Towards Automated Library Migrations with Error Prone and Refaster

Rick Ossendrijver
rick.ossendrijver@gmail.com

December 23, 2021 61 pages

Academic supervisor:	Dr. Clemens Grelck, c.grelck@uva.nl
Daily supervisor:	Stephan Schroevers, MSc., stephan.schroevers@teampicnic.com
Host organisation:	Picnic Technologies, https://picnic.app

UNIVERSITY OF AMSTERDAM
FACULTY OF PHYSICS, MATHEMATICS AND INFORMATICS
MASTER SOFTWARE ENGINEERING
https://www.software-engineering-amsterdam.nl

Abstract

Software projects use a significant number of external libraries to speed up development and deliver high-quality software in a cost-effective way. Requirements placed on external libraries can change over time and libraries may cease to be maintained or evolve in incompatible ways. Consequently, a library may no longer be fit for purpose and require replacement. Such a *library migration* is generally performed manually and may be complex, error-prone, and time-consuming, especially for large codebases. It follows that an automated approach to library migrations is preferred.

We consider the suitability of Error Prone and Refaster, two Google tools capable of refactoring Java programs, for automated library migration. The former is a compiler plugin capable of emitting diagnostics and suggested code fixes, while the latter exposes a Java templating domain-specific language (DSL) with before-and-after rewrite semantics.

We propose extensions that improve the accuracy with which Refaster templates are matched, as well as the correctness of their replacement operations. We additionally propose a set of Error Prone extensions which facilitate rewriting method return types. Lastly, several generalizations and improvements of existing Error Prone plugins are discussed. Together these improvements enable a new class of rewrite operations, especially relevant to library migrations.

As a case study, the reactive programming library RxJava 2 is migrated to Project Reactor. We consider a private industrial codebase as well as an actively developed open source codebase. We show that using out-of-the-box Error Prone and Refaster full migration of RxJava 2-referencing files is possible in only 35% and 0% of such files, respectively. Using the proposed extensions, we manage to fully migrate an additional 35% of the RxJava 2-referencing files in the former codebase, and an initial 39% of such files in the latter codebase.

Contents

1 Introduction **3**

2 Background **6**
2.1 Static analysis . . . 6
2.2 Contextual knowledge during compilation . . . 7
2.3 Refactoring . . . 7
2.4 Error Prone . . . 9
2.5 Refaster . . . 11

3 Library migration automation with Error Prone and Refaster **13**
3.1 Select library migration . . . 13
3.2 Migration of RxJava to Reactor . . . 14
3.3 Refaster's support with RxJava to Reactor . . . 16
3.4 Error Prone and inlining deprecated methods . . . 19

4 Extensions to improve automation of library migration **22**
4.1 Extension for Refaster templates . . . 22
4.2 Return type extension . . . 25
4.3 Inlining method invocations in tests . . . 32
4.4 Extension Inliner plugin for method references . . . 34
4.5 Migration cleanup . . . 35

5 Validate impact of extensions **38**
5.1 Setup experiments . . . 38
5.2 Case study on an industrial codebase . . . 39
5.3 Case study on an open source repository . . . 41

6 Discussion **44**
6.1 Research question 1 . . . 45
6.2 Research question 2 . . . 45
6.3 Research question 3 . . . 46
6.4 Generalisability . . . 46
6.5 Future-proofness . . . 46

7 Related work **47**
7.1 Refactoring in IDEs . . . 47
7.2 Type changes . . . 47
7.3 Semantic patches . . . 47
7.4 Existing tools . . . 48
7.5 Library migrations . . . 48

8 Conclusion **51**
8.1 Threats to validity . . . 52
8.2 Future work . . . 53

Acronyms **56**

Bibliography **57**

Chapter 1

Introduction

Software projects heavily depend on software libraries for development [1]. Libraries allow developers to reuse software and cope with the expense of software evolution [2]. This means keeping up with the continuously evolving best practices around software development in general and library usage in particular. As a result, implementation cost are reduced, the quality of the software increases, and time is saved [3].

Since libraries are widely used, they play an important role in the quality and maintainability of a codebase. A library that offers an extensive feature that can be used with only a few lines of code can significantly improve quality and maintainability. For instance, the testing library JUnit[1] provides a sophisticated framework to write software tests. As a result, a developer can quickly start writing tests using a mature framework without having to create one. This can have a direct impact on the quality, maintainability, and cost of the software.

Generally, the libraries against which a product integrates are chosen during its development phase. These libraries should be updated periodically to resolve security vulnerabilities and improve overall code quality [4]. The relevance, suitability, and availability of libraries can change during the product's life cycle. Teyton et al. defined the following categories of reasons for library migration [1]:

- **convenience**: improved or new features;
- **outdated**: a library may be outdated or no longer maintained;
- **incompatibilities**: to prevent conflicts between libraries and code;
- **license**: there may be license incompatibilities [5].

Consequently, software developers need to periodically reconsider the set of libraries they depend on and replace existing integrations where necessary or beneficial [2, 6]. This is called *library migration.*

A definition by Teyton et al., "*library migration refers to the migration of a project from one library to another where the project is dependent on the library*" [6]. Hussein et al. add "*while preserving the same program behavior*" [2].

How cumbersome and time-consuming it is to manually migrate a specific library depends on many factors. For instance, it depends on what library one is migrating, how many people work on the project, the size of the codebase, how widespread the usage of the library is, as well as the project's workflow.

When a migration mostly consists of renaming a method or variable throughout the codebase, which can be the case with logging libraries, a *search and replace* operation in an IDE or editor can be used to migrate all library calls at once. However, typically, a migration consists of multiple types of refactorings which makes it more difficult.

Refactoring is a common term for changing software to improve readability and maintainability without changing the external behavior of the code [7, 8, Ch.0]. The definition of refactoring overlaps with the definition of library migration, as defined by Hussein et al.: library migration is "*the process of replacing a library with a different one while preserving the same program behavior*" [2]. In both definitions, *ensuring behavior preserving changes* is essential. However, we argue that library migration is a specific type of refactoring. Library migration has a specific goal in comparison to refactoring. With refactoring for readability and maintainability, it is negotiable what the impact of the changes is, as it is open to interpretation whether the result of a rewrite operation is more readable. The end goal of a migration is more narrowly defined because one can verify whether any references to the old library are left.

[1] https://junit.org/

Library migration is generally a time-consuming manual process. A vast amount of code may need to be rewritten in order to completely migrate a large codebase. When a library is used, it is not uncommon that references to its API are scattered throughout the codebase. This generally makes library migration an error-prone and complex task [6, 9, 10]. Besides having to rewrite the code, the developer needs to comprehend the terminology, structure, and semantics of the old and of the new library.

In practice, library migration is more than just performing the migration in the code. After rewriting the code, it must go through the code review process, which can be time-consuming as well, especially when the change is large. The quality of the review decreases because the useful comment density decreases as the number of files increases [11]. Additionally, "*the review latency increases as the size of the change increases*" [12]. As a result, the quality of the migration is negatively impacted.

Therefore, automating library migration or parts thereof can have a major impact on time spent, cost, and quality. Existing research focuses mostly on mining migrations to establish a mapping between libraries [13, 14]. Tools that automatically refactor code are mostly researched in the context of library evolution, not migration [15–17].

We use the tool Refaster [18] to investigate how the automation of library migration can be improved. Refaster is a tool that allows one to define rewrite rules in Java using before and after-templates. It scans the codebase for code that matches the structure of the before-template. Matching code is subsequently rewritten in accordance with the associated after-template. Using such templates Refaster can apply changes to an entire codebase in a manner of seconds or minutes. Currently, Refaster can only be used to rewrite Java expressions and statements; it does not support modifying other source code constructs such as annotations, variable declarations, return types, method names, or method signatures.

Refaster is built on top of Error Prone [19]. Both tools are built by Google and hosted in the same GitHub source code repository[2]. Error Prone is a Java compiler plugin which analyzes the code's Abstract Syntax Tree (AST) as created by the compiler. The aim of Error Prone is to identify bugs and anti-patterns and, where possible, produce suggested fixes. Any issues found are flagged by emitting a compilation warning or error. Additionally, Error Prone can produce patch files describing suggested fixes, optionally directly applying the changes to the codebase.

Our goal is to explore the how tools Error Prone and Refaster can support the automation of library migration in Java and extend research on this. To the best of our knowledge, the impact of tools on library migration automation has not yet been studied. To assess this, we start by exploring how Error Prone and Refaster can currently support library migration. At the same time, we identify some limitations of these tools. We investigate this by looking at a specific library migration between two reactive programming libraries: from RxJava 2 to Reactor. Based on our experience with this migration, we propose several extensions and generalizations for Error Prone and Refaster. For instance, we introduce an extension to automatically rewrite the return type of declared methods. Together with the existing inlining functionality of Error Prone this allows us to migrate almost all callers of the aforementioned methods to Reactor. Additionally, we propose several extensions to improve the accuracy with which Refaster templates are matched as well as the correctness of their replacement operations.

To validate and measure the impact of the extensions, we conduct experiments involving two codebases. We choose an industrial codebase and an open source codebase to ensure the validity of the research. The industrial codebase is a private repository provided by Picnic. The codebase contains the core services and exists of more than 165.000 lines of code. For example, the core provides services for users, orders, deliveries, articles, and promotions. The other repository is called Gravitee [20]. Gravitee.io Access Management is an open source Identity and Access Management solution that offers a centralized authentication and authorization Service for applications and APIs [20]. The codebase consists of more than 265,000 lines of code.

The experiments identify how much RxJava code we can automatically migrate to Reactor. First, we use stock Error Prone and Refaster to migrate as much RxJava usages from the codebase as possible. Second, we use the proposed extensions to migrate more RxJava code. After the automated part of the migration is complete, the different states can be compared to collect data on how far we can push the migration.

The fork of the open source repository that we use for conducting the experiment is publicly available on GitHub[3].

[2]Error Prone: `https://github.com/google/error-prone`
[3]Gravitee fork: `https://github.com/rickie/gravitee-access-management`

In this thesis, we answer the following research questions:

- **RQ1**: To what extent can Error Prone and Refaster support automation of library migration?
- **RQ2**: What extensions of Error Prone and Refaster benefit library migration?
- **RQ3**: What is the impact of the extensions on automation of library migration?

Our research makes the following contributions:

- We gain insight into how Error Prone and Refaster can support automation of library migration.
- We propose multiple Error Prone and Refaster extensions that improve library migration automation.
- We make various contributions to the Error Prone base code, consisting of generalisations and minor bug fixes of existing checks.
- We create a significant number of Refaster templates and migration templates to migrate from RxJava 2 to Reactor.

All contributions to Error Prone and Refaster can be found in a fork of the GitHub repository[4]. The body of Refaster templates is also available on GitHub[5].

The remainder of this thesis is organised as follows: in Chapter 2 we describe the background of this thesis. Chapter 3 describes how the current features of Error Prone and Refaster can support a library migration. Chapter 4 describes the Error Prone and Refaster extensions. The validation, experiment, and results are described and discussed in Chapter 5. Chapter 7, describes the work related to this thesis. Finally, we present our concluding remarks in Chapter 8 together with future work.

[4]Error Prone fork: `https://github.com/PicnicSupermarket/error-prone`

[5]Refaster templates: `https://github.com/rickie/msc-thesis-source-code`

Chapter 2

Background

This chapter presents some background information for this thesis. The background focuses on the Java ecosystem to provide better context on the topic. First, we start by explaining static analysis. Next, we describe contextual knowledge that is available during compilation. Then we provide context on refactoring. Finally, we look into tools Error Prone and Refaster are and how they work.

2.1 Static analysis

Static analysis tools aim to find bugs and weaknesses in the source code of a system [21, 22]. Currently, many static analysis tools exist with the intent to detect bugs early in the software life cycle. Correcting an error that is detected early in the software development life cycle can improve productivity as the cost of correcting an error is reduced, resulting in lower development costs [22].

It is common practice to combine several static analysis instruments. This is because tools often excel at solving different types of problems. A few benefits of using source code are low labor costs because it can run automatically [21]. In addition, the tools can be integrated in the developers workflow.

For static analysis in Java, the source code and the bytecode can be analysed [23]. For the thesis, we will use a static analysis tool for Java source code called Error Prone. Before we explain this tool, we will provide context on static analysis and what knowledge is available during compilation.

Source code analysis

Source code analysis works directly on the source code of a program [23]. Two well-known source code analysis tools for Java are CheckStyle and PMD. CheckStyle analyses whether the source code adheres to the specified coding standard [24]. PMD tries to find "*common programming flaws, like unused variables, and empty blocks*" [25]. Both tools are capable of identifying a wide variety of flaws. However, there are limitations to using source code analysis tools. Since the provided context is limited, it is sometimes hard to produce accurate results. Additionally, "*because static analysis attempts to predict behavior based on models of the source code*" this can lead to false positives. Together, these limitations can lead to a high amount of false positives [21].

Bytecode analysis

Compiling Java source code results in bytecode, which can be executed by a Java Virtual Machine (JVM). Bytecode analysis works directly on the compiled bytecode of a program [23]. Compared to analyzing the source code, this analyzes code that is closer to what is actually executed by the JVM [26].

During compilation, the compiler optimizes the code for execution. As a result, the resulting bytecode cannot, in general, be traced back to its original place in the source code [23]. This can make it harder to fix problems that are found in the bytecode. We give an example to show how hard it can be for static analysis tools, that analyze bytecode, to know the location of the bug. A Java example where the bytecode is significantly more verbose than the source code is the *Try With Resources* statement. The Java Language Specification (JLS) specifies how a basic *Try With Resources* statement can be written in Java. The statement consists of two lines of code. Under the hood, the compiler translates the single *Try With Resources* statement to three blocks, a *try*, *catch*, and *finally* block [27]. Suppose there is an exception in the *finally* block. Since the source code does not contain a *finally* block, it is difficult to trace back the error in the original code. This is a disadvantage of bytecode analysis.

An advantage of analyzing the bytecode is that it can find issues that were not visible or present during source code analysis. In addition, other checks and knowledge are available in the bytecode. For instance, the compiler performed type checking and name resolutions [23].

SpotBugs (previously FindBugs) is a well-known bytecode analyzer [28, 29]. SpotBugs looks for instances of "*bug patterns, code instances that are likely to be errors*" [29]. A few examples of such bug patterns are null pointer dereferences, casting errors, and infinite loops [28].

2.2 Contextual knowledge during compilation

The aforementioned analysis tools either focus on a program's source code or the bytecode produced by the compilation process. What sets Error Prone apart is that it hooks into the compilation process and thereby has access to both a program's source code and the compiler's intermediate representation (IR). The IR contains structural and semantic information at least on par with the information contained in a program's bytecode. As such Error Prone is able to leverage "the best of both worlds".

A compiler starts compiling source code and creates an Abstract Syntax Tree (AST). An AST is a tree representation of the source code. The AST consists of various *nodes*, where each node represents a "construct" that is in the source code [30]. For instance, a class node contains method nodes, and a method node contains statement nodes. The tree is an *abstract* representation of the code, which means that irrelevant information, such as white space, is usually left out. An advantage of a tree over a text-based source is that a "*tree is more convenient and reliable to analyze and modify programmatically*" [30].

Additionally, a Concrete Syntax Tree (CST) can be created during the compilation. A CST is a one-to-one mapping from the grammar in the input program to a tree representation [31, 32, ch.5]. The CST ensures that the exact location of a node in the tree can be traced back to the actual source code.

In the case of Error Prone, the CST *and* the AST are created, which makes it a powerful tool. How Error Prone uses these trees and what we can do with them is explained in more detail in Section 2.4.

Finally, we discuss data-flow analysis. Data-flow analysis "*is a process of deriving information about the run time behavior of a program*" [33]. By looking at the "flow" of the *data* in a program, it is possible to derive information about the use of data items in a program [34]. For instance, where methods are called, how variables are used, what might be affected by changing a variable, and whether data is used before or after a specific state in a program [34]. Data-flow analysis is supported by Error Prone but in a limited way, which can result in even more information about the execution of the program. Error Prone integrates the checker-framework [35] which allows for data-flow analysis.

To illustrate what can be achieved with compile-time analysis, we provide some examples. With such analysis, we can [36]:

- enforce dependency constraints between components (e.g. to reduce the size of JARs),
- prevent anti-patterns from occurring in the codebase,
- enforce style rules.

2.3 Refactoring

Library migration is a form of refactoring. The origin of refactoring goes back to one of the first books on refactoring that was written by Martin Fowler [8]. Martin Fowler stated that it is not known by whom or when the term refactoring was coined [8, Ch.2]. The definition Martin Fowler provides is "*the process of changing a software system in a way that does not alter the external behavior of the code yet improves its internal structure*" [8, Ch.0].

Since refactoring is a widely used term, it is hard to capture the essence in one definition. Therefore we also provide a second definition. Silva et al. define refactoring as "*a widespread practice that helps developers to improve the maintainability and readability of their code*" [7].

Much research is done on the reasons why we refactor software; we list the most occurring reasons [7, 8, 37–40]:

- to improve the readability,
- to prevent the code from becoming overly complex (comprehensibility),
- to make the code more modifiable (extensibility),
- to improve the reusability of the code,
- to improve the performance of the system.

Improving these separate characteristics of the code improves the overall *maintainability* of the code. Better maintainability then ensures faster development of new features.

If code is not refactored, it can *decay* and become "*harder to change than it should be*" [41]. The world changes, the developers gain experience and have a better understanding of the software. If the software design is not improved accordingly, and the code is not maintained, the code will decay and no longer represent the environment. This will make the codebase unnecessarily complex, unreliable, and harder to add new features [42]. To prevent this, time needs to be invested in the codebase. Regular refactoring helps keep the code in shape. Refactoring should be a programmer's habit [8, Ch.2], which helps to deal with software evolution. The software evolves together with the environment and it will be easier to add new features and maintain the codebase.

Incremental changes

The goal of refactoring is to improve the maintainability of the code. However, refactoring has drawbacks because it can be a risky task.

During the life cycle of a project, developers join and leave. When a developer works on the project for a long time, specific knowledge of the system is gathered. Important knowledge of the project is lost when a developer leaves the project [43]. This makes it harder to maintain the project. When someone hasn't worked on the project for a long period, the limited understanding of the project can cause productivity and product quality to decline [44]. It is hard to find out how a program works and what it does. This makes each attempt to change it a possibility to introduce mysterious bugs [45]. For instance, a developer can make a small change in the code with the expectation that it does not have a significant impact. However, the change has a system-wide impact that was unnoticed because of the poor readability of the system. Verification tools can support in ensuring the quality of a program, especially after doing a localized change [42]. This can be done by running the whole verification on the program. Rerunning the verification after every improvement can help the developer identifying bugs [8]. This is a good way to identify bugs in the code and it prevents the need for the developer to debug the code, which can save time.

Doing multiple small incremental refactorings is better than doing one large refactoring. When the refactor becomes larger, it is more likely that the way of thinking about the problem will radically change [43]. Thus it will be harder to review and harder to assess whether the refactor improved the readability. Fowler adds "with refactoring, small steps are the best; that way less tends to go wrong" [8, Ch.2]. Fowler explains that a developer can compile and run the test after each refactoring for verification [8, Ch.2]. Logically it is important to have a significant number of tests that can identify mistakes.

Syntax and semantics

A programming language has two fundamental features: syntax and semantics. The syntax can consider the logical meaning of sentences and validate the grammar and correctness of sentences [46, Chapter 3]. Semantics is focused on the meaning of these programs [47, Chapter 1].

In the definition of refactoring by Fowler [8], the *external behavior* of the program is mentioned. Changing the program without changing the external behavior can be done in two ways. First, by simply changing the syntax to make a code change that is behavior preserving. For instance, improving the code style to adhere to coding standards. Second, changing the semantics of the code to improve the internal behavior of the code. For example, migrating a library can have a major impact on the code while not changing external behavior.

Tools

Software maintenance was identified as a common problem decades ago and since then much research is done on this topic [43–45, 48, 49]. Especially for large software systems, software maintenance is time-consuming and therefore costly [43, 48, 50]. Since the problem is identified, much research is done to see how tools can solve the problem [9, 15, 51, 52]. The goal is to reduce costs and improve the process. To cope with the evolution of large-scale systems tools can have a tremendous impact. An example where tools can have such an impact is Google. At Google, they have a monolithic repository with 2 billion lines of code [53]. They built multiple tools to continuously analyze and automatically refactor their codebase. To handle all the analysis, they created a Program Analysis Ecosystem Tricorder [54], which integrates refactoring tools such as ClangMR [55] and Error Prone [19]. We will explain what Error Prone is and how it works in the next section.

2.4 Error Prone

One of the tools that will be used for the thesis is Error Prone, which is a compile-time static analysis tool for Java [19]. The goal of Error Prone is to catch common Java mistakes as compile-time errors. Error Prone is created by Google and open-sourced on GitHub[1]. Error Prone defines a set of checks that can be configured by the developer. For instance, each check can be turned on or off and warnings can be set to show as errors. Error Prone is actively maintained and improved.

The compiler does much valuable analysis and creates data structures (e.g. an AST) to assess the validity of the code and run the code for a given language. Thus, according to Aftandilian et al. the compiler "*is in the best position to understand source code*" [36]. Since the compiler has to do its calculations anyway, using this information at compile-time will not affect performance.

Aftandilian et al. explain the implementation of Error Prone in detail [36]. Error Prone is implemented on top of the OpenJDK javac compiler as an additional compiler pass. Right after the *flow* phase of the compiler, the error checking pass of Error Prone is performed. This means that type checking is already done and all the symbol information is available together with the AST. At this point in compilation, "*all errors detected by the standard javac have been reported.*" [36]. If Error Prone finds anything, it will be presented as compiler errors.

Error Prone plugins

In Error Prone one can create plugins, which are called *BugPatterns* in the code. These plugins will run during compilation. A new compile-time plugin in Error Prone must adhere to the following criteria [36]:

- be easily understandable,
- actionable and easy to fix,
- produce no effective false positives,
- and report issues that only affect correctness.

Additionally, Error Prone "*provides utility for deleting whole AST nodes, replacing or swapping nodes, reflowing overly-long lines, and adding or removing import statements when necessary*" [36].

It is important to note that information about the syntax is available in Error Prone. Within Error Prone a *tree* is provided which contains the exact information about the source code. We can compare this with the node from a CST. In Error Prone we can retrieve a so called *symbol* based on the tree. This *symbol* is a more abstract representation of the tree and does not know specifics of the syntax. However, the *symbol* has properties that can be easily queried. A *symbol* can be seen as a node from the AST. Using the CST and the AST it is possible to find improvements in the code and update the existing source code accordingly.

A BugPattern in Error Prone uses the visitor pattern to traverse the CST. This means that Error Prone will go over the various nodes of the CST and visit them in a specific order. Error Prone will go over each source file and use the BugPatterns to check the code for flaws. In a BugPattern a developer can implement one or more `TreeMatchers` of the node that should be visited in the check. Suppose we want to write a BugPattern that checks whether a method body is empty, and flags the method if this is the case. In the BugPattern we would implement the `MethodTreeMatcher` and override the method `matchMethod(MethodTree tree, VisitorState state)`. In this method we get the information of the method that is currently matched. To illustrate what data can be retrieved from a `MethodTree` the interface is shown in Listing 2.1. The comment behind each method in Listing 2.1 explain what data is returned after calling it. To find out whether the body is empty, we can ask the `MethodTree` for the `BlockTree` by using `getBody()`. A BlockTree consists of a `List<Statement>` which we can query by using `.isEmpty()`. This will tell us whether the body of the current method contains statements. Additionally, we can use the `MethodTree` combined with the `state` to retrieve other information from the AST. The AST allows us to retrieve more information then just the method. For instance, we can ask the AST to provide the node of the enclosing class of the method or what the annotations on the method are.

A BugPattern contains a description on how to solve the problem, in Error Prone this is called a

[1]https://github.com/google/error-prone

```java
public interface MethodTree extends Tree {
   ModifiersTree getModifiers(); // The modifiers of the method (e.g.‘
      public’, ‘static’ or ‘default’.

   Name getName(); // The name of the method

   Tree getReturnType(); // The return type of the method

   List<? extends TypeParameterTree> getTypeParameters(); // The
      typeparameters of the method (e.g. ‘T’ when the returntype is ‘
      Single <T>’)

   List<? extends VariableTree> getParameters(); // The parameters of
      the method

   VariableTree getReceiverParameter(); // The VariableTree of the
      receiver parameter

   List<? extends ExpressionTree> getThrows(); // The checked
      exceptions of the method

   BlockTree getBody(); // The body of the method
}
```

Listing 2.1: Partial content of the MethodTree interface [19]

SuggestedFix. A suggested fix is not required to be semantics-preserving [36]. When the developer is working on the code, the detected problems with suggested fixes are presented as compiler errors to the user. This means that Error Prone is completely integrated into the developer workflow [56]. Since these errors will prevent the code from compiling, the errors cannot enter the codebase. This is a major advantage of integrating compile-time analysis into the developer's workflow. Additionally, using the suggested fix, Error Prone can automatically repair errors at a large scale by patching the code. The developer can specify what rules should run and patch the code. As a result, Error Prone can provide good support in the process of Software Evolution.

Throughout the thesis, we use the term *plugin* rather than *BugPattern*.

Limitations

There exist many builds of the OpenJDK, which is the open source reference implementation of the Java SE Platform [57]. For instance, Oracle JDK, OpenJDK, AdoptOpenJDK, and Red Hat OpenJDK.

As explained by Aftandilian, when Error Prone was created, Google used its own compiler implementation of OpenJDK [36]. The reason for this is that the public API of the compiler did not provide enough functionality for non-trivial static analysis. Since they are using an internal build of the OpenJDK at Google, no-one else is relying on the specific implementation of the API. This means that Error Prone uses an internal representation of the AST. They can decide to change the internals of the API at any time. This can result in a significant number of unexpected breaking changes and compatibility issues. Especially when we compare the use of an internal API with the use of a public API, such as the Eclipse JDT, a standalone Java compiler. The Eclipse JDT is publicly available on GitHub[2] and is used throughout the Eclipse ecosystem [58]. Introducing a (breaking) change can have an enormous impact because the JDT is widely used. Therefore, it is carefully considered before introducing such a change.

On the GitHub repository of Error Prone, many issues show compatibility issues with different versions of the JDK. If Error Prone used a publicly available version of the OpenJDK, these incompatibility issues are less likely to occur.

[2]https://github.com/eclipse/eclipse.jdt.core

2.5 Refaster

Refaster is a tool to automatically refactor Java code. It supports template-based refactoring at compile-time using before-and-after templates, which we call Refaster templates [59]. Powerful features of Error Prone are used for the execution of the Refaster templates.

Listing 2.2 shows an example of a Refaster template. Refaster provides a DSL that consists of compilable Java code but is never *executed*. The template is compiled and the required information is extracted. Using Java for the templates makes it easy, accessible, and familiar for Java developers to define refactorings. A template consists of at least one *before*-template and one *after*-template, which is indicated by the annotations `@BeforeTemplate` and `@AfterTemplate`. The goal of Refaster is to find code that matches the expression that is provided in the before-template. For each method of the template, there is a parameter `String string`. Refaster uses the parameter when traversing through the AST to find expressions of type `String` in the code that also match the structure of the before-template. When a match is found, Refaster will automatically refactor the code as described in the after-template.

Listing 2.2 shows a Refaster template that identifies any `String`-typed expressions on which the method `equals` is invoked with the argument ””, and replaces said method call with an invocation of the nullary method `isEmpty`. The result of such a rewrite operation is illustrated in Listing 2.2.

```java
import com.google.errorprone.refaster.annotation.AfterTemplate;
import com.google.errorprone.refaster.annotation.BeforeTemplate;

public class StringIsEmpty {
  @BeforeTemplate
  boolean equalsEmptyString(String string) {
    return string.equals("");
  }

  @BeforeTemplate
  boolean lengthEquals0(String string) {
    return string.length() == 0;
  }

  @AfterTemplate
  boolean optimizedMethod(String string) {
    return string.isEmpty();
  }
}
```

Listing 2.2: An example of a Refaster template [59]

```java
// Running the Refaster template on this code:
boolean b = methodCall().returningAString().equals("");

// Will result in an automatic refactor to:
boolean b = methodCall().returningAString().isEmpty();
```

Listing 2.3: Example refactor based on the Refaster template of Listing 2.2

The example shown in Listing 2.2 is fairly simple. Refaster allows for more complicated refactorings. For instance, it supports the use of generic types, expression placeholders, and the specification of import policies.

Features

According to the documentation, Refaster is useful for the following types of refactoring [59]:

- Migrate usages of method A to method B.
- Migrate usages of method A with an arbitrary method signature to method B.
- Migrate a particular fluent sequence of method invocations to some other pattern.
- Migrate a sequence of consecutive statements to an alternative.

Refaster aims to support "*a large class of useful refactorings that can be expressed in pure Java, without a specialized domain-specific language (DSL), while keeping the tool easily accessible to average Java developers*" [18].

Besides *usability*, the key goals of Refaster are *scalability* and *expressiveness* [18].

Limitations

A downside to Refaster being fairly simple to understand is that the number of features in Refaster is limited. Currently, Refaster can only be used to refactor expressions and statements. Consequently, Refaster does not support modifying annotations, access modifiers, return types, method names, or method signatures. Unlike Refaster, Error Prone does support these types of refactorings. The problem is that Refaster has no way of defining these refactorings. In other words, the expressivity of the tool is limited.

Chapter 3

Library migration automation with Error Prone and Refaster

We want to use Refaster and Error Prone, with their current set of features, to automate part(s) of library migration. In this chapter, the goal is to identify ways in which the tools can support a specific migration. First, we research how a migration is performed and what refactoring operations are required. Per operation, we show whether the tools currently support the refactoring operation. Subsequently, we identify operations where Error Prone and Refaster do not provide features to support the migration. To study this, we want to research a common migration.

3.1 Select library migration

We found that RxJava is commonly used in the industry. RxJava is a library that provides reactive extensions to allow for reactive programming [60]. Reactive programming has "*gained popularity as a paradigm that is well-suited for developing event-driven and interactive applications*" [61]. RxJava has over 45.400 stars on GitHub[1] at the time of writing (02-10-2021). In 2014, Teyton et al. found that Spring is amongst the most used libraries [6]. In addition, the Stack Overflow Developer Survey of 2021 shows that Spring is the most used library for Java-based enterprise applications [62]. So the Spring framework is popular and commonly used. In the Spring ecosystem, an equivalent of the RxJava library is available, namely Project Reactor [63]. Since Reactor is integrated with the Spring framework, migrating from RxJava to Reactor is a logical choice. Besides that, Reactor has a newer implementation and is not focused on backward compatibility but on newer Java versions. Whereas the goal of RxJava is to be backward compatible and support older Java versions. Since Spring and RxJava are not uncommon libraries in the industry, this will not be an uncommon or illogical migration.

For the research, we set the scope on one specific migration. This is because focusing on multiple migrations can make it hard to produce accurate results. Since library migrations can differ greatly from each other, different refactoring operations are required to migrate from one to another. For example, consider the migration from TestNG to JUnit. Which was one of the most used libraries in 2018, according to Kula et al. [4]. A migration between these libraries mainly consists of adding, modifying, and removing annotations. While in another library migration, for example, JUnit to AssertJ, there is no need to do anything with annotations. Therefore, it is hard to find migrations that require similar refactoring operations. As a result, we decide to focus on one library migration.

To validate the research, we want to do a case study. Within the host company RxJava 2 is extensively used. While the newest version of RxJava is 3. As a result, RxJava 2 reached end of life status in March 2021[2] and will receive no more updates. Therefore the RxJava 2 code needs to be updated either to the new RxJava version or to another library, for instance Reactor. The host company uses the Spring framework for most applications. In some places, the code already uses Reactor instead of RxJava. However, RxJava is still widely used throughout the codebase, while Reactor is the preferred library. There is a significant body of RxJava code to perform a case study on automating this to Reactor.

It is important that not every part of the migration can already be automated with Refaster. The goal is to create an extension for Refaster and Error Prone that can improve automation. Therefore, the

[1] `https://github.com/ReactiveX/RxJava`
[2] `https://github.com/ReactiveX/RxJava/issues/7197`

library migration must have multiple options for improvements of the automation.

In summary, the migration of RxJava to Reactor is not an uncommon choice in the Java ecosystem and the migration has yet to be done in the Picnic codebase. Refaster cannot perform a full migration with the current features, so there are many opportunities for improvements. In the next section, we want to investigate what the migration looks like and which parts can be automated.

3.2 Migration of RxJava to Reactor

The APIs of both libraries are extensive. The 5 primary types of RxJava contain over 1000 public methods in total[3]. Most of the methods have an equivalent in Reactor. However, some methods do not have an interchangeable equivalent. The documentation of both libraries provides detailed information on the libraries and their methods. Using the documentation, we can identify and verify whether the mapping from an RxJava method to Reactor method is correct.

To see how a migration is performed in the industry, we examine pull requests from RxJava to Reactor migrations that have already been performed. Examining already performed migrations shows what a migration entails.

Primary types

In RxJava and Reactor there are primary types that represent the core of the library. These are the primary types of RxJava [60]:

- Flowable: 0..N flows, with the option for consumers to signal producers (backpressure),
- Observable: 0..N flows, no option to signal the producers (no backpressure),
- Single: a flow of exactly 1 item or an error,
- Completable: a flow without items but only a completion or error signal,
- Maybe: a flow with no items, exactly one item, or an error.

RxJava and Reactor have quite similar APIs. To encourage migrating from one to the other, there is an additional module that contains an adapter class named `RxJava2Adapter` [64]. Its adapter methods allow for easy conversion of RxJava types to corresponding Reactor types, and vice versa. Consequently, the interoperability between the libraries is substantial, enabling a codebase to be gradually migrated from one library to another.

That said, the intermediate steps of such a migration are not without cost. Adapter usages introduce boilerplate code and the concurrent use of both libraries increases cognitive load, for example by requiring developers to be familiar with both APIs.

Using the methods of the `RxJava2Adapter`, we can easily detect what the mapping of the types is. In Listing 3.1, an example is provided on how to convert an object from RxJava to Reactor and from Reactor to RxJava. To see how the base types of RxJava are mapped to Reactor, see Table 3.1.

```java
// Create a Flowable
Flowable<Integer> flowable = Flowable.just(1);

// Convert the Flowable to a Flux
Flux<Integer> flux = flowable.as(RxJava2Adapter::flowableToFlux);

// Convert the Flux back to a Flowable
Flowable<Integer> originalFlowable = flux.as(RxJava2Adapter::
    fluxToFlowable);
```

Listing 3.1: Using the `RxJava2Adapter` to convert between RxJava and Reactor

[3]`https://github.com/ReactiveX/RxJava#base-classes`

RxJava	**Reactor**
Flowable	Flux
Observable	Flux
Single	Mono
Completable	Mono
Maybe	Mono

Table 3.1: The mapping of the primary RxJava types to the primary Reactor types

Code changes

Many operators are defined on each of the primary types. For example, the `Flowable` class contains methods like *empty(), fromArray(), just(), zip(), blockingFirst(), collect(), map(), and flatMap()*. Each operator that is used, must be migrated to the equivalent of the matching Reactor type. This can sometimes be difficult if there is no direct mapping to a Reactor operator.

In Listing 3.2 we give an example of how an RxJava method can be rewritten to a Reactor method. Multiple things are happening for the rewrite of a method. We can see that the RxJava type *Single* is rewritten to the Reactor type *Mono* in the method body. The *map* and *switchIfEmpty* operators on the *Single* type, are the same for the *Mono*. However, the *flatMapMaybe* needs to be rewritten to a *flatMap*. Finally, the return type of the method is changed from *Single<Address>* to *Mono<Address>*.

```java
// The RxJava method
public Single<Address> getAddress(String customerId) {
  return Single.just(customerId)
    .map(AddressServiceImpl::customerIdToAddress)
    .flatMapMaybe(addressService::complete)
    .switchIfEmpty(Single.error(new ItemNotFoundException(customerId));
}

// The same method migrated to Reactor
public Mono<Address> getAddress(String customerId) {
  return Mono.just(customerId)
    .map(AddressServiceImpl::customerIdToAddress)
    .flatMap(addressService::complete)
    .switchIfEmpty(Mono.error(new ItemNotFoundException(customerId));
}
```

Listing 3.2: Complete rewrite of an RxJava method to Reactor

Important to note is that most changes also imply changes to the import statements. For instance, replacing a *Flowable* with a *Flux* means removing `import io.reactivex.Flowable` and inserting `import reactor.core.publisher.Flux`.

After changing the implementation of a method, it is time to update the callers of the method. Every line of code where the original method is called, an object of type *Single* is expected. To make sure the code compiles, each caller must be updated accordingly. In Listing 3.3, there is an example on the first line of such a caller.

In the migration, there is one last rewrite operation that often occurs. It is common in Object-Oriented Programming to use the Facade pattern. A facade defines a higher-level interface of a complex system which makes it easier to use [65, Ch.4]. This will result in the system being easier to use, promotes weak coupling and reusability, and promote the independence of subsystems and portability [65, Ch.4]. However, the migration we have described so far does not take into account method declarations in interfaces. If we would rewrite a method and its callers, the code would not compile if the interface is not updated accordingly. In Listing 3.4 we provide an example of the rewrite operation that is required to update the method definition in the interface.

```java
// Arbitrary code that calls the RxJava method 'getAddress(String
    customerId)'.
Single<String> streetName = getAddress("1")
    .map(Address::getStreetName);

// Now assume that the return type of the RxJava method is changed to
    the Reactor equivalent type 'Mono'.
// Then the following rewrite is required:
Single<String> streetName =
    RxJava2Adapter.monoToSingle(getAddress("1"))
        .map(Address::getStreetName);
```

Listing 3.3: Rewrite callers of RxJava method to handle the new Reactor return type

```java
public interface AddressService {
  // The declaration with an RxJava type
  Single<Address> getAddress(String customerId);

  // Will be replaced by this declaration using a Reactor type
  Mono<Address> getAddress(String customerId);
}
```

Listing 3.4: Rewrite RxJava method in interface to Reactor

Unit Tests

Updating the callers of a migrated method consists of two parts. The first part consists of updating the callers in non-test code. In Listing 3.3 is shown what updating the callers of non-test code looks like. However, after that, it is likely the tests for that code will break. This is a problem because the tests are in place to ensure the quality of the code.

Rewriting the tests to handle the new return types usually does not require the same rewrite operations as in Listing 3.3. Zerouali & Mens researched the usage of test libraries on a body of open source Java projects on GitHub. They found that 97% of the projects use JUnit [66]. Therefore we can say that it is not uncommon that testing libraries are used in Java projects. Additionally, Zerouali & Mens found that different testing libraries are often combined in one project. When JUnit is used, it is most likely this is combined with the *Mockito*[4] library [66]. Mockito is a mocking framework for unit tests written in Java. Writing tests with JUnit and Mockito can result in various ways of calling and using methods. This makes it harder to rewrite the code. Writing tests for a method can result in multiple tests that each test something different. Therefore, there is not a single way of calling a method that can be rewritten in one way. Listing 3.5 shows the various ways in which one can test a method. To highlight the difference, for each test the same method is called.

The last example of Listing 3.5 uses the RxJava test method *.test().await().assertXXX()*, where 'XXX' can be many different assert methods. Each RxJava type has a method *test()* which returns either a *TestSubscriber* or a *TestObserver*. These objects support writing unit tests, perform assertions, and inspect received events for reactive methods and objects.

In Reactor, there is a slightly different way of doing this. One can use the *StepVerifier* and the *TestPublisher* in the unit tests for Reactor code. To make the tests work, the TestSubscriber and TestObserver have to be rewritten to use the Reactor StepVerifier and TestPublisher.

3.3 Refaster's support with RxJava to Reactor

Refaster Templates

As explained in the background (Chapter 2, currently Refaster can be used to refactor code inside the body of a method. To make sure that everything is rewritten with Refaster, it is important to write

[4]https://github.com/mockito/mockito

```
// Mock the class Foo, which contains the method 'getName(String
    customerId)'.
Foo foo = mock(Foo.class);

when(foo.getName(CUSTOMER_ID))
    .thenReturn(Flowable.just(NAME1))
    .thenReturn(Flowable.just(NAME2));

// The return type of the method 'getLongName()' is Answer<Flowable<
    String>>
when(foo.getName(any()))
    .thenAnswer(getLongName());

doReturn(NAME1)
    .when(foo).getName(any());

foo.getName(CUSTOMER_ID)
    .test()
    .await()
    .assertResult(NAME1);
```

Listing 3.5: Multiple test method invocations that depend on the return type

Refaster templates that rewrite code in small steps. There are two reasons for this. Firstly, there are many different ways a developer can write a specific piece of code. In a large codebase, these different ways are likely all used. If we want to migrate all of these pieces, we need to ensure the Refaster templates gradually rewrites the code by taking small steps. When the steps taken by Refaster are too large, there is a chance one is not able to rewrite some pieces, simply because some intermediate steps will be skipped. Secondly, when taking small steps it is easier to ensure that the rewrite does not break the code.

The primary RxJava types have many public methods. We want to show how we can use Refaster to rewrite usages of these public methods to the equivalent in Reactor. In Listing 3.6 we show an example of a non-static public method on the Flowable class. It is important to notice that the class has a generic type `T`. A generic type variable "*can be any non-primitive type you specify: any class type, any interface type, any array type, or even another type variable*" [67]. The @BeforeTemplate matches every Flowable object of type `T` with the method `firstElement()`. In the example below the Refaster template, is shown how a rewrite would be performed on actual code. The `Flowable.just(1)` will be matched in the @BeforeTemplate and be the parameter of the before and after-template, the `Flowable<T> flowable`.

Listing 3.7 contains a Refaster template that rewrites a static method on the Flowable class. The @BeforeTemplate we specify that any call to `Flowable.just` is matched for any `T`. Notice that in the example `Flowable.just(1)` is matched by the @BeforeTemplate and is rewritten as is specified in the @AfterTemplate. In the example of Listing 3.7, the Refaster template we continue with the result of Listing 3.6. We continue with this example to demonstrate how we can partially migrate the method body of an RxJava method to Reactor. The goal is to push the RxJava type to the "boundaries" of the method body. In the example, we see a `Flux` object followed by a conversion to a `Flowable` and then back to a `Flux` again. For this, we can write a Refaster template to remove the unnecessary conversion. After removing the double conversion, the method starts with creating a Reactor type, calling a method on that type, and in the end, convert it to an RxJava type to comply with the method definition. This means that with many Refaster templates a large part of the method bodies can be migrated to Reactor. This way a part of the migration can be automated. Since there are many public methods on the RxJava types, many Refaster templates need to be written.

Important to note, is that with the strategy we explained, after applying the Refaster rules the code will start by moving further away from the end state of a completed migration. As shown in Listing 3.6 we start with a single line of code: `Flowable.just(1).firstElement();`. In Listing 3.7 the code is rewritten to five lines. Only after changing the return type of the method, we can remove the last `RxJava2Adapter` calls. Hence, before we can move to the end state of the migration, we have to add a significant body of code.

```java
  /** Refaster template to rewrite ‘flowable.firstElement()’ to the Reactor equivalent. */
static final class FlowableFirstElement<T> {
  // T is a type parameter for the entire rule, which will be inferred from the match found by Refaster

  @BeforeTemplate
  Maybe<T> before(Flowable<T> flowable) {
    return flowable.firstElement();
  }

  @AfterTemplate
  Maybe<T> after(Flowable<T> flowable) {
    return flowable.as(RxJava2Adapter::flowableToFlux)
        .next()
        .as(RxJava2Adapter::monoToMaybe);
  }
}

// The Refaster template will match the following code:
Maybe<Integer> testFlowableFirstElement() {
  return Flowable.just(1).firstElement();
}

// After matching, the code will be rewritten to:
Maybe<Integer> testFlowableFirstElement() {
  return Flowable.just(1)
      .as(RxJava2Adapter::flowableToFlux)
      .next()
      .as(RxJava2Adapter::monoToMaybe);
}
```

Listing 3.6: Refaster template that rewrites RxJava Flowable *.map()*

The approach described so far only addresses the benefits of rewriting code with Refaster. However, there are some edge cases and downsides to the approach. There is a problem with method signatures in RxJava compared to Reactor. For instance, RxJava uses the Function object located in the package `io.reactivex.functions.Function` whereas Reactor uses a Function located in the package `java.util.function.Function`. The two Functions are almost identical, except that the Reactor Function has a *throws Exception* in the method signature. Besides Function, the problem also holds for BiFunction, Callable, and Predicate. This is a problem if we want to rewrite methods on RxJava types to Reactor as is shown in the examples. For instance, the `Flowable.map(io.reactivex.functions.Function)` cannot be easily rewritten to a `Flux.map(java.util.function.Function)`. While the method signature contains `throws Exception`, most Functions do not throw an exception. In addition, if the Function is a lambda expression, it would be safe to reuse it as a Reactor function, because a lambda expression cannot throw any checked exceptions.

There is a way to work around the problem. We can introduce a method that works similarly to the `RxJava2Adapter`. In Listing 3.8 we show one of the methods that are introduced. The utility method helps with creating Refaster templates that are fully compatible. In Listing 3.9 we see an example of how one can use the method. The example rewrites a `map` on a Flowable to a `map` on a Flux. Notice that on line 11 the `RxJavaReactorMigrationUtil.toJdkFunction` is wrapped around the Function object. This means that more code is introduced, but the rewrite is safe. Since there is no safe way to automatically remove this, it is up to the developer to do this manually.

The `RxJavaReactorMigrationUtil` contains methods for various types. For instance, for the Consumer, Callable, and Predicate. The `RxJavaReactorMigrationUtil` is in a separate module. This module must be imported by the codebase on which the migration is performed. Otherwise, the code would not compile because it does not have a reference to this class. Ideally, after introducing the

```java
/** Refaster template to rewrite 'Flowable.just()' to the Reactor
   equivalent. */
static final class FlowableJust<T> {
  @BeforeTemplate
  Flowable<T> before(T t) {
    return Flowable.just(t);
  }

  @AfterTemplate
  Flowable<T> after(T t) {
    return Flux.just(t).as(RxJava2Adapter::fluxToFlowable);
  }
}

// The Refaster template will match the following code:
Maybe<Integer> testFlowableFirstElement() {
  return Flowable.just(1)
      .as(RxJava2Adapter::flowableToFlux)
      .next()
      .as(RxJava2Adapter::monoToMaybe);
}

// After matching, the code will be rewritten to:
Maybe<Integer> testFlowableFirstElement() {
  return Flux.just(1)
      .as(RxJava2Adapter::fluxToFlowable)
      .as(RxJava2Adapter::flowableToFlux)
      .next()
      .as(RxJava2Adapter::monoToMaybe);
}
```

Listing 3.7: Refaster template that rewrites RxJava Flowable *.just()*

`RxJavaReactorMigrationUtil`, we want to create a Refaster template to automatically remove it after the migration to the Reactor type is done. Currently, there is nothing in Refaster to express such a thing that also guarantees the validity and correctness of the rewrite.

3.4 Error Prone and inlining deprecated methods

The existing plugins in Error Prone focus mostly on catching common bugs, preventing anti-patterns, cleaning up code, and improving specific code constructs. Given the current features of Refaster, there are almost no plugins that can significantly support the automation of a migration. A reason for this could be that the focus of Error Prone is not on supporting library migrations. Nevertheless, since Error Prone is such a powerful tool, there are plenty of options for contributing to migrations.

There is one feature in Error Prone that could be of use during a migration. However, currently, it would require manual labor in advance to use the plugins. The feature in Error Prone is a mechanism to automate updating callers of deprecated methods. The feature consists of several plugins that replace the caller to the method with the body of the called function. This process is called the *inlining* of a method. The feature consists of two plugins, the InlinerSuggester, and Inliner. It starts with a method that is annotated with `@Deprecated` to denote that the method should not be used anymore. An example of the inlining feature is shown in Listing 3.10. When the InlinerSuggester finds a deprecated method that does not have an `@InlineMe` annotation, it will suggest one. The result of such a suggestion is line 2 of Listing 3.10. So the content of the `replacement` in the `@InlineMe` is the method body of the deprecated method. Now the InlinerSuggester created the annotation, the Inliner can perform the inlining actions. The Inliner checks whether a method invocation calls a deprecated method that also has the `InlineMe` annotation. Line 8 of Listing 3.10 is an example of such a method invocation. The

```java
public static <T, R> java.util.function.Function<T, R> toJdkFunction(
      io.reactivex.functions.Function<T, R> function) {
  return (t) -> {
      try {
        return function.apply(t);
      } catch (Exception e) {
        throw new IllegalArgumentException("Function threw checked
            exception", e);
      }
  };
}
```

Listing 3.8: Utility method from the `RxJavaReactorMigrationUtil`

```java
static final class FlowableMap<I, T extends I, O> {
  @BeforeTemplate
  Flowable<O> before(Flowable<T> flowable, Function<I, O> function) {
    return flowable.map(function);
  }

  @AfterTemplate
  Flowable<O> after(Flowable<T> flowable, Function<I, O> function) {
    return flowable
        .as(RxJava2Adapter::flowableToFlux)
        .map(RxJavaReactorMigrationUtil.toJdkFunction(function))
        .as(RxJava2Adapter::fluxToFlowable);
  }
}
```

Listing 3.9: The `RxJavaReactorMigrationUtil` introduced in a Refaster template

Inliner can automatically patch the code and replace the right side as is shown on line 10.

Since the Inliner only updates the right side of the expression, the rest of the code at the call site will not change. Imagine that the line below the String object *text* there would be a line with `int length = text.length();`. If we would update the entire expression of line 8 and also update the type, the code would not compile. Thus only updating the right side of the expression can be a good step for the migration.

Although this is a powerful feature of Error Prone, currently it cannot contribute to automation. The problem is that before this feature can be used, a developer needs to manually change the code. Hence, it is not part of automated migration.

Rewriting tests

In Listing 3.5 is shown that there are many ways a method can be called in a test. Notice that the structure of calling the method and defining the return type is different for each example. In Refaster we can write a rule that matches a `when` and a `thenReturn`. The best Refaster can do next, is add a conversion to the method invocation and the return type. For instance, the method invocation could be wrapped in `RxJava2Adapter.flowableToFlux(...)`. Additionally, the `thenReturn` could return a `Flux.just(NAME1)`. However, there would be no actual improvement to the code, only additional code in the test that is being validated. The goal of the tests is to validate the code that is written. Introducing more conversion logic in the tests, will not improve the validation. Besides that, the extra code makes it harder for the developer to understand what is going on.

Assume we are testing a method of which we migrated the inner side of the method body but still returns an RxJava type. The end goal is that the return type of the method is changed and that the test expects a Reactor type. Therefore, we argue that updating the tests with conversions brings us closer to the desired end state of the migration. For instance, take the `Flowable.just(NAME1)`, which end the end will be migrated to `Flux.just(NAME1)` when the return type of the method is a Reactor

```
@Deprecated
@InlineMe(
   replacement = "String.valueOf(this.getInteger())")
public String bar() {
  return String.valueOf(getInteger());
}

// An example of a caller of the method in an arbitrary place in the
    code:
String text = bar();

// The Inliner will automatically replace the code with:
String text = String.valueOf(getInteger());
```

Listing 3.10: An example of the InlineMe annotation

type. The Flowable with a conversion, namely `RxJava2Adapter.fluxToFlowable(Flux.just(NAME1))`, will be closer to the end state because the last thing that needs to happen is to remove the conversion. The conversion does not break the test because the original behavior is preserved. However, after the conversion is added, currently there is no way to verify whether it is safe to remove the conversion. As a result, we cannot go any further in the migration than adding the conversion.

In the previous section 3.4 we showed how the `InlineMe` of Error Prone works and how it could be of use. However, the `InlineMe` feature does not take into account whether or not the code is in a test framework. Using `InlineMe` on the examples in Listing 3.5 would cause problems for the tests. For instance, the method invocation would be changed and migrated to the new type, while the specified return type is not updated at all. The test will compare an RxJava type with a Reactor type which will cause the test to fail. This would be harmful because it is hard to prove the migration is still correct. Additionally, it would require manual labor performed by the developer to fix the tests.

In the next Chapter 4 we address this problem. We explain how we use Error Prone to correctly rewrite the code in the tests. The existing `Inliner` is extended to exclude rewrites that are in the tests. In addition, we add a plugin that handles the special inlining cases to ensure compilable code.

Scope of Refaster

For the migration of RxJava to Reactor we see many type changes. Especially since there are many places where types can be defined. For instance, in parameters, field declarations, local variables, type variables, subclasses, and return types.

Within Refaster there is support for two types of templates. First, expression templates can match a single expression statement. Second, a block template matches a sequence of expression statements. As a result, most of the places where RxJava types can occur cannot be modified with Refaster. Therefore, only using Refaster for a migration makes it hard to account for all these cases.

Chapter 4

Extensions to improve automation of library migration

In the previous chapter, we focused on the migration of RxJava to Reactor. We showed what it takes to completely migrate. There are not many things that can currently be automated with Refaster and Error Prone. However, Refaster and Error Prone can potentially have a significant impact on the migration. Therefore we elaborate on multiple extensions that we created to overcome the limitations discussed in the previous chapter. Overcoming these limitations improves automation of the migration.

4.1 Extension for Refaster templates

In section 3.3 we discussed limitations that occurred when writing Refaster templates. If we cannot assure that a rewrite is safe, it is better to not execute the Refaster template. We create an extension in Refaster and Error Prone to help with assuring the validity and correctness of a rewrite.

Note that in section 3.3 we explained that we can rewrite this with the template that is in Listing 3.8, which migrates to a Flux `.map` and introduces the `RxJavaReactorMigrationUtil` wrapper. However, we want to explain the problem in more detail, so we dive into a more specific example. In Listing 4.1 we see an example of valid RxJava code. There is a Flowable object with a lambda expression passed to the `map` method. The different Function classes are shown in Listing 4.2. Note that the RxJava method has a different method signature than the Reactor method. Rewriting the `Flowable` of Listing 4.1 to a `Flux` with the same content for the map function, would not work. To compile the code with a `Flux`, the `throw new Exception()` should be wrapped in a try-catch. That way the Exception would be handled and therefore match the method signature of Reactors Function method `apply`.

```java
Flowable.just(1, 2)
    .map(
        x -> {
          if (x > 1) {
            throw new Exception();
          }
          return x + 1;
        });
```

Listing 4.1: An expression with a `Flowable#map` that throws an `Exception`

The problem does not only occur with the `.map` operator. There are many operators on each RxJava type that require a Function object as a parameter. Besides the Function object, there are other types with the same problem. For instance, the Consumer, Predicate, BiFunction, Callable to Supplier, and Action to Runnable. This extension can be used for many different rewrites operations.

In Listing 3.8 we show a way to partially deal with this issue. By introducing a call to a wrapper (the RxJavaReactorMigrationUtil), one can do a valid and correct rewrite. With this extension we can remove the RxJavaReactorMigrationUtil wrapper where it is safe and correct to do so. As a result, the migration will be one step closer to the desired end state. In Listing 4.3 we use the `@CanTransformToTargetType`

```java
// The io.reactivex.functions.Function that is used in RxJava
public interface Function<T, R> {
  R apply(@NonNull T t) throws Exception;
}

// The java.util.function.Function that is used in Reactor
public interface Function<T, R> {
  R apply(T var1);
}
```

Listing 4.2: The RxJava and Reactor Function classes

annotation on a parameter in the before-template, see lines 4 and 5. This Refaster template will remove the unnecessary conversion if it is valid and correct to remove it. Listing 4.4 is an example of code that can now be rewritten with the new annotation. As a result, we can fully migrate a `map` method from RxJava to Reactor.

Note that for the example with the `throw new Exception();` the annotation would prevent Refaster from unwrapping the `RxJavaReactorMigrationUtil.toJdkFunction` because it would not be correct. The extension would see that the exception would result in non-compilable code if it would perform the rewrite.

```java
1    static final class UnnecessaryFunctionConversion<I, O> {
2      @BeforeTemplate
3      java.util.function.Function<I, O> before(
4         @CanTransformToTargetType
5                          Function<I, O> function) {
6        return RxJavaReactorMigrationUtil.toJdkFunction(function);
7      }
8
9      @AfterTemplate
10     java.util.function.Function<I, O> after(java.util.function.Function
          <I, O> function) {
11       return function;
12     }
13   }
```

Listing 4.3: Refaster template to remove the RxJavaReactorMigrationUtil.toJdkFunction

In addition to validating the checked exceptions, the annotation can validate more things. Before we explain that, we explain how the annotation can perform these extra checks.

Initially, there was no way in Refaster to retrieve data of the after-template during the matching of the before-template in Refaster. Therefore we modified the builder of the Refaster templates. During the compilation of the templates, the whole `MethodTree` of the after-template is passed to the creation of the before-template. Therefore extra information on the parameters and type arguments are available in the before-template.

When Refaster tries to match an expression in the codebase, Refaster tries to *unify* the target expression with the expression specified in the before-template. For every type of Tree, there is a `unify` method implemented in Refaster. For instance, there is an UIf, ULiteral, UReturn, UClassDecl, and UMethodDecl class which defines how each respective tree is unified. Using the `@CanTransformToTargetType` we can add our logic for performing the unification. Since the information of the after-template is provided, we can add the following checks:

- Verify that the return type of the before-template matches the return type of the after-template.
- Verify that the number of parameters is equal.
- Verify that the types of the parameters match.
- Verify that the defined checked exceptions match.

When we use the word *match* it does not mean that the types have to be exactly the same. A

```java
// Before executing the Refaster rule
return Flowable.just(1)
    .as(RxJava2Adapter::flowableToFlux)
    .map(RxJavaReactorMigrationUtil.toJdkFunction(i -> i + 1))
    .as(RxJava2Adapter::fluxToFlowable);

// After executing the Refaster rule
return Flowable.just(1)
    .as(RxJava2Adapter::flowableToFlux)
    .map(i -> i + 1) // Note that this .map operation is now called on
       a Flux object.
    .as(RxJava2Adapter::fluxToFlowable);
```

Listing 4.4: Safely remove the wrapper method because of the `@CanTransformToTargetType`

type parameter can have a specific upper- and lowerbound. Suppose we have a generic type `T` in the before-template and the after-template the same `T` with `extends Number`. When Refaster tries to match three expressions with the types *Integer, Number,* and *Object*, only the latter expression will be rejected. Integer matches the `T extends Number` because it is a subtype of Number.

For validating the return types and checked exceptions of lambdas we had to account for some special cases. Since a lambda expression can have multiple paths and return statements, there is additional logic required to validate that the return types and checked exceptions match. The following example is used to show an instance of the special case. For validity, note that the specific example that is shown in Listing 4.5 is already filtered out by the current features of Refaster. However, the purpose of the example is to demonstrate what we are trying to achieve for multiple checks, not only for the matching of a return type. Suppose there is a Refaster template that matches a `map` operation with a `java.util.function.Function` as an argument. In the after-template we want to rewrite it to an `mapToInt` operation that takes a `java.util.function.IntFunction` as an argument. This rewrite operation can only be performed if we make sure that the input and output of the original `Function` are of type `int` or `Integer`. Listing 4.5 shows a lambda expression that has two different paths and two different return statements. The `.stream()` ensures that we can stream the items of the `ImmutableList`. On this *stream*, we can perform operations in a functional style. There is an extra cast expression to make clear why the expression should not be matched. The return statement in the body of the `if` statement is not compatible with the desired `mapToInt` method. This is an example of where the `@CanTransformToTargetType` can be used to add restrictions to the expression Refaster matches.

```java
ImmutableList.of(1, 2, 3)
    .stream()
    .map(
        y -> {
          if (false) {
            return (Object) y;
          }
          return (int) y;
        });
```

Listing 4.5: Edge case for a lambda expression with multiple return statements

To summarize, without extension, the before-template is the only place to express what should be matched by Refaster. With extension, we provide a way to use more data and add extra restrictions that help Refaster decide what should be matched. The goal of the extension is to create even more specific rewrite operations and improve validity of performed rewrites.

Unwrap nested lambdas

Some RxJava methods do not have a direct equivalent in Reactor. Consequently, we have to use intermediate steps to ensure that the code compiles and we end up with a correct Reactor counterpart. Therefore we use some Refaster templates that introduce extra method invocations, conversion calls, and adapter calls. Sometimes, this means introducing nested lambda expressions with additional type information. The result of this intermediate steps can result in complex and deeply nested expressions. Therefore, we have to introduce complex Refaster templates to ensure simplifications of the code. In Listing 4.6 we show a Refaster rule that contains a complex before-template and an after-template. The example contains a *placeholder* method with associated annotation. This allows for matching an arbitrary expression of a type that *may* use the specified `input`.

```java
abstract static class MaybeFlatMapSingleElementUnwrapLambda<T, R> {
  @Placeholder
  abstract Mono<R> placeholder(@MayOptionallyUse T input);

  @BeforeTemplate
  java.util.function.Function<T, ? extends Mono<? extends R>> before()
     {
    return e ->
      RxJava2Adapter.singleToMono(
          Single.wrap(
              RxJavaReactorMigrationUtil.<T, Single<R>>toJdkFunction(
                      (T ident) -> RxJava2Adapter.monoToSingle(
                         placeholder(ident)))
                     .apply(e)))
  }

  @AfterTemplate
  java.util.function.Function<T, ? extends Mono<? extends R>> after() {
    return e -> placeholder(e);
  }
}
```

Listing 4.6: Refaster template to simplify a nested lambda expression

4.2 Return type extension

To this point, we discussed ways to rewrite code that is inside method bodies and how to improve upon that. While by far the most common rewrite operation in the RxJava to Reactor migration is changing the return type of a method. Therefore we now focus on migrating return types of methods.

For performing a migration on a large codebase there are several things we need to take into account. Again, because the code needs to compile every step during the migration, we need to take some extra steps to ensure this. When it comes to migrating the return type of a method, we identify three different parts:

- Changing return type of a method that has an implementation.
- Updating callers of methods to match new return types.
- Updating interface definitions to comply with the updated return type of implementing methods.

Performing these steps incrementally and safe is a challenging task. There are many small steps in the migration to prevent the code from breaking. Since there are steps combined and we want to rewrite a large codebase, there will be many (edge) cases to account for. We want our extension to be applicable on other codebases as well, so we need to take into account many use cases.

In section 3.3 we discussed how we want to use Refaster templates to migrate the body of a method. Imagine that after applying Refaster rules, we end up in a state where we have a method as is shown in Listing 4.7. There is a method that returns an RxJava type but starts with a Reactor type. Multiple operations are performed on the Reactor type before it is converted to an RxJava type to match the

```java
Maybe<Integer> exampleMethod() {
    return Flux.just(1, 2, 3)
        .filter(i -> i > 1)
        .map(String::valueOf)
        .next() // The ‘next()’ is called on a Flux and returns a Mono
        .as(RxJava2Adapter::monoToMaybe);
}
```

Listing 4.7: A method that contains a Reactor expression, in the end, returns an RxJava type

method return type. The state in Listing 4.7 is the state of the migration we want to continue working with for the return type extension.

We will explain how the extension works by going over the various steps. First, we explain how we want to define the type migrations that are required during each step. Secondly, we explain how we extend the interfaces. Thirdly, we elaborate on migrating return types of normal methods while not introducing breaking changes. Finally, we update callers of the methods to handle new return types.

Define a type migration

For each step of the migration, we need to know how to move from one type to another and vice versa. To this point in the migration, the migration is performed by only Refaster templates. When we use Refaster templates, the conversion is defined in @Before and @After templates. In Listing 4.7, we see the state to which we can come with Refaster. The method is ready to change to the new return type. Since we want to do that automatically and for multiple types, we need an extension that can perform this rewrite based on a migration definition. With a migration definition, we mean that there needs to be a way to express how one can migrate a method with a type to another type. It is important that the method body needs to be updated as well. Therefore we need a way to define these conversions, such that Refaster and Error Prone can perform these migrations automatically.

To show what the result after Listing 4.7 is, we created Listing 4.8. The example shows an updated return type and an extra conversion to comply with a Reactor type.

```java
Mono<Integer> exampleMethod() {
  return Flux.just(1, 2, 3)
      .filter(i -> i > 1)
      .map(String::valueOf)
      .next()
      .as(RxJava2Adapter::monoToMaybe)
      .as(RxJava2Adapter::maybeToMono);
}
```

Listing 4.8: A method with a new return type and corresponding conversion as final statement

As can be seen in the examples, we want to express this conversion in Java code. There is a logical option here, namely using a Refaster template. When we use a Refaster template for this conversion, some benefits come with the tool. Refaster has multiple convenient features that will help us with writing extensions:

- A clear and concise way to express conversions.
- A serialization mechanism to save Refaster templates.
- The RefasterBuilderScanner supports exposing the content of Refaster templates in a convenient way to Error Prone.
- We can use the matching logic of Refaster to create SuggestedFixes in Error Prone for updating method bodies.

Refaster cannot be used to define these migrations without modifying the current features. Since we want to create an extension and not tweak the existing logic too much, we want to create a clear distinction between normal Refaster templates and our migration definitions. We introduce an annotation, the `@MigrationTemplate` annotation, that has a boolean value as an argument. The boolean value indicates whether the `MigrationTemplate` expresses a type conversion that is desired or not. The *desired* migration means that we want to move away from the type that is defined in the migration.

For every RxJava type, we defined a Refaster `MigrationTemplate`. Each `MigrationTemplate` has the conversion as specified in Table 3.1. All of the `MigrationTemplate`s look similar to the FlowableToFlux `MigrationTemplate` that is in Listing 4.9.

```java
public final class FlowableToFluxMigrationTemplate {
  private FlowableToFluxMigrationTemplate() {}

  static final class FlowableToFlux {
    @MigrationTemplate(
        value = /* desired = */ false)
    static final class MigrateFlowableToFlux<T> {
      @BeforeTemplate
      Flowable<T> before(Flowable<T> flowable) {
        return flowable;
      }

      @AfterTemplate
      Flux<T> after(Flowable<T> flowable) {
        return RxJava2Adapter.flowableToFlux(flowable);
      }
    }

    @MigrationTemplate(
        value = /* desired = */ true)
    static final class MigrateFluxToFlowable<T> {
      @BeforeTemplate
      Flux<T> before(Flux<T> flux) {
        return flux;
      }

      @AfterTemplate
      Flowable<T> after(Flux<T> flux) {
        return RxJava2Adapter.fluxToFlowable(flux);
      }
    }
  }
}
```

Listing 4.9: The Refaster template that describes the migration definition for the Flowable to Flux migration

Extending interfaces

Rewriting the return type of a method without callers is a straightforward operation. Since the code does not depend on the return type it can be updated directly. However, if there are callers of the method, they need to handle the new return type at the call site. Rewriting the return type of various methods and update the callers in one run of Error Prone and Refaster is not an option. There is no option in Error Prone to guarantee that one plugin is performed on the entire codebase before another plugin can start. Therefore it is important to take into account that migrating something always takes multiple steps to run.

Besides the callers, there is another problem when updating the return type of a method that is defined in an interface. If we change the return type a method, of which all callers are migrated to the new return type, the interface will complain that it is missing an implementation of a method with the old return type. It is not possible to perform the migration without updating the interfaces. Updating the interface is the first step of the migration, as it will allow us to subsequently update the methods and callers of the methods.

The goal is to extend the interface so that implementing classes and callers of the methods can update to the new return type. So the first part of the migration consists of introducing an extra method in the interfaces. We scan for methods in interfaces that have an undesired return type, according to the migration definitions, and introduce a method with the new return type. In addition, both methods are extended with a `default` implementation. An example of performing this rewrite operation is provided in Listing 4.10. Since both methods in the interface have a default implementation, it is safe to migrate the method in the implementing class. Suppose we have a class that implements the `AddressService` and overrides the method `validateAndComplete`. If we delete the `validateAndComplete` method and replace that with the `validateAndComplete_migrated` method, the code would now call the `validateAndComplete` in the interface. There we would be redirected to the `validateAndComplete_migrated` method that is implemented in the class.

At first sight, it looks like we introduce an infinite loop to the code. However, before performing the rewrite, there was no default implementation provided in the interface. Thus, there must exist an implementation in the class that implements the interface. In addition, during the migration, we make sure that at least one of the two methods defined in the interface is always implemented in the class. Therefore it is not possible to get into a loop.

The challenging part of this rewrite operation is the content of the default implementation. Since we have the Refaster templates with the conversions, obtaining the content of the default method body should be easy. However, there is one challenge in using the Refaster templates on the non-existent default implementations. When Refaster runs during the compilation phase, the compiler provides all sorts of data. For instance, the types of (sub)expressions, the exact locations of expressions in a source file, and provides access to all sorts of managers and helpers via the `Context`. In this case, we want Refaster to perform a rewrite operation on a method invocation to obtain the content as we can see on line 9 of Listing 4.10. Thus, we want to give a method invocation of a not yet existing method, in this case, `validateAndComplete_migrated(address)`, to Refaster which should then use the migration definition to wrap the content with the `RxJava2Adapter.monoToSingle()`. Since we try to match on a non-existing method in the code, Refaster misses information to match the method invocation and to perform the rewrite operation. Therefore, Refaster cannot directly provide the content of the default methods.

To overcome this limitation of Refaster, we have to "fool" Refaster. We built our own `Tree` objects and added these to `JavaFileObject`s that are created in memory. Next, the `JavaFileObject` is heavily modified with data that would in a normal case be set by the parser. Through trial and error, we managed to set up just enough data to make Refaster believe it is a "normal" Java file that can run the regular matching of Refaster. This is performed for both default implementations. Then, the original `MethodTree` is duplicated and the method name will get a suffix (_migrated) and the return type is updated to the Reactor type. The original method is annotated with `@Deprecated`. Next up, we modify both `MethodTree`s with the new method bodies, the `default` modifier, and the return statements. Finally, this is merged into a suggested fix which can be applied by Error Prone.

There is an additional case that is important when rewriting methods of interfaces. We need to perform a different rewrite when Error Prone finds a method in an interface that is eligible for migration *and* already contains a default implementation. In that case, most of the steps are handled in the same manner as we explained till now. However, the body of the `_migrated` method cannot be a delegating method call to the original method. The original implementation of the default method will be moved to the `_migrated` method. Without changing the implementation, the return type would not be compatible with the original method body. For this, we use the same logic as is explained in the following section 4.2 on updating methods.

Note that on line 7 of Listing 4.10 the `@Deprecated` annotation is introduced. That is useful for the part of the migration where we update the callers of the method. How we use this, is explained in Listing 4.2.

In Java, there exists an annotation to denote that an interface has exactly one abstract method, namely the `@FunctionalInterface`. A limitation to our approach is that interfaces marked with `@FunctionalInterface` cannot be migrated because the compiler will complain when an extra method is added to the interface. Therefore, we do not migrate interfaces that are annotated with this annotation.

```
interface AddressService {
  // This is the original method
  Single<CompletedCustomerAddress> validateAndComplete(CustomerAddress
     address);

  // Which is replaced by the following two methods
  @Deprecated
  default Single<CustomerAddress> validateAndComplete(CustomerAddress
     address) {
    return RxJava2Adapter.monoToSingle(validateAndComplete_migrated(
       address));
  }

  default Mono<CustomerAddress>
      validateAndComplete_migrated(CustomerAddress address) {
    return RxJava2Adapter.singleToMono(validateAndComplete(address));
  }
}
```

Listing 4.10: Rewrite a method of an interface and add two default implementations

Updating methods

This part of the extension finds methods in classes that have undesired return types. First, the content of the original method is changed to a delegating method. This is done in the same way as is described for rewriting the methods in interfaces. The content of the new method, with the new return type, is a little different than with the interfaces. Now we can use Refaster to update the method body according to the new return type. Refaster can run on the code that exists in a source file. As a result, Refaster automatically rewrites every usage of the type in the method body. In most cases this is valid and the rewrite will look like the code in Listing 4.11.

```
  // This is the original method
  public Flowable<String> findAll() {
    return collection
        .flowable(c -> c.find(getBadWordDocumentFilter()).first())
        .flatMapIterable(BadWordDocument::getBadWords);
 }

  // Which is replaced by the following two methods
  @Deprecated
  public Flowable<String> findAll() {
    return RxJava2Adapter.fluxToFlowable(findAll_migrated());
  }

  public Flux<String> findAll_migrated() {
    return RxJava2Adapter.flowableToFlux(
        collection
            .flowable(c -> c.find(getBadWordDocumentFilter()).first())
            .flatMapIterable(BadWordDocument::getBadWords));
  }
```

Listing 4.11: Rewrite a method of a class

Before the rewrite of a method starts, there is a check to see whether the class implements an interface that also defines the method. If so, then we check if the method in the interface has already been migrated. If not, we will skip the migration for this method so that the interface can migrate first.

There could be a state where we migrate the method before it is defined in the interface. That would be a problem because the annotations of a method are also duplicated. When the original method has the `@Override` annotation and is duplicated to the new method, that would result in a non-compilable state. The compiler will complain that the new method with `@Override` is not defined in the interface.

Running Refaster on the whole method body does not always result in compilable code if all matches are applied. In Listing 4.12 two methods can result in non-compilable code if every type of matching statement would be rewritten. When the return type of the method is modified, it is important to *only* update the return statements to match the new return type. This would result in the first example of Listing 4.12 to output valid code. However, the second example would still result in non-compilable code. To be completely certain about the output state, we need to ensure that only *direct* return statements are updated. Since the statements `return "2"` and `return "3"` are nested they are not changed at all. To ensure the rewrites follow these guidelines, we added a matcher using the `@Matches` annotation. In Listing 4.13 on Line 3 is shown how the Matcher is used in one of the migration definitions. The Matcher validates that the expression that is matched, is a return statement and that it has no parent return statement of the method.

```java
// Suppose we are migrating 'Single' to 'Mono' in this example.
public Single<String> bar() {
  Single<String> temp = Single.just("1");

  if (true) {
    return Single.error(new RuntimeException("Error!"));
  }
  return Single.just("2");
}

// Suppose we are migrating 'String' to 'Integer' in this example.
public String baz() {
  return ImmutableList.of(1, 2).stream()
      .map(
          e -> {
            if (true) {
              return "2";
            }
            return "3";
          })
      .findFirst().get();
}
```

Listing 4.12: Example of rewrite that has multiple code paths

```java
1 @BeforeTemplate
2 Flowable<T> before(
3     @Matches(IsParentReturnTree.class)
4     Flowable<T> flowable) {
5   return flowable;
6 }
```

Listing 4.13: Before-template of the FlowableToFlux migration definition with the `@Matches` annotation

See Listing 4.14 for a migration performed on both the interface and an implementing class. Since the delegating method bodies are present in the interface, one could argue that the old method of the class, `MappingService#getQueueSchemaMappings()`, can be deleted. After removing that method, without updating the callers, it would mean that the default method `getQueueSchemaMappings()` is called in the interface. That method, delegates to the `getQueueSchemaMappings_migrated()` method in the class. Therefore, removing the method should not alter the external behavior. The only extra thing that should happen is an additional method call via the delegation. Unfortunately, this appears not to be the case.

In the example, we see that besides the `@Override` there is a Spring framework annotation `@Secured()` that is used. The `@Secured(NAME_OF_ROLE)` annotation specifies the roles a user should have to access the method. The problem is that by removing the original implementing method, the `@Secured` annotation could be bypassed. Without updating the caller, the call now goes via the interface with the default implementations. As a result, the `@Secured` annotation does not function in a proper way anymore. This can cause major issues if this is not handled correctly. Therefore we choose to keep the original method if there is another annotation then the `Deprecated`, `Override`, or `InlineMe` declared on the method. We only remove the method if all callers are migrated to the new method, and the method is unused. Error Prone provides a plugin out-of-the-box that flags *private* unused methods for deletion. This can be used to clean up the code afterward.

```java
public interface QueueSchemaMappingService {
  @InlineMe(
      replacement = "RxJava2Adapter.fluxToFlowable(this.
         getQueueSchemaMappings_migrated())",
      imports = "reactor.adapter.rxjava.RxJava2Adapter")
  @Deprecated
  default Flowable<QueueSchemaMapping>
      getQueueSchemaMappings() {
    return RxJava2Adapter.fluxToFlowable(
        getQueueSchemaMappings_migrated());
  }

  default Flux<QueueSchemaMapping>
      getQueueSchemaMappings_migrated() {
    return RxJava2Adapter.flowableToFlux(getQueueSchemaMappings());
  }
}

public class MappingService implements QueueSchemaMappingService {
  // Part of the class omitted for brevity

  @InlineMe(
      replacement = "RxJava2Adapter.fluxToFlowable(this.
         getQueueSchemaMappings_migrated())",
      imports = "reactor.adapter.rxjava.RxJava2Adapter")
  @Deprecated
  @Override
  @Secured(QUEUE_SCHEMA_MAPPING_SERVICE_READ)
  public Flowable<QueueSchemaMapping> getQueueSchemaMappings() {
    return RxJava2Adapter.fluxToFlowable(
        getQueueSchemaMappings_migrated());
  }

  @Override
  @Secured(QUEUE_SCHEMA_MAPPING_SERVICE_READ)
  public Flux<QueueSchemaMapping> getQueueSchemaMappings_migrated() {
    return RxJava2Adapter.flowableToFlux(
         queueSchemaRepository.findAll());
  }
}
```

Listing 4.14: Completely migrated return types of method declarations for an interface and implementing class

Migrating callers

Now that the new methods are in place, both in the interface and in the classes, we can start migrating the callers. The first step of this part starts with the introduced `@Deprecated`. The annotation will trigger the InlinerSuggester plugin. The InlinerSuggester sees the deprecated method and adds the `@InlineMe` annotation with a suggestion for the replacement. In Listing 4.10 an example of the `@InlineMe` is shown. The content of line 2 is automatically added by Error Prone.

In the next run of Error Prone, the Inliner plugin can use the `@InlineMe` annotations. With the information that is provided in the annotation, the Inliner can start migrating callers of the deprecated methods. The Inliner goes over method invocations and checks whether the method contains an InlineMe annotation. If that is the case, the code is automatically replaced by the content of the InlineMe annotation. As a result, the caller of the method is successfully migrated to the new return type. This way, we use the Inliner to push the migration of the RxJava types further to the call site of the methods. The callee is in this case a method with only Reactor code. That means that the migration is pushed out of the method to the call site.

```
1 @Deprecated
2 @InlineMe(
3   replacement = "RxJava2Adapter.monoToSingle(
        validateAndComplete_migrated(address))",
4   imports = "reactor.adapter.rxjava.RxJava2Adapter")
5 default Single<CustomerAddress> validateAndComplete(CustomerAddress
      address) {
6   return RxJava2Adapter.monoToSingle(
7     validateAndComplete_migrated(address));
8 }
```

Listing 4.15: The `@InlineMe` annotation

4.3 Inlining method invocations in tests

For most callers, we can use the Inliner plugin. The method invocations in non-test code can be successfully migrated this way. For method invocations in test code, this works a little differently. In section 3.2 we showed that updating tests is more challenging than normal code. The tests are an important part of the migration. To ensure the quality of the migration, we aim to migrate as many tests as possible. Tests can be written in many different ways. Besides the various testing frameworks, there are many different ways of testing an application. As a result, some test setups are not completely rewritable.

In Listing 4.16 there is an example of a rewrite that the Inliner would perform, given that the migration is from a String to an Integer. Two things are not correct after the update. While the method invocation is successfully updated, the wrapper around the method call should not be present. In addition to that, the return type is untouched while another type is expected. The content of the `thenReturn` needs to be wrapped to comply with the new type of the updated method invocation.

```
// The Inliner plugin would rewrite the following statement
when(foo.getId()).thenReturn("1");

// To this, which is incorrect
when(Integer.valueOf(foo.getId_migrated()).thenReturn("1");
```

Listing 4.16: Faulty rewrite that the Inliner plugin would perform on a Mockito statement

Since there are various ways of writing and passing method invocations, we created a new plugin, named *InlineMockitoStatements*. The purpose of the plugin is to migrate Mockito code that tests the methods annotated with the `@InlineMe` annotation. In Listing 4.17 we the various rewrites that are performed by the InlineMockitoStatements. The plugin tries to match the various Mockito operators and migrates the method invocation together with the return statements. For each operator, the plugin performs a slightly different migration.

```java
public final class Foo {
  @Deprecated
  @InlineMe(
      replacement = "String.valueOf(this.getId_migrated())")
  public String getId() {
    return String.valueOf(getId_migrated());
  }

  public Integer getId_migrated() {
    return 1;
  }
}

public final class BarTest {
  // The original test method
  @Test
  public void simpleTest() {
    Foo foo = mock(Foo.class);

    when(foo.getId()).thenReturn("1").thenReturn("2");
    when(foo.getId()).thenAnswer(inv -> String.valueOf("3"));

    doReturn("1", "2").when(foo).getId();
    doAnswer(inv -> String.valueOf("3")).when(foo).getId();

    verify(foo).getId();
  }

  // The method after running the plugin
  @Test
  public void simpleTest() {
    Foo foo = mock(Foo.class);

    when(foo.getId_migrated())
        .thenReturn(Integer.valueOf("1"))
        .thenReturn(Integer.valueOf("2"));
    when(foo.getId_migrated())
        .thenAnswer(inv -> Integer.valueOf(String.valueOf("3")));

    doReturn(Integer.valueOf("1"), Integer.valueOf("2"))
        .when(foo).getId_migrated();
    doAnswer(inv -> Integer.valueOf(String.valueOf("3")))
        .when(foo).getId_migrated();

    verify(foo).getId_migrated();
  }
}
```

Listing 4.17: Rewrites performed by the InlineMockitoStatements plugin

4.4 Extension Inliner plugin for method references

A limitation of the Inliner plugin is that it does not work in method references. This prevents the last usages of the RxJava methods from migrating to Reactor. As a result, we cannot remove the methods with RxJava return types. In Listing 4.18 we show an example of a limitation of the migration. The state that is shown occurs after rewriting the `Single#onErrorResumeNext` to a `Mono#onErrorResume`. Since the method signatures do not match, we introduce an extensive lambda expression with the `.apply(e)` to migrate the method. Without improving the Inliner, we would not be able to clean this up.

```java
@Override
public Mono<CompleteAddress> validate_migrated(Address address) {
  return foo_migrated(address)
      .single()
      .onErrorResume(
          e ->
              RxJava2Adapter.singleToMono(
                 RxJavaReactorMigrationUtil.toJdkFunction(this::handle)
                    .apply(e)));
}
```

Listing 4.18: Example of a method in the migration that contains a method reference

Therefore we decided to improve the existing Inliner plugin and use it in combination with other plugins to migrate this code. How the migration for this part goes step by step, is shown in Listing 4.19.

```java
// The following code is matched in the plugin
this::handle

// This will be rewritten to
ident -> handle(ident)

// Next, the Inliner will see that handle can be rewritten
ident -> RxJava2Adapter.monoToSingle(handle_migrated())
```

Listing 4.19: The steps for migrating a method reference

The result of Listing 4.19 will replace the `this::handle` in Listing 4.18. As a result, we have an extensive nested lambda expression that can be optimized. We write a Refaster template to match the nested lambda expression and optimize the code. After running Refaster, we end up with the code that is on the first line of Listing 4.20. Then, we can do one final optimization, and that is rewriting the simple lambda expression to a method reference. For this rewrite, we created a plugin to improve this. As a result, we have a concise, fully rewritten method that is now returning a Reactor type.

```java
return foo_migrated(address).single().onErrorResume(e ->
    handle_migrated(e));

// The Inliner will see the above lambda expression with '
    handle_migrated' can be rewritten.
// The end result of the migration for the whole method will look like
    this:
@Override
public Mono<CompleteAddress> validate_migrated(Address address) {
    return foo_migrated(address).single().onErrorResume(this::
        handle_migrated);
}
```

Listing 4.20: The final steps for migrating a method reference

4.5 Migration cleanup

By now, we migrated the biggest part of the code to Reactor. There are a few edge cases left which could not be rewritten. For instance, methods with RxJava types as parameters, methods with local RxJava variables, or tests that have a too difficult setup to rewrite. Apart from the edge cases, all callers are now updated to use the Reactor types. Hence, it is time to clean up the code and try to remove all code that is not required any more. The aim is to stay as close as possible to the original code. We want the minimum amount of changes compared to the original version of the code. By limiting the number of changes and unnecessary statements we improve the quality of the migration. A migration that introduces many unnecessary statements requires manual labor to remove these afterward. Therefore we discuss three last operations that aim to minimize the difference with the original code.

Delete redundant wrappers

After performing the migration, it became apparent that many unnecessary conversions appeared in the code. In Listing 4.21 are a few examples. The interesting thing is that for every case, the method also accepts the expression that is wrapped in a call to the `RxJava2Adapter`. For instance, the `thenMany` and `switchIfEmpty` both accept a `Publisher` as an argument. The `Flux` and `Flowable` implement the `Publisher` interface. Therefore, the code is correct and will not give any problems. To reduce the unnecessary statements and improve the quality of the migration, we want to remove the wrappers.

```java
repository
    .upsert_migrated(MAPPING_1)
    .thenMany(RxJava2Adapter.fluxToFlowable(
        repository.findAll_migrated()));

repository.flux()
   .switchIfEmpty(
        RxJava2Adapter.fluxToFlowable(
            Flux.error(illegalState("No connection"))));

verifyAccess(
    () -> RxJava2Adapter.monoToSingle(
        adressService.validate_migrated(IRRELEVANT_ADDRESS)));
```

Listing 4.21: Three `RxJava2Adapter` conversions that may be omitted

We introduce a plugin in which we mark specific methods on classes as conversion methods. For instance, all the methods of the `RxJava2Adapter`, the `Mono#from`, and the `String#valueOf` method. Once Error Prone finds such a conversion method, the receiver type of that statement is compared with the type of the parameter that is inside the `RxJava2Adapter`. When the conversion is not required, we drop it.

Remove RxJava methods and default implementations

At the start of the migration, we introduce a significant amount of code. Mainly by duplicating existing methods and introducing default implementations for methods in interfaces. Now that we migrated the callers, we can remove the old methods that have RxJava return types. Besides removing the methods, we can also remove the default implementations of the interfaces.

We create a plugin that can do both, removing methods and removing the default implementations where that is required. Listing 4.22 is an example of rewriting an interface. There the default implementation of the method with the RxJava return type is removed. In addition, the default implementation of the Reactor type is removed.

Before performing the migration, there already are default implementations for some interfaces. Removing these default implementations would result reaching a non-compilable state. Therefore, we add a check in the plugin to verify that the method is introduced by us for migration purposes. This can be verified by checking that the method contains exactly one statement that is a delegating method. If this is the case, we can remove the default implementation, otherwise the default implementation will remain unchanged.

```java
// The following code block
@InlineMe(
  replacement = "RxJava2Adapter.fluxToFlowable(this.
     getMappings_migrated())",
  imports = "reactor.adapter.rxjava.RxJava2Adapter")
@Deprecated
default Flowable<Mapping> getMappings() {
  return RxJava2Adapter.fluxToFlowable(getMappings_migrated());
}

default Flux<Mapping> getMappings_migrated() {
  return RxJava2Adapter.flowableToFlux(getMappings());
}

// Is replaced by the following line
Flux<Mapping> getMappings_migrated();
```

Listing 4.22: Final steps of migrating an interface

While working on the migration, we encounter some methods that we could not migrate. In addition, there are callers that we cannot migrate. For instance, methods with RxJava types as parameters, or Mockito statements that use a difficult setup that we, therefore, mark as out of scope. Therefore, we implement a feature that prevents methods with specific names from being removed.

Remove _migrated suffix

The difference between the code before and after the migration is now confined to a simple replacement of the type, name of the method, and statements containing RxJava calls. An example of such a difference is given in Listing 4.23. The `-` marks what line is removed and the `+` indicates what line is added. It shows the difference for a method in an interface that is completely migrated. Besides changes in the interface, many callers have been updated according to the name changes for the migration. As a result, when looking at the difference, there are many changes that add an extra suffix `_migrated`.

```diff
- Flowable<String> saveMapping(Mapping mapping);
+ Flux<String> saveMapping_migrated(Mapping mapping);
```

Listing 4.23: Current state of the migration

To make the difference even smaller, we remove the `_migrated` suffix where possible. We use the *search and replace* feature of the IDE for this.

After performing this last step of the migration, we end up with concise rewrites. See Listing 4.24 for some examples. These are actual results of performing a migration.

```
// Rewrite an interface
- Flowable<String> saveMapping(Mapping mapping);
+ Flux<String> saveMapping(Mapping mapping);

// Rewrite a method
- Completable unsubscribeFromQueues(QueueSchemaMapping mapping) {
-   return Completable.fromAction(schemaRepositoryClient::refresh)
-       .andThen(
-           Maybe.fromCallable(
-               () -> consumers.remove(mapping.getExchange(), mapping.
  getRoutingTemplate())))
-       .flatMapCompletable(
-           consumer -> Completable.fromAction(consumer::close));
  }

+ Mono<Void> unsubscribeFromQueues(QueueSchemaMapping mapping) {
+   return Mono.fromRunnable(schemaRepositoryClient::refresh)
+       .then(
+           Mono.fromSupplier(
+               () -> consumers.remove(mapping.getExchange(), mapping.
  getRoutingTemplate())))
+       .flatMap(consumer -> Mono.fromRunnable(consumer::close))
+       .then();
  }

// Rewrite a test
  service
        .deleteMapping("ex-1", "ex-2")
-       .test()
-       .await()
-       .assertError(ItemNotFoundException.class);
+       .as(StepVerifier::create)
+       .verifyError(ItemNotFoundException.class);
```

Listing 4.24: Results of performing the migration using the proposed extensions

Chapter 5

Validate impact of extensions

In this chapter we elaborate on the design of the experiments that we use to validate our research. We want to set up experiments with the purpose to show how useful Refaster and Error Prone are for the library migration of RxJava to Reactor. Additionally, we want to assess how the proposed extensions contribute to the automation of this library migration. In this chapter, we explain how we approach this. We elaborate on challenges and how we measure the impact on library migration automation. Furthermore, we describe the results and discuss them.

5.1 Setup experiments

We want to be able to express to what extent the migration is automated. However, to the best of our knowledge, there is no standardized way of measuring the automation of library migration. For the research we are not creating a benchmark. The only goal is to assess and validate the impact of our proposed extensions.

For the validation of the research we set up two experiments that we perform on two codebases. To overcome codebase-specific biases we choose two completely different projects. One project is an industrial codebase and the other is an open source codebase. We aim to use two codebases that are of considerable size and contain at least 150,000 lines of code. Therefore, these codebases together are representative of a significant body of RxJava code.

The first part of the experiments is focused on using stock Error Prone and Refaster to migrate RxJava code to Reactor. Since Error Prone does not have out-of-the-box features that we can use for the migration we only use Refaster for the first part. For the second part, we focus on the proposed extensions so that we can validate their impact.

For both parts of the migration, we run Error Prone and Refaster many times. Since we are taking small and incremental steps when performing the migration, there will be many consecutive runs that rewrite code. We do not know in advance how many iterations are needed for the migration. Therefore, we keep iterating until there are no changes and we reach a fixed point.

We hypothesize that stock Error Prone and Refaster can support automating library migration. In the case of RxJava and Reactor, they can support migrating method bodies. Furthermore, we hypothesize that automated library migrations can benefit from features that can modify the structure of the code. That means modifying code other than the method body. We hypothesize that such extensions can positively impact the automation of library migration. In other words, such extensions can push the automation of library migration.

To get an overview of how the code evolves during the migration, we decide to count the identifier occurrences of the libraries RxJava, Reactor, and adapter. Therefore, we create an Error Prone plugin that analyzes the codebase and outputs all the occurrences of the libraries to a file of the respective library. For example, we count class variables, local variables, imports, return types, method invocations, method references, and parameters. When we come across an identifier, we check if the type comes from one of the libraries. In that case, we write it to a file so we can see how many times it occurs.

Additionally, we count how many files have RxJava 2 references, we do that by checking the import statements. A command used to count how many files have an RxJava 2 reference is in Listing 5.1. Similar commands are used for counting (Maven) modules and packages.

```
git grep -l io.reactivex -- '*.java' | wc -l
```

Listing 5.1: Bash command to count RxJava 2-referencing files in a directory

5.2 Case study on an industrial codebase

The industrial codebase is a private repository provided by Picnic. The codebase contains the core services and exists of more than 165.000 lines of code. For example, the core provides services for users, orders, deliveries, articles, and promotions.

Table 5.2, column *Original code* to *Automatic migration without extensions*, shows the extent to which RxJava code can be replaced using stock Error Prone and Refaster. Then, column *Automatic migration without extensions* to *Automatic migration with extensions* shows the further improvements unlocked by our proposed extensions to Error Prone and Refaster. The extensions show a significant impact on completely migrating files, modules, and packages. Lastly, we see that over 99% of the return types are removed by the proposed extensions.

Industrial codebase	Original code	Automatic migration without extensions	Automatic migration with extensions
RxJava identifier occurrences	2137	788	203
Adapter identifier occurrences	274	1526	590
Reactor identifier occurrences	9079	10269	10751
Files containing RxJava references	188	122	56
Maven modules containing RxJava references	65	61	32
Packages containing RxJava references	94	84	42
RxJava return types	225	225	2

Table 5.1: Codebase evolution during the industrial codebase migration

In Figure 5.1 we see how each iteration of the migration impacted the number of calls to the respective libraries. Note that Figure 5.1 uses a logarithmic scale on the y-axis to get a good view of how the RxJava and adapter calls change during the migration.

After iteration 16, the migration using stock Error Prone and Refaster reaches a fixed point. We then turn on the proposed extensions. Around iteration 28, we again reach a fixed point and turn on the plugins for the deletion of RxJava methods, together with the final cleanup.

Discussion

The industrial codebase starts with 2,137 RxJava identifier occurrences. After using stock Error Prone and Refaster that number is reduced to 788, which means that 63% of the RxJava code is migrated automatically. Note that the number of adapter calls increased from 274 to 1,526. With the proposed extensions we can migrate another 27% of the RxJava usages. Additionally, the number of adapter calls is reduced to 590, a reduction of 61%. In total, 90% of the RxJava identifier occurrences are removed. Notice that the number of adapter identifiers only slightly grows. This is mainly because call sites are updated using the `RxJava2Adapter`, to comply with the Reactor return type, and to ensure that the last 203 RxJava identifiers are handled correctly.

In Table 5.2 we see that the extensions have a significant impact on migrating whole classes and modules. The initial part of the migration can completely migrate 35% of the classes, meaning they no longer contain RxJava 2 references. Only 4 modules and 10 packages are fully migrated. With the extensions, an additional 44% of the modules and 44% of the packages are completely migrated.

Additionally, using the extensions we could migrate over 99% of the RxJava return types of method declarations. This means that the extension has a significant impact on migrating the RxJava return

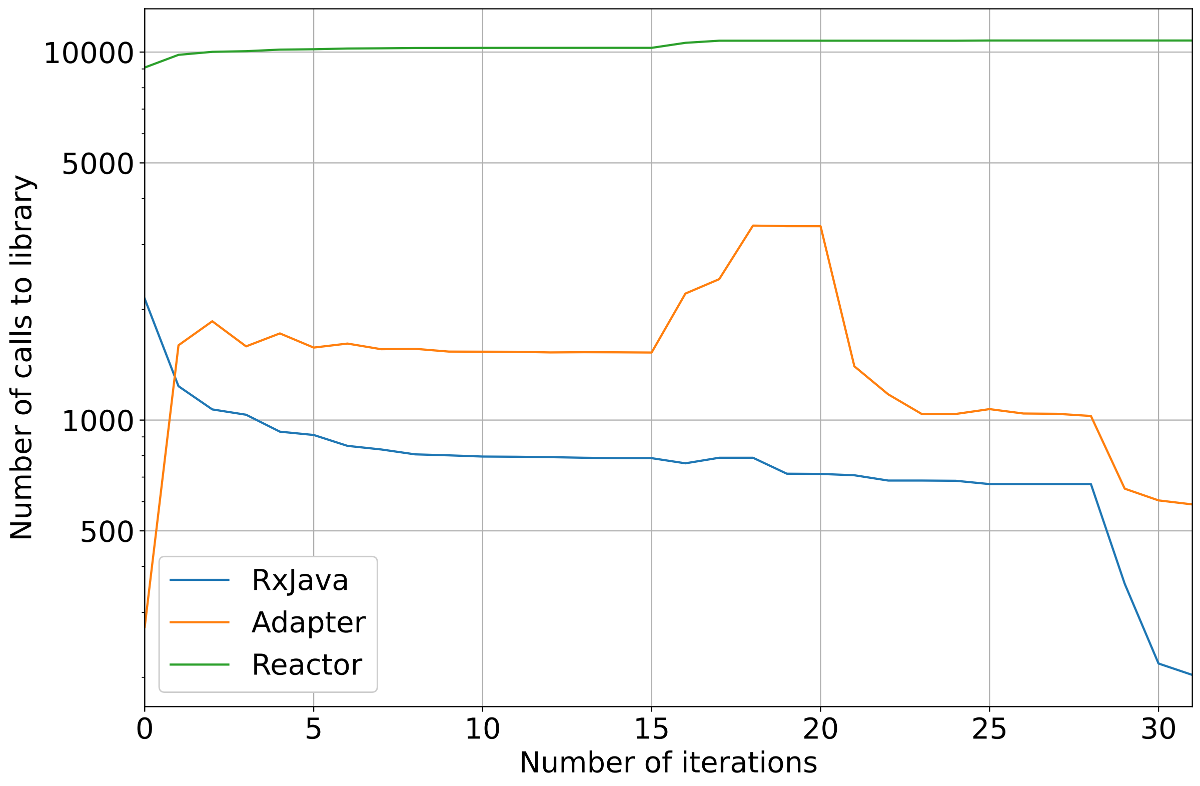

Figure 5.1: The progress of migrating the industrial codebase

types for the industrial codebase. Based on the number of RxJava identifier occurrences we know that there are still usages left inside the method bodies. While for the structure of the code this means we now almost exclusively use Reactor.

The last 10% of the remaining RxJava usages are mostly due to known limitations of the implementation. For instance, migration of local variables, class variables, and parameters is currently not supported. This in turn requires many adapter invocations to be retained. These limitations can be lifted by extending our proposed extensions. The setup of our extensions can largely be reused when lifting these limitations. However, we consider these limitations as out of scope for this thesis.

Additionally, we did not account for method references to the RxJava library. For instance, `Flowable::fromArray` and `Maybe::just` are left as is. We consider rewriting method references as out of scope.

Besides that, there are edge cases that are hard to automatically rewrite. For instance, the parameter `MaybeOnSubscribe` of the method `Maybe#create(MaybeOnSubscribe)` has a rather RxJava-specific implementation that has no counterpart in Reactor. This makes it difficult to rewrite such occurrences to Reactor. Finally, there are RxJava methods for which no Refaster template is provided, simply because they have very few usages. For example, `Observable#publish`, `Completable#timeOut`, and `Maybe#create` had only a few method invocations.

Another limitation in the industrial codebase is the usage of other libraries that use RxJava. See `getFlowable` in Listing 5.2 for an example of such a method call. The method call `getFlowable` is in an abstract class that is extended in the class. The method is almost completely migrated to Reactor, except for the `getFlowable` and the adapter call. For the industrial codebase, this prevents more than 200 adapter calls from being removed.

There are many different test setups that can prevent us from rewriting RxJava or adapter method invocations to Reactor. For instance, in Listing 5.3 we show a method that is in an abstract class. The abstract class is extended in two test classes. All usages of the methods, which is a total of 59, have to introduce adapter calls that cannot be optimized later. Since the method invocations `.test().await()` are now located in the abstract class, the test setup looks different, which can currently not be handled by Refaster. Therefore we are not able to rewrite these tests.

```java
@Override
public Mono<User> getUserById_migrated(String userId) {
  return RxJava2Adapter.flowableToFlux(
      getFlowable(
          "/api/{api}/user/{id}",
          User.class,
          API,
          userId))
      .next()
      .single();
}
```

Listing 5.2: A third-party method that cannot be inlined

```java
static <T> TestSubscriber<T> test(Completable completable) throws
    InterruptedException {
  return test(completable.toFlowable());
}

static <T> TestSubscriber<T> test(Publisher<T> publisher) throws
    InterruptedException {
return Flowable.fromPublisher(publisher).test().await();
}
```

Listing 5.3: Subset of the methods in an abstract class that are used to test RxJava code

5.3 Case study on an open source repository

For choosing an open source repository we use the GitHub search [68]. The search terms that are used to narrow the search is `language=Java`, `type=Code`. Finally, we want an actively maintained project of with at least 150,000 lines of code and uses RxJava 2. To make sure we find a repository that uses RxJava 2, we search specifically for `io.reactivex.Flowable`. We consider the first 10 pages on GitHub[1]. Based on the aforementioned requirements, we choose to use Gravitee [2] as open source repository for the case study. Gravitee is the only repository that meets all requirements *and* has over 150,000 lines of code.

Gravitee.io Access Management is an open source *Identity and Access Management* solution that offers a centralized authentication and authorization Service for applications and APIs [20]. The codebase consists of more than 265,000 lines of code. For the experiment, we branch from a specific commit on master[3].

Table 5.3 follows the same structure as Table 5.2. After running stock Error Prone and Refaster on the codebase, we see a small increase in the number of files and packages that have RxJava references. This is a side-effect of some Refaster templates. For instance, intermediate rewrite operations required the introduction of calls to RxJava factory methods and associated imports. In general, these new operations are elided again by subsequent rewrite operations.

In Figure 5.2 we see how each iteration of the migration impacts the number of calls to the respective libraries. Note that starting from iteration 15, we omit 20 iterations, each of which changed at most 15 method calls. This omission is indicated by the red line. At iteration 37 we turn on the extensions. Finally, at iteration 75 we run the plugins for deletion of the RxJava methods and additional clean up.

Discussion

For the open source project stock Error Prone and Refaster can migrate 48% of RxJava identifier occurrences. At the same time, 17,000 calls to the adapter are introduced. After applying the extensions, we end up with 19,8% fewer RxJava identifier occurrences. This entails that we migrated over 67% of all RxJava occurrences. Important to note is that at the same time we introduce over 13,000 adapter calls

[1]`https://github.com/search?l=&o=desc&q=io.reactivex.Flowable+language%3AJava&s=indexed&type=Code`
[2]`https://github.com/gravitee-io/gravitee-access-management`
[3]Gravitee commit SHA: `15b7250fba64ea41477ca5d5bb1cbfa96f2e3fc5`

Gravitee	Original code	Automatic migration without extensions	Automatic migration with extensions
RxJava identifier occurrences	29506	15188	9342
Adapter identifier occurrences	55	17089	13373
Reactor identifier occurrences	923	18032	24460
Files containing RxJava references	830	855	504
Maven modules containing RxJava references	58	58	42
Packages containing RxJava references	205	211	138
RxJava return types	2083	2083	751

Table 5.2: Codebase evolution during the Gravitee migration

throughout the migration.

We see a significant increase of adapter calls in Figure 5.2 just before iteration 20. Within four iterations the existing 17,089 adapter calls double to more than 35,171. This is primarily due to the introduction of Reactor type-returning methods, thereby requiring the introduction of conversions at each of their call sites. Subsequent iterations simplify these call sites by dropping pairs of conversions that cancel each other out.

In Table 5.3 we see that 40% of the files and 45% of packages are completely migrated to Reactor. Additionally, using the extensions, 64% of RxJava return types are migrated. Together with the 99% migrated RxJava return types on the industrial codebase, we conclude that the return type extension works well for these codebases.

The Gravitee migration significantly underperforms compared to the industrial codebase. There are several technical reasons for this. The main factor is the difference in code quality and style.

To start, the extensive nesting of lambda expressions in Gravitee causes problems for some Refaster templates. Some Refaster templates introduce a new lambda expression with an argument identifier that is expected to be unused in the expression's outer scope. This assumption is violated if such templates are applied in a nested scope, as the inner and outer applications introduce the same identifier. As a result, we reach a non-compilable state.

Gravitee contains many nested lambda expressions, with many RxJava variables and operators. Sometimes, this prevents our Refaster templates from matching. This particularly impacts Refaster templates that aim to simplify ("clean up") complex lambda expression introduced during earlier migration steps. In Listing 5.4 we show an example of a nested lambda expression. Note the nested `flatMap` with the parameter *e* which causes issues. In the example, code is omitted to better demonstrate the problem.

```java
@Override
public Mono<Domain> patch_migrated(String domainId, PatchDomain
    patchDomain, User principal) {
  return domainRepository.findById_migrated(domainId).flatMap(e -> {
    // Omitted code for clarity
    return validateDomain_migrated().then(Mono.defer(() ->
        domainRepoistory(update_migrated(toPatch))).flatMap(e -> /*
        Omitted code for clarity */);
  });
```

Listing 5.4: Method from Gravitee with nested `flatMap` operator

The tests of Gravitee are written in a different way than in the industrial codebase. Every RxJava type has a method `.test()` which returns an object that enables testing of the associated reactive streams. However, Gravitee tests save the resulting object in a local variable prior to dereferencing it.

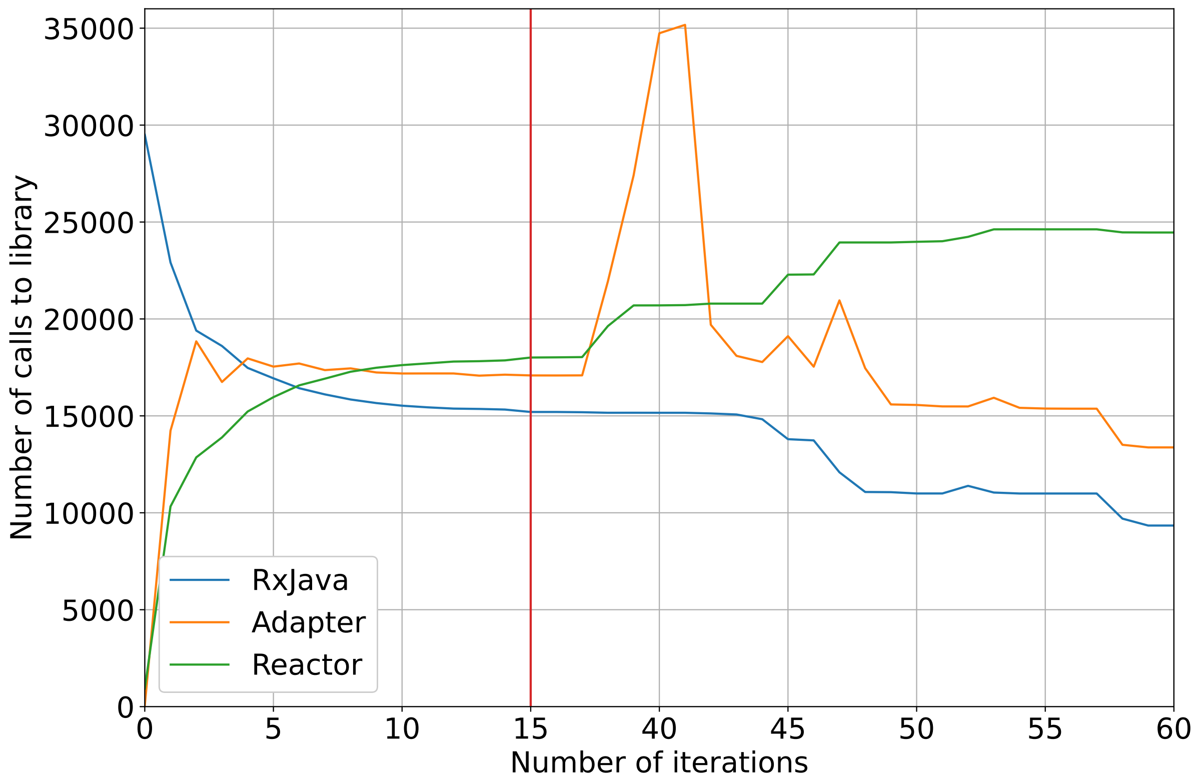

Figure 5.2: The progress of the Gravitee migration

See an example of such a local variable in Listing 5.5. In contrast, the industrial codebase does not utilize such local variables, instead producing longer chains of method invocations. Given current limitations, the latter allows for a significantly more complete migration of the test code.

```
TestObserver<ExtensionGrant> testObserver = RxJava2Adapter.monoToSingle
    (extensionGrantRepository.update_migrated(updatedExtension)).test();
testObserver.awaitTerminalEvent();
testObserver.assertComplete();
testObserver.assertNoErrors();
testObserver.assertValue(e -> e.getName().equals(updatedExtension.
    getName()));
```

Listing 5.5: Usage of the TestObserver that is saved in a local variable

The plugin for deleting RxJava-returning methods has a limitation that prevents us from deleting some of these methods. If a method cannot be deleted because of such an edge case, we manually exclude it from the deletion process. This exclusion process currently considers only method names, rather than full signatures. This means that in case of overloaded methods we delete strictly fewer methods than possible.

Chapter 6

Discussion

First, we discuss some limitations that apply to both codebases. Second, we discuss the answers to the research questions. Finally, we elaborate on the generalizability of this research and how future-proof it is.

Limitations for both codebases

In the previous sections we discussed limitations that were codebase specific. Now we describe limitations that apply to both codebases.

Although we have plugins specifically for rewriting tests, there are exceptions we cannot rewrite. The Mockito `thenAnswer` and `doAnswer` both return an `Answer` object. In most cases, we can do a valid migration for their method invocations. However, in some cases, the answer object is extracted or explicitly declared. Listing 6.1 shows examples of such cases. The underlying problem is that for both cases we cannot rewrite the return type because it is wrapped in `Answer<>`.

```
// Test in industrial codebase
when(ctarMock.findAssignmentsByTopics(any(), eq(true))).thenAnswer(
    getTopicLookup(idByTopic));
// Method signature
private static Answer<Flowable<Assignment>> getTopicLookup(Map<String,
    String> idByTopic) {

// Test in Gravitee
when(jweService.encryptAuthorization_migrated(anyString(), eq(client)))
    .then(new Answer<Single<String>>() {
  @Override
  public Single<String> answer(InvocationOnMock invocation) throws
      Throwable {
    return RxJava2Adapter.monoToSingle(Mono.just((String)invocation.
        getArguments()[0]));
  }
});
```

Listing 6.1: Edge case for rewriting a test with a method invocation in the `thenAnswer`

Table 5.2 shows that only two method declarations with RxJava return types are left in the final result of the industrial codebase migration. Rewriting these last two return types is prevented by the first example shown in Listing 6.1. We currently have no way of rewriting this return type to the Reactor counterpart. This is a rather specific edge case that we consider out of scope and occurs only twice in the industrial codebase.

In some Refaster templates, we specify extra type information for method calls: `RxJavaReactorMigrationUtil.<T, R>toJdkFunction`. When the template is applied, this can result in `<Throwable, SingleSource<Boolean>>toJdkFunction`. In most cases, this is a correct rewrite. However, sometimes we come across a method with a specific signature. Two examples of specific method signatures are shown in Listing 6.2. The `SingleSource` will in both cases result in non-compilable code because the method signatures do not completely match. For the first example, `SingleSource` is not compatible with `Single`. The latter will not match because of the question mark.

```java
// From the industrial codebase
private static <T, E extends RuntimeException> Function<Throwable,
    Single<T>> convertSingleError(Class<E> error, Function<E, ? extends
    RuntimeException> converter) {

// From Gravitee codebase
private Function<BotDetection, SingleSource<? extends BotDetection>>
    checkBotDetectionReleasedByApp(String domainId, String
    botDetectionId) {
```

Listing 6.2: Two examples of method signatures with RxJava types

6.1 RQ 1: To what extent can Error Prone and Refaster support automation of library migration?

We hypothesized that Error Prone and Refaster can support the automation of library migration. Stock Error Prone has features that can be useful while performing a migration. For instance, improving and optimizing code constructs, and support with inlining method calls to deprecated methods. However, without any manual labor, these features cannot be directly used. So currently Error Prone cannot support the automation of library migrations.

Stock Refaster offers *Refaster templates* that can directly be used to rewrite expressions and statements in method bodies. With these templates, method bodies can almost completely be automatically migrated. Since there are only changes inside method bodies, the structure of the code does not change. Refaster can be used the automate parts of library migration.

6.2 RQ 2: What extensions of Error Prone and Refaster benefit library migration?

While researching the current Error Prone and Refaster features, we identified opportunities for extensions. The focus was on the migration from *source library* RxJava 2 to *target library* Reactor.

When migrating from the source library to the target library, there can be method signatures that slightly differ. For instance, the checked exceptions or parameter types are not an exact match. To ensure the correctness of the rewrites, we created an extension to increase the accuracy with which we match code. Consequently, we can write a new body of Refaster templates that further optimize the code and benefit the migration.

Additionally, we found that automatically rewriting the return type of method declarations would impact the migration. The extension consists of several steps that safely migrate all method calls from the source library to the target library. An out-of-the-box feature of Error Prone that we extended is the Inliner plugin. To increase the support for the automation of library migration it can now inline private methods, default methods of interfaces, and member references. In addition, we have created plugins that migrate tests so that we can migrate the remaining callers. After migrating all callers, we can safely remove method declarations with return types of the source library. Furthermore, we built new and generalised existing extensions to remove unnecessary code and clean up code.

The aforementioned Error Prone and Refaster extensions, can support in the process of automating library migration and thus benefit library migrations.

6.3 RQ 3: What is the impact of the extensions on automation of library migration?

Our goal was to push the automation of library migration. To validate the impact of the extensions, we conducted a case study on an industrial codebase and an open source codebase. The migration consisted of migrating as much as possible with stock Error Prone and Refaster. Subsequently, the goal was to push the automated migration by using the proposed extensions. To validate the impact we compared these two states of the migration.

Stock Error Prone and Refaster are capable of migrating 63% of the industrial codebase and 48% for the open source codebase. The proposed extensions migrated another 27% and 20% respectively.

Another way of validating the impact is by counting the number of files, packages, and modules that contain RxJava 2 references. Removing all RxJava references means that a whole file, package, or module is completely migrated. With stock Error Prone and Refaster the impact overall was minimal. For the industrial codebase 35% of the files, 10% of the packages, and 6% of the modules were migrated. The open source codebase was negatively impacted because due to the side effects of Refaster templates that introduced RxJava 2 references. As a result, now 15 files and 6 packages also contain RxJava 2 references.

Using the extensions completely migrated another 35% of the classes, 42% of the packages, and 44% of the modules for the industrial codebase. For the open source codebase the percentages are 39%, 32%, and 27% respectively.

In summary, we validated that the proposed extensions have a significant impact. Therefore we can confirm that they have a positive impact on the automation of library migration.

6.4 Generalisability

There is another reactive programming library besides RxJava and Reactor, namely Smallrye Mutiny [69]. We investigate whether our extensions can support automation of library migrations from and to Smallrye Mutiny. Smallrye Mutiny provides adapters that convert between its API and the RxJava 2, RxJava 3, and Reactor APIs[1]. The adapters are essential for defining the conversions between the libraries in `MigrationTemplates`. Using these templates, our proposed extensions can migrate return types of method declarations in classes, in interfaces, and their associated callers, from the source library to the target library. Additionally, many Refaster templates have to be written to create a mapping between the API of the source library and the target library.

In summary, our extensions proposed can be used to migrate codebases between Smallrye Mutiny and the other three reactive libraries. This demonstrates the generalisability of our approach.

6.5 Future-proofness

The Java language has a rich history, having evolved over the past 26 years. While generally new versions of the language are backwards compatible, over the years significant new syntactical constructs and APIs have been introduced. Such new features normally require corresponding support to be added to Error Prone and Refaster. The former exists since 2012, while the latter was introduced in 2014. At the time of writing these tools are actively maintained, supporting all released versions from Java 8 through Java 17. New language syntax generally requires an extension of the AST. Error Prone and Refaster can support such extensions relatively easily, as they integrate directly against the Java compiler API.

As long as Error Prone is maintained, support for new Java versions will be added in a timely manner. Additionally, Java compiler internals have shown to be remarkably stable over the years. Combined, this shows that the proposed extensions will prove compatible with future versions or can be made so with little effort.

[1]Smallrye Mutiny adapters: `https://smallrye.io/smallrye-mutiny/guides/converters`

Chapter 7

Related work

We divide the related work into multiple categories. First, we discuss how IDEs can be used to refactor code. Second, we discuss research on type changes in Java. Next, we describe research on (automatically) applying semantic patches. Subsequently, we discuss existing tools that support a form of migration. Finally, we discuss research on library migrations.

7.1 Refactoring in IDEs

One way of performing migrations is to manually apply them using an integrated development environment (IDE). One can argue that an IDE also provides support for the operations required for a migration. An IDE often performs these actions correctly for small projects. However, when a project significantly grows in size, often the features of an IDE will not suffice. As a result, mistakes can be made or the refactoring operation is inaccurate. Wright et al. [55] found that because the IDE runs "*on a developer's local workstation, processing large collections of source code is often intractable*".

Besides IDEs, there are existing tools that use traditional regular expression-based matching to perform analysis and rewrite source code [70, 71]. However, these traditional tools lack semantic knowledge that is often required to perform complex transformations [55]. For instance, regular expression-based matching cannot differentiate between similarly named methods that are members of different classes. Aleksyuk & Itsykson [72] provide another example, the *structural search and replace*, which supports refactoring in general, but not so much migrations in particular. The structural search and replace is focused on simple replacements like a method name change or call arguments order. A migration typically goes beyond these operations, which are too complex or impossible to define in the structural search and replace [72]. As a result, an IDE provides limited support for manually performing migrations on large codebases.

7.2 Type changes

Dig & Johnson [73] found that, for evolving APIs, argument type and return type changes occur more often than method renaming. This was later validated by the research of Ketkar et al. [74] into type changes in Java. An interesting finding is that "*type changes are more common than renaming*" [74]. They conclude with a call for more tool support and further research in assisting developers by automating type changes. Their research considered: *variable*, *return*, *parameter*, and *field* type changes. Our research tries to improve the research on and support for automated changes of return types.

7.3 Semantic patches

Kang et al. [75] propose a tool that enables rewriting using semantic patches. Using a declarative syntax, which looks similar to the output of the Unix `diff` command (called a *patch*), one can apply these patches to a codebase. Since it looks like a *patch*, writing a semantic patch is relatively easy. The semantic patches are type-aware but the type information is not as extensive as Refaster. For instance, when an identifier is referenced, the type information like its class hierarchy is not resolved. Since one of the goals of the tools is to be lightweight, less type information is resolved. Hence, Refaster provides more type information. The tool described was ported from a program transformation tool designed for changes in C code. How the tool performs was only tested against Android code, which uses the Java ecosystem. In their test, only method calls to deprecated APIs were patched.

Besides that, there is one other tool that we already discussed in the background section. Wright et al. [55] created a tool called *ClangMR* which can be used to refactor large C++ codebases efficiently.

7.4 Existing tools

Besides the already mentioned tools and sources, we want to mention a few other tools in the industry. However, to the best of our knowledge, these tools have no associated academic source. We researched refactoring tools that support library migration to some extent. There are various tools that each have their way of supporting in this process.

There is a tool, called Rewrite [76], that contains predefined rewrites for some library migrations. For instance, refactoring JUnit test assertions to AssertJ and migrating the logging library Log4J to SLF4J. These migrations mostly consist of changing statements in method bodies and updating import statements.

Another tool is Sourcegraph that has a feature called *Batch Changes*[77], which allows for automatically applying rewrites to different repositories on Git. In YAML files one can define steps that should be executed on repositories. The actions that can be performed are commands that can be executed in bash. Sourcegraph provides Sourcegraph CLI [78] that supports additional features on top of regular bash features.

Finally, Comby [79] is a tool that matches and rewrites syntactic structures using a parse tree. Comby works for over 40 languages, it understands the basic syntax of code, strings, and comments. Since a parser is never fully implemented for a language, the expressivity is limited. With the syntactic structures, Comby offers more features than only using regular expressions.

7.5 Library migrations

Much research is done on library migrations in software development. Partially the focus is on trying to understand the nature of library migrations and collecting data on how a migration is performed. Alrubaye et al. [2] does much research in this area and researched the impact of library migrations on software quality. Based on this research Alrubaye et al. [3] created a tool to recommend how to migrate between two different Java libraries, at the method level.

Besides providing recommendations, tools exist to support rule-based transformations [80–82]. These tools can perform semantic changes that run over the whole codebase using the AST or the syntactic structure of a program. Most of these tools are used in the context of upgrading libraries. How easy it is to define the transformation rules for the tools is continuously improving.

There is also research on identifying these transformations automatically [15–17, 83]. Kim et al. [15] and Liu et al. [83] created tools that identify often recurring refactoring of fixes and automatically fix these. The research of Winter & Mametjanov [17] provides declarative-style rules to enable automatically upgrade Java libraries (like `java.lang` and `java.util`). The limitation of this research is that it does not have support for full semantic type information. This means that type information cannot be used. In Refaster and Error Prone this information is available.

Balaban et al. created a framework to update declarations and allocations sites to legacy classes for Java [16]. The framework only supports rewriting Java's standard library classes. Their plans for future work entail "*supporting migration of user-defined subclasses of legacy classes, and the introduction of the Adapter design pattern to wrap allocation sites and method parameters that could otherwise not be migrated*" [16]. In addition, they want to support migrations between methods that have different method signatures. Balaban et al. specifically mention the difference in the thrown exceptions. Our extension addresses and solves both these issues.

Logging libraries

Logging libraries are extensively used and often migrated [6, 9, 84]. However, for this research, we left logging libraries out of scope. Logging libraries often have small APIs that also have many similarities. This makes migrating a logging library easier. For instance, the SLF4J Migrator [85] can migrate from multiple logging libraries to SLF4J. In addition, many scripts can perform migrations between logging libraries. When the API of a library becomes richer it is harder to automatically migrate the library. As a result, the migration cannot be performed with a manageable script.

Migration detection

According to Alrubaye et al. [2, 9], a migration consists of migration rules and method mappings. Therefore, automating this part can support the process of completely automating a migration. The research on migration detection is related because it can help with the input for Refaster templates. When there is a need to have a more detailed mapping between the RxJava and Reactor APIs, the automatic detection of migration is useful.

Chen et al. [14] state that "*manually establishing API mappings is tedious due to a large number of APIs to be examined*". Besides that, it is hard to decide to what library one wants to migrate because there are many options to pick from. Therefore, much research is done on identifying mappings between migrations and library versions [13, 14, 86].

Automating library migration

As early as 2006 Dig et al. [73] stated that developers want to have a safe, easy, and behavior-preserving way to update their applications. The purpose is to overcome the manual labor that needs to be done to adapt to change.

In 2014, Teyton & Falleri [1] researched how existing projects have performed migrations. They used *library migration graphs* to identify relevant library replacements. Furthermore, their future work described plans to analyze the source code to detect migration patterns. With the goal "*to automatically apply them in new projects that are migrating*" [1].

Combining these goals and features could be the next step. Our research focused on automatically applying a self-defined migrating mapping.

Tool LibraryMigration

Research into tools that primarily focus on performing migrations from one library to another is scarce. In 2018, Aleksyuk & Itsykson [72] researched library migration and created a proof-of-concept tool **LibraryMigration**[1]. To specify a migration with the tool, one needs to comprehend a custom DSL, while writing such a specification for our implementation only requires knowledge of plain Java, as Refaster's DSL is Java.

The approach for this tool is to create a library metamodel, which is a set of extended finite state machines (EFSM). Each EFSM "*corresponds to a semantic object of the program that can be processed by the library or used for its action*" [72]. In other words, using the DSL, one has to model *all* the states and actions of the libraries. To add migration support for a new pair of libraries, a new metamodel for these specific libraries must be created. In our implementation, the equivalent of states and actions from the metamodel is the mapping of methods in Refaster templates.

The implementation of LibraryMigration is fundamentally different from how Error Prone works directly with the AST. The migration method is completely based on *traces* and consists of the following steps: trace extraction, trace mapping, equivalent trace calculation, mapping of a new trace back into AST nodes, and program transformation. In the research, the description of the step *trace extraction* contains a definition of a trace: "*A trace contains a list of code elements placed in the order of activation*" [72]. Every time LibraryMigration is unable to find an equivalent path, the user is required to help *during* the migration. Consequently, not all migrations can run completely automatically without manual intervention. Additionally, since the migration method is based on traces, it is unclear how it handles highly configurable code. As it is not uncommon to have a configurable code setup with feature flags for example. The *trace extraction* step may consider parts of the code as "dead code" due to the configurable setup, and therefore not migrate all code. This limitation is not discussed in the research.

A side-effect of the library metamodel is that the migration can be performed in two ways, namely from the source library to the target library and vice versa. In the tests of their tool, this two-way migration is used as extra validation of the quality of the migration. Our implementation only allows for migrating from the source to the target library, not the other way around.

As validation of LibraryMigration, only two minor library migrations were performed and one real-world project was used to validate. The tool contains a few examples of migrations, such as two logging libraries SLF4J and Log4J, and three libraries to make an HTTP request, namely Apache HttpClient,

[1]`https://github.com/h31/LibraryMigration`

OkHttp, and Java HttpClient. We do not consider migrating logging libraries because these libraries often have small and quite similar APIs, which makes migrating such libraries easier. Furthermore, since the example migrations of LibraryMigration contain only a limited body of code, we cannot be sure that their tool will work for a migration with the size of an API from RxJava and Reactor.

In the example migrations of LibraryMigration we observe that *all* rewrites are applied on code *inside* method bodies or import statements. This is comparable to the features of Refaster. Our implementation goes beyond that and also changes the return types of method declarations in classes and interfaces. It is unclear whether such rewrites are possible with LibraryMigration.

LibraryMigration is released on GitHub and is last modified in 2017. Additionally, it only supports Java 8 [87], which is the oldest existing long-term support (LTS) version. Currently, there are *two* newer LTS versions, Java 11 and Java 17. One of the reasons we do not run a quantitative comparison is that LibraryMigration only supports Java 8. Our implementation is based on Error Prone and Refaster which support Java versions from 8 till 17. In addition to this, as long as Error Prone is maintained, support for new Java versions will be added in a timely manner.

The implementation of LibraryMigration does not contain the same defined migrations as our implementation. To do a quantitative comparison, a migration needs to be defined in both tools. For LibraryMigration, the library metamodels have to be implemented using the custom DSL. For our implementation, this would mean that the Refaster templates have to be implemented for the mapping of the methods and the `MigrationTemplate`s. Consequently, implementing the migration for either one of the libraries would cost a significant amount of time.

In summary, the discussed limitations and the time required to describe a migration using one of the tools make a quantitative comparison infeasible labour-wise.

Chapter 8

Conclusion

We researched how Error Prone and Refaster can be used for automated library migration. The case study showed that using stock Error Prone and Refaster we can migrate RxJava code to Reactor. With Refaster templates we can rewrite expressions and statements in method bodies. Hence, we can partially migrate code inside the method body whilst remaining compatible with the RxJava return type. Consequently, some method bodies are fully migrated to Reactor and contain only one adapter conversion call to match the return type. Without extensions, we managed to migrate 63% of all RxJava identifier occurrences in an industrial codebase and 48% of such occurrences in an open source codebase.

Next, we identified what extensions could be useful to increase the automation of the library migration. The first extension improves the accuracy with which we match Refaster templates to increase the correctness and safety of the rewrites. Additionally, we propose a set of Error Prone extensions which facilitate rewriting return types of method declarations. Together, the extensions have a significant impact on library migration automation. The application of the proposed extensions showed that we can push the RxJava migration by another 27% and 19% respectively. As a result, we are able to migrate a total of 90% of all RxJava identifiers and 67% for the open source codebase.

Another way of validating the impact of the extensions is to count how many files contain RxJava 2 references. When all references are automatically removed from a file, it means that we migrated it completely. Stock Error Prone and Refaster removed RxJava references from 35% of the RxJava-containing files of the industrial codebase. For the open source codebase, no files were completely migrated. Using the proposed extensions, we managed to fully migrate an additional 35% of the RxJava 2-referencing files in the former codebase, and an initial 39% of such files in the latter codebase.

The extensions for Error Prone are mainly focused on rewriting return types for method declarations. The impact of said extensions was significant. For the industrial codebase we migrated 99% of the return types and for the open source codebase 64%. This means that for all these methods, the resulting method body consists of Reactor code and *all* callers are safely migrated to use the new return type.

In summary, we can conclude that Error Prone and Refaster are indeed suitable tools to help automate library migration. However, our proposed extensions to both Error Prone and Refaster have proven essential to bring automation of library migration to a new level. Of course, the quality of application code and the style of library use still have some impact on the effectiveness of our approach, as illustrated by the different levels of automation reached in the two case studies.

We have shown the effectiveness of the proposed extensions by applying them to two codebases. We believe these codebases to be representative of a wide range of software that uses RxJava, due to the select projects' distinct provenance and considerable size. Therefore, we are confident to achieve similar results for other migrations from RxJava to Reactor.

Generalization to other library migrations

After discussing the proposed extensions, we want to consider how they can be used by developers.

The research required the implementation of Error Prone extensions and improvements, as well as the definition of a large number of Refaster templates. The former are sufficiently generic so as to not be specific to RxJava and Reactor, enabling their application to a wide range of library migrations. The latter, on the other hand, are largely applicable *just* to the migration from RxJava to Reactor.

It follows that the research in this thesis provides a foundation for the definition of library migrations.

This foundation does require users to define a significant number of Refaster templates for any new pair of libraries between which one wishes to migrate. However, this effort does not require extensive Error Prone expertise. As Refaster templates are defined in Java, a language developers must already be familiar with, we expect that the proposed framework significantly reduces the effort involved in and experience required to define library migrations.

To validate the research, we have shown that the proposed extensions work on two completely different codebases. In addition, we considered another Reactive programming library Smallrye Mutiny [69], to increase the validity of the research (see section 6.4). If we were to generalize this research further, a new pair of libraries would have to be considered, which are not in the same domain as the original pair.

8.1 Threats to validity

Pair of libraries

Throughout the thesis, we focused on migrating a specific pair of libraries. Besides that, both the source library and target library are Reactive programming libraries that have similar APIs. They only have 5 and 2 base classes respectively, that can be converted to each other. Additionally, for most operations in the source library, there is a single direct equivalent in the target library. The aforementioned facts together can affect the validity.

Edge cases plugins

We make use of built-in Error Prone plugins and have created additional ones ourselves. In practice, we see that these plugins work well in the common case. This means that not all edge cases and arcane ways of writing Java code are supported. Consequently, the quality of the migration is dependent on how idiomatic and consistent the codebase is. The case study on the industrial codebase and on the open source codebase illustrate this. The former is on average much more idiomatic and consistent than the latter. This is one of the reasons that we could migrate less for the latter.

In general, these limitations can be lifted by solving more edge cases. However, we would expect diminishing returns on investment in writing code for these edge cases. Especially since creating an Error Prone plugin can be a quite complex task.

Experiments data collection

For the validation and assessment of the experiments we choose to count the identifier occurrences. We made this decision with the aim of proving an unambiguous metric that is easy to establish and comprehend.

Another option would have been to retrieve more data on the RxJava uses in the codebase. For instance, if we know how an identifier is used, we can reason about how difficult it is to replace that usage. On the other hand, it is hard to be concrete about this as it is difficult to quantify. Reasoning about the cognitive load for migrating an identifier is dependent on many factors. Such factors may include, for example, a developer's general level of experience and familiarity with the libraries involved.

We want to point out that not all remaining changes, that are left after performing the automated part of the migration, are equivalent. For example, suppose that after the automatic migration is done, there are still 100 minor rewrite operations left. If these 100 operations are all the same change, it can be done relatively easily. In contrast, if there are 100 different operations in 100 different places, this will be much harder. In the latter case, the cognitive load is higher because it is not something one can do on "autopilot". Additionally, it is much less likely that a set of heterogeneous changes can be performed through some ad hoc scripting.

Passing tests

We consider rewriting test code as part of the migration. However, ensuring that all tests pass is not feasible within the scope of this research. It is not the common case that makes rewriting tests hard. There are so many ways in which a developer can write tests, which results in many edge cases. Besides that, *what* the developer wants to test and *how*, impacts the test setup. Finally, there are many libraries and combinations thereof that can be used for testing. This results in many edge cases that must be accounted for in order to make all tests pass. Rewriting the code is often not the problem and is done correctly. However, due to the challenges mentioned above, it is easy to perform a rewrite that is incorrect with respect to the test setup. Consequently, the rewrite is valid but the test does not pass.

8.2 Future work

In the future we plan to extend the proposed extensions. For example, we want to extend by adding support for rewriting method parameters, local variables, and abstract methods. These improvements should increase library migration automation.

Furthermore, we plan to resolve the nested identifier problem in Refaster. This occurs when a Refaster template introduces an identifier for a lambda expression and the template is executed twice in nested code. For instance, the parameter for a `flatMap` is a lambda expression that also contains a lambda expression that invokes a `flatMap`.

Additionally, we want to write more Refaster templates. Currently, we have around 275 Refaster templates available[1], while the RxJava API contains over 1000 methods. In the future we plan to increase the coverage of the RxJava API by writing more templates. Besides writing the templates themselves, we plan to improve the test coverage for the templates.

Tests of Error Prone plugins

While working on the migration, we add tests for the edge cases that we encounter. Since it is hard to track every edge case, writing lots of tests can help to keep an overview. Performing the migration on other codebases may reveal more edge cases and potential improvements. We plan to extend the body of tests and therefore generalise and improve the stability of the plugins.

Remove methods plugin

Currently, for the last step of the migration, we remove the `_migrated` suffix by using the *search and replace* feature provided by the IDE. This is a step in the migration that needs to be performed manually. Alternatively, we could fairly easily create an Error Prone plugin for removing the `_migrated` suffix. The plugin has to match all the method invocations, method references, and methods names to flag `_migrated` suffixes.

Dependency management

Throughout this thesis, we only focused on code written in Java. In modern Java, developers do not directly invoke the compiler but rather use a build system such as Maven or Gradle. Therefore, a complete migration would also involve resolving declared dependencies. The dependencies with the target library are introduced, while the libraries associated with the source library would have to be manually removed. This research can be extended by exploring automated dependency resolution during a migration.

Adapter conversions

The RxJava to Reactor migration, extensively discussed in this research, relies on a set of adapter methods enabling conversions from source to target types and vice versa. In the specific case discussed, these adapter operations were provided out-of-the-box by one of the libraries involved. Generalizing the approach taken to other sets of libraries crucially depends on similar conversion operations to be available. Such adapter methods may not exist, requiring a developer tasked with the migration to manually define them. If a migration can be automated to the extent that upon completion few or no conversion methods remain, it is not required for adapter methods to actually be *functional*. They may exist merely as syntactical constructs that allow intermediate states of the migration to comprise valid ("compilable") source code. Such adapter methods can even accept arbitrary additional parameters with the sole purpose of transferring the state between consecutive migration iterations. Once the migration has reached a fixed point the remaining adapter usages must then be manually replaced with functional code.

[1]`https://github.com/rickie/msc-thesis-source-code`

Comparison

In section 7.5 we discussed related work from Aleksyuk & Itsykson [72]. Their work is the closest related to our research that we have found. Fundamentally, however, there is a significant difference in how their LibraryMigration tool works compared to Error Prone. LibraryMigration depends on traces whereas Error Prone depends on the AST.

There are a few limitations to LibraryMigration, namely that it is not maintained since 2017 and is only compatible with Java 8. Additionally, only two minor library migrations and one real-world project are migrated as validation. Finally, the migrations performed by LibraryMigration only rewrite code that is inside method bodies.

In the future, it would be interesting to make a more detailed comparison between LibraryMigration and our implementation. A comparison would consist of two parts. First, for LibraryMigration, create a library metamodel that implements all states and actions using the custom DSL. Second, for our implementation, this would mean implementing the Refaster templates for mapping the methods and writing the `MigrationTemplate`s. Consequently, defining a library migration is a labour-intensive task for both tools. Therefore, performing a detailed comparison would be a significant amount of work.

Acknowledgements

This project is the result of a journey of several months, involving people to which I owe deep gratitude. First and foremost, Stephan Schroevers.

It all started with a few video calls from Stephan's famous corner at Picnic. I was looking for a project where I could really go all the way. A project in which I could truly put my heart and soul. I am grateful to Stephan for answering this request fully and wholeheartedly, as it is safe to say that this most definitely worked out as desired.

I am grateful for the countless conversations, answers to a plethora of questions, and for all the pair programming sessions. These moments, combined with your competence in explaining even the hardest topics, have elevated my knowledge and experience to a great extent. Despite your busy schedule, you were always able to find time for me. Your passion and motivation are endless and unmatched and make it a pleasure to work with you.

At times the project seemed impossible, yet you helped me take the hurdles necessary to produce a work of which we can be proud. Stephan, I truly enjoyed working together and would like to thank you for that.

Furthermore, I would like to thank my dearest family for your unconditional love and support. I cannot thank you enough. You are always willing to help and encourage me, especially when I have a tough deadline and need all the focus I can get. Without you, this project would have been significantly more difficult.

Last but not least, I would like to thank my academic supervisor, Clemens Grelck. You have always been able to elucidate the direction of my research when I needed it. Thank you for all the effort you put in providing feedback in a timely manner every time. Additionally, I am grateful that you suggested submitting the paper to SAC 2022. With a 12-day deadline, the schedule was tight as the thesis was not yet finished. This worked out however, and the submission was a success. I am happy that we took this opportunity and that we created something I am very proud of. I am equally thrilled that Stephan and you are co-authors of the paper.

Acronyms

AST Abstract Syntax Tree. 4, 7, 9–11, 48, 49, 54
CST Concrete Syntax Tree. 7, 9
DSL domain-specific language. 1, 11, 12, 49, 50
EFSM extended finite state machines. 49
IDE integrated development environment. 3, 36, 47, 53
IR intermediate representation. 7
JDK Java Development Kit. 10
JLS Java Language Specification. 6
JVM Java Virtual Machine. 6
LTS long-term support. 50

Bibliography

[1] C. Teyton, J. Falleri, and X. Blanc, "Mining library migration graphs," in *2012 19th Working Conference on Reverse Engineering*, 2012, pp. 289–298. DOI: 10.1109/WCRE.2012.38.

[2] H. Alrubaye, D. Alshoaibi, E. Alomar, M. W. Mkaouer, and A. Ouni, "How Does Library Migration Impact Software Quality and Comprehension? An Empirical Study," en, in *Reuse in Emerging Software Engineering Practices*, S. Ben Sassi, S. Ducasse, and H. Mili, Eds., vol. 12541, Cham: Springer International Publishing, 2020, pp. 245–260, ISBN: 978-3-030-64693-6 978-3-030-64694-3. DOI: 10.1007/978-3-030-64694-3_15.

[3] H. Alrubaye, M. W. Mkaouer, I. Khokhlov, L. Reznik, A. Ouni, and J. Mcgoff, "Learning to recommend third-party library migration opportunities at the API level," en, *Applied Soft Computing*, vol. 90, p. 106 140, May 2020, ISSN: 15684946. DOI: 10.1016/j.asoc.2020.106140.

[4] R. G. Kula, D. M. German, A. Ouni, T. Ishio, and K. Inoue, "Do developers update their library dependencies?" *Empirical Software Engineering*, vol. 23, no. 1, pp. 384–417, 2018.

[5] H. He, Y. Xu, Y. Ma, Y. Xu, G. Liang, and M. Zhou, "A multi-metric ranking approach for library migration recommendations," in *2021 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, Mar. 2021, pp. 72–83. DOI: 10.1109/SANER50967.2021.00016.

[6] C. Teyton, J.-R. Falleri, M. Palyart, and X. Blanc, "A study of library migrations in java," en, *Journal of Software: Evolution and Process*, vol. 26, no. 11, pp. 1030–1052, 2014, ISSN: 2047-7481. DOI: 10.1002/smr.1660.

[7] D. Silva, N. Tsantalis, and M. T. Valente, "Why we refactor? confessions of github contributors," in *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. FSE 2016, Seattle, WA, USA: Association for Computing Machinery, 2016, pp. 858–870, ISBN: 9781450342186. DOI: 10.1145/2950290.2950305. [Online]. Available: https://doi.org/10.1145/2950290.2950305.

[8] M. Fowler, *Refactoring: improving the design of existing code*. Addison-Wesley Professional, 2018.

[9] H. A. T. Al-Rubaye, "Towards the Automation of Migration and Safety of Third-Party Libraries," en, p. 191, 2020.

[10] H. Alrubaye, M. W. Mkaouer, and A. Ouni, "On the Use of Information Retrieval to Automate the Detection of Third-Party Java Library Migration at the Method Level," in *2019 IEEE/ACM 27th International Conference on Program Comprehension (ICPC)*, May 2019, pp. 347–357. DOI: 10.1109/ICPC.2019.00053.

[11] A. Bosu, M. Greiler, and C. Bird, "Characteristics of useful code reviews: An empirical study at microsoft," in *2015 IEEE/ACM 12th Working Conference on Mining Software Repositories*, 2015, pp. 146–156. DOI: 10.1109/MSR.2015.21.

[12] C. Sadowski, E. Soderberg, L. Church, M. Sipko, and A. Bacchelli, "Modern code review: A case study at google," in *Proceedings of the 40th International Conference on Software Engineering: Software Engineering in Practice*, ser. ICSE-SEIP '18, Gothenburg, Sweden: Association for Computing Machinery, 2018, pp. 181–190, ISBN: 9781450356596. DOI: 10.1145/3183519.3183525. [Online]. Available: https://doi.org/10.1145/3183519.3183525.

[13] H. Alrubaye, M. W. Mkaouer, and A. Ouni, "MigrationMiner: An Automated Detection Tool of Third-Party Java Library Migration at the Method Level," in *2019 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, Sep. 2019, pp. 414–417. DOI: 10.1109/ICSME.2019.00072.

[14] C. Chen, "Similarapi: Mining analogical apis for library migration," in *2020 IEEE/ACM 42nd International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*, Oct. 2020, pp. 37–40.

[15] M. Kim, M. Gee, A. Loh, and N. Rachatasumrit, "Ref-finder: A refactoring reconstruction tool based on logic query templates," in *Proceedings of the Eighteenth ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. FSE '10, Santa Fe, New Mexico, USA: Association for Computing Machinery, 2010, pp. 371–372, ISBN: 9781605587912. DOI: `10.1145/1882291.1882353`. [Online]. Available: `https://doi.org/10.1145/1882291.1882353`.

[16] I. Balaban, F. Tip, and R. Fuhrer, "Refactoring support for class library migration," in *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, ser. OOPSLA '05, San Diego, CA, USA: Association for Computing Machinery, 2005, pp. 265–279, ISBN: 1595930310. DOI: `10.1145/1094811.1094832`. [Online]. Available: `https://doi-org.proxy.uba.uva.nl/10.1145/1094811.1094832`.

[17] V. L. Winter and A. Mametjanov, "Generative programming techniques for java library migration," in *Proceedings of the 6th international conference on Generative programming and component engineering*, 2007, pp. 185–196.

[18] L. Wasserman, "Scalable, example-based refactorings with refaster," en, in *Proceedings of the 2013 ACM Workshop on Workshop on Refactoring Tools - WRT '13*, Indianapolis, Indiana, USA: ACM Press, 2013, pp. 25–28, ISBN: 978-1-4503-2604-9. DOI: `10.1145/2541348.2541355`.

[19] Google, *Google/error-prone*, Google. [Online]. Available: `https://github.com/google/error-prone` (visited on 10/23/2021).

[20] Gravitee.io. (2021). "Open source API management platform - gravitee," [Online]. Available: `https://www.gravitee.io` (visited on 11/01/2021).

[21] B. Graham, P. N. Leroux, and T. Landry, "Using static and runtime analysis to improve developer productivity and product quality," p. 17, Apr. 2008.

[22] C. Flanagan, K. R. M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, and R. Stata, "Extended static checking for java," *SIGPLAN Not.*, vol. 37, no. 5, pp. 234–245, May 2002, ISSN: 0362-1340. DOI: `10.1145/543552.512558`. [Online]. Available: `https://doi.org/10.1145/543552.512558`.

[23] P. Louridas, "Static code analysis," *IEEE Software*, vol. 23, no. 4, pp. 58–61, Jul. 2006, ISSN: 1937-4194. DOI: `10.1109/MS.2006.114`.

[24] Checkstyle, *Checkstyle/checkstyle*. [Online]. Available: `https://github.com/checkstyle/checkstyle` (visited on 04/21/2021).

[25] P. S. C. A. Project. (Aug. 2017). "Documentation index — PMD source code analyzer," [Online]. Available: `https://pmd.github.io/pmd-6.33.0/` (visited on 04/21/2021).

[26] F. Logozzo and M. Fähndrich, "On the relative completeness of bytecode analysis versus source code analysis," in *International Conference on Compiler Construction*, Springer, 2008, pp. 197–212.

[27] Oracle. (Jul. 2011). "Chapter 14. blocks and statements," [Online]. Available: `https://docs.oracle.com/javase/specs/jls/se7/html/jls-14.html#jls-14.20.3` (visited on 04/23/2021).

[28] D. A. Tomassi, "Bugs in the wild: Examining the effectiveness of static analyzers at finding real-world bugs," in *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2018, Lake Buena Vista, FL, USA: Association for Computing Machinery, 2018, pp. 980–982, ISBN: 9781450355735. DOI: `10.1145/3236024.3275439`. [Online]. Available: `https://doi.org/10.1145/3236024.3275439`.

[29] SpotBugs. (2021). "Introduction — spotbugs 4.2.3 documentation," [Online]. Available: `https://spotbugs.readthedocs.io/en/stable/introduction.html` (visited on 04/21/2021).

[30] T. Kuhn and O. Thomann. (Nov. 20, 2006). "Abstract syntax tree," [Online]. Available: `https://www.eclipse.org/articles/Article-JavaCodeManipulation_AST/index.html` (visited on 04/20/2021).

[31] K. D. (D. Cooper, *Intermediate Representations*, 2nd ed. Amsterdam ; Boston : Elsevier/Morgan Kaufmann, 2012, ISBN: 9780120884780.

[32] A. V. Aho and A. V. Aho, Eds., *Compilers: principles, techniques, & tools*, 2nd ed, OCLC: ocm70775643, Boston: Pearson/Addison Wesley, 2007, 1009 pp., ISBN: 978-0-321-48681-3.

[33] U. Khedker, A. Sanyal, and B. Sathe, *Data flow analysis: theory and practice*. CRC Press, 2009, pp. 1–19.

[34] F. E. Allen and J. Cocke, "A program data flow analysis procedure," *Communications of the ACM*, vol. 19, no. 3, p. 137, 1976.

[35] Checker Framework. (2021). "The checker framework," The Checker Framework, [Online]. Available: `https://checkerframework.org/` (visited on 11/02/2021).

[36] E. Aftandilian, R. Sauciuc, S. Priya, and S. Krishnan, "Building Useful Program Analysis Tools Using an Extensible Java Compiler," in *2012 IEEE 12th International Working Conference on Source Code Analysis and Manipulation*, Sep. 2012, pp. 14–23. DOI: `10.1109/SCAM.2012.28`.

[37] C. Vassallo, G. Grano, F. Palomba, H. C. Gall, and A. Bacchelli, "A large-scale empirical exploration on refactoring activities in open source software projects," *Science of Computer Programming*, vol. 180, pp. 1–15, 2019.

[38] G. Bavota, A. De Lucia, M. Di Penta, R. Oliveto, and F. Palomba, "An experimental investigation on the innate relationship between quality and refactoring," *Journal of Systems and Software*, vol. 107, pp. 1–14, 2015, ISSN: 0164-1212. DOI: `https://doi.org/10.1016/j.jss.2015.05.024`. [Online]. Available: `https://www.sciencedirect.com/science/article/pii/S0164121215001053`.

[39] M. Kim, T. Zimmermann, and N. Nagappan, "An empirical study of refactoringchallenges and benefits at microsoft," *IEEE Transactions on Software Engineering*, vol. 40, no. 7, pp. 633–649, Jul. 2014, ISSN: 1939-3520. DOI: `10.1109/TSE.2014.2318734`.

[40] J. Pantiuchina, F. Zampetti, S. Scalabrino, V. Piantadosi, R. Oliveto, G. Bavota, and M. D. Penta, "Why developers refactor source code: A mining-based study," *ACM Trans. Softw. Eng. Methodol.*, vol. 29, no. 4, Sep. 2020, ISSN: 1049-331X. DOI: `10.1145/3408302`. [Online]. Available: `https://doi.org/10.1145/3408302`.

[41] S. G. Eick, T. L. Graves, A. F. Karr, J. S. Marron, and A. Mockus, "Does code decay? assessing the evidence from change management data," *IEEE Transactions on Software Engineering*, vol. 27, no. 1, pp. 1–12, 2001. DOI: `10.1109/32.895984`.

[42] T. Mens, M. Wermelinger, S. Ducasse, S. Demeyer, R. Hirschfeld, and M. Jazayeri, "Challenges in software evolution," in *Eighth International Workshop on Principles of Software Evolution (IWPSE'05)*, Sep. 2005, pp. 13–22. DOI: `10.1109/IWPSE.2005.7`.

[43] K. H. Bennett and V. T. Rajlich, "Software maintenance and evolution: A roadmap," in *Proceedings of the Conference on The Future of Software Engineering*, ser. ICSE '00, Limerick, Ireland: Association for Computing Machinery, 2000, pp. 73–87, ISBN: 1581132530. DOI: `10.1145/336512.336534`. [Online]. Available: `https://doi.org/10.1145/336512.336534`.

[44] B. P. Lientz and E. B. Swanson, "Problems in application software maintenance," *Commun. ACM*, vol. 24, no. 11, pp. 763–769, Nov. 1981, ISSN: 0001-0782. DOI: `10.1145/358790.358796`. [Online]. Available: `https://doi-org.proxy.uba.uva.nl/10.1145/358790.358796`.

[45] N. Schneidewind, "The state of software maintenance," *IEEE Transactions on Software Engineering*, vol. SE-13, no. 3, pp. 303–310, Mar. 1987, ISSN: 1939-3520. DOI: `10.1109/TSE.1987.233161`.

[46] J. Thanaki, *Python natural language processing*. Packt Publishing Ltd, 2017.

[47] C. Goddard, *Semantic analysis: A practical introduction*. Oxford University Press, 2011.

[48] B. P. Lientz, E. B. Swanson, and G. E. Tompkins, "Characteristics of application software maintenance," *Commun. ACM*, vol. 21, no. 6, pp. 466–471, Jun. 1978, ISSN: 0001-0782. DOI: `10.1145/359511.359522`. [Online]. Available: `https://doi-org.proxy.uba.uva.nl/10.1145/359511.359522`.

[49] N. Chapin, J. E. Hale, K. M. Khan, J. F. Ramil, and W.-G. Tan, "Types of software evolution and software maintenance," *Journal of software maintenance and evolution: Research and Practice*, vol. 13, no. 1, pp. 3–30, 2001.

[50] W. F. Opdyke, "Refactoring object-oriented frameworks," 1992.

[51] T. C. Lethbridge and J. Singer, "Understanding software maintenance tools: Some empirical research," in *in Workshop on Empirical Studies of Software Maintenance*, 1997.

[52] R. Rolim, G. Soares, R. Gheyi, T. Barik, and L. D'Antoni, "Learning Quick Fixes from Code Repositories," *arXiv:1803.03806 [cs]*, Sep. 2018. arXiv: `1803.03806 [cs]`.

[53] R. Potvin and J. Levenberg, "Why google stores billions of lines of code in a single repository," *Commun. ACM*, vol. 59, no. 7, pp. 78–87, Jun. 2016, ISSN: 0001-0782. DOI: `10.1145/2854146`. [Online]. Available: `https://doi.org/10.1145/2854146`.

[54] C. Sadowski, J. van Gogh, C. Jaspan, E. Soederberg, and C. Winter, "Tricorder: Building a program analysis ecosystem," in *International Conference on Software Engineering (ICSE)*, 2015.

[55] H. K. Wright, D. Jasper, M. Klimek, C. Carruth, and Z. Wan, "Large-Scale Automated Refactoring Using ClangMR," in *2013 IEEE International Conference on Software Maintenance*, Sep. 2013, pp. 548–551. DOI: `10.1109/ICSM.2013.93`.

[56] C. Sadowski, E. Aftandilian, A. Eagle, L. Miller-Cushon, and C. Jaspan, "Lessons from building static analysis tools at Google," *Communications of the ACM*, vol. 61, no. 4, pp. 58–66, Mar. 2018, ISSN: 0001-0782, 1557-7317. DOI: `10.1145/3188720`.

[57] OpenJDK, *JDK*, Accessed: 7-20-2021, Mar. 16, 2021. [Online]. Available: `http://openjdk.java.net/projects/jdk/` (visited on 04/30/2021).

[58] J. Team, *Eclipse java development tools (JDT) overview — the eclipse foundation*, 2021. [Online]. Available: `https://www.eclipse.org/jdt/overview.php` (visited on 05/01/2021).

[59] Google, *Refaster templates*, https://errorprone.info/docs/refaster, Accessed: 10-23-2021. [Online]. Available: `https://errorprone.info/docs/refaster`.

[60] RxJava, *RxJava: Reactive extensions for the JVM*, Sep. 7, 2021. [Online]. Available: `https://github.com/ReactiveX/RxJava` (visited on 09/07/2021).

[61] E. Bainomugisha, A. L. Carreton, T. v. Cutsem, S. Mostinckx, and W. d. Meuter, "A survey on reactive programming," *ACM Comput. Surv.*, vol. 45, no. 4, Aug. 2013, ISSN: 0360-0300. DOI: `10.1145/2501654.2501666`. [Online]. Available: `https://doi-org.proxy.uba.uva.nl/10.1145/2501654.2501666`.

[62] Stack Overflow. (2021). "Stack overflow developer survey 2021," [Online]. Available: `https://insights.stackoverflow.com/survey/2021/?utm_source=social-share&utm_medium=social&utm_campaign=dev-survey-2021` (visited on 08/22/2021).

[63] Reactor, *Reactor core*, Sep. 6, 2021. [Online]. Available: `https://github.com/reactor/reactor-core` (visited on 09/07/2021).

[64] Project Reactor. (2021). "RxJava2adapter (reactor-adapter 3.4.4)." Accessed: 10-23-2021, [Online]. Available: `https://projectreactor.io/docs/adapter/release/api/reactor/adapter/rxjava/RxJava2Adapter.html` (visited on 09/07/2021).

[65] E. Gamma, R. Helm, R. Johnson, J. Vlissides, and D. Patterns, *Elements of reusable object-oriented software*. Addison-Wesley Reading, Massachusetts, 1995, vol. 99.

[66] A. Zerouali and T. Mens, "Analyzing the evolution of testing library usage in open source java projects," in *2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, Feb. 2017, pp. 417–421. DOI: `10.1109/SANER.2017.7884645`.

[67] Oracle. (2021). "Generic types (the java™ tutorials > learning the java language > generics (updated))," [Online]. Available: `https://docs.oracle.com/javase/tutorial/java/generics/types.html` (visited on 09/10/2021).

[68] *Build software better, together*, https://github.com, Accessed: 11-17-2021.

[69] Smallrye, *Mutiny!* Accessed: 23-10-21. [Online]. Available: `https://smallrye.io/smallrye-mutiny` (visited on 10/23/2021).

[70] M. C. Franky and J. A. Pavlich-Mariscal, "Improving implementation of code generators: A regular-expression approach," in *2012 XXXVIII Conferencia Latinoamericana En Informatica (CLEI)*, Oct. 2012, pp. 1–10. DOI: `10.1109/CLEI.2012.6427199`.

[71] Z. Cai, L. Zhao, X. Wang, X. Yang, J. Qin, and K. Yin, "A pattern-based code transformation approach for cloud application migration," in *2015 IEEE 8th International Conference on Cloud Computing*, Jun. 2015, pp. 33–40. DOI: `10.1109/CLOUD.2015.15`.

[72] A. O. Aleksyuk and V. M. Itsykson, "Semantics-driven migration of java programs: A practical application," *Aut. Control Comp. Sci.*, vol. 52, no. 7, pp. 581–588, Dec. 1, 2018, ISSN: 1558-108X. DOI: `10.3103/S0146411618070027`. [Online]. Available: `https://doi.org/10.3103/S0146411618070027` (visited on 11/02/2021).

[73] D. Dig and R. Johnson, "How do APIs evolve? A story of refactoring," en, *Journal of Software Maintenance and Evolution: Research and Practice*, vol. 18, no. 2, pp. 83–107, 2006, ISSN: 1532-0618. DOI: `10.1002/smr.328`.

[74] A. Ketkar, N. Tsantalis, and D. Dig, "Understanding type changes in java," in *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2020, Virtual Event, USA: Association for Computing Machinery, 2020, pp. 629–641, ISBN: 9781450370431. DOI: `10.1145/3368089.3409725`. [Online]. Available: `https://doi-org.proxy.uba.uva.nl/10.1145/3368089.3409725`.

[75] H. J. Kang, F. Thung, J. L. Lawall, G. Muller, L. Jiang, and D. Lo, "Automating Program Transformation for Java Using Semantic Patches," RR-9256, Feb. 2019. [Online]. Available: `https://hal.inria.fr/hal-02023368`.

[76] OpenRewrite, *Semantic code search and transformation*, Accessed: 10-23-2021, Oct. 16, 2021. [Online]. Available: `https://github.com/openrewrite/rewrite` (visited on 10/16/2021).

[77] Sourcegraph. (2021). "Batch Changes - Sourcegraph docs," [Online]. Available: `https://docs.sourcegraph.com/batch_changes` (visited on 10/16/2021).

[78] ——, *Sourcegraph CLI*, Oct. 14, 2021. [Online]. Available: `https://github.com/sourcegraph/src-cli` (visited on 10/16/2021).

[79] Comby, *Comby · structural code search and replace for every language.* [Online]. Available: `https://comby.dev` (visited on 10/16/2021).

[80] Chow and Notkin, "Semi-automatic update of applications in response to library changes," in *1996 Proceedings of International Conference on Software Maintenance*, Nov. 1996, pp. 359–368. DOI: `10.1109/ICSM.1996.565039`.

[81] W. Wu, Y.-G. Guéhéneuc, G. Antoniol, and M. Kim, "Aura: A hybrid approach to identify framework evolution," in *2010 ACM/IEEE 32nd International Conference on Software Engineering*, vol. 1, May 2010, pp. 325–334. DOI: `10.1145/1806799.1806848`.

[82] L. Wu, Q. Wu, G. Liang, Q. Wang, and Z. Jin, "Transforming code with compositional mappings for api-library switching," in *2015 IEEE 39th Annual Computer Software and Applications Conference*, vol. 2, Jul. 2015, pp. 316–325. DOI: `10.1109/COMPSAC.2015.29`.

[83] K. Liu, A. Koyuncu, D. Kim, and T. F. Bissyandé, "Tbar: Revisiting template-based automated program repair," in *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA 2019, Beijing, China: Association for Computing Machinery, 2019, pp. 31–42, ISBN: 9781450362245. DOI: `10.1145/3293882.3330577`. [Online]. Available: `https://doi.org/10.1145/3293882.3330577`.

[84] S. Kabinna, C.-P. Bezemer, W. Shang, and A. E. Hassan, "Logging library migrations: A case study for the apache software foundation projects," in *2016 IEEE/ACM 13th Working Conference on Mining Software Repositories (MSR)*, May 2016, pp. 154–164.

[85] SLF4J Migrator. (2021). "SLF4j migrator," [Online]. Available: `http://www.slf4j.org/migrator.html` (visited on 10/16/2021).

[86] A. Brito, L. Xavier, A. Hora, and M. T. Valente, "Apidiff: Detecting api breaking changes," in *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, 2018, pp. 507–511. DOI: `10.1109/SANER.2018.8330249`.

[87] A. Aleksyuk and V. Itsykson, "Automated semantics-driven source code migration: A pilot prototype," Accessed: 11-16-2021, Jun. 26, 2017, [Online]. Available: `https://persons.iis.nsk.su/files/persons/pages/istykson.pdf` (visited on 11/16/2021).