RAID: TOOL SUPPORT FOR REFACTORING-AWARE CODE REVIEWS

RODRIGO FERREIRA DE BRITO

RAID: TOOL SUPPORT FOR REFACTORING-AWARE CODE REVIEWS

Dissertação apresentada ao Programa de Pós-Graduação em Ciência da Computação do Instituto de Ciências Exatas da Universidade Federal de Minas Gerais como requisito parcial para a obtenção do grau de Mestre em Ciência da Computação.

Orientador: Marco Túlio de Oliveira Valente

Belo Horizonte

Agosto de 2021

RODRIGO FERREIRA DE BRITO

RAID: TOOL SUPPORT FOR REFACTORING-AWARE CODE REVIEWS

Dissertation presented to the Graduate Program in Computer Science of the Federal University of Minas Gerais in partial fulfillment of the requirements for the degree of Master in Computer Science.

Advisor: Marco Túlio de Oliveira Valente

Belo Horizonte

August 2021

© 2021, Rodrigo Ferreira de Brito.
Todos os direitos reservados.

Brito, Rodrigo Ferreira de

B862r RAID: tool support for refactoring-aware code reviews [manuscrito]. / Rodrigo Ferreira de Brito. — Belo Horizonte, 2021.
xiv, 58 f. : il. ; 29cm

Orientador: Marco Túlio de Oliveira Valente.

Dissertação (mestrado) - Universidade Federal de Minas Gerais, Instituto de Ciências Exatas, Departamento de Ciência da Computação.
Referências: f. 53-58

1. Computação – Teses. 2. Engenharia de Software - Teses. 3. Software - Refatoração – Teses. 4.– Linguagem de programação (Computadores) – Teses. I. Valente, Marco Túlio de Oliveira. II. Universidade Federal de Minas Gerais, Instituto de Ciências Exatas, Departamento de Ciência da Computação. III. Título.

CDU 519.6*32(043)

Ficha catalográfica elaborada pela bibliotecária Irénquer Vismeg Lucas Cruz - CRB 6ª Região nº 819.

UNIVERSIDADE FEDERAL DE MINAS GERAIS
INSTITUTO DE CIÊNCIAS EXATAS
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

FOLHA DE APROVAÇÃO

RAID: Tool Support for Refactoring-Aware Code Reviews

RODRIGO FERREIRA DE BRITO

Dissertação defendida e aprovada pela banca examinadora constituída pelos Senhores:

[illegible]

Prof. Marco Túlio de Oliveira Valente - Orientador
Departamento de Ciência da Computação - UFMG

[illegible]

Profa. Ingrid Oliveira de Nunes
Departamento de Informática Aplicada - UFRGS

[illegible]

Prof. André Cavalcante Hora
Departamento de Ciência da Computação - UFMG

Belo Horizonte, 20 de Agosto de 2021.

Acknowledgments

Agradeço aos meus familiares, amigos e professores! A presença de vocês foi indispensável para a realização desse sonho. Agradeço em especial:

A Deus, por permitir a conclusão desta etapa.

Aos meus familiares, pelo incentivo durante este curso de mestrado. Especialmente, aos meus pais Bernadete e José, e à minha irmã Aline, por todo o carinho, apoio e conselhos recebidos.

Aos meus amigos, pela amizade, suporte e por todos os momentos incríveis que passamos juntos. Em especial aos meus amigos do Aserg, pela parceria, amizade e aprendizado.

Aos meus companheiros de trabalho da Studio Sol, pelo companheirismo e contribuição no meu crescimento profissional e pessoal.

Ao meu orientador, Prof. Marco Tulio Valente, pela paciência, confiança e todos os ensinamentos passados durante esse período.

Aos meus professores, pelo incentivo e conhecimento compartilhado durante o mestrado, graduação e ensino básico.

Aos membros da banca, Prof. Ingrid Nunes e Prof. André Hora, pela disponibilidade em participar deste trabalho.

Ao DCC/UFMG, FAPEMIG e CNPq, pelo suporte financeiro e acadêmico.

“Great things are done by a series of small things brought together.”
(Vincent Van Gogh)

Resumo

Revisão do código é uma importante prática no desenvolvimento de software moderno. Além de detectar falhas, revisão de código contribui para a qualidade do código e transferência de conhecimento. No entanto, revisão do código leva tempo e exige uma análise detalhada e demorada de diffs textuais. Particularmente, detectar refatorações durante as revisões não é uma tarefa trivial, uma vez que as refatorações não são representadas em diffs. Nesta dissertação, nós inicialmente estendemos RefDiff – uma ferramenta de detecção de refatoração multilinguagem - para identificar refatorações na linguagem de programação Go. Nossa extensão—chamada RefDiff4Go—detecta 13 tipos de refatoração e obteve uma precisão de 92% e um recall de 80% em uma avaliação com seis projetos de código aberto populares. Em seguida, como nossa principal contribuição, nós apresentamos RAID: uma ferramenta de diff inteligente que identifica atividades de refatoração e instrumenta os diffs textuais—particularmente, os diffs fornecidos pelo GitHub—com informações de atividades de refatoração. Nosso objetivo é aliviar o esforço cognitivo associado a revisões de código, detectando automaticamente as operações de refatoração incluídas nas solicitações de *pull requests*. Além de propor uma arquitetura para o RAID, implementamos um plug-in para o navegador Chrome que suporta nossa solução. Também avaliamos RAID em um experimento de campo por três meses, quando oito desenvolvedores profissionais usaram nossa ferramenta em quatro projetos Go. Concluímos que RAID reduz o esforço cognitivo necessário para detectar e revisar atividades de refatoração em diffs textuais. Particularmente, RAID também reduz o número de linhas necessárias para revisar tais operações. Por exemplo, o número médio de linhas a serem revisadas diminuiu de 14,5 para 2 linhas no caso de refatorações envolvendo movimentação e de 113 para 55 linhas no caso de extrações.

Palavras-chave: Refatoração, Refactoring-Aware, Revisão de Código, Diffs Textuais.

Abstract

Code review is a key practice in modern software development. Besides detecting bugs, code review contributes to code quality and knowledge transfer. However, code review takes time and demands detailed and time-consuming analysis of textual diffs. Particularly, detecting refactorings during reviews is not a trivial task, since refactorings are not represented in diffs. In this dissertation, we initially extended RefDiff—a multi-language refactoring detection tool—to identify refactorings in the Go programming language. Our extension—called RefDiff4Go—detects 13 refactoring types and achieved 92% of precision and 80% of recall in an evaluation with six well-known open source projects. Then, as our key contribution, we proposed RAID: a refactoring-aware and intelligent diff tool for instrumenting textual diffs—particularly, the ones provided by GitHub—with information about refactorings. Our goal is to alleviate the cognitive effort associated with code reviews, by automatically highlighting refactoring operations included in pull requests. Besides proposing an architecture for RAID, we implemented a Chrome browser plug-in that supports our solution. We also evaluated RAID in a field experiment for three months, when eight professional developers used our tool in four Go projects. We concluded that RAID can reduce the cognitive effort required for detecting and reviewing refactorings in textual diff. Particularly, RAID reduces the number of lines required for reviewing such operations. For example, the median number of lines to be reviewed decreased from 14.5 to 2 lines in the case of move refactorings and from 113 to 55 lines in the case of extractions.

Palavras-chave: Refactoring, Refactoring-Aware Code Review, Code Review, Textual Diffs.

List of Figures

List of Tables

Contents

Chapter 1

Introduction

1.1 Motivation

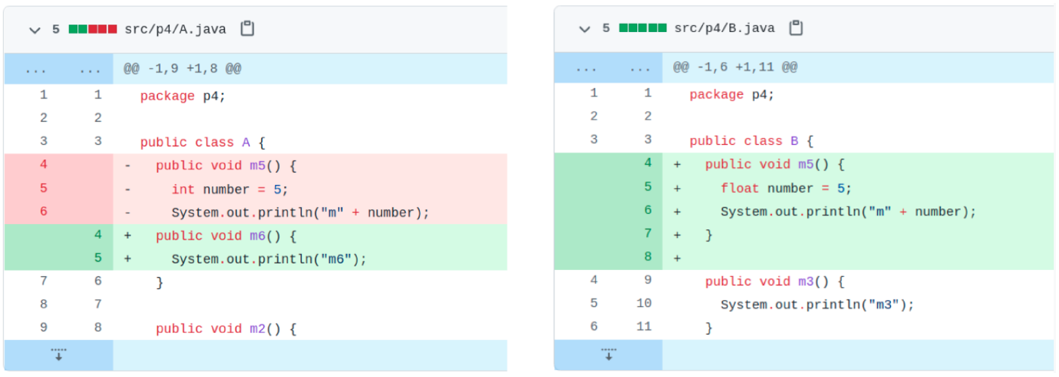
Refactoring is a key software development practice that seeks to improve the internal structure of the code without changing its external behavior [Fowler, 2018]. Recent studies show that refactoring is often used in high-quality software systems and plays an important role in supporting maintenance and code evolution activities [Murphy-Hill et al., 2009; Murphy-Hill et al., 2012].

In this way, the identification of refactorings has a key relevance for researchers intended to conduct empirical studies, providing important information about source code transformations. For example, refactoring data is crucial to understand key aspects of software evolution. Therefore, there is a large body of studies on the motivations to perform refactoring operations [Silva et al., 2016; Mazinanian et al., 2017; Tsantalis et al., 2013; Pantiuchina et al., 2020], on the challenges of refactoring [Kim et al., 2012, 2014], impact on software quality and evolution [Bibiano et al., 2019; Chaparro et al., 2014; Lacerda et al., 2020], and security [Abid et al., 2020; Mumtaz et al., 2018; amd Kensuke Tokoda, 2008].

Refactoring activities also impact software development practices, such as code review. In a study presented by Ge et al. [2017]. with 35 developers, the authors show that 95% of participants consider that refactorings can cause delays in review of non-refactoring changes. In addition, 91% of respondents mention that automatic identification of refactorings could help on code review activities.

However, existing diff tools do not automatically detect refactorings in their results. In Figure 1.1, we have a refactoring that moves a function `m5()` from file `A.java` to file `B.java`. This move is represented in current state-of-the-practice diff tools as a set of lines deleted (−) from `A.java` and by a set of lines added (+) to `B.java`.

Therefore, reviewers need to infer that these added/deleted lines indeed represent a move function. This is a not trivial inference when the change is complex, distributed over multiple files and when the refactoring is combined with other changes, such as bug fixes and the implementation of new features, as usual in practice [Silva et al., 2016]. After inferring the refactoring, reviewers should also compare the code after and before the change, to review possible changes performed in the moved function. The reason is that refactorings are usually followed by minor edits in the code, particularly when we compare the changes at the level of commits [Silva et al., 2016]. In the presented example, the developer moved the function and also created a new method `m6` and changed a variable type from `int` to `float`.



```
5 src/p4/A.java
@@ -1,9 +1,8 @@
  package p4;

  public class A {
-   public void m5() {
-     int number = 5;
-     System.out.println("m" + number);
+   public void m6() {
+     System.out.println("m6");
    }

    public void m2() {
```

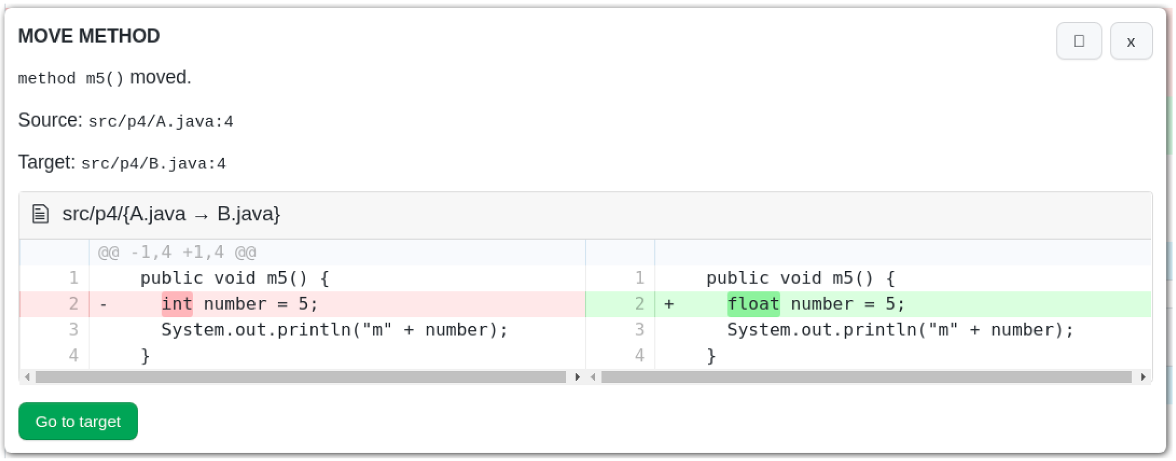
```
5 src/p4/B.java
@@ -1,6 +1,11 @@
  package p4;

  public class B {
+   public void m5() {
+     float number = 5;
+     System.out.println("m" + number);
+   }
+
    public void m3() {
      System.out.println("m3");
    }
```

Figure 1.1. Example of Move Function in a textual based diff

To tackle this challenge, there are in the literature some tools that use refactoring activities to support developers in development and maintenance activities [Ge and Murphy-Hill, 2011; Shen et al., 2019; Terra et al., 2018; Alizadeh et al., 2019; Lahiri et al., 2012]. In particular, a class of tools exposes refactoring activities to assist developers in the code review process [Alves et al., 2014]. Figure 1.2 presents the previous example but considering the refactoring performed, in which the diff was reduced from six modified lines to only two lines.

Furthermore, existing tools do not support multiple programming languages and do not work directly with continuous integration systems of popular platforms for code review, such as GitHub. Thus, the objective of this dissertation is to reduce the cognitive effort during code review by the design, implementation, and evaluation of refactoring aware diff tools. The proposed approach is based on RefDiff [Silva and Valente, 2017; Silva et al., 2021], a tool that detects refactoring activities in the commit history of a software repository. RefDiff currently supports Java, JavaScript, and C programming languages. Thus, in this dissertation we also propose the extension of



method `m5()` moved.

Source: `src/p4/A.java:4`

Target: `src/p4/B.java:4`

src/p4/{A.java → B.java}

```
@@ -1,4 +1,4 @@
    public void m5() {                          public void m5() {
-       int number = 5;                     +       float number = 5;
        System.out.println("m" + number);           System.out.println("m" + number);
    }                                           }
```

Figure 1.2. Example of Move Function in a refactoring-aware based diff

RefDiff to the Go programming language, expanding the range of languages supported by the tool.

1.2 Proposed Work

First, in this dissertation, we implemented a RefDiff extension for Go, named RefDiff4Go. We also evaluated this tool with six well-known Go projects. RefDiff4Go is a preliminary step to make our second study possible, when we conducted a field experiment for evaluating our refactoring-aware code review solution.

Next, we designed and implemented **RAID: Refactoring-aware and Intelligent Diff tool**. Basically, RAID is an open source tool that adds refactorings annotations in the GitHub diff in order to support developers in code review activities. In a nutshell, RAID seamlessly provides a diff result similar to the one we showed in Figure 1.2. As mentioned, we also evaluated the tool, aiming to better understand the benefits and challenges of adopting refactoring-aware tools in code reviews. Particularly, we evaluated the cognitive effort reduction achieved with RAID

To this purpose, we conducted a field experiment in a medium-size technology company when two development teams used RAID in their daily activities during three months. We conducted this study with eight participants and four Go projects.

In summary, we performed the following working units:

- We designed and implemented an open source tool that detects refactoring activities through a continuous integration pipeline and annotates Pull Request diffs with this information.

- We conducted a field experiment to assess our refactoring-aware code review solution and to collect metrics to analyze the cognitive effort reduction achieved when using this solution.
- We conducted a post-experiment survey to understand the participants' perception about the use of refactoring-aware tools during code review.

1.3 Contributions

The main contributions of this master's dissertation are:

- RefDiff4Go, an extension of RefDiff, which identify 13 types of refactoring activities for Go with 92% of precision and 80% of recall.
- An oracle with over 68K refactoring operations performed in six well-known Go projects.
- RAID, an open-source tool for instrumenting textual diffs—particularly, the ones provided by GitHub—with information about refactorings. In a field experiment, we show that RAID can reduce the cognitive effort required for reviewing refactorings when using textual diffs. For example, in the case of move refactorings, the number of lines decreases from 14.5 to 2 lines (median values); and from 113 to 55 lines in the case of extractions.

1.4 Publications

The dissertation results are published in the following publications and therefore contains parts of them:

- **Rodrigo Brito**, Marco Tulio Valente. RAID: Tool Support for Refactoring-Aware Code Reviews. In 29th International Conference on Program Comprehension (ICPC), pages 1–11, 2021.
- **Rodrigo Brito**, Marco Tulio Valente. RefDiff4Go: Detecting Refactorings in Go. In 14th Brazilian Symposium on Software Components, Architectures, and Reuse (SBCARS), pages 1–10, 2020. **2nd Best Paper Award**.

The following publications were produced in preliminary activities conducted as part of this master's work.

- Laerte Xavier, Fabio Ferreira, **Rodrigo Brito**, Marco Tulio Valente. Beyond the Code: Mining Self-Admitted Technical Debt in Issue Tracker Systems. In 17th International Conference on Mining Software Repositories (MSR), pages 1–10, 2020.

- Hudson Borges, **Rodrigo Brito**, Marco Tulio Valente. Beyond Textual Issues: Understanding the Usage and Impact of GitHub Reactions. In 33rd Brazilian Symposium on Software Engineering (SBES), pages 1–10, 2019.

- **Rodrigo Brito**, Aline Brito, Gleison Brito, Marco Tulio Valente. GoCity: Code City for Go. In 26th International Conference on Software Analysis, Evolution and Reengineering (SANER), Tool Track, pages 649–653, 2019.

1.5 Outline of the Dissertation

The remaining of this master's thesis is divided in the following chapters:

- **Chapter 2** discuss related work and introduces the main concepts involving RefDiff, Go, GitHub Actions. Essentially, these are technologies and tool that will be used in the next chapters.

- **Chapter 3** describes RefDiff4Go, an extension of RefDiff to identify refactoring activities in the Go programming language. In this chapter, we present the architecture of RefDiff4Go and an evaluation with six projects.

- **Chapter 4** presents RAID, a refactoring-aware and intelligent diff tool, a pipeline tool that assists developers in code review activities. We also discuss the lessons learned in a field experiment with eight professional developers who used RAID during three months.

- **Chapter 5** concludes this master dissertation, presents the final remarks, limitations, and future work.

Chapter 2

Background and Related Work

In this chapter, we briefly present the technical background needed to understand our work. First, in Section 2.1, we introduce RefDiff, a refactoring detection tool proposed by Silva et al. [2021]. RefDiff4Go and RAID presented in Chapters 3 and 4, respectively, rely on this tool. Next, in Section 2.2, we present Go, a popular programming language, which is used in both evaluations conducted in this master dissertation (RefDiff4Go's evaluation and RAID's evaluation). In Section 2.3, we describe GitHub Actions, a pipeline automation tool proposed to execute RAID on GitHub repositories. Finally, in Section 2.4 we present related work.

2.1 RefDiff

Recently, Silva et al. [2021] proposed a technique and tool for detecting refactorings called RefDiff. The proposed tool consists of processing two revisions of a system through static analysis. The extraction process is divided into two main stages: Code Analysis and Relationship Analysis.

In the first step, RefDiff receives two revisions of the source code as input and builds a model called Code Structure Tree (CST), which is a tree structure where each node represents a code element, similar to an Abstract Syntax Tree (AST) but simpler. For example, CSTs provide information about the main elements of the code, such as functions, classes, and interfaces. This step is only applied to modified files.

Typically, a CST is created from an AST. The selection of AST elements depends on the analyzed programming language and a set of rules defined by a RefDiff plugin. The plugin identifies and converts each node and establishes relationships between them. For example, from Java ASTs, RefDiff extracts Classes, Interfaces, Enums, and

Methods nodes. In the case of C, the plugin selects only Files and Functions. Each node type contains a unique identifier, and may receive the following custom attributes:

- **Name:** Identifier of the element in the code, usually its name. In the case of functions, the name is followed by its parameters. For example, the `print` function in Figure 2.1 has the name `print(string)`.
- **Namespace:** An optional field, which in conjunction with the name field, identifies the global location of the node within the project. The namespace is associated only with high-level nodes like classes in Java and files in C. In the example shown in Figure 2.1, the namespace of the file is `main`, i.e., its package name.
- **Type:** A string that represents the type of the node in the analyzed language. For example, in Go the valid types are *Struct*, *Function*, *Interface*, *Type*, and *File*. In Figure 2.1, `print` has type *Function*.
- **Parameters List:** An optional field that identifies the names of the parameters in a function. In the example in Figure 2.1, the print function parameters is a list with a single element named value.
- **Tokenized source code:** This data is the source code of the node in the form of a list of tokens. For example, the list of tokens of the print function is: `func`, `println`, `(`, `value`, `string`, `)`, `{`, `println`, `(`, `value`, `)`, and `}`.

```
package main

func print(value string) {
        println(value)
}
```

Figure 2.1. Example of Go file

The CST structure is independent of the language grammar, which allows its integration with different programming languages. RefDiff extension is supported by a plugin system, which are key components responsible for CST nodes identification, nodes relationship, and communication with RefDiff core. Currently, RefDiff has three programming languages: Java, C, and JavaScript.

In the second processing stage, RefDiff compares the CST nodes between the two analyzed revisions. The ultimate goal of this stage is to discover relations between nodes that denote moves and modifications in the tree of nodes.

The matching algorithm consists of two steps: search and classification. The first stage executes a recursive search in the node tree, analyzing elements, such as signature, hierarchy, and similarity of the list of tokens. Subsequently, RefDiff follows a series of rules to classify the type of the relationship between each pair of nodes. For example, if two nodes have the same type, name, and parent they are connected by a *Same* relationship. Similarly, RefDiff applies a set of rules to identify each of the 13 supported refactorings. For a detailed specification of each of such rules, we refer the reader to RefDiff's original paper [Silva et al., 2021].

Often, developers do not apply refactorings in isolation, i.e., commits can also fix bugs or include new features. Therefore, RefDiff also uses a similarity metric to identify possible variations in function bodies. This metric is computed using a Term Frequency-Inverse Document Frequency Algorithm (TF-IDF) algorithm, which weighs less frequent tokens with a higher score and reduces the relevance of common terms, such as brackets and semicolons [Salton and McGill, 1986].

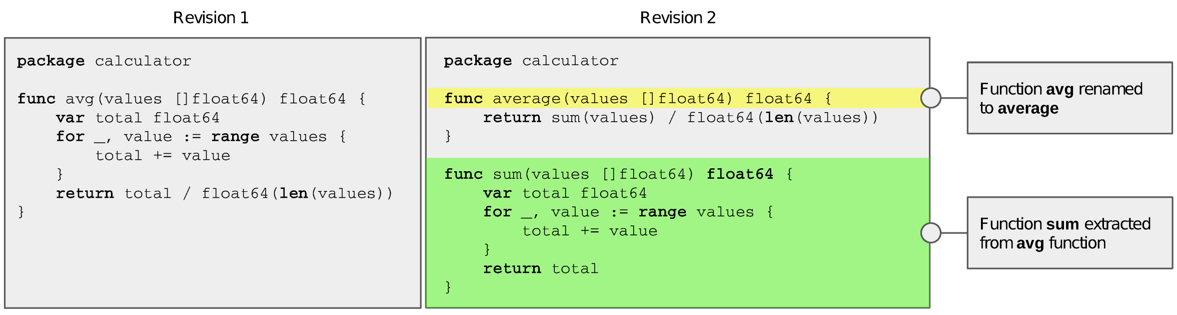
At the end of the process, RefDiff returns a list of relationships in the form of a tuple (n_1, n_2, Rel), where n_1 is a CST node in the initial revision, n_2 is a node in the compared revision, and Rel is the relationship identified between them. Table 2.1 lists the types of relationships, i.e., refactorings detected by RefDiff by default.

Table 2.1. Refactorings supported by RefDiff

Refactoring	Description
Same	Detection of identical elements in code
Convert Type	Transformation of definitions types
Change Signature	Changes function signature
Pull Up Method	Move a function to a superclass
Push Down Method	Move a function to subclasses
Rename	Renaming of code components
Move	Moving component location
Move and Rename	Move operation with renaming
Extract Function	Extraction of code to a new function
Extract Supertype	Extraction to a new shared superclass
Inline Function	Replace of a function call with its content

Figure 2.2 shows an example of relationships identified between two versions of a Go file. In Revision 1, we have one function named `avg`, which receives a list of values and return the average value. Next, we have the Revision 2 with two refactorings. The

developer renames the function `avg` to `average` and extracts a new function from the original code with the summary logic.


Revision 1
Revision 2
Function **avg** renamed to **average**
Function **sum** extracted from **avg** function

```go
package calculator

func avg(values []float64) float64 {
    var total float64
    for _, value := range values {
        total += value
    }
    return total / float64(len(values))
}
```

```go
package calculator

func average(values []float64) float64 {
    return sum(values) / float64(len(values))
}

func sum(values []float64) float64 {
    var total float64
    for _, value := range values {
        total += value
    }
    return total
}
```

Figure 2.2. Example of rename and extract function

In this example, as follows: RefDiff returns a list with two tuples representing the relationship between revisions, $(avg, average, Rename)$ which identifies the rename refactoring, and $(avg, sum, Extract)$ which represents the extraction of `sum` function.

2.2 Go

Go is a statically typed and compiled programming language created by Google in 2007 to supply demands of large-scale and complex systems [Pike, 2021]. Currently, there are approximately 20K Go projects hosted on GitHub. Among these projects, there are popular and large software systems like Kubernetes (a container management system), Docker (a software containerization platform) and Terraform (a tool that manages and versions server configurations).

The syntax of Go is similar to C and C++, since the language is statically typed and has only 25 keywords. Similar to C, Go uses structs to define entities and functions. Besides that, the language supports an abstraction mechanism using interfaces. However, Go does not have support for inheritance relations [Donovan and Kernighan, 2015].

2.3 GitHub Actions

Nowadays, teams that employ agile methodologies use automated pipelines to apply modern software development practices, such as continuous integration (CI) and continuous delivery (CD). Popular platforms like GitHub and Gitlab provides these functionalities, in which developers can include automation tasks in their projects. In the case of GitHub, this tool receives the name of GitHub Actions.

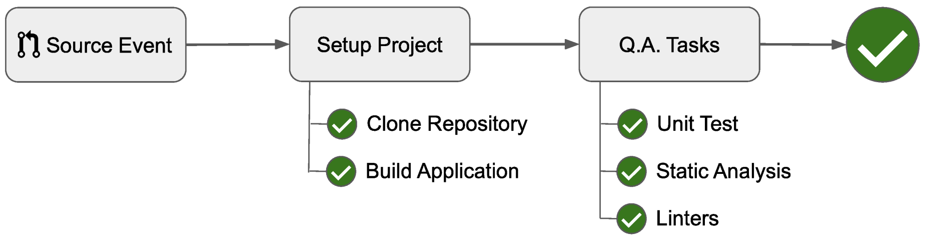


Figure 2.3. Flow of a continuous integration pipeline

Figure 2.3 presents an example of pipeline execution. The task flow is started from an event in the software repository, such as the opening of a Pull Request. In this way, GitHub Action starts a sequence of one or more groups of tasks. These groups are customizable and may involve different activities, such as application build and unit test execution. If all steps are successfully executed, the pipeline issues a signal, allowing the code to be integrated.

In summary, GitHub Actions allows developers to run scripts in different programming languages with Docker containers. GitHub execute these tasks internally on its servers, facilitating the integration process with software repositories already hosted on the platform. In addition, GitHub Actions provides several pre-configured plugins, but the developer can also create customized steps and include his own tools in the pipeline.

2.4 Related Work

Refactoring is a key practice to evolve and preserve software quality. Due to the importance of such activities, there is a large body of studies on the motivations to perform refactoring operations [Silva et al., 2016; Mazinanian et al., 2017; Tsantalis et al., 2013; Pantiuchina et al., 2020], on the challenges of refactoring [Kim et al., 2012, 2014], impact on software quality and evolution [Bibiano et al., 2019; Chaparro et al., 2014; Lacerda et al., 2020], security [Abid et al., 2020; Mumtaz et al., 2018; amd Kensuke Tokoda, 2008] and, providing tools to support developers in evolution and maintenance activities [Ge and Murphy-Hill, 2011; Shen et al., 2019; Alves et al., 2014; Terra et al., 2018; Alizadeh et al., 2019; Lahiri et al., 2012; Brito et al., 2019].

2.4.1 Detection of Refactoring Activities

There are also several studies in the literature based on the identification of refactoring activities. It is possible to find different approaches, such as plugins integrated to

IDEs during software development [Negara et al., 2013; Prete et al., 2010; Kim et al., 2010], metadata analysis of revision control systems [Ratzinger et al., 2008; Krasniqi and Cleland-Huang, 2020], and static analysis [Silva and Valente, 2017; Silva et al., 2021; Tsantalis et al., 2013].

The most common approach for detecting refactorings is static analysis. Currently, we find in the literature several studies that apply this technique. Particularly, there are tools focused on detecting refactorings in a single programming language [Dig et al., 2006; Silva and Valente, 2017; Tsantalis et al., 2018, 2020] and approaches for identifying refactorings in multiple languages [Silva et al., 2021].

Dig et al. [2006] proposed Refactoring Crawler, a static analysis tool that detects seven types of refactoring: *Change Method Signature*, *Rename Package / Class / Method*, *Pull Up Method*, *Push Down Method*, and *Move Method*. The tool is based on a lightweight static analysis using the Shingle encoding technique combined with a semantic analysis to refine the results. The approach proposed by the authors presented good results in a study with three relevant projects: EclipseUI, Struts, and JHotDraw, reporting accuracy of 85%.

Ref-Finder, proposed by Kim et al. [2010], is a tool that covers a wide range of refactorings. Through an Eclipse plugin, the tool identifies atomic and composite refactorings using a template-based refactoring reconstruction approach, covering 63 types of refactorings from Fowler's catalog [Fowler, 2018]. The authors evaluated the tool with examples from Fowler's book and also with real open-source projects, reporting a precision of 0.79 and recall of 0.95.

Tsantalis et al. [2013] proposed Refactoring Miner, a tool capable of identifying 14 types of refactoring in Java projects: *Move Class / Method / Field*, *Extract Method*, *Inline Method*, *Rename Package / Class / Method*, *Pull Up Method / Field*, *Push Down Method / Field*, and *Extract Superclass / Interface*. The approach proposed by the authors consists of a lightweight version of UMLDiff [Xing and Stroulia, 2005] for analyzing object-oriented models. The tool was applied in an empirical study in three well-known projects: JUnit, HTTPCore, and HTTPClient. The results include an accuracy of 96.4% for *Extract Method*, 97.6% for *Rename Class*, and 100% accuracy for the other types of refactorings.

In 2018, Tsantalis et al. [2018] proposed a new tool called RMiner, an evolution of the approach previously presented. RMiner is based on an AST matching algorithm and supports 15 different types of refactorings. In a new study, the authors evaluated the tool with 3,188 real refactorings from an oracle. In this large oracle, RMiner showed a precision of 98% and a recall of 87%. Currently, RAID does not use this tool. However, in the future we plan to extend RAID to also work with RefactoringMiner.

In a recent study, published in 2020, Tsantalis et al. [2020] presented Refactoring Miner 2.0, an evolution of the tool previously presented with improvements in the matching algorithm. The authors extended the tool to support 40 different types of refactoring. In a large-scale study, involving 7,226 real instances of refactoring, Refactoring Miner showed an average precision of 99.6% and recall of 94%, surpassing the result of six other compared tools, both in accuracy and execution time.

Silva and Valente [2017] proposed a new approach for detecting refactorings activities called RefDiff 1.0. The tool uses a combination of heuristics through static analysis and TF-IDF as a measure of similarity. RefDiff is able to identify 13 types of refactorings. Besides, the tool was evaluated through an empirical study and compared with three known tools: Refactoring Miner, Refactoring Crawler and Ref-Finder. The tool proposed by the authors outperformed the others, recording 100% of precision and an 88% of recall.

Recently, Silva et al. [2021] proposed a new version of RefDiff, now named RefDiff 2.0, and with multi-language support. The new approach is able to identify 11 different types of refactoring in three programming languages: Java, JavaScript, and C. The proposed algorithm is based in a Code Structure Tree (CST) that abstracts the structural representation of the code and, therefore, is independent of the syntax of programming languages. RefDiff 2.0 also allows extension via plugins to support new languages. In the study, the authors reported a 96% accuracy and 80% recall. RAID—as proposed and evaluated in this dissertation—relies on this tool.

2.4.2 Code Review

Code review review is a key practice in software engineering that seeks to improve software quality [Bacchelli and Bird, 2013; Sadowski et al., 2018]. Since its origins in the 1970s, when a formal process with strict guidelines was proposed by Fagan [1976], it is widely discussed in the literature. We found studies about the impact on software quality [McIntosh et al., 2016], security [McGraw, 2008; Edmundson et al., 2013], and knowledge transfer [Caulo et al., 2020].

Sadowski et al. [2018] presented an exploratory investigation of code review practices at Google. The authors performed 12 semi-structured interviews with Google developers, an internal survey with 44 respondents, and analyzed 9 million reviews over two years. The results showed that Google performs a lightweight and quick code review with interactions over small changes. The developers spend 2.6 hours per week reviewing changes, in median values. The focus of code review is centered in code readability and maintainability instead of being centered only in problem solving. The authors also reports a important educational aspect over this practice.

Other studies on modern code review are discussed by Davila and Nunes [2021]. The authors performed a systematic review of 139 papers, where foundational studies, proposals, and evaluations from four digital libraries were analyzed.

2.4.3 Refactoring-Aware Code Review

Refactoring activities are also used in the context of code review. In the literature, we found several studies that explored the use of this information to understand the benefits of this technique and on providing tools to support developers during the development process [Ge et al., 2017].

The benefits of refactoring-aware code reviews have also been explored in the literature. For example, Ge et al. [2017] presented a formative study with 35 developers to investigate the motivation and challenges of reviewing refactorings during code review. They report that 94% (33) of the study participants consider that refactorings can slow down the review of non-refactoring changes. Furthermore, the automatic identification of refactorings may assist code reviewers for 91% of the participants (32). The authors also presented a refactoring-aware tool called ReviewFactor, which provides a separation of refactorings and non-refactorings changes for the purpose of code review. This tool separates code reviews in two steps: first, non-refactored code should be reviewed using a specific interface that is part of an IDE; after that, reviewers should focus on the refactored code. With RAID we decided to follow a different approach by seamlessly integrating our tool to a state-of-the-practice code review workflow. For this purpose, RAID provides the “R” buttons and also the floating windows with detailed data about refactorings. It is also relevant to mention that refactorings are usually followed by minor edits in the changed code [Silva et al., 2016, 2021; Tsantalis et al., 2020], as illustrated in Figures 4.3 and 4.4. However, it is not clear how ReviewFactor handles such interleaved operations, which might for example require a duplicated effort by code reviewers.

Textual diffs also present issues when merging changes in version control systems. Shen et al. [2019] proposed a graph-based refactoring-aware algorithm to detect and resolve refactoring-related conflicts. The algorithm was evaluated with 1,070 merge scenarios from popular open source Java projects and archived a 58.90% reduction of merge conflicts comparing with GitMerge and 11.84% over jFSTMerge. Other studies on refactoring-aware code reviews are summarized in a survey by Coelho et al. [2019].

2.5 Final Remarks

In this chapter, we briefly presented the technical background related to this dissertation. Specifically, we detailed the concepts of RefDiff, presented the motivation in use and the main features of Go programming language, and described the main concepts of pipeline automation with GitHub Actions. Finally, we presented related work.

Chapter 3

RefDiff4Go: Detecting Refactorings in Go

In this chapter, we describe RefDiff4Go, an extension of RefDiff to identify refactoring activities in the Go programming language. We briefly present our motivations in Section 3.1. In Section 3.2, we present the architecture of RefDiff4Go. Finally, in Section 3.3 we describe an evaluation with six projects and the results are reported in Section 3.4.

3.1 Introduction

Recently, several studies have proposed automated approaches to detect refactorings activities [Silva and Valente, 2017; Silva et al., 2021; Tsantalis et al., 2018, 2020]. Tsantalis et al. [2020] presented Refactoring Miner 2.0, a tool that identifies 40 types of refactorings in Java systems with 99.6% of precision and 94% of recall. Other approaches, such as RefDiff, also present accurate detection measures. The tool proposed by Silva et al. [2021] follows an extensible architecture, capable of detecting refactorings for Java, JavaScript, and C languages with 96.4% of precision and 80.4% of recall.

Currently, several studies focus on maintenance activities and the evolution of Java systems [Brito et al., 2020; Silva et al., 2016; Terra et al., 2018]. Therefore, tools such as RefDiff allow the expansion of such studies to other popular languages such as JavaScript and C. Also, RefDiff supports the addition of new programming languages, through a plugin system, permitting expansion to other relevant programming languages. Thus, there is a lack of tools that support developers and researchers to study refactorings in emergent programming languages. To address these challenges, we present RefDiff4Go, a RefDiff extension that detects 13 types of refactoring activities in Go projects. To our knowledge, we are the first to implement refactoring detector for Go. RefDiff4Go is a preliminary step to make the second study possible, when we conducted a field experiment with four software systems implemented in Go, presented in Chapter 4. Particularly, our contributions are:

- An open-source RefDiff plugin for Go.
- A precision and recall evaluation with six well-known open-source projects.
- An oracle with over 68K performed refactorings.

The remainder of this chapter is organized as follows: In Section 3.2, we present the proposed tool in detail. Then, in Section 3.3, we describe the design of the evaluation of RefDiff4Go. Our results are reported in Section 3.4. In Section 3.5, we discussed threats to validity. Finally, we conclude the chapter in Section 3.6.

3.2 RefDiff4Go

Figure 3.1 shows the architecture of our implementation of RefDiff for Go. The system is implemented as a plugin with two key components: a Parser for analyzing Go's source code and an JSON Adapter that integrates the previous module with RefDiff's core, i.e., it receives the Go output and converts it to a valid Java format.

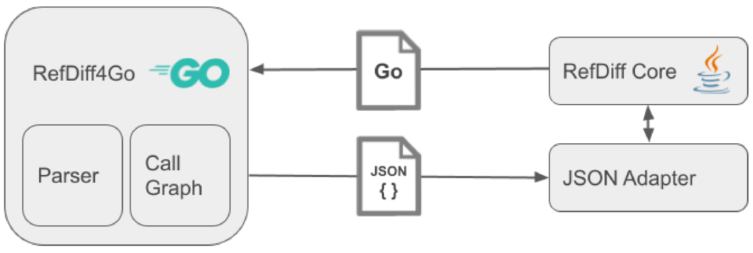
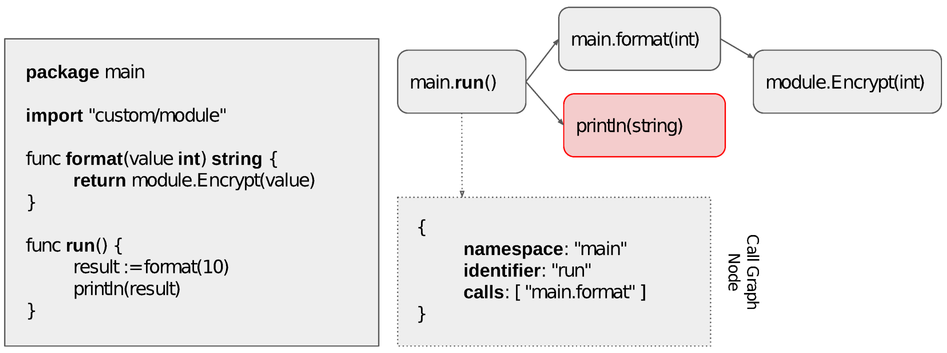


Figure 3.1. Refdiff4Go architecture

The Parser module is implemented in Go and relies on the language's API to detect nodes and to extract information about code elements. More specifically, the parser is a Command-Line Interface (CLI), where the input consists of a valid Go file and the output is a list of CST nodes. Since the Parser Module is written in Go and RefDiff Core is written in Java, we used a common data format for exchanging messages. Specifically, RefDiff4Go exports each CST node information in JSON format, which is a compatible format between both programming languages.

The extraction step uses an AST to represent the analyzed code. Through this structure, the Parser Module detects valid elements and extracts relevant information, such as name, position and namespace. Besides, this module is also responsible for extracting all the tokens from the source code, storing their position, and dealing with any text encoding problems (e.g., non UTF-8 characters are valid tokens to define variables and may affect the extraction). The Parser module is also responsible for building the hierarchical tree of the analyzed code. When traversing the nodes in the AST, the module saves the reference of the last valid CST ancestor that is identified as the parent of the node.

Finally, the Call Graph module uses this information to create a graph of functions calls. Figure 3.2 shows an example of a Call Graph. The source code contains two function declarations: *format* and *run*. The function format contains a single call to an external module. In this way, it includes a single edge to *Encrypt* node. The function *run*, on the other hand, invokes two other functions, however, only *format* is included in the final graph, because *println* is a built-in method and therefore does not represent a possible CST node relationship. Figure 3.2 also shows the call graph generated for function *main.run*, in the JSON format.



```
package main

import "custom/module"

func format(value int) string {
        return module.Encrypt(value)
}

func run() {
        result :=format(10)
        println(result)
}
```

```
{
        namespace: "main"
        identifier: "run"
        calls: [ "main.format" ]
}
```

Figure 3.2. Example of call graph representation

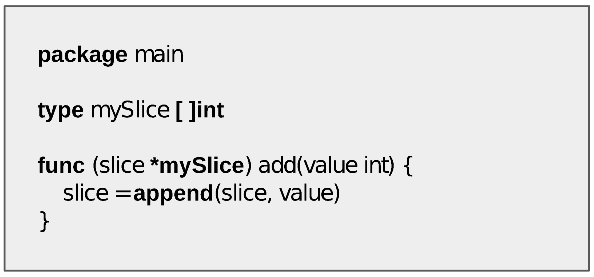
RefDiff4Go identifies five main types of nodes: *Structs*, *Functions*, *Interfaces*, *Type Definitions* (*TypeDef*), and five different types of refactorings. Table 3.1 shows the relationship between the types of nodes identified and the possible refactorings.

Function, *Interface*, and *Struct* types are the basic syntactic structures in Go. *TypeDef* represents a custom type definition, which may have methods as in a *Struct*. Figure 3.3 shows an example of a *TypeDef* declaration, in which the user declared an array of integers under the name `mySlice`. Additionally, this new type also provides a function, which appends a new value in the array.

Similar to what happens in procedural languages (such as C), a file in Go represents a valid CST node type. Therefore, independent functions can be declared in

Table 3.1. Refactorings supported by RefDiff4Go

	Move	Rename	Extract	Inline	Change Signature
Function	✓	✓	✓	✓	✓
Struct	✓	✓			
TypeDef	✓	✓			
Interface	✓	✓			
File	✓	✓			



```
package main

type mySlice [ ]int

func (slice *mySlice) add(value int) {
   slice =append(slice, value)
}
```

Figure 3.3. Example of type definition with associated method

files, allowing direct calls without previous object instantiation (i.e., such functions are similar to static methods in Java, for example). For this reason, functions also represent a relevant type in the node hierarchy.

As shown in Table 3.1, all defined types are eligible for *Move* and *Rename* refactorings. However, only the *Function* type is a possible candidate for the *Extract*, *Inline*, and *Change Signature* refactorings.

3.3 Evaluation

3.3.1 Evaluation Design

In this section, we present an evaluation of precision and recall for RefDiff4Go. We also present a comparison with results reported by Silva et al. [2021] in a recent study, in which the authors evaluated the precision and recall of three RefDiff plugins (Java, JavaScript, and C). Finally, we discuss the execution time of RefDiff4Go in six well-known Go projects. To the best of our knowledge, there is no dataset with real refactorings information for the Go language. In this way, we decided to create our own dataset as described next.

3.3.2 Computing Precision

To assess the accuracy of the plugin, we ran RefDiff on the commit history of popular Go open-source projects and computed precision through manual validation. Specifically, we take the following steps:

Step 1. We selected ten open-source Go projects available on GitHub, ranked by their number of stars. According to recent research, the number of stars is a reliable popularity metric [Silva and Valente, 2018; Borges et al., 2016]. Next, we removed non-software systems like awesome lists, books, and sample repositories. We remained with six well-known projects, as described in Table 3.2.

Table 3.2. Open source projects used in evaluation

Name	Description	Commits	Stars (K)	KLOC
Kubernetes	Container scheduling and management	63,726	68.5	4,521
Moby	Platform for OS-level virtualization	38,521	57.7	1,398
Hugo	Framework for building websites	5,731	45.7	135
Gin	Web framework written in Go	1,369	40.3	15
Frp	A fast reverse proxy	845	37.7	19
Gogs	Self-hosted Git service	5,360	35.1	89

Step 2. Then, we ran RefDiff4Go in the commit history of the six selected projects, from the most recent to the oldest commit in the main branch. We compared each revision with its ancestor. As usual when using refactoring detection systems, such as RefDiff, we also discarded commits with more than one parent, particularly merge commits (since in this case RefDiff might report the same refactorings multiple times, after comparing the results with each parent commit).

We also exclude from evaluation test files, automatically generated code, project dependencies, and testing utilities. We use a regular expression to validate file extensions and suffixes (e.g. `test.go`, `gen.go`, `pb.go`). We also removed common utility folders such as `test`, `testdata`, and `vendor` (project dependencies).

Table 3.3 shows the number of refactorings detected in the analyzed projects. The project with the highest number of refactorings is Kubernetes, with approximately 52K refactorings. Besides that, the most frequent refactoring is *Change Signature*, with approximately 37K instances. By contrast, the lowest number of refactorings was detected in Gin, 203 refactoring instances, and the less frequent type is *Rename* Type with 57 refactorings.

Refactoring	Kubernetes	Moby	Hugo	Gin	Frp	Gogs	Total
Change Signature	30,035	5,241	859	60	245	1,016	37,456
Extract Function	832	395	96	26	17	105	1,471
Inline Function	69	13	8	1	3	4	98
Move Struct	1,525	203	53	18	19	77	1,895
Move Type	524	31	10	5	2	4	576
Move File	5,470	745	68	7	99	307	6696
Move Function	8,924	2,178	675	41	69	330	12,217
Move Interface	80	18	18	1	1	3	121
Rename Struct	580	150	38	4	8	56	836
Rename Type	41	8	7	0	0	1	57
Rename File	451	177	30	6	11	47	722
Rename Function	5,991	1,330	449	38	112	348	8,268
Rename Interface	70	16	7	2	0	1	96
Total	52,907	10,079	2,202	203	559	2,254	68,204

Table 3.3. Number of refactorings per project

Step 3. After creating a list of refactorings detected by RefDiff4Go, we randomly select a sample of ten refactorings for each type. It is also important to highlight that we selected at most one refactoring per commit (i.e., to avoid possible bias due to preferences of a given developer or due to a major maintenance work under progress in the project).

Once the refactorings list was defined, we manually inspected the textual diff of each commit to check whether the identified refactoring was a True Positive (TP) or a False Positive (FP). A TP is a refactoring detected by RefDiff4Go that was indeed performed in the code, as confirmed in our manual validation. By contrast, a FP is a source code transformation that was not considered a refactoring in our manual validation.

Step 4. After completing the manual inspection, we compute precision using the following formula:

$$P = \frac{TP}{TP + FP} \tag{3.1}$$

3.3.3 Computing Recall

To measure recall, we relied on the textual description of each commit of each project searching for possible documented refactorings. We also discarded commits with more than one parent in this step.

For example, to identify refactorings of type *Move*, we select commit messages with words such as *move*, *moving*, or *migrate*. The following example represents a commit message extracted from the Moby/Moby project, describing a *Move Struct* refactoring:[1]

"Move Termios struct to os specific file"

However, some refactorings like Inline Function are not explicitly described in the commit message, so we decided to use terms with a related meaning, such as simplify or merge. The following commit message documents an *Inline Function* refactoring performed in a commit from Kubernetes:[2]

"Merge 3 resource allocation priority functions"

We finished our search after finding ten distinct commits for each refactoring operation supported by RefDiff4Go. In summary, our goal was to create a "ground truth" (or oracle) of refactoring operations. However, not all types of refactorings had documented changes. An example is inline functions, from which we only found eight occurrences. In total, we collected 128 commits to compute recall.

However, it is also possible that a documented refactoring (in a commit description message) was not performed in the code. To discard such cases, we manually checked each commit in our initial ground truth, searching for the documented refactoring in the respective commit diff. To replace these commits, we performed a new draw with a second manual inspection.

A False Negative (FN) is a refactoring from the ground truth that is not detected by RefDiff4Go. Finally, to compute the recall score, we use the following formula:

$$R = \frac{TP}{TP + FN} \tag{3.2}$$

We also combine the precision (P) and recall (R) using a harmonic mean, also known as F1-Score, as in the following formula:

$$F1 = 2 \cdot \frac{P \cdot R}{P + R} \tag{3.3}$$

Table 3.4 shows the total number of commits analyzed per refactoring type for the precision and recall evaluation. In total, we relied on 258 refactorings, 130 for precision and 128 for recall.

[1] `https://github.com/moby/moby/commit/a70dd659`

[2] `https://github.com/kubernetes/kubernetes/commit/c65225ee`

Table 3.4. Number of commits used for evaluating of precision an recall

Refactoring	Precision	Recall
Change Signature	10	10
Extract Function	10	10
Inline Function	10	8
Move Struct	10	10
Move TypeDef	10	10
Move File	10	10
Move Function	10	10
Move Interface	10	10
Rename Struct	10	10
Rename TypeDef	10	10
Rename File	10	10
Rename Function	10	10
Rename Interface	10	10
Total	130	128

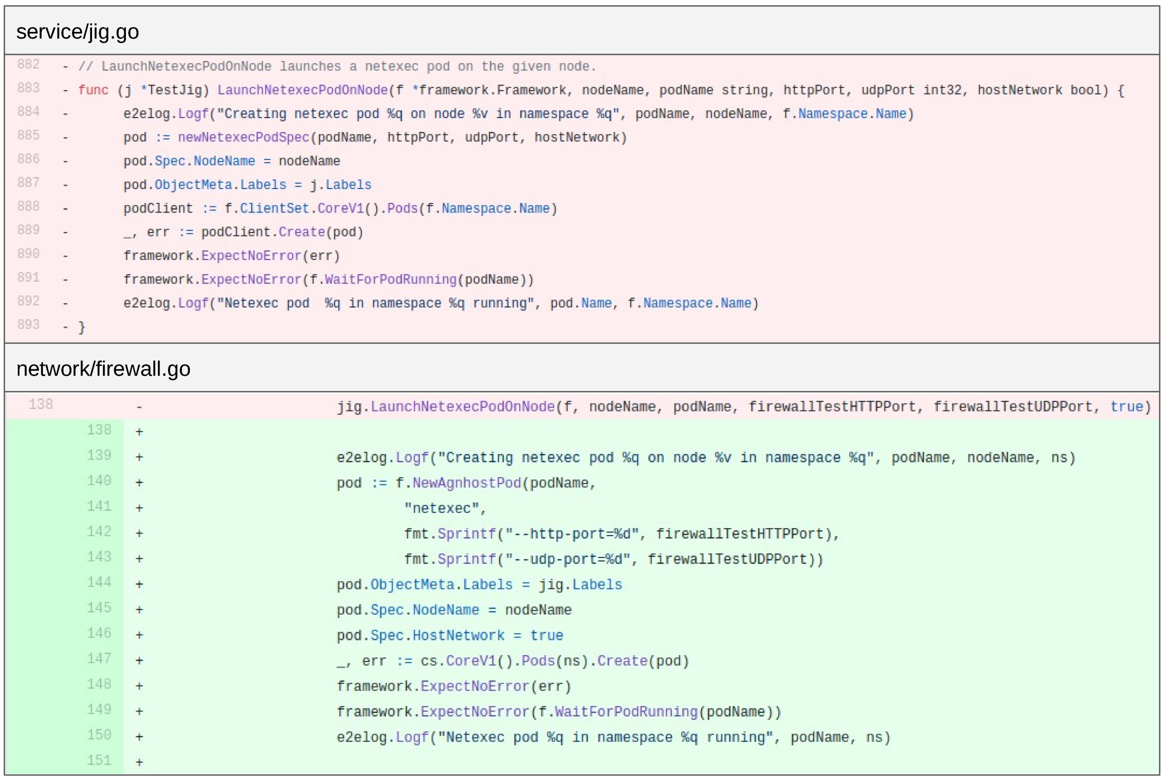
3.4 Results

In this section, we present the results of the RefDiff4Go evaluation. Table 3.5 presents the results for precision, recall and F1-Score for each type of refactoring. Precision ranges from 0.70 (*Move Function*) to 1.0 (*Inline Function*, *Move File*, and *Rename Struct*). The average precision is 0.92. Recall ranges from 0.5 (*Inline Function*) to 1.0 (*Rename Struct* and *Rename File*), reporting an average of 0.80, which is therefore slightly lower than precision. Table 3.5 also shows the F1 scores for each refactoring operation. The average F1 score is 0.86.

As presented in Table 3.5, some refactorings like *Inline Function* have a low recall value (0.67). In this way, we investigated some commits related to this class of refactoring searching for possible causes. By definition, *Inline Function* replaces a function call by the function body. However, after the replacement, developers often modify the inlined code. Since RefDiff4Go has only access to the full modification performed in the commit (i.e., the tool works in a commit granularity level), it considers that the new code is very different from the original function body. As a result, the tool does not detect an Inline Function refactoring operation in such case.

Next, we give a real example. Commit 6a42e1, from the Kubernetes project, received the FN classification for the *Inline Function* refactoring.[3] The commit message documents the refactoring with the following message: *"Inline/simplify two used-*

[3] `https://github.com/kubernetes/kubernetes/commit/6a42e1`


service/jig.go

```
- // LaunchNetexecPodOnNode launches a netexec pod on the given node.
- func (j *TestJig) LaunchNetexecPodOnNode(f *framework.Framework, nodeName, podName string, httpPort, udpPort int32, hostNetwork bool) {
-       e2elog.Logf("Creating netexec pod %q on node %v in namespace %q", podName, nodeName, f.Namespace.Name)
-       pod := newNetexecPodSpec(podName, httpPort, udpPort, hostNetwork)
-       pod.Spec.NodeName = nodeName
-       pod.ObjectMeta.Labels = j.Labels
-       podClient := f.ClientSet.CoreV1().Pods(f.Namespace.Name)
-       _, err := podClient.Create(pod)
-       framework.ExpectNoError(err)
-       framework.ExpectNoError(f.WaitForPodRunning(podName))
-       e2elog.Logf("Netexec pod  %q in namespace %q running", pod.Name, f.Namespace.Name)
- }
```

network/firewall.go

```
-                       jig.LaunchNetexecPodOnNode(f, nodeName, podName, firewallTestHTTPPort, firewallTestUDPPort, true)
+
+                       e2elog.Logf("Creating netexec pod %q on node %v in namespace %q", podName, nodeName, ns)
+                       pod := f.NewAgnhostPod(podName,
+                               "netexec",
+                               fmt.Sprintf("--http-port=%d", firewallTestHTTPPort),
+                               fmt.Sprintf("--udp-port=%d", firewallTestUDPPort))
+                       pod.ObjectMeta.Labels = jig.Labels
+                       pod.Spec.NodeName = nodeName
+                       pod.Spec.HostNetwork = true
+                       _, err := cs.CoreV1().Pods(ns).Create(pod)
+                       framework.ExpectNoError(err)
+                       framework.ExpectNoError(f.WaitForPodRunning(podName))
+                       e2elog.Logf("Netexec pod %q in namespace %q running", podName, ns)
+
```

Figure 3.4. Example of Inline Function from Kubernetes project

Table 3.5. Evaluation results by refactoring type

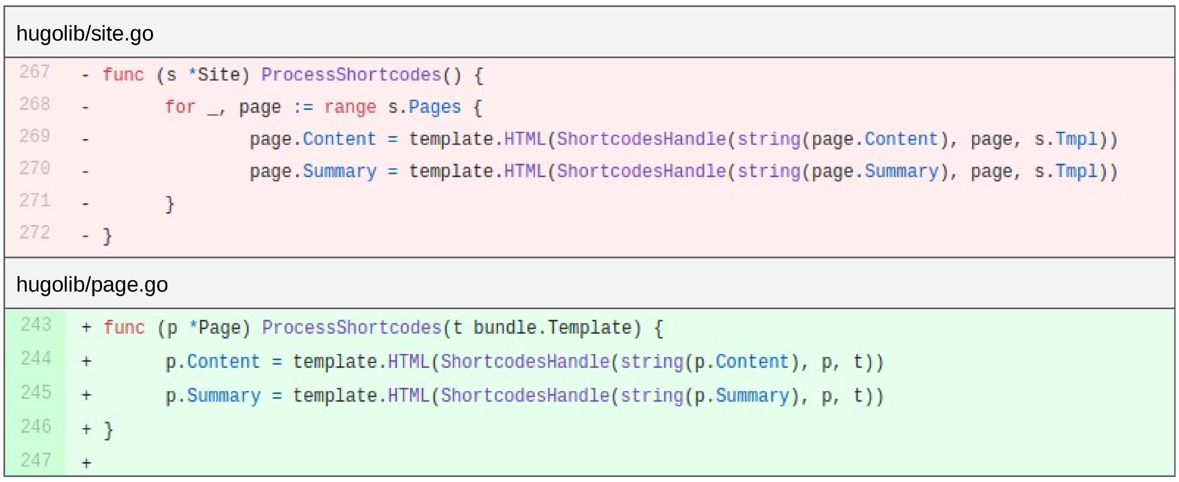
Refactoring	Precision	Recall	F1-Score
Change Signature	0.90	0.80	0.85
Extract Function	0.90	0.90	0.90
Inline Function	1.00	0.50	0.67
Move Struct	0.90	0.80	0.85
Move TypeDef	0.80	0.90	0.85
Move File	1.00	0.70	0.82
Move Function	0.70	0.80	0.75
Move Interface	0.90	0.60	0.72
Rename Struct	1.00	1.00	1.00
Rename TypeDef	0.90	0.60	0.72
Rename File	1.00	1.00	1.00
Rename Function	1.00	0.90	0.95
Rename Interface	1.00	0.90	0.95
All	0.92	0.80	0.86

only-once service test helper functions”. As a result, we included the *Inline Function* refactoring shown in Figure 3.4 in our ground truth, where the call of function

`LaunchNetexecPodOnNode` is replaced by its body.

However, as we can see, the code transformation was not a pure *Inline Function*, since the inlined function body was also modified. The manual validator considered that this modification does not mischaracterize an *Inline Function* refactoring. On the other side, this modification prevented the trigger of an *Inline Function* by RefDiff4Go. In summary, this example shows that there is a degree of subjectivity when refactoring operations are evaluated under the granularity of commits.

A similar case is observed in Commit 0e04b9 from Hugo project, where the developer has documented a commit with the following message:[4] *"Moving processing short codes to the page"*. As illustrated in Figure 3.5, the operation performed by the developer consists of moving a function between two files. RefDiff4Go did not identify the refactoring activity, which received an FN classification. Similar to the *Inline Function* refactoring presented earlier, the developer moved the code, but also changed its content.


hugolib/site.go

```
- func (s *Site) ProcessShortcodes() {
-       for _, page := range s.Pages {
-               page.Content = template.HTML(ShortcodesHandle(string(page.Content), page, s.Tmpl))
-               page.Summary = template.HTML(ShortcodesHandle(string(page.Summary), page, s.Tmpl))
-       }
- }
```

hugolib/page.go

```
+ func (p *Page) ProcessShortcodes(t bundle.Template) {
+       p.Content = template.HTML(ShortcodesHandle(string(p.Content), p, t))
+       p.Summary = template.HTML(ShortcodesHandle(string(p.Summary), p, t))
+ }
+
```

Figure 3.5. Example of Move Function refactoring in Hugo Project

Therefore, the results reveal that refactorings related to code movement—such as extract, inline, and move—are more susceptible to failures when the developer performs the operation and also modifies the behavior of the target code.

3.4.1 Comparison with Java, JavaScript, and C

In this section, we present a comparison between RefDiff4Go and the other programming languages supported by RefDiff: Java, JavaScript, and C. However, it is not

[4]https://github.com/gohugoio/hugo/commit/0e04b9

possible to compare all refactoring types. For example, *Pull Up Method* and *Push-Down Method* refactorings are only detected by RefDiff for Java.

On the other side, some refactoring types have similarities, allowing a direct comparison. For example, a Struct component in Go is similar to a Class in Java and JavaScript. Table 3.6 shows the selection of compatible refactorings for each language.

Table 3.6. Refactorings compatibility considered in the evaluation

Go	Java	JavaScript	C
Change Sign	-	-	Change Sign
Extract Func	Extract Meth	Extract Func	Extract Func
Inline Func	Inline Meth	Inline Func	Inline Func
Move Struct	Move Class	Move Class	-
Move File	-	Move File	Move File
Move Func	Move Meth	Move Func	Move Func
Rename Struct	Rename Class	Rename Class	-
Rename File	-	Rename File	Rename File
Rename Func	Rename Meth	Rename Func	Rename Func

3.4.2 Java

Table 3.7 shows the comparison between RefDiff4Go and the RefDiff plugin for Java. We present the F1 score for each refactoring and the number of refactorings used in the precision and recall analysis. It is also important to observe that the number of refactorings in Java is greater than in Go. In total, we compared 3,023 refactorings in Java with 118 refactorings in Go.

Table 3.7. Comparison of results between Refdiff4Go and Java

Refactoring	Go		Java	
	F1-Score	#	F1-Score	#
Extract Function / Method	0.90	20	0.78	1,037
Inline Function / Method	0.67	18	0.82	122
Move Class / Struct	0.85	20	0.98	1,100
Move Function / Method	0.75	20	0.84	319
Rename Class / Struct	1.00	20	0.90	95
Rename Function / Method	0.95	20	0.80	350
All	0.85	118	0.85	3,023

Considering the average F1-score, the results for Go and Java are exactly the same: F1-score is 0.85 for both languages. RefDiff4Go performed better than the Java plugin for *Extract Function*, *Rename Struct* and *Rename Function* refactorings.

Summary: RefDiff4Go has a similar result to the Java plugin, recording the final F1-Score of 0.85, which is exactly the one reported by the Java plugin.

3.4.3 JavaScript

We also compared the results obtained by RefDiff4Go with the ones achieved by the JavaScript plugin. Table 3.8 shows the F1-Score for compatible refactorings. In this case, the number of refactorings used in the Go results is higher (158 refactorings) than the ones used for JavaScript (122 refactorings).

Table 3.8. Comparison of results between Refdiff4Go and JavaScript

Refactoring	Go		JavaScript	
	F1-Score	#	F1-Score	#
Extract Function	0.90	20	0.90	20
Inline Function	0.67	18	0.53	15
Move Struct / Class	0.85	20	-	2
Move File	0.82	20	1.00	20
Move Function	0.75	20	0.95	20
Rename Struct / Class	1.00	20	-	5
Rename File	1.00	20	0.89	20
Rename Function	0.95	20	0.85	20
All	0.87	158	0.83	122

JavaScript and Go plugins present similar results. RefDiff4Go presented a lowest score only in *Move Function* (0.75) against 0.95 and *Move File* (0.82) when the JavaScript plugin presents 1.0 of F1 score. It is worth mentioning that we could not calculate the F1-Score for *Move* and *Rename Struct/Class* obtained by JavaScript, once the authors did not report (in their original study) the precision value.

Summary: RefDiff4Go presents a better result than the JavaScript plugin, reporting a F1 score of 0.87 in comparison of 0.83 achieved by JavaScript.

3.4.4 C

Table 3.9 presents the results of the comparison between RefDiff4Go and the C plugin. Differently from other languages, C and Go have similar types of refactorings. Thus, we did not rely on equivalences when comparing the results. However, although both languages have *Struct* as the principal data structure abstraction, the C plugin does not support refactorings involving this kind of component, such as *Move Struct* and *Rename Struct*.

Table 3.9. Comparison of results between Refdiff4Go and C

Refactoring	Go		C	
	F1-Score	#	F1-Score	#
Change Signature	0.85	20	0.95	20
Extract Function	0.90	20	0.82	20
Inline Function	0.67	18	0.64	20
Move File	0.82	20	1.00	20
Move Function	0.75	20	0.80	20
Rename File	1.00	20	1.00	20
Rename Function	0.95	20	0.90	20
All	0.85	138	0.87	140

In general, both tools achieved similar results. For example, RefDiff4Go has a better result for *Extract Function* (0.9), *Inline Function* (0,67), against 0.82 and 0.64 for the C plugin, respectively. However, Refdiff4Go has a worst result mainly for *Move File*, reporting a F1-Score of 0.82 against 1.0 of C.

Summary: RefDiff C plugin presents a slightly better result, reporting an F1 score of 0.87 against 0.85 presented by Go.

3.4.5 Execution Time

Besides comparing recall and precision, we also evaluated the execution time spent by RefDiff4Go. For this purpose, we started with the 110,728 commits from our initial dataset. Then, we discarded commits with more than one parent (e.g. merge commits) and commits that did not contain changes in source code. The final dataset used to measure the execution time contains 70,645 commits and 68,204 refactorings.

Figure 3.6 presents a graph of the execution time by commit, using a logarithmic scale. To perform the tests, we used a Core I5-7400 computer with 16 GB of RAM running Ubuntu 18.04 Linux distribution.

We can notice that RefDiff4Go presented a median execution time of 171 milliseconds. However, we also identify a large number of commits (14.9%) that are processed in less than ten milliseconds, which are basically commits with trivial changes in small

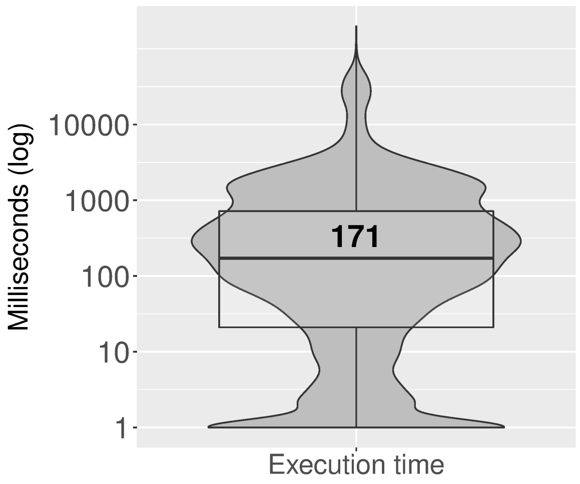


Figure 3.6. Execution time per commit (logarithmic scale)

files. As another point to remark, a small fraction of the commits (1.7%) required more than 10 seconds to execute. Normally, these commits perform massive project changes. An example can be found in Kubernetes where more than 1,900 files were modified in a single commit.[5] Figure 3.7 shows the execution time for each analyzed system. Gogs is the system with the highest execution time, with median of 406 milliseconds, followed by Hugo (median 278 ms) and Moby (median of 246 ms).

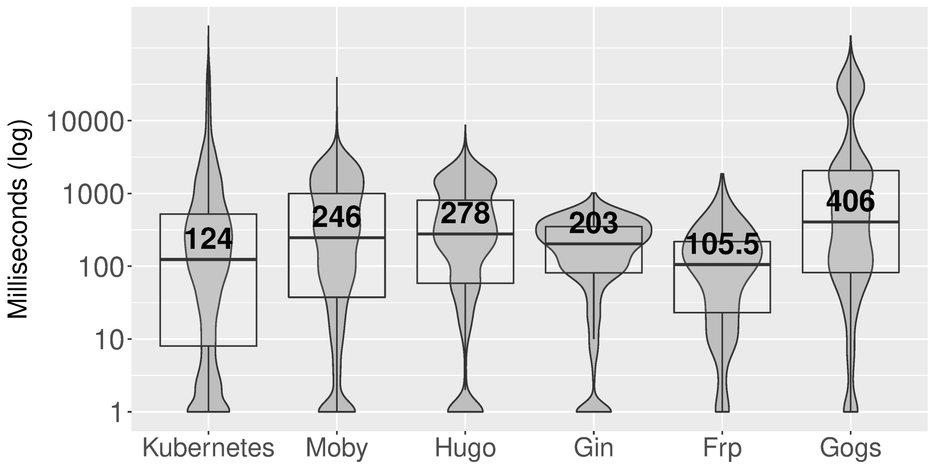


Figure 3.7. Execution time of RefDiff4Go by project

[5]https://github.com/kubernetes/kubernetes/commit/fb9bb501

3.5 Threats to Validity

Dataset: GitHub has thousands of Go projects and we have built our dataset using the top-6 code projects by number of stars. Therefore, our dataset represents a small sample of the entire universe of Go projects. On the other hand, relevant projects such as Kubernetes and Moby are included in our evaluation. We are also aware that the analyzed refactorings do not cover the entire code base and is a threat to validity.

Evaluation: Another threat to validity is found in the comparison between language plugins. The evaluation conducted by Silva et al. [2021] uses an oracle of 3,249 refactorings for Java. However, we have not found an oracle of refactorings for Go that covered the same order of refactorings. Therefore, as an alternative, we created our own oracle with much less refactorings (130 refactorings). On the other side, for the C and JavaScript languages, the study conducted by Silva et al. [2021] relies on a similar number of refactorings (122 for JavaScript and 140 for C). Finally, we acknowledge that refactoring classification is a subjective task, particularly when conducted by comparing commits that include other source code modifications, performed after the refactoring, as we discussed in Section 3.6.

Go Syntax: To build the CST, it is necessary to identify the functions present in a source code file and all the function calls made by each one. In Go, it is necessary to load all the files in a given scope to identify the source of an external function call. However, due to performance reasons, RefDiff processes only the files modified in a commit. Consequently, the function call identification supported by RefDiff4Go is limited to the scope of the modified files, similar to the C and JavaScript plugins.

3.6 Final Remarks

In this chapter, we presented RefDiff4Go, an extension of RefDiff for detecting refactorings in Go software projects, which is capable to identify 13 different types of refactoring. We evaluated the tool with six well-known open-source projects and compared them with plugins for other languages: Java, JavaScript, and C. The results report a precision of 92% and 80% of recall, resulting in an F1 Score of 0.86. Our evaluation also showed that RefDiff4Go has a similar accuracy than other language plugins and good execution time, recording a median of 171 milliseconds per commit.

RefDiff4Go is publicly available on GitHub.[6]

[6]`https://github.com/rodrigo-brito/refdiff-go`

Chapter 4

RAID: Refactoring-Aware Code Reviews

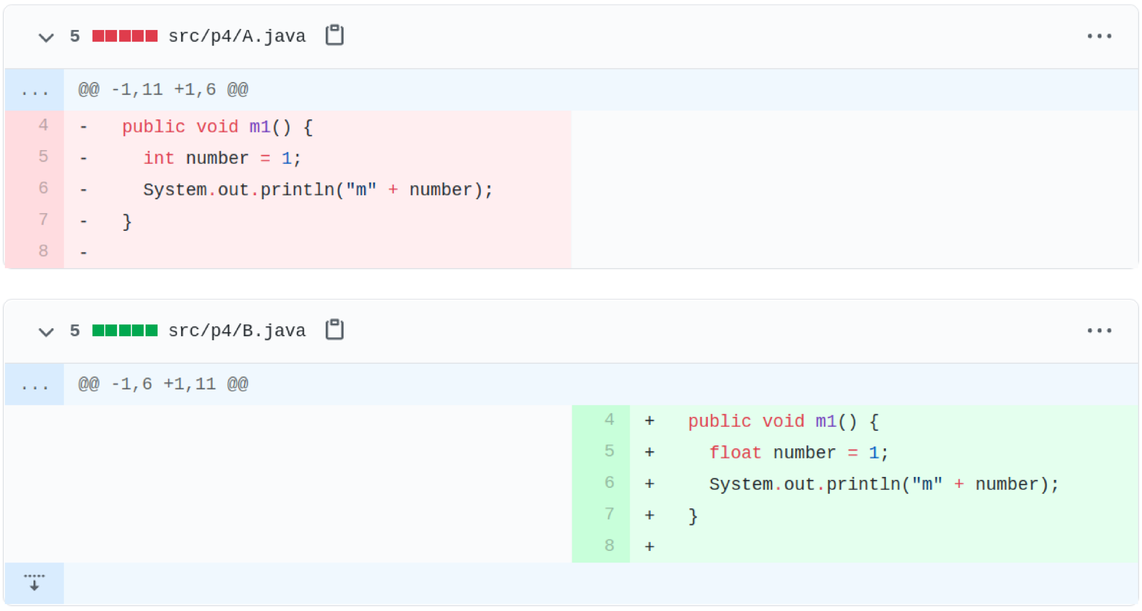
In this chapter, we present RAID, a refactoring-aware and intelligent diff tool. RAID assists developers in code review activities by automatically detecting refactoring operations included in pull requests. In Section 4.1, we present the motivation and challenges of diff analysis during code reviews. RAID architecture is presented in Section 4.2. Finally, in Section 4.4 we discuss the lessons learned after a field experiment when eight professional developers used RAID during three months.

4.1 Introduction

Code review is a widely used software engineering practice [Bacchelli and Bird, 2013; Sadowski et al., 2018]. It has its origins in the 70s, when it was performed according to formal and strict guidelines, such as the ones proposed by Fagan [1976]. Over the years, lightweight code review practices have emerged and gained popularity, in order to make the process more agile. Nowadays, software companies of all sizes and impact require their developers to engage in code reviews. For example, "code review is one of the most important and critical processes at Google" [Winters et al., 2020]. Besides that, modern version control platforms—such as GitHub and GitLab—are contributing to popularize code reviews by means of pull/merge requests.

However, code review takes time and may introduce delays in the release of new versions. The reason is that it is a manual process that requires expertise in the codebase and careful inspection of textual diffs. In fact, diffs only provide a low level representation of code changes, which is based on added (+) and deleted lines of code (-). As a combined consequence of these two factors—manual inspections using line-based tools—software developers spend a considerable time performing code reviews. For example, Bosu and Carver [2013] and Sadowski et al. [2018] report that, in specific contexts, the time spent in review tasks ranges from 3 to 6.4 hours per week, on average.

Particularly, current diff tools do not automatically detect refactorings in the inspected code change. For example, suppose a refactoring that moves a function `f()` from file `A` to file `B`. This move is represented in current state-of-the-practice diff tools as a set of lines deleted (-) from `A` and by a set of lines added (+) to `B`. Therefore, reviewers need first infer that these added/deleted lines indeed represent a move function. This is a not trivial inference when the change is complex, distributed over multiple files and when the refactoring is combined with other changes, such as bug fixes and the implementation of new features, as usual in practice [Silva et al., 2016]. After inferring the refactoring, reviewers should also compare the code after and before the change, to review possible changes performed in the moved function. The reason is that refactorings are usually followed by minor edits in the code, particularly when we compare the changes at the level of commits.



```
5 src/p4/A.java
@@ -1,11 +1,6 @@
-   public void m1() {
-     int number = 1;
-     System.out.println("m" + number);
-   }
-
```

```
5 src/p4/B.java
@@ -1,6 +1,11 @@
+   public void m1() {
+     float number = 1;
+     System.out.println("m" + number);
+   }
+
```

Figure 4.1. Default diff including a Move Function refactoring (`m1` is moved from `A.java` to `B.java`)

In this chapter, we describe the key features and architecture of a tool for supporting *refactoring-aware code reviews*. Our goal is to alleviate the cognitive effort associated with code reviews, by automatically detecting refactoring operations included in pull requests. Besides supporting refactoring detection, the proposed tool—called

RAID (or Refactoring-aware and Intelligent Diffs)—seamlessly instruments current diff tools with information about refactorings. As a result, reviewers can easily inspect the changes performed in the refactoring code after the operation. Regarding its architecture, RAID is built on the top of recent and automatic refactoring tools. Particularly, RAID relies on RefDiff [Silva et al., 2021; Silva and Valente, 2017], which is the first tool that detects refactorings in other programming languages, besides Java. Finally, RAID operates with a low runtime overhead and it is fully integrated with state-of-the-practice continuous integration pipelines (GitHub Actions) and browsers (Google Chrome).

This chapter is divided into three main sections:

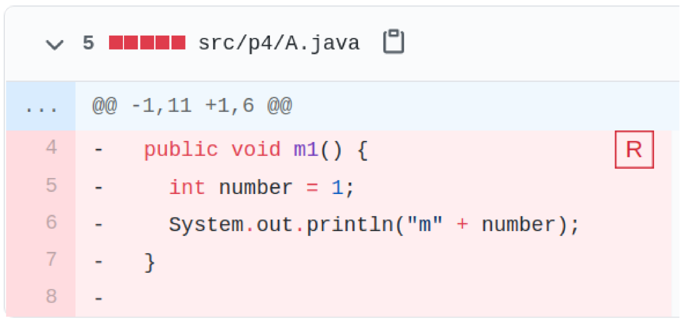
1. In Section 4.2, we present RAID main features and Web-based interface.
2. In Section 4.3, we describe RAID internal architecture, including its integration with third-party tools, such as GitHub (Actions and pull requests) and browsers (Chrome).
3. In Section 4.4, we document the results and lessons learned in a field experiment with eight professional developers who used RAID during three months. We concluded that RAID can indeed reduce the cognitive effort required for detecting and reviewing refactorings. For example, the number of lines that need to be reviewed decreases from 14.5 to 2 lines (median values) in the case of move refactorings and from 113 to 55 lines (median values) in the case of extractions. We also collected the perceptions of the study participants about our tool, using to this purpose a post-experiment survey.

RAID—our key contribution in this chapter—is publicly available in this URL: `https://github.com/rodrigo-brito/refactoring-aware-diff`.

4.2 RAID in a Nutshell

RAID is a tool for instrumenting textual diffs—particularly, the ones provided by GitHub—with information about refactorings. Typically, refactorings are represented in textual diffs as a sequence of removed lines in the left (-) and a sequence of added lines in the right. Therefore, code reviewers must infer by themselves whether this "difference" represents a refactoring operation, which requires an amount of cognitive effort. Figure 4.1 shows an example with a *Move Function* refactoring. In this case, method `m1` is moved from `A.java` (removed lines, in the left) to `B.java` (added lines, in the right).

Essentially, RAID preprocesses the default diff—provided by GitHub—in a seamless way and generates an enhanced diff visualization with explicit refactoring information. Figure 4.2 presents an example of refactoring annotation, in which RAID included an "R" button at the end of line four. This button is included in the refactoring source and destination, allowing for two-way navigation.



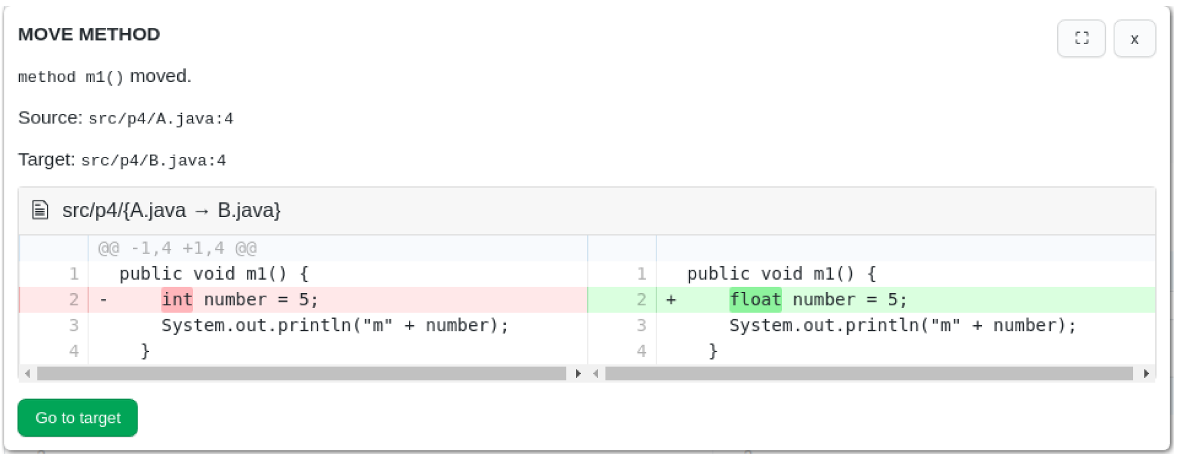
```
5 src/p4/A.java
@@ -1,11 +1,6 @@
-   public void m1() {                         R
-     int number = 1;
-     System.out.println("m" + number);
-   }
-
```

Figure 4.2. Diff instrumented by RAID (an "R" button is added to the diff indicating the function is the part of a refactoring)

When the developer clicks in this button, a float window provided by RAID is displayed, as shown in Figure 4.3. As we can see, this window includes detailed information about the detected refactoring. First, on the top, we have the following information about the operation: name (*Move Function*), short description (method `m1()` moved), source (`A.java`, line 4) and target file (`B.java`, line 4). Next, we can see a side by side view of the code before and after the move operation. Finally, in this internal diff, the lines changed in the moved code are highlighted, i.e., reviewers can easily review them. For the aforementioned reasons, we claim RAID reduces the cognitive effort required to review refactorings.

Extract Function is another refactoring that benefits from RAID features. As illustrated in Figure 4.4, suppose a method `isEven` is extracted from a method `m1`. In this case, reviewers need to compare the code of the source method (`m1`) before and after the extraction. They also need to locate and review the code of the extracted method (`isEven`). As the reader can check in Figure 4.4, RAID helps in both tasks. Particularly, the floating window provided by RAID includes, in the bottom, the code of the extracted method (`isEven`). Therefore, reviewers do not need to search for this method in the textual diff.

Besides move and extract refactorings, RAID provides information about other refactorings (when they are detected in a pull request). For example, in the case of pull up/push down refactorings, the information is similar to the one we described for



MOVE METHOD

`method m1()` moved.

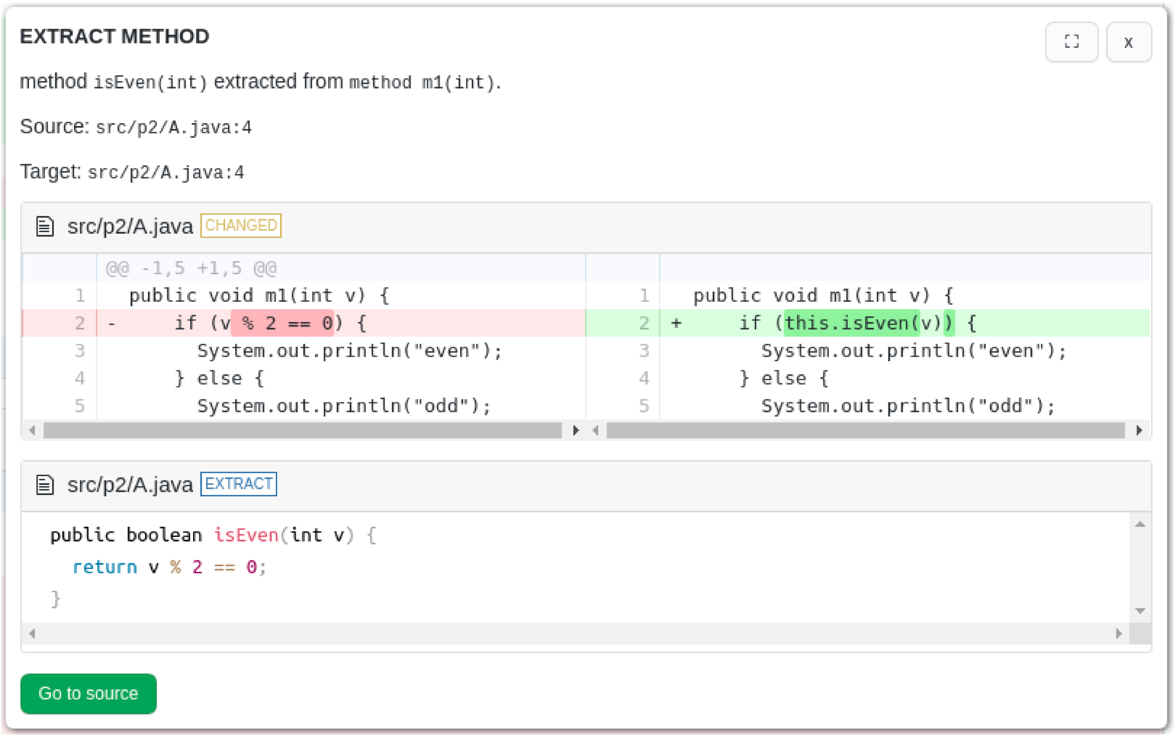
Source: `src/p4/A.java:4`

Target: `src/p4/B.java:4`

src/p4/{A.java → B.java}

```
@@ -1,4 +1,4 @@
  public void m1() {                     |    public void m1() {
-     int number = 5;                    | +      float number = 5;
      System.out.println("m" + number);  |        System.out.println("m" + number);
    }                                    |      }
```

Figure 4.3. Example of diff including a Move Function, as presented by RAID (this window is opened after clicking in the "R" button shown in Figure 4.2)



EXTRACT METHOD

method `isEven(int)` extracted from `method m1(int)`.

Source: `src/p2/A.java:4`

Target: `src/p2/A.java:4`

src/p2/A.java

```
@@ -1,5 +1,5 @@
  public void m1(int v) {                |    public void m1(int v) {
-     if (v % 2 == 0) {                  | +      if (this.isEven(v)) {
        System.out.println("even");      |          System.out.println("even");
      } else {                           |        } else {
        System.out.println("odd");       |          System.out.println("odd");
```

src/p2/A.java

```
public boolean isEven(int v) {
  return v % 2 == 0;
}
```

Figure 4.4. Example of window documenting an Extract Function, as presented by RAID. We can see the lines changes in the original method (top) and also the code of the extracted method (bottom)

move operations. In the case of *Rename* and *Change Signature* refactorings, RAID highlights the changes in the component's name and signatures. Table 4.1 lists the refactorings detected by RefDiff/RAID.

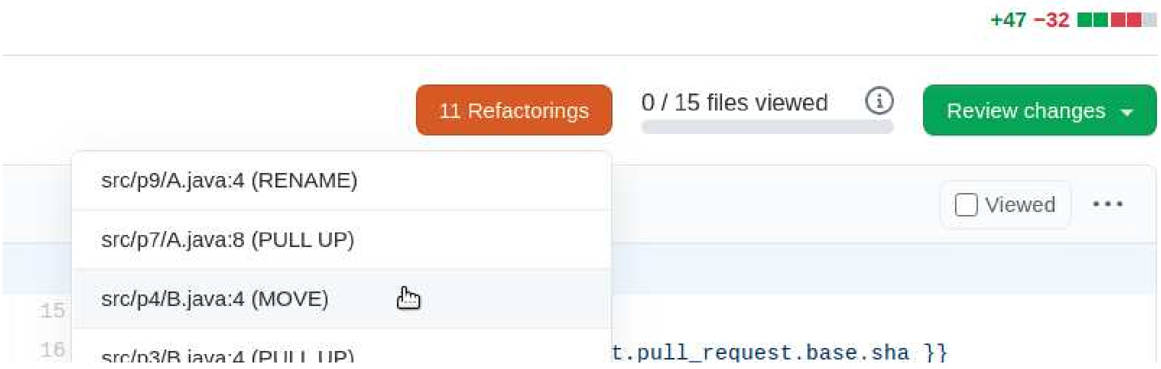
All windows provided by RAID include a "Go to source" button, which allows the

Table 4.1. Refactorings detected by RefDiff/RAID

Language	Refactorings
C	Move, Extract Function, Inline Function, Rename, Change Signature
Go	Move, Extract Function, Inline Function, Rename, Change Signature
JavaScript	Move, Extract Function, Inline Function, Rename, Change Signature
Java	Move, Extract Function, Inline Function, Rename, Change Signature, Pull Up, Push Down

reviewer to locate the source of the performed refactoring, thus reducing the effort to find the code in the default diff provided by GitHub.

Finally, RAID instruments the default diff's toolbar with a button that provides a list with all refactorings detected in the pull request. Figure 4.5 shows an example of a pull request including 11 refactorings.



src/p9/A.java:4 (RENAME)
src/p7/A.java:8 (PULL UP)
src/p4/B.java:4 (MOVE)
src/p3/B.java:4 (PULL UP)

```
t.pull_request.base.sha }}
```

Figure 4.5. RAID adds a button in the toolbar providing easy access to the list of refactorings in a pull request

4.3 RAID Architecture

RAID architecture is composed of three components: (1) RAID GitHub Action (RGA), which is responsible for detecting and extracting refactorings; (2) RAID Chrome Extension (RCE), which is a browser extension that instruments GitHub's default diff pages with refactoring information; and (3) RAID Server, which provides a REST API for storing refactoring metadata and to connect the previous mentioned applications.

Figure 4.6 presents RAID's workflow. First, when a developer submits a pull request, the RGA component is automatically called to detect the refactorings performed in the changed code. This data is then transmitted and stored by the RAID server. Finally, when a developer opens a diff page in his browser, RCE is automatically called to instrument the page with the refactoring information retrieved from the RAID server.

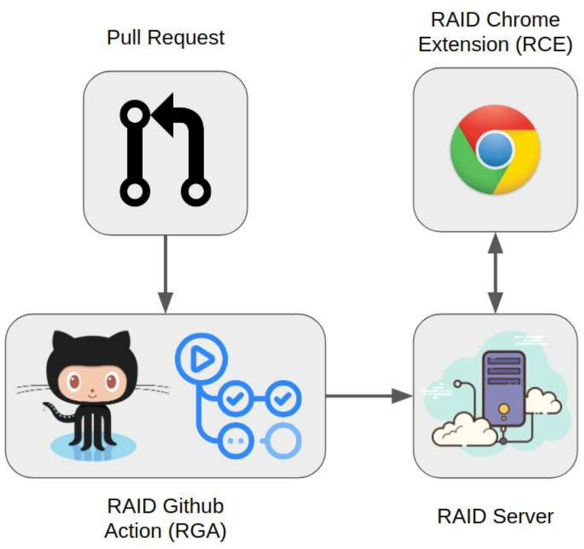


Figure 4.6. RAID main components and execution flow

In the following subsections, we describe each of these architectural components in details.

4.3.1 RAID GitHub Action (RGA)

RAID GitHub Action component is responsible for analyzing the source code submitted by developers as part of a pull request. RGA is implemented in Java and it is executed in a Docker container. The main function of RGA is to detect the refactorings performed in a pull request. To this purpose, RGA fully relies on a third-party tool: RefDiff [Silva et al., 2021; Silva and Valente, 2017], which is a multi-programming language refactoring detection tool. Specifically, RefDiff detects 13 types of refactorings in four programming languages: Java, JavaScript, C and Go. Consequently, RAID can be seamlessly used with any of such languages.

Figure 4.7 shows RGA internal modules. As we mentioned, RGA is called after a pull request event is sent by GitHub Actions. In Step 1, RGA receives information about the pull request, such as branches, revisions, and commits. After that, the Git module loads the project and stores the commit history in a temporary directory.

At this point, it is important to remind that GitHub provides two distinct diffs for a given pull request: (1) the main diff, which compares changes performed in the last commit of the pull request with the repository HEAD; and (2) individual commit diffs,

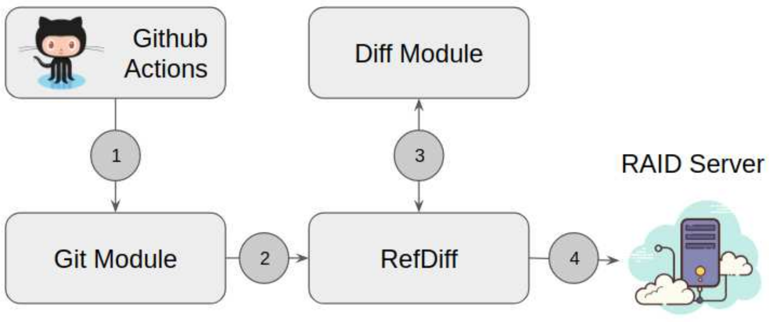


Figure 4.7. RAID GitHub Action (RGA) modules and workflow

which compare the changes performed in a given commit C_i with its parent commit. Therefore, RefDiff is called by RAID to extract the refactorings included in each of these diffs. To be clearer, assume a pull request with three commits: C_1, C_2, and C_3. In this case, RAID calls RefDiff four times, with the following pairs of commits: (C_3, HEAD), (C_3, C_2), (C_2, C_1), and (C_1, HEAD).

Next, the Diff Module computes the internal diffs that are showed in RAID's floating windows (see Figures 4.3 and 4.4). For example, in the case of a *Move Function* refactoring, RAID highlights the changes performed in the moved method (since move refactorings might include minor changes in the code after its movement [Silva et al., 2021; Tsantalis et al., 2020]). Finally, RGA transmits the refactorings data to the RAID Server.

4.3.2 RAID Chrome Extension (RCE)

The RAID Chrome Extension (RCE) presents the refactoring data provided by the RAID server. RCE is implemented in JavaScript (∼1.1 KLOC) and it currently supports Google Chrome browser. The extension is automatically called by Chrome after a valid GitHub's pull request URL is opened by the browser. This URL includes information about the pull request, which is used by RCE to contact the RAID server and to obtain data about possible refactorings.

For each refactoring, RCE transparently modifies the DOM (Document Object Model) of the default visualization provided by GitHub and inserts a button, identified by the letter "R", at the end of the line where the refactoring was identified, as we already illustrated in Figure 4.2. The RCE module is also responsible for displaying the information of refactoring activities, as presented in Figure 4.3. Finally, RCE assists the reviewer to navigate between the refactored code elements.

4.3.3 RAID Server

The RAID Server acts as a bridge between the RCA and RCE components, i.e., it receives the refactoring information after each RGA execution and sends the refactorings to RCE when requested. To this purpose, the server provides a REST API implemented in the Go language. Internally, the server stores the refactoring data in a non-relational database at Google Firebase.[1] For open-source projects, RAID provides a public API. For private repositories, the server supports a custom credentials configuration to control user access.

4.4 Field Experiment

4.4.1 Methodology

To evaluate RAID, we relied on a field experiment. More specifically, we obtained permission to include the tool in the development workflow of a medium-sized technology company that develops software for musical products. In this way, two development teams (with 5 and 3 developers) used the tool during three months, from June to September 2020. Table 4.2 describes basic information about the software projects and the pull requests performed during the experiment. All four projects are implemented in the Go programming language.

Table 4.2. Number of pull requests, commits, and lines of code by project

Name	Pull Requests	Commits	KLOC	Team
Project A	94	102	370	1
Project B	86	89	196	1
Project C	54	65	34	1
Project D	91	118	154	2
All	325	374	754	

In this company, all development teams use GitHub for version control. Developers also use pull requests followed by code reviews to integrate new code in the projects' repository. First, developers perform modifications in private forks and submit pull requests to the main repository. Then, a CI server is used to run linters and a suite of unit tests. Finally, code review is carried out before integrating the change in the mainline. The code needs to be approved by two team members who did not work in the change.

[1] `https://firebase.google.com`

Before starting the experiment, the participants received instructions and training on our tool (~30 minutes), followed by one week of warm-up and to get used with RAID. In addition, we provided support during the study period to clarify questions.

For the experiment, we instrumented RAID to collect all refactorings performed during the study period and also the user events. These events include the clicks in the "R" and "Go to source" buttons, and the time interval between opening and closing the floating windows with refactoring descriptions.

During the experiment, 325 pull requests were created. Out of them, 84 pull requests (26%) included refactoring activities. Figure 4.8 presents the distribution of refactorings for these 84 pull requests. The project with the highest number of refactorings per pull request is project B (median of 4 refactorings per PR), followed by project D (median of 2).

The difference in the number of refactorings per pull request can be explained by the level of maturity of the projects. Projects A and C are more stable, with more than 5 years of development. While projects B and D were created less than a year ago.

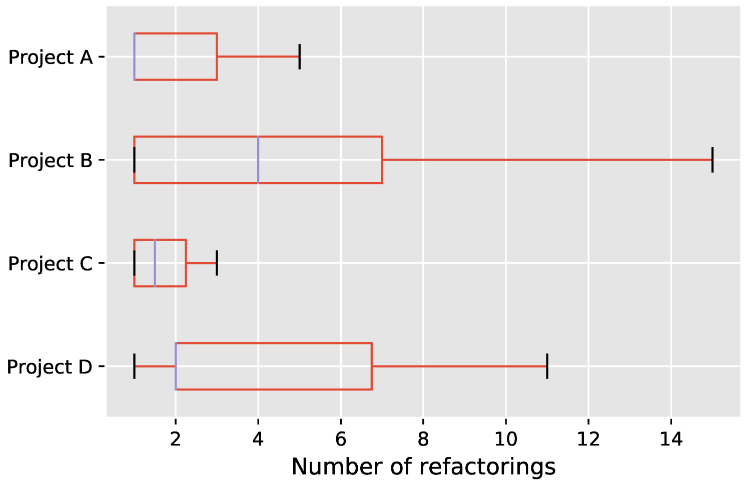


Figure 4.8. Number of refactorings per pull request

In total, we collected data about 685 refactorings. As we can see in Figure 4.9, the top-3 refactorings are *Change Signature* (498 occurrences, 73%), *Rename Function* (123 occurrences, 18%), and *Extract Function* (44 occurrences, 6%).

After the experiment, we sent a short survey to the participants asking their perceptions about RAID. In this survey, we asked three questions:

- What are the key benefits of RAID?

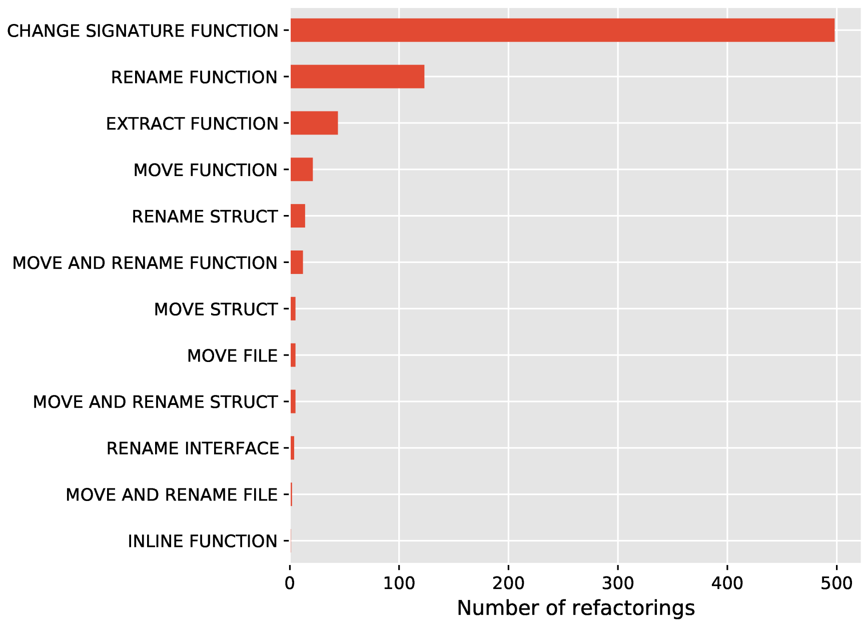


Figure 4.9. Most frequent refactorings

- What are the difficulties you experienced using RAID?
- Any additional comments or suggestions?

We received responses from all 8 participants.

4.4.2 Research Questions

We used the data collected in the field experiment to answer four research questions:

RQ1: What is the runtime overhead introduced by RAID?

We started with this question since it is important to check whether RAID introduces (or not) a delay in the code review process. In other words, after submitting a pull request, it is important that the information provided by RAID becomes available to code reviewers as soon as possible.

RQ2: How did developers use RAID during code review?

With this second RQ, we characterize the usage of RAID during the three months of our experiment. For example, we report and analyze the collected usage data, including most frequent refactorings with user events, reviewing time, and UI events performed by users on RAID's web-based interface.

RQ3: How much cognitive effort is reduced with RAID?

Although this is the key goal of RAID, it is not trivial to estimate the amount of cognitive effort that is saved when RAID is used to support code review tasks, particularly in the case of an experiment performed in the context of a real software company. For this reason, we estimate this effort using two proxies: Diff Code Churn (DCC), which is the number of lines needed to represent a code change in a textual diff and the distance of the code moved by a refactoring (if any).

RQ4: How did developers perceive RAID?

In this last research question, we report the perceptions of the participants about our tool, as commented by them in the post-experiment survey.

4.4.3 Experiment Results

RQ1: What is the runtime overhead introduced by RAID?

Regarding the execution time required by RAID to identify the refactorings, Projects A, C, and D have a median time lower than 40 seconds, while Project B has a median of 95.7 seconds. However, as previously presented in Figure 4.8, project B has the highest median number of refactorings per pull request. Moreover, most high runtime results observed in Project B refer to atypical changes performed in the code base, such as updating third party code. However, since the CI server runs RAID in parallel with other time-consuming activities such as unit tests, even a median time of 95 seconds is acceptable in practical code review scenarios.

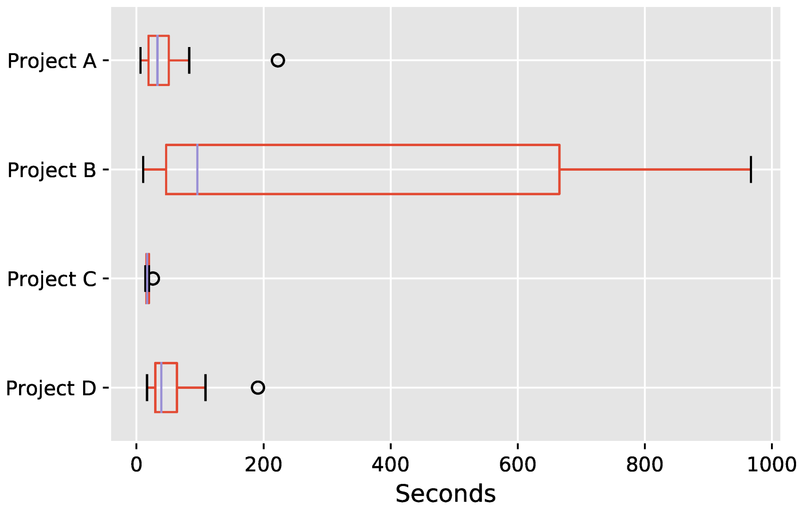


Figure 4.10. Execution time per project

Summary: Typically, the information provided by RAID is available to code reviewers in less than 2 minutes after submitting the pull request. Therefore, in practical terms, RAID does not delay the start of code reviews.

RQ2: How did developers use RAID during code review?

Figure 4.11 presents the most common refactorings with user events, i.e, refactoring operations performed in pull requests that were inspected by code reviewers through RAID, by clicking in the "R" buttons. As we can notice, most cases refer to refactorings performed over functions. Specifically, the top-3 operations comprise *Change Signature Function* (78 occurrences, 43.6%), *Extract Function* (35 occurrences, 19.6%), and *Move Function* (28 occurrences, 15.7%).

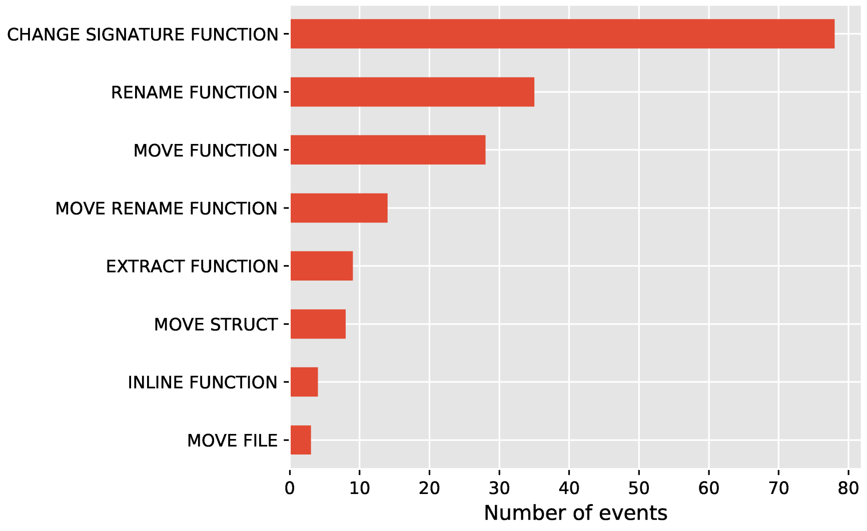


Figure 4.11. Most common refactorings with clicks in the "R" buttons

We also analyzed the percentage of clicks in the "R" buttons. Assume a button is presented x times. As a result, it received y clicks by code reviewers. The percentage represented in Figure 4.12 is the ration y/x. However, in this figure we only considered refactorings that appeared at least 10 times, i.e., $x \geq 10$. The reason is that the described ratio is not representative in the case of rare refactoring operations (e.g., 100% is a very different result depending on whether the button appeared 1 or 100 times).

As we can see in Figure 4.12, the top-3 most clicked refactorings (in percentage values) represent move operations, i.e., *Move Struct* (60%), *Move and Rename Function* (58%), and *Move Function* (29%). Interestingly, *Change Signature* is the most common refactoring (in absolute terms, Figure 11). However, only 13% of the "R" buttons with instances of this particular refactoring received clicks. A probable reason is that *Change Signature* is a relatively simple refactoring, when compared with move ones. On the other side, *Move Struct* and *Move Function* are the refactorings that benefit most from the diff improvements provided by RAID.

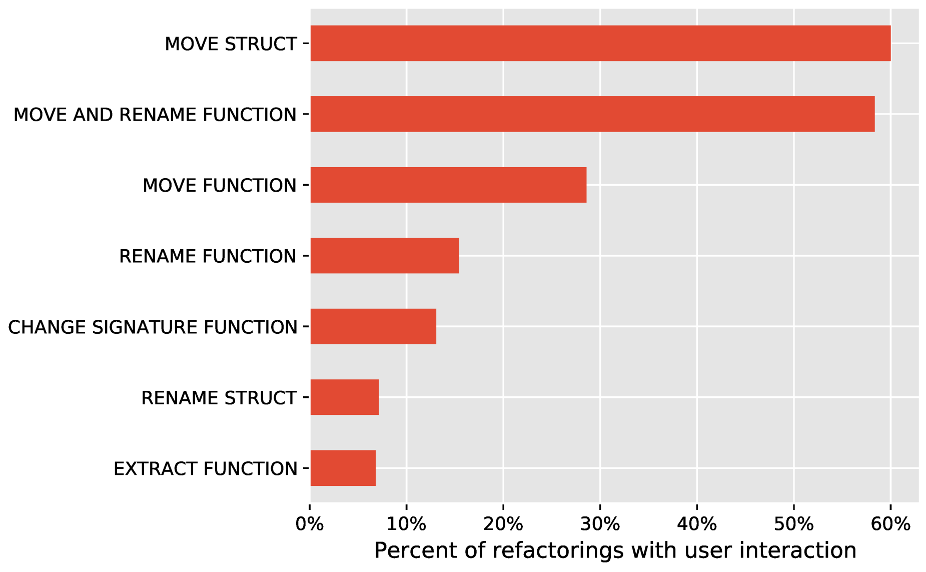


Figure 4.12. Ratio of clicks in "R" buttons, i.e., number of times the button received a click divided by number of times it was shown

GitHub also allows users to split the screen for a side-by-side comparison of the changes in a textual diff, i.e., the original version is presented on the left side and the modifications are presented on the right side (see an example in Figure 4.1). In this case, when a refactoring is detected, RAID adds an "R" button on both sides of the diff. For example, when a method is moved, on the left side a "R" appears on top of the deleted lines; and an identical button is added on the right side, on top of the new position of the method. Therefore, when reviewing refactorings, developers can click on both buttons. We instrumented RAID to collect the number of such clicks and the results are presented in Figure 4.13. As we can see, the click events are almost equally distributed between the two sides of a diff, with a minor preference for clicks on the left side (53.9%).

Finally, we computed the time spent by code reviewers when inspecting the information provided by RAID in the floating windows. The results are presented in Figure 4.14. Each distribution refers to the amount of time the reviewer kept the window opened. Considering the median of the distributions, the top-5 refactorings that required more reviewing time are *Inline Function* (12.4 sec), *Move Struct* (9.4 sec), *Move and Rename Function* (8.7 sec), *Extract Function* (8.7 sec), and *Move Function*

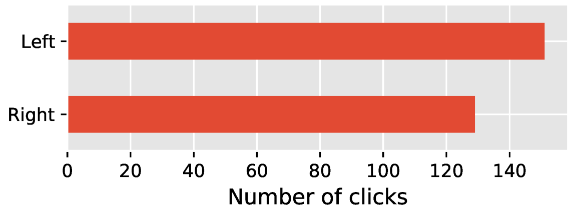


Figure 4.13. Number of clicks on the left and right buttons provided by RAID

(3.55 sec). Therefore, we can see that refactorings that involve moving code demand more time to consume the information provided by RAID.

By contrast, refactorings such as *Change Signature Function*, *Move File*, and *Rename Function* have low inspection time, with a median of less than 3 seconds. Indeed, both *Change Signature* and *Rename Function* are simple operations, where RAID does not provide improvements to the traditional diff, which probably explains their lower time.

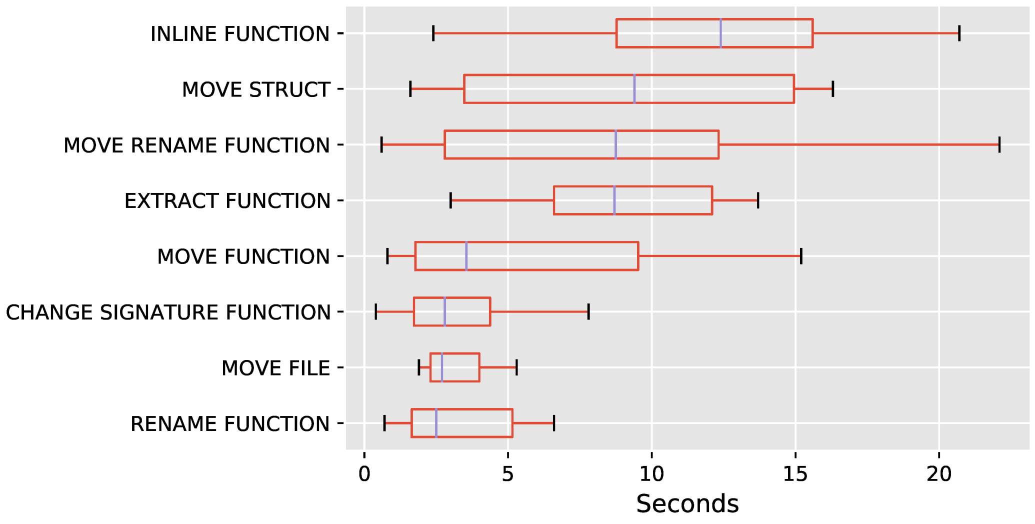


Figure 4.14. Time spent reviewing the information provided by RAID (per refactoring operation)

Summary: (1) in relative terms, reviewers relied on RAID to obtain information mostly for *Move Struct*, *Move and Rename Function*, and *Move Function*; (2) reviewers used both "R" buttons—available on the left and right side of the original diff—to request refactoring information provided by RAID; (3) on the median, reviewers spent from 2.5 seconds (*Rename Function*) to 12.4 seconds (*Inline Function*) to interpret the refactoring information provided by RAID.

RQ3: How much cognitive effort is reduced with RAID?

In this RQ, we provide indicators of the impact on cognitive effort achieved with RAID. For this purpose, we concentrate on two refactorings: *Move and Extract Function*, since they are more complex than simple refactorings, such as a rename.

Move Refactorings. Out of 32 move refactorings performed during the experiment, 11 operations move code between files located in different packages and 21 operations are performed in files from the same package. Among these, 9 operations are performed within the same file but between internal `structs`. Therefore, most of the move refactorings impact distinct files (71.8%). We claim it is more difficult to detect and review such moves when using traditional diffs. The reason is that the default visualization lacks information about the refactored files. On the other hand, with RAID developers can easily access this information, as illustrated in Figure 4.3.

We also found nine move refactorings that were performed in the same file. In Go, an internal move occurs when a function is moved to a new `struct` or moved between existing `structs` located in the same file. In this case, we computed the distance of the moved code, i.e., suppose a move from line x to line y, the distance in this case is $|y - x|$. The rationale is that the higher this distance, the higher the effort for a reviewer to detect and inspect the move, assuming she is using a non-refactoring aware diff tool. By contrast, with RAID, this effort does not exist since the tool automatically detects the refactoring and provides buttons to navigate from its source to target lines (and back).

Table 4.3 shows the values of the distance metric for the nine move refactorings restricted to the same file. As we can see, eight refactorings moved the code at least 40 lines above or below the original position. In one case, this distance is zero due to a coincidence, when multiple refactoring operations ended up aligning the diffs after the move. Therefore, the effort to locate the moved code—even when it is located in the same file—is not trivial, assuming code reviewers rely on traditional diffs. For example, a distance of 41 lines means the function is one page above or below the code under the reviewer focus in the browser.

Table 4.3. Distance of Move Function refactorings, i.e., line of the moved code after the operation minus line of the code before the operation

Refactoring	1	2	3	4	5	6	7	8	9
Distance	0	40	40	40	41	466	466	474	4,870

We also computed the Diff Code Churn (DCC) of the move refactorings. We refer to Diff Code Churn as the number of added (+) and deleted (-) lines showed in the diff interface to represent a refactoring. Considering the 32 move refactoring performed in the experiment, DCC is 14.5 lines of code (median values), as presented in Figure 4.15. On the other hand, when using RAID to review these moves, DCC is reduced to just 2 lines of code (median values). The reason is clear: RAID aligns the moved lines of code and only shows the minor changes performed in the code after the move (using + and - lines). Examples are presented in Figures 4.1 and 4.3. In Figure 4.1, we have a move represented in a traditional diff, where DCC is 10 lines (5 added plus 5 deleted lines). In Figure 4.3, we have the same move, as detected and presented by RAID. As we can see, DCC is just 2 lines.

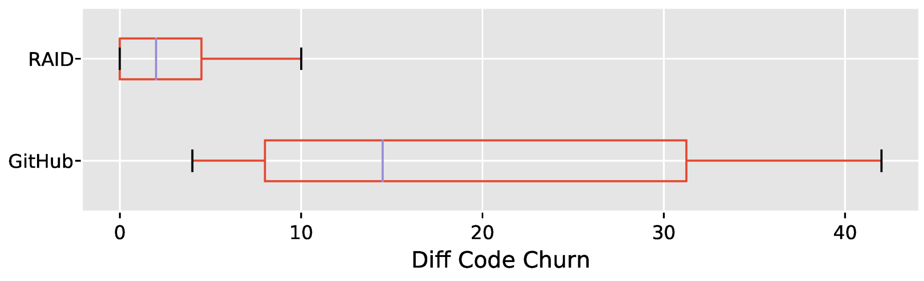


Figure 4.15. Diff Code Churn (DCC) when reviewing Move Function refactorings, when using RAID and GitHub diff

Summary: (1) 71.8% of the *Move Function* refactorings are inter-files, when RAID clearly outperforms non-refactoring aware diffs; (2) In the case of intra-file moves, the code is moved 41 lines after or below its original position (median results); (3) Diff Code Churn reduces from 14.5 lines to only 2 lines (median results), when moves are reviewed using RAID.

Extract Refactorings. We found 44 occurrences of *Extract Function*. First, we compared the distance between the first line of the source code and the first line of the extracted code. Figure 4.16 shows the distribution of the distance values. We can see that the extracted code is implemented 25 lines after or before the source method (median values). Therefore, these numbers suggest that it is probably not difficult to locate the method. On the other side, before locating the extracted method, reviewers should infer whether the diff contains an *Extract Function*, which might not be trivial.

We also computed the Diff Code Churn for extract refactorings. Figure 4.17 shows the distribution of DCC values, when using non-refactoring aware diffs and RAID. The median values decrease from 113 lines (when using GitHub's diff) to 55 lines when using RAID. The reason is that RAID reduces the number of lines to be analyzed, due to the precise identification of the extracted code. Figure 4.3 shows an example, when

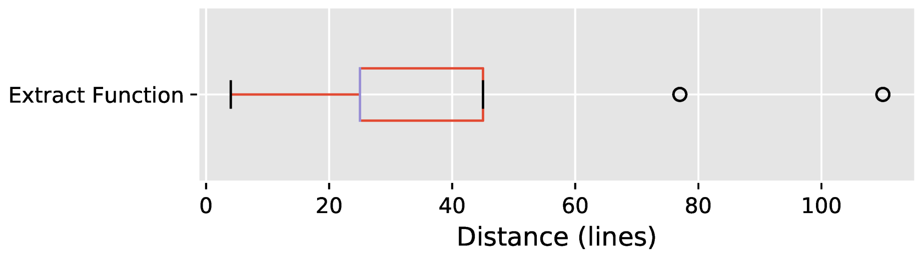


Figure 4.16. Distance of Extract Method refactorings, i.e., initial line of the extracted method minus initial line of the source method, in absolute terms

RAID highlighted the single line of code changed in the extracted method (in this case, just the variable type changed from `int` to `float`).

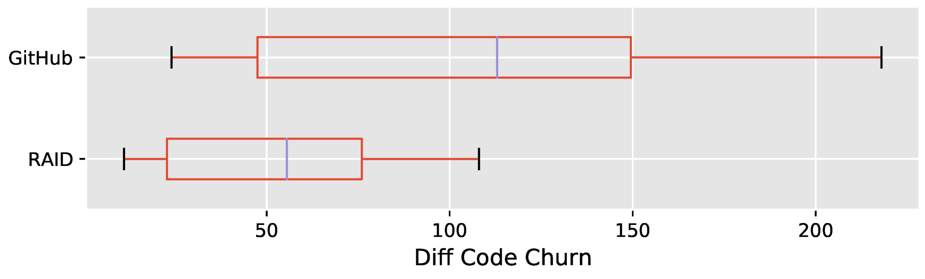


Figure 4.17. Diff Code Churn when reviewing Extract Function refactorings, when using GitHub Diff and RAID

Summary: (1) Methods are extracted 25 lines before or after the source method; (2) Diff Code Churn decreases from 113 lines to 55 lines when using RAID to review *Extract Function* refactorings.

RQ4: How did developers perceive RAID?

In this section, we report the results of the post-experiment survey.

RAID Benefits. All participants mentioned that RAID represents an improvement regarding GitHub's traditional diff. As examples, we received the following answers.

RAID highlights large migrations of code, easing code review when the snippet was just moved without major changes. (S1)

RAID allows a more efficient code review with a realistic diff, which shows the real changes that occurred in the code. (S2)

The tool makes the refactor recognition process instantaneous, and shows a more targeted and clean diff, facilitating and accelerating the understanding of the code. (S8)

Limitations. We received five responses pointing out limitations in RAID. For example, one reviewer reported that RAID did not identify refactorings in specific cases.

I particularly experienced some bugs where RAID did not detect possible refactors. (S7)

Indeed, RAID is based on RefDiff, which has 92% of precision and 80% of recall for Go [Brito and Valente, 2020]. Particularly, the tool might miss refactorings when developers, for example, make considerable changes in the code after the operation. In these cases, the tool might not detect a refactoring although a reviewer might still consider it as such.

Finally, two reviewers reported the limitation of browser support. Currently, RAID supports only Google Chrome. As future work, we intend to extend the tool to other browsers, such as Safari and Mozilla Firefox.

4.5 Threats to Validity

External Validity. In our field experiment, we evaluated four Go language projects from a medium-sized company. The participants used RAID during three months. Therefore, on the one hand, we strived to conduct a real experiment, with real-world systems and professional developers. On the other hand, we acknowledge that our results may change if we consider more projects, more developers, a longer time box, and other programming languages.

Construct Validity. First, the refactoring detection module relies on RefDiff [Silva et al., 2021], which has a precision of 92% and recall of 80%, in the case of Go projects [Brito and Valente, 2020]. In this way, RAID visualizations might miss refactorings (false negatives) or may incorrectly detect refactorings (false positives). Indeed, the presence of false negatives was mentioned by one of the experiment participants. However, we highlight that RefDiff accuracy is compatible with the state-of-the art in refactoring detection tools [Tsantalis et al., 2020, 2018]. Furthermore, to mitigate the impact of this threat on the field experiments results, we manually evaluated each of the refactorings used to compute the proxy metrics presented and discussed when answering RQ3. In total, we found 10 false positives (13.2%) among moves and extracts.

Internal Validity. A first possible internal threat to validity can be found in RQ1, where the size of pull requests (i.e., the number of modified lines) might have an influence on the execution time. In our third research question, we proposed two proxies to estimate the amount of cognitive effort reduced by RAID: Diff Code Churn (DCC) and distance of move refactorings. On the one hand, claim that such proxies have a positive correlation with the cognitive effort required in code reviews. On the other hand, they did not allow us to provide a clear measure of the effort reduction. Therefore, further studies—such as controlled ones—might contribute to clarify the effort saved with RAID. The inspection time of the float window is also a threat to validity, since a developer can have left the window opened beyond the time used in the review.

4.6 Final Remarks

In this chapter, we presented RAID, a refactoring-aware tool that instruments GitHub diff with refactoring information. We also conducted a field experiment with eight professional developers during three months and we concluded that RAID can reduce the cognitive effort required for reviewing refactorings when using textual diffs. In addition, our study reports a reduction in the number of lines required for reviewing such operations. In the case of move refactorings, the number of lines decrease form 14.5 to 2 lines (median values); and from 113 to 55 lines in the case of extractions.

RAID is public available at GitHub.[2]

[2]https://github.com/rodrigo-brito/refactoring-aware-diff

Chapter 5

Conclusion

In this chapter, we present the conclusion of this dissertation. First, in Section 5.1, we present the lessons and contributions provided by our work. Next, in Section 5.4, we present suggestions of future work.

5.1 Overview and Contributions

Our main contributions in this work are RefDiff4Go and RAID, as described next.

RefDiff4Go. In Chapter 3, we presented RefDiff4Go, an extension for RefDiff that detect refactoring activities for Go projects. The proposed tool identify 13 different types of refactorings, achieved a precision of 92% and recall of 80% in an evaluation with six well-known Go projects. Also, RefDiff4Go presented a similar accuracy with other RefDiff plugins, such as Java, JavaScript, and C.

RAID. In Chapter 4, we proposed a refactoring-aware tool that instrument refactoring activities in GitHub diffs. We evaluated the tool in a field experiment with eight developers. This study reported a reduction in the number of lines required to analyze a code review. For example, in the case of *Move* refactorings, the number of lines reduces from 14.5 to 2 in medium values, when using RAID.

5.2 Comparison with Existing Tools and Studies

There are also open source projects proposing refactoring-aware tools supporting visualization of diff. The tool presented by Tsantalis et al. [2020], called Refactoring Aware Commit Review, identifies refactorings in open source Java projects and lists the refactorings activities on GitHub diffs. RefactorInsight is another tool for visualizing refactorings through the history of commits, but within a IDE for Java and Kotlin.[1]

Both tools instrument textual diff with a list of refactoring operations, which are detected by RefactoringMiner [Tsantalis et al., 2020]. In Refactoring Aware Commit Review, after reviewers click on a given list element that describes a refactoring, they are directed to its specific line in the right side of GitHub's diff. Therefore, both tools do not instrument diff lines with refactoring data, unlike RAID that seamlessly provides the "R" buttons, which are added to both sides of a diff (see Figure 4.2). Also, these tools do not provide detailed information about *Move Function* refactorings—with the moved code appearing side by side, as in Figure 4.3—or detailed information about *Extract Function* refactorings—by highlighting the extracted code and the textual differences in the source method, as illustrated in Figure 4.4. Finally, unlike RAID, both tools are not integrated with a CI server, which is critical to provide refactoring data right after pull requests are submitted.

5.3 Limitations

The work presented in this dissertation has the following limitations:

- The evaluation of RefDiff4Go was limited to the list of refactorings evaluated in the initial study presented by Silva et al. [2021]. Furthermore, due to the incompatibility of programming languages, some refactorings are not directly comparable due to the lack of inheritance mechanism in languages such as C and Go.

- Due to limitations of the Go programming language, the field experiment did not cover all possible refactorings supported by RAID. Go does not have an inheritance structure, then refactorings such as *Pull Up* and *Push Down*, which were not covered, could also benefit from the diff provided by RAID.

- RAID was designed to instrument refactorings in diffs in the GitHub platform. However, other popular platforms like Gitlab and Bitbucket can also be used to perform code reviews.

[1]`https://github.com/JetBrains-Research/RefactorInsight`

5.4 Future Work

In this dissertation we presented RefDiff4Go, an extension for Go programming language, and RAID, a refactoring-aware and intelligent diff tool. We suggest as extension of this work:

- It would be interesting to conduct a survey with developers to better understand their needs and include support to new promising languages to RefDiff. Consequently, this will also include support for new languages for RAID.
- Configure and evaluate RAID with other refactoring detection tools, such as RefactoringMiner [Tsantalis et al., 2020].
- Provide plugin support to other browsers, such as Mozilla Firefox and Safari.
- Evaluate RAID using a controlled experiment to better understand the benefits and impacts of the tool.

Bibliography

Abid, C., Kessentini, M., Alizadeh, V., Dhouadi, M., and Kazman, R. (2020). How does refactoring impact security when improving quality? a security-aware refactoring approach. *IEEE Transactions on Software Engineering*, pages 1--1.

Alizadeh, V., Ouali, M. A., Kessentini, M., and Chater, M. (2019). Refbot: intelligent software refactoring bot. In *34th International Conference on Automated Software Engineering (ASE)*, pages 823--834.

Alves, E. L. G., Song, M., and Kim, M. (2014). RefDistiller: A refactoring aware code review tool for inspecting manual refactoring edits. In *22nd International Symposium on Foundations of Software Engineering (FSE)*, pages 751--754.

amd Kensuke Tokoda, K. M. (2008). Security-aware refactoring alerting its impact on code vulnerabilities. In *15th Asia-Pacific Software Engineering Conference*, pages 445--452.

Bacchelli, A. and Bird, C. (2013). Expectations, outcomes, and challenges of modern code review. In *35th International Conference on Software Engineering (ICSE)*, pages 712--721.

Bibiano, A. C., Fernandes, E., Oliveira, D., Garcia, A., Kalinowski, M., Fonseca, B., Oliveira, R., Oliveira, A., and Cedrim, D. (2019). A quantitative study on characteristics and effect of batch refactoring on code smells. In *13th International Symposium on Empirical Software Engineering and Measurement (ESEM)*, pages 1--11.

Borges, H., Hora, A., and Valente, M. T. (2016). Understanding the factors that impact the popularity of GitHub repositories. In *32nd IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 334--344.

Bosu, A. and Carver, J. C. (2013). Impact of peer code review on peer impression formation: A survey. In *7th International Symposium on Empirical Software Engineering and Measurement (ESEM)*, pages 133--142.

Brito, A., Hora, A., and Valente, M. T. (2020). Refactoring graphs: Assessing refactoring over time. In *27th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 367–377.

Brito, R., Brito, A., Brito, G., and Valente, M. T. (2019). GoCity: Code city for Go. In *26th International Conference on Software Analysis, Evolution and Reengineering (SANER), Tool Track*, pages 649--653.

Brito, R. and Valente, M. T. (2020). RefDiff4Go: Detecting refactorings in Go. In *14th Brazilian Symposium on Software Components, Architectures, and Reuse (SBCARS)*, pages 1--10.

Caulo, M., Lin, B., Bavota, G., Scanniello, G., and Lanza, M. (2020). Knowledge transfer in modern code review. In *28th International Conference on Program Comprehension (ICPC)*, pages 230—240, New York, NY, USA. Association for Computing Machinery.

Chaparro, O., Bavota, G., Marcus, A., and Penta, M. D. (2014). On the impact of refactoring operations on code quality metrics. In *30th International Conference on Software Maintenance and Evolution (ICSME)*, pages 456--460.

Coelho, F., Massoni, T., and L. G. Alves, E. (2019). Refactoring-aware code review: A systematic mapping study. In *3rd International Workshop on Refactoring (IWoR)*, pages 63--66.

Davila, N. and Nunes, I. (2021). A systematic literature review and taxonomy of modern code review. *Journal of Systems and Software*, 177:1--36.

Dig, D., Comertoglu, C., Marinov, D., and Johnson, R. (2006). Automated detection of refactorings in evolving components. In *20th European Conference on Object-Oriented Programming (ECOOP)*, pages 404--428.

Donovan, A. A. and Kernighan, B. W. (2015). *The Go programming language*. Addison-Wesley Professional.

Edmundson, A., Holtkamp, B., Rivera, E., Finifter, M., Mettler, A., and Wagner, D. (2013). An empirical study on the effectiveness of security code review. In

International Symposium on Engineering Secure Software and Systems, pages 197--212. Springer.

Fagan, M. E. (1976). Design and code inspections to reduce errors in program development. *IBM Systems Journal*, 15(3):182--211.

Fowler, M. (2018). *Refactoring: improving the design of existing code.* Addison-Wesley Professional.

Ge, X. and Murphy-Hill, E. (2011). Benefactor: a flexible refactoring tool for eclipse. In *International Conference Companion on Object Oriented Programming Systems Languages and Applications Companion (OOPSLA)*, pages 19--20.

Ge, X., Sarkar, S., Witschey, J., and Murphy-Hill, E. (2017). Refactoring-aware code review. In *IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, pages 71--79.

Kim, M., Gee, M., Loh, A., and Rachatasumrit, N. (2010). Ref-finder: Arefactoring reconstruction tool based on logic query templates. In *8th Symposium on Foundations of Software Engineering (FSE)*, pages 371--372.

Kim, M., Zimmermann, T., and Nagappan, N. (2012). A field study of refactoring challenges and benefits. In *20th International Symposium on the Foundations of Software Engineering (FSE)*, pages 50:1--50:11.

Kim, M., Zimmermann, T., and Nagappan, N. (2014). An empirical study of refactoring challenge and benefits at Microsoft. *IEEE Transactions on Software Engineering*, 40(7):633--649.

Krasniqi, R. and Cleland-Huang, J. (2020). Enhancing source code refactoring detection with explanations from commit messages. In *27th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 512--516.

Lacerda, G., Petrillo, F., Pimenta, M., and Guéhéneuc, Y. G. (2020). Code smells and refactoring: a tertiary systematic review of challenges and observations. *Journal of Systems and Software*, page 110610.

Lahiri, S. K., Hawblitzel, C., Kawaguchi, M., and Rebêlo, H. (2012). SYMDIFF: A Language-Agnostic Semantic Diff Tool for Imperative Programs. In Madhusudan, P. and Seshia, S. A., editors, *Computer Aided Verification*, pages 712--717, Berlin, Heidelberg. Springer Berlin Heidelberg.

Mazinanian, D., Ketkar, A., Tsantalis, N., and Dig, D. (2017). Understanding the use of lambda expressions in Java. *Programming Languages*, 1(85):85:1--85:31.

McGraw, G. (2008). Automated code review tools for security. *Computer*, 41(12):108--111.

McIntosh, S., Kamei, Y., Adams, B., and Hassan, A. E. (2016). An empirical study of the impact of modern code review practices on software quality. *Empirical Software Engineering*, 21(5):2146--2189.

Mumtaz, H., Alshayeb, M., Mahmood, S., and Niazi, M. (2018). An empirical study to improve software security through the application of code refactoring. *Information and Software Technology*, 96:112--125.

Murphy-Hill, E., Parnin, C., and Black, A. P. (2009). How we refactor, and how we know it. In *31st International Conference on Software Engineering (ICSE)*, pages 287--297.

Murphy-Hill, E., Parnin, C., and Black, A. P. (2012). How we refactor, and how we know it. *IEEE Transactions on Software Engineering*, 38(1):5--18.

Negara, S., Chen, N., Vakilian, M., Johnson, R. E., and Dig, D. (2013). A Comparative Study of Manual and Automated Refactorings. In *27th European Conference on Object-Oriented Programming (ECOOP)*, pages 552--576.

Pantiuchina, J., Zampetti, F., Scalabrino, S., Piantadosi, V., Oliveto, R., Bavota, G., and Penta, M. D. (2020). Why developers refactor source code: A mining-based study. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 37(4):1--32.

Pike, R. (2012 (accessed June 25, 2021)). Go at Google: Language design in the service of software engineering. In *Golang Talks*. `http://talks.golang.org/2012/splash.article`.

Prete, K., Rachatasumrit, N., Sudan, N., and Kim, M. (2010). Template-based reconstruction of complex refactorings. In *26th International Conference on Software Maintenance (ICSM)*, pages 1--10.

Ratzinger, J., Sigmund, T., and Gall, H. C. (2008). On the relation of refactorings and software defect prediction. In *5th Working Conference on Mining Software Repositories (MSR)*, pages 35--38.

Sadowski, C., Söderberg, E., Church, L., Sipko, M., and Bacchelli, A. (2018). Modern code review: A case study at Google. In *40th International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*, pages 181--190.

Salton, G. and McGill, M. J. (1986). *Introduction to Modern Information Retrieval.* McGraw-Hill, Inc., USA. ISBN 0070544840.

Shen, B., Zhang, W., Zhao, H., Liang, G., Jin, Z., and Wang, Q. (2019). IntelliMerge: A refactoring-aware software merging technique. *Programming Languages*, 3(170):170:1--170:28.

Silva, D., da Silva, J. P., Santos, G., Terra, R., and Valente, M. T. (2021). RefDiff 2.0: A multi-language refactoring detection tool. *IEEE Transactions on Software Engineering*, 1(1):1--17.

Silva, D., Tsantalis, N., and Valente, M. T. (2016). Why we refactor? confessions of GitHub contributors. In *24th International Symposium on the Foundations of Software Engineering (FSE)*, pages 858--870.

Silva, D. and Valente, M. T. (2017). RefDiff: Detecting refactorings in version histories. In *14th International Conference on Mining Software Repositories (MSR)*, pages 1--11.

Silva, H. and Valente, M. T. (2018). What's in a GitHub star? understanding repository starring practices in a social coding platform. *Journal of Systems and Software*, 146:112--129.

Terra, R., Valente, M. T., Miranda, S., , and Sales, V. (2018). JMove: A novel heuristic and tool to detect move method refactoring opportunities. *Journal of Systems and Software*, 138:19--36.

Tsantalis, N., Guana, V., Stroulia, E., and Hindle, A. (2013). A multidimensional empirical study on refactoring activity. In *Conference of the Centre for Advanced Studies on Collaborative Research (CASCON)*, pages 132--146.

Tsantalis, N., Ketkar, A., and Dig, D. (2020). RefactoringMiner 2.0. *IEEE Transactions on Software Engineering*, pages 1--21.

Tsantalis, N., Mansouri, M., Eshkevari, L. M., Mazinanian, D., and Dig, D. (2018). Accurate and efficient refactoring detection in commit history. In *40th International Conference on Software Engineering (ICSE)*, pages 483--494.

Winters, T., Wright, H., and Manshreck, T. (2020). *Software Engineering at Google: Lessons Learned from Programming over Time*. O'Reilly Media.

Xing, Z. and Stroulia, E. (2005). UMLDiff: An algorithm for object-oriented design differencing. In *20th International Conference on Automated Software Engineering (ASE)*, pages 54--65.