Systems Group, TU Delft*

The final report for this project can be found at: `http://repository.tudelft.nl`

`info@cafejojo.org` `cafejojo/schaapi`

B

Original Project Description

Library developers are often unaware of how their library is used in practice. This can cause discomfort to library users, as their usage of the library may need to change significantly after a library upgrade, sometimes with unforeseen consequences. However, it also is an obstacle to library developers, as they lack a realistic picture of how interface changes will impact library users.

As a library developer, you want to see how a change would affect users before deploying, and notify affected users directly after deploying.

The goal for this project is to design and build a general pipeline for such a tool. In the process of building this pipeline, stage interfaces for "checker" modules need to be designed, a library of example implementations for those checkers needs to be added, and extensibility for other, custom implementations needs to be maintained.

C

Software Improvement Group Evaluation

These evaluations were provided in Dutch. Please contact the authors for an informal translation to English.

C.1. First Submission

De code van het systeem scoort 3.6 sterren op ons onderhoudbaarheidsmodel, wat betekent dat de code marktgemiddeld onderhoudbaar is. We zien Unit Size en Duplication vanwege de lagere deelscores als mogelijke verbeterpunten.

Voor Unit Size wordt er gekeken naar het percentage code dat bovengemiddeld lang is. Het opsplitsen van dit soort methodes in kleinere stukken zorgt ervoor dat elk onderdeel makkelijker te begrijpen, te testen en daardoor eenvoudiger te onderhouden wordt. Binnen de langere methodes in dit systeem, zoals bijvoorbeeld, zijn aparte stukken functionaliteit te vinden welke ge-refactored kunnen worden naar aparte methodes.

Een voorbeeld uit jullie project is `generalizedValuesAreEqual()` in `GeneralizedNodeComparator.kt`. Die methode wordt vanzelf wat langer door de grote hoeveelheid herhaling. Het stuk "`templateHasTag && !instanceHasTag`" komt bijvoorbeeld 4x voor. Door dit anders te implementeren zou de methode vanzelf korter worden.

Voor Duplicatie wordt er gekeken naar het percentage van de code welke redundant is, oftewel de code die meerdere keren in het systeem voorkomt en in principe verwijderd zou kunnen worden. Vanuit het oogpunt van onderhoudbaarheid is het wenselijk om een laag percentage redundantie te hebben omdat aanpassingen aan deze stukken code doorgaans op meerdere plaatsen moet gebeuren.

In jullie project zitten drie varianten van dezelfde class, `PatternDetector`. Het is ons niet duidelijk waarom deze class meerdere kopieën nodig heeft, maar dit op één of andere manier generaliseren zou de duplicatie verminderen.

De aanwezigheid van testcode is in ieder geval veelbelovend. De hoeveelheid tests blijft nog wel wat achter bij de hoeveelheid productiecode, hopelijk lukt het nog om dat tijdens het vervolg van het project te laten stijgen.

Over het algemeen is er dus nog wat verbetering mogelijk, hopelijk lukt het om dit tijdens de rest van de ontwikkelfase te realiseren.

C.2. Second Submission

In de tweede upload zien we dat het project een stuk groter is geworden. De score voor onderhoudbaarheid is in vergelijking met de eerste upload ongeveer gelijk gebleven.

Qua Duplication en Unit Size, de verbeterpunten uit de eerste upload, zien we beperkte verbetering. Jullie hebben de genoemde voorbeelden weliswaar aangepast, maar dit soort refactorings niet structureel doorgevoerd. Daarnaast zijn er weer nieuwe issues bijgekomen in de nieuwe code die sinds de eerste upload is toegevoegd.

Naast de toename in de hoeveelheid productiecode is het goed om te zien dat jullie ook nieuwe testcode hebben toegevoegd. De hoeveelheid tests ziet er dan ook nog steeds goed uit.

Uit deze observaties kunnen we concluderen dat de aanbevelingen uit de feedback op de eerste upload deels zijn meegenomen tijdens het ontwikkeltraject.

D

Ethical Implications

Because Schaapi interacts with and collects data from users, there is a number of ethical implications that we have taken into account during the design and implementation of Schaapi. Because Schaapi does not gather any non-public data, the most important concerns are about the interactions with its users.

In particular, a naïve implementation of the user notification system is ripe to abuse. For example, someone using Schaapi could forge an identity to look like a commonly used library and forcefully trigger a breaking change to allow them to notify users in an attempt to deceive them into introducing a new bug that will later allow the impostor to break into the user's application. Additionally, a naïve implementation could allow library developers to send inappropriate or undesired notices to their users. Therefore, users should have the ability to block notices, either completely or per project.

Finally, Schaapi will not always be correct and may give false positives or false negatives, especially considering its somewhat experimental state. False negatives may give the library developers a false sense of confidence that their changes did not affect any of their users, possibly causing library developers to roll out a breaking change that they would otherwise have questioned. False positives may result in library developers losing confidence in Schaapi's usefulness and may cause them to ignore the true positives. While this could mean that library developers publish changes that break their users, they would have done this without Schaapi as well and the only real effect is that Schaapi was unable to achieve its goal. A simple solution for both problems is to properly inform Schaapi's users of its capabilities and limitations.

Bibliography

[1] Mithun Acharya, Tao Xie, Jian Pei, and Jun Xu. Mining API patterns as partial orders from source code. *Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering - ESEC-FSE '07*, page 25, 2007. doi: 10.1145/1287624.1287630. URL http://portal.acm.org/citation.cfm?doid=1287624.1287630.

[2] Mauricio Aniche. RepoDriller. https://github.com/mauricioaniche/repodriller, 2018.

[3] Jay Ayres, Johannes Gehrke, Tomi Yiu, and Jason Flannick. Sequential pattern mining using a bitmap representation. *Proceedings of the eighth ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 429–435, 2002. doi: 10.1145/775107.775109. URL http://dl.acm.org/citation.cfm?id=775109.

[4] Luciano Baresi, Pier Luca Lanzi, and Matteo Miraz. TestFul: An evolutionary test approach for Java. *ICST 2010 - 3rd International Conference on Software Testing, Verification and Validation*, pages 185–194, 2010. ISSN 2159-4848. doi: 10.1109/ICST.2010.54.

[5] Len Bass, Paul Clements, and Rick Kazman. *Software Architecture in Practice*. Addison-Wesley Professional, 2 edition, 2003. ISBN 978-0321154958.

[6] Eric Bruneton, Romain Lenglet, and Thierry Coupaye. ASM: A Code Manipulation Tool to Implement Adaptable Systems. *Adaptable and Extensible Component Systems*, 2002. doi: 10.1.1.117.5769.

[7] Valerio Cosentino, Javier Luis, and Jordi Cabot. Findings from GitHub. *Proceedings of the 13th International Workshop on Mining Software Repositories - MSR '16*, pages 137–141, 2016. doi: 10.1145/2901739.2901776. URL http://dl.acm.org/citation.cfm?doid=2901739.2901776.

[8] Gordon Fraser and Andrea Arcuri. EvoSuite : Automatic Test Suite Generation for Object-Oriented Software. In *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*, pages 416–419, 2011. ISBN 9781450304436. doi: 10.1145/2025113.2025179.

[9] Eirini Kalliamvakou, Georgios Gousios, Kelly Blincoe, Leif Singer, Daniel M. German, and Daniela Damian. The Promises and Perils of Mining GitHub Categories and Subject Descriptors. *Proceedings of the 11th Working Conference on Mining Software Repositorie (MSR 2014)*, pages 92–101, 2014. doi: 10.1145/2597073.2597074.

[10] Patrick Lam, Eric Bodden, Ondrej Lhoták, and Laurie Hendren. The soot framework for java program analysis: a retrospective. In *Cetus Users and Compiler Infastructure Workshop (CETUS 2011)*, volume 15, page 35, 2011.

[11] Mario Linares-Vásquez, Gabriele Bavota, Carlos Bernal-Cárdenas, Massimiliano Di Penta, Rocco Oliveto, and Denys Poshyvanyk. API change and fault proneness: a threat to the success of Android apps. *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering - ESEC/FSE 2013*, page 477, 2013. ISSN 9781450322379. doi: 10.1145/2491411.2491428. URL http://dl.acm.org/citation.cfm?doid=2491411.2491428.

[12] Carlos Pacheco and Michael D. Ernst. Randoop: Feedback-Directed Random Testing for Java. *Companion to the 22nd ACM SIGPLAN conference on Object oriented programming systems and applications companion - OOPSLA '07*, 5:815, 2007. ISSN 0270-5257. doi: 10.1145/1297846.1297902. URL http://dl.acm.org/citation.cfm?id=1297846.1297902.

[13] Jian Pei, Jiawei Han, Behzad Mortazavi-Asl, Jianyong Wang, Helen Pinto, Qiming Chen, Umeshwar Dayal, and Mei Chun Hsu. Mining sequential patterns by pattern-growth: The prefixspan approach. *IEEE Transactions on Knowledge and Data Engineering*, 16(11):1424–1440, 2004. ISSN 10414347. doi: 10.1109/TKDE.2004.77.

[14] Mauro Pezze and Michal Young. *Software Testing and Analysis: Process, Principles and Techniques*. John Wiley & Sons, 2008. ISBN 9780471455936.

[15] Steven Raemaekers, Arie van Deursen, and Joost Visser. Semantic Versioning versus Breaking Changes : A Study of the Maven Repository. Technical report, TU Delft, 2014.

[16] Nick Rozanski and Eoin Woods. *Software Systems Architecture*. Addison Wesley, 2005. ISBN 0321112296.

[17] D C Schmidt, Michael Stal, Hans Rohnert, Frank Buschmann, and J Wiley. *Pattern-Oriented Software Architecture, Patterns for Concurrent and Networked Objects, Volume 2*, volume 2. John Wiley & Sons, 2000. ISBN 0471606952. doi: 10.1080/13581650120105534.

[18] K. Taneja and Tao Xie. Diffgen: Automated regression unit-test generation. In *Proceedings of the 2008 23rd IEEE/ACM International Conference on Automated Software Engineering*, ASE '08, pages 407–410, Washington, DC, USA, 2008. IEEE Computer Society. ISBN 978-1-4244-2187-9. doi: 10.1109/ASE.2008.60. URL `http://dx.doi.org/10.1109/ASE.2008.60`.

[19] Gias Uddin, Barthélémy Dagenais, and Martin P. Robillard. Analyzing temporal API usage patterns. *2011 26th IEEE/ACM International Conference on Automated Software Engineering, ASE 2011, Proceedings*, pages 456–459, 2011. ISSN 1938-4300. doi: 10.1109/ASE.2011.6100098.

[20] J. Wang and J. Han. BIDE: efficient mining of frequent closed sequences. *Proceedings. 20th International Conference on Data Engineering*, pages 79–90, 2004. ISSN 10636382. doi: 10.1109/ICDE.2004.1319986. URL `http://ieeexplore.ieee.org/document/1319986/`.

[21] Andrzej Wasylkowski, Andreas Zeller, and Christian Lindig. Detecting object usage anomalies. In *Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering - ESEC-FSE '07*, page 35, New York, New York, USA, 2007. ACM Press. ISBN 9781595938114. doi: 10.1145/1287624.1287632.

[22] Laerte Xavier, Aline Brito, Andre Hora, and Marco Tulio Valente. Historical and impact analysis of API breaking changes: A large-scale study. *SANER 2017 - 24th IEEE International Conference on Software Analysis, Evolution, and Reengineering*, pages 138–147, 2017. doi: 10.1109/SANER.2017.7884616.

[23] Tao Xie. Augmenting Automatically Generated Unit-Test Suites with Regression Oracle Checking. In *ECOOP 2006 – Object-Oriented Programming*, pages 380–403. Springer Berlin Heidelberg, 2006. ISBN 978-3-540-35726-1. doi: 10.1007/11785477_23. URL `http://link.springer.com/10.1007/11785477{_}23`.

[24] Mohammed J. Zaki. Efficiently mining frequent trees in a forest. *Proceedings of the eighth ACM SIGKDD international conference on Knowledge discovery and data mining - KDD '02*, page 71, 2002. ISSN 158113567X. doi: 10.1145/775047.775058. URL `http://portal.acm.org/citation.cfm?doid=775047.775058`.

[25] Jingsong Zhang, Yinglin Wang, and Dingyu Yang. CCSpan: Mining closed contiguous sequential patterns. *Knowledge-Based Systems*, 89:1–13, 2015. ISSN 09507051. doi: 10.1016/j.knosys.2015.06.014. URL `http://dx.doi.org/10.1016/j.knosys.2015.06.014`.